

Implementierung einer Squad KI für Shooter in Unity

A Implementation of a Squad AI in Unity

Philip Nuss

Bachelor-Abschlussarbeit

Betreuer: Prof. Dr. Christoph Lürig

Saarbrücken, 15.02.2022

---

## **Kurzfassung**

In dieser Arbeit wurde mithilfe der Spieleengine "Unity" ein Prototyp eines Top Down Shooters erstellt, indem der Spieler gegen 4 Gegner, die innerhalb einer 4er Gruppe auch Squad genannt kämpft. Die KI der einzelnen Squadmitglieder wurde über eine State Machine realisiert, dessen Fundament das Scriptableobject System von Unity ist. Darüber hinaus sind die Entscheidungen der KI von einem Utilitiesystem abhängig. Zusätzlich erhalten die Squadmitglieder von einer übergeordneten KI Anweisungen. Dadurch interagieren die Gegner miteinander und agieren sowohl individuell als auch als Team gegen den Spieler.

---

# Inhaltsverzeichnis

<b>1 Einleitung</b> .....	1
1.0.1 Motivation .....	1
1.0.2 Einführung .....	1
1.0.3 Vorhandene Lösungsansätze .....	4
1.0.4 Zielsetzung .....	5
<b>2 Grundlagen</b> .....	6
2.0.1 Finite State Machine .....	6
2.0.2 Utility AI .....	7
2.0.3 Pathfinding .....	11
2.0.4 Unity .....	13
<b>3 Umsetzung</b> .....	17
3.0.1 Vorgehensweise .....	17
3.0.2 Spieler und Allgemeine Systeme .....	18
3.0.3 A* Navigationsalgorithmus .....	19
3.0.4 StateMachine mit Scriptableobjects .....	21
3.0.5 Squad AI .....	26
3.0.6 Utility System .....	33
<b>4 Zusammenfassung und Ausblick</b> .....	41
4.1 Zusammenfassung .....	41
4.2 Weiterentwicklung .....	42
<b>Literaturverzeichnis</b> .....	43
<b>A Selbstständigkeitserklärung</b> .....	46

---

## Abbildungsverzeichnis

1.1	Enter the Gungeon .....	2
1.2	Direktor AI Alien Isolation .....	5
2.1	Finite State Machine .....	6
2.2	Entscheidungsfaktor .....	8
2.3	Lineare Funktion .....	9
2.4	Quadratisch Funktion .....	9
2.5	Logistische Funktion .....	10
2.6	Teilweise Lineare Funktion .....	10
2.7	A* Algorithmus berechneter Weg .....	11
2.8	Komponenten eines Gameobjects in Unity .....	13
2.9	Standart Nutzerinterface in Unity .....	14
2.10	Renderpipeline Schritte .....	16
3.1	A* Navigationsgitter im Spiel .....	19
3.2	A* Bewegung im Spiel .....	20
3.3	Drag & Drop einer Action in die Liste des States .....	21
3.4	Vergleich von normale States und geklonte States .....	22
3.5	Inspector eines Scriptableobjects von PlugableAction .....	24
3.6	Inspector eines Scriptableobjects von Decision .....	25
3.7	Darstellung der Statepakete .....	26
3.8	Darstellung Squadposition Suchalgorithmus .....	28
3.9	Deckungspunkt Überprüfung .....	29
3.10	öffentliche Utilityliste des UtilityManagers in der UnityEngine .....	34
3.11	Utilityscore Funktion : Agressiv .....	35
3.12	Utilityscore Funktion : Flankable .....	36
3.13	Utilityscore Funktion : Flankable .....	36
3.14	Utilityscore Funktion : RunTowardsCoverSprint .....	38
3.15	Utilityscore Funktion : RunTowardsCoverAttack .....	38
3.16	Utilityscore Funktion : CanGiveFireAssistance .....	39
3.17	Utilityscore Funktion : CanGivePickUpAssistance .....	40

---

## Listings

3.1	Code zum Statewechsel . . . . .	23
3.2	Code zum bestimmen des besten UtilityAIFlags . . . . .	34

# 1

---

## Einleitung

### 1.0.1 Motivation

2020 haben 40% der Weltbevölkerung regelmäßig Spiele gespielt. Dies sind fast 3 Milliarden Menschen. [Iri20] Dabei sind Shooter vor allem Ego-Shooter sehr beliebt. Gerade Moderne Shooter wie Call of Duty, Counterstrike und Battlefield haben eher einen Fokus auf Mehrspieler als auf Einzelspieler. Manchmal wird der Einzelspieler auch komplett ignoriert, obwohl er in vorherigen Teilen eine große Rolle gespielt hat, wie es in Battlefield 2042 der Fall ist. [Dam22] Dennoch haben Spiele wie Doom Eternal eine große Beliebtheit und eine aktive Community. [Mat20] Im Falle von Doom Eternal kämpfen verschiedene Gegentypen gegen den Spieler. Diese Synagieren zwar in der Art und Weise, wie sie gegen den Spieler kämpfen, aber sie interagieren nicht miteinander. Außer das manche gegeneinander Kämpfen.

Nun stellt sich die Frage, wie eine KI designt und aufgebaut ist, die gemeinsam als 4 Personen Gruppe, auch Squad genannt, gegen den Spieler kämpfen.

### 1.0.2 Einführung

#### Shooter

In Einzelspieler Shooter Spielen kämpft der Spieler für gewöhnlich gegen eine oder mehrere vom Computer gesteuerte Gegnern oft auch einfach KI genannt.

Dabei funktioniert das Grunddesign von Shootern primär immer gleich.  
So schreibt Chiossi Clarissa<sup>1</sup>:

”Im Sinne einer Minimaldefinition wäre dann jedes Videospiel (zumindest passagenweise) ein Shooter, in dem Bildschirmobjekte durch Handlungen beeinflusst werden, die vom Spieler als Kombination von Zielen, Schießen und Treffen interpretiert werden.”

---

<sup>1</sup> [Chi13] , S.2

Dabei gibt es verschiedene Kategorien von Shootern. Zum Beispiel :

- Ego Shooter welcher die ich Sicht zeigt.
- Ein Top Down Shooter welcher die Sicht aus der Vogelperspektive sieht.
- Ein Side Scroller Shooter welcher die Sicht von der Seite zeigt.



Abbildung 1.1: Enter the Gungeon [NEL22]

Darüberhinaus wird das Design oft mit anderen Spiele Genres gemischt.

## KI in Shootern

Uwe Schick beschreibt KI wie folgt : "Künstliche Intelligenz ist der Überbegriff für Anwendungen, bei denen Maschinen menschenähnliche Intelligenzleistungen erbringen. Darunter fallen das maschinelle Lernen oder Machine Learning, das Verarbeiten natürlicher Sprache (NLP – Natural Language Processing) und Deep Learning. "[Uwe18]

Dies ist eine sehr allgemeine Definition einer KI, da sich die KI in Videospielen mit anderen Problemen beschäftigen muss. Dabei wird in der Regel nicht auf Machine Learning zurückgegriffen. Eine Ausnahme ist hierbei die von DeepMind entwickelte KI AlphaStar die durch Reinforcement Learning in dem komplexen Strategiespiel "Starcraft 2" gegen Profis gewinnen kann. [Ste19]

Dabei ist das Ziel einer Videospiel KI nicht, die Aufgabe optimal zu lösen, sondern dem Spieler zu unterhalten und auf dessen Handlungen zu reagieren.

Heiko Klinge betont: "Einfach ausgedrückt definieren die Programmierer für alle Einheiten, Gebäude, Nicht-Spieler-Charaktere, Feinde und so weiter, wie sie sich in bestimmten Situationen eines Spiels zu verhalten haben. Je mehr solcher Verhaltensweisen in Verbindung mit bestimmten Situationen definiert werden, desto schlauer wirkt die künstliche Intelligenz eines Computerspiels. "[Hei08]

Betrachtet man die gegnerische KI in Shootern so reagiert diese meist auf den Spieler und dessen unmittelbare Umgebung. Dennoch gibt es Abweichungen im Design der Gegner.

Dabei besprechen Gabriel Rivera, Kenneth Hullett und Jim Whitehead in ihrer Arbeit folgende Designmuster:

- Soldat: Die Hauptfunktion ist es, den Spieler durch das Level zu führen. Er hält eine mittlere Distanz.
- Grenadier: Erhöht die Herausforderung und Spannung durch die Art der Waffe, da diese Flächenschaden verursacht.
- Aggresor: Erhöht die Spannung und den Druck auf den Spieler. Diese Einheit bewegt sich meist schnell und hat eine kurze Angriffsdistanz.
- Berserker: Ähnlich wie der Aggressor nur stärker Dieses Designpattern soll eine hohe Herausforderung bieten.
- Carrier: Dieser spawnt mehrere Gegner, während dieser gegen den Spieler kämpft. Je länger der Carrier lebt, desto höher wird der Druck auf den Spieler.
- Tank: Diese haben meist viel Leben und bewegen sich langsam. Sie sind da um den Fortschritt des Spielers zu behindern.
- Shield: Verhält sich wie der Tank außer, dass dieser einen Schild besitzt, der Schaden des Spielers blockiert. Dieser soll die Herausforderung, das Pacing und den Druck auf den Spieler erhöhen. Zusätzlich soll der Spieler zum Flanken angeregt werden.

(vgl.[RHW12])

Diese werden genutzt, um dem Spieler Abwechslung beim Spielen zu bieten. Durch die verschiedenen Design Muster erhalten die Gegner unterschiedliche Priorisierungen für den Spieler. Dadurch entsteht Dynamik während dem spielen. Ein gutes Beispiel wären Doom oder Enter The Gungeon. Dennoch existiert diese Dynamik in jedem Shooter selbst wenn man dieselbe Gegnerart nutzt. Da die Position dieser Gegner eine große Rolle spielen und damit auch ein Priorisierungsfaktor sein kann. Als Beispiel kann man davon ausgehen das ein Spieler ein Gegner, der keine Deckung hat, gegenüber einem mit Deckung bevorzugt.

## Squad

Ein Squad wird meist in einem militärischen Zusammenhang genannt. Dieser beschreibt eine Gruppe aus meist 6 bis 10 Soldaten oder Polizisten und ist somit die kleinste Militärische Einheit. Dabei existiert meist ein sogenannter Squadleader der die Gruppe anführt. In Videospielen wird die Squadgröße oft auf 4 Personen reduziert wie in dem Spiel Battlefield Bad Company 2. (vgl. [Fab10])

### 1.0.3 Vorhandene Lösungsansätze

In Videospielen werden neben normalen Finite State Machine, Behaviourtrees auch andere Systeme genutzt, damit KI dynamisch auf verschiedene Begebenheiten reagieren können.

In „BROTHERS IN ARMS: ROAD TO HILL 30“ von „Gearbox Software“ ist der Spieler ein Squad Leader und muss durch Befehlen einzelner Squads, die aus individuellen Einheiten bestehen, gegen den Feind kämpfen. Hierbei haben verschiedene Einheiten eigene Aufgaben wie Unterdrückungsfeuer oder die gegnerische Stellung zu flankieren. Gegnerische Einheiten versuchen genauso wie der Spieler, andere Einheiten mit Unterdrückungsfeuer anzugreifen und diese zu flankieren.[Sco06]

In Spielen wie „Halo“ von „Bungie Inc“, „Half-Life“ von „Valve“ und „Dragon Age Inquisition“ von „BioWare“ wird „Utility KI“ verwendet. (vgl. [Tom21]) Utility KI oder zu deutsch Nutzen KI gibt einer Aktion meist ein Wert zwischen 0 und 1, um zu berechnen, wie hoch der Nutzen in der jeweiligen Situation ist. Dies wird genutzt, um dynamisch auf Situationen reagieren zu können. Darauf wird im Kapitel 3 weiter eingegangen.

Ein weiterer Lösungsansatz ist die Nutzung einer sogenannten „Director AI“. Hierbei beschreibt Tommy Thomson [Tom20] in seinem Video „Director AI for Balancing In-Game Experiences — AI 101“ wie diese KI in Spielen wie „Left for Dead“ verwendet wird.

Dabei wird in dem Video die Director KI und dessen Funktionen beschrieben. Dazu gehört das Aufzeichnen von Informationen der Spieler und das Anpassen der Spielwelt anhand dieser Informationen. Sie sorgt zusätzlich dafür, dass die aufgestellten Regeln der Spielwelt eingehalten werden.

Somit ist die KI dafür verantwortlich, die Erfahrung, die ein Spiel verursachen soll, zu bewerkstelligen, indem die Schwierigkeit und das Pacing des Spiels geändert wird.

Im Beispiel von „Left for Dead 2“ wird Musik geändert, Events wie das spawnen von Gegnerwellen ausgelöst, das geben von Pausen nach einem langem Kampf sowie das Bestrafen von Spielern die sich von dem Team entfernen.

Tommy Thomson [Tom16] beschreibt in dem Video „The AI of Alien: Isolation — AI and Games“ wie die Director KI in dem Spiel „Alien Isolation“ funktioniert. Dies ist ein besonders interessantes Beispiel, da die KI des Aliens mit Informationen der Director KI gefüttert wird.

Damit wird erreicht, dass das Alien immer in der Nähe des Spielers ist und Hinweise bekommt, wo sich dieser befindet, ohne diese Informationen komplett freizugeben.



Abbildung 1.2: Alien Isolation

#### 1.0.4 Zielsetzung

Ziel dieser Arbeit ist es, einen Prototypen eines Top Down Shooters mit einer Squad KI in Unity zu entwickeln. Dem Spieler soll es möglich sein, sich im Rahmen des Spiels umzuschauen, zu schießen und zu bewegen. Darüber hinaus kämpft dieser gegen ein Team aus bis zu 4 KI-Einheiten. Das Squad das aus 4 Gegnern besteht, soll hierbei als geschlossene Einheit gegen den Spieler kämpfen. Das Squad und der Spieler sollen sich auf einer Karte mit möglichen Deckungen bewegen. Um diese nutzen zu können, soll es für die Charaktere möglich sein, sich hinzuknien.

Dabei sollen sich die einzelnen computergesteuerten Einheiten des Squads unterstützen und auf Situationen sowohl individuell für sich als auch für die Gruppe reagieren. Dabei sollen Situationen abgedeckt werden wie das nicht mehr Sichten des Spielers, wenn ein anderes Squadmitglied Hilfe benötigt und das gemeinsame Bewegen als Squad Einheit.

Dies wird erreicht durch eine Squad KI die einer Director KI nachempfunden ist. Diese soll auf Situationen des Squads oder der einzelnen Squadmitglieder reagieren und dann den individuellen Squadmitgliedern Anweisungen geben. Die Einheiten sollen ihr individuelles Verhalten durch eine State Machine erhalten, die durch Utility KI unterstützt wird.

## 2

---

# Grundlagen

## 2.0.1 Finite State Machine

Laut Fernando Bevilacqua [Bev13] ist eine Finite State Machine oder zu deutsch ein Endlicher Zustandsautomat ein Automat mit mehreren Zuständen. Dabei hält der Automat immer nur einen Zustand gleichzeitig. Im Kontext der Spieleentwicklung werden den Zuständen des Automaten meist Aktionen zugewiesen wie Angreifen oder Flucht. Dabei werden Übergänge zwischen den Zuständen genutzt um diese zu wechseln. (vgl. Abbildung 2.1) Übergänge werden meist durch einen boolean ausgelöst.

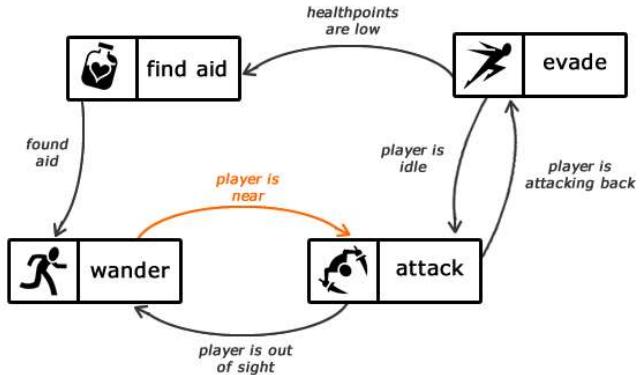


Abbildung 2.1: Finite State Machine [Bev13]

In Videospielen werden Endliche Zustandsautomaten meist genutzt um den Ablauf von künstlicher Intelligenz zu koordinieren. Zum Beispiel um den Ablauf eines Gegners zu koordinieren der gegen einen Spieler vorgeht.

Dabei steht die Komplexität der KI und die Anzahl der States und dessen möglichen Übergängen im direkten Zusammenhang.

## 2.0.2 Utility AI

David Rez Graham [Gra13] beschreibt wie ein System mit Utility AI zu deutsch Nutzen KI funktioniert. Das Grundkonzept von Utility AI ist hierbei, dass jeder möglichen Aktion oder Zustand der KI einen Utilityscore, manchmal auch Gewicht genannt, zuschreibt. Dann wird die Aktion oder der Zustand ausgeführt mit dem höchsten Utilityscore. Dieser Wert kann von mehreren Faktoren abhängen. Betrachtet man einen Gegner aus einem Shooter wären Faktoren beispielsweise Position des Spielers, vorhandene Waffen, vorhandene Deckung usw.

Dabei ist es wichtig einen einheitlichen Maximal und Minimal Utilityscore zu definiert wird. Dabei wird meist ein Normalisierter Wert von 0 bis 1 genommen um die Utility einer Aktion zu bewerten.

### Das Prinzip der maximal erwarteten Utility

Anstatt zu berechnen welche Konsequenz eine Aktion hat wird berechnet wie gut diese mögliche Aktion ist. Da Spiele nicht deterministisch sind, also bei der selben Eingabe verschiedene Ergebnisse entstehen können, wird dies genutzt um möglichst nah an einen genauen Utility Score zu kommen ohne präzise Informationen zu haben.

Dabei wird meist diese Formel<sup>1</sup> genutzt :

$$EU = \sum_{n=1}^N D_i * P_i \quad (2.1)$$

- EU ist der neue Utility Score
- D ist der Wunsch, dass die Aktion ausgeführt wird bzw. der Utilityscore für die Aktion.
- P ist die Wahrscheinlichkeit das die Aktion ausgeführt wird. Dabei ist der Wert normalisiert damit die Summe aller Möglichkeiten 1 ergibt.

Nun wird jede mögliche Aktion berechnet um die Möglichst beste darunter zu wählen.

Beispiel: Eine Aktion hat einen vorherigen Utility Score von 0.6 und hat eine Wahrscheinlichkeit von 0.85 also 85% durchgeführt zu werden. Dann liegt der neu berechnete Utility Score bei 0.51.

Eine weitere Aktion hat einen vorherigen Utility Score von 0.9 aber die Wahrscheinlichkeit diese Aktion auszuführen liegt bei 0.6 also bei 60%. Der neu berechnete Utility Score liegt bei 0.54. Somit wäre die zweite Aktion besser obwohl sie nicht so oft Eintritt.

---

<sup>1</sup> [Gra13], S.115

## Entscheidungsfaktoren

Decisionfaktors oder zu deutsch Entscheidungsfaktoren ist ein normalisierter Wert der abhängig von verschiedenen Faktoren berechnet wird. Diese Faktoren müssen vorher definiert werden.

Entscheidungsfaktoren werden genutzt um den Utilityscore einer Aktion zu berechnen. Betrachtet man einen Gegner in einem Shooter könnte ein Entscheidungsfaktor die Anzahl der übrigen Kugeln in einem Magazin sein. Zum Beispiel hat ein Magazin der Waffe des Gegners 30 Schuss aber es sind nurnoch 10 in diesem vorhanden. Somit wäre der Entscheidungsfaktor für die übrige Munition des Gegners 0.33. Hierbei kann man verschiedenen Entscheidungsfaktoren noch jeweils ein Gewicht hinzufügen falls die Decision für die KI keine große Bedeutung hat.

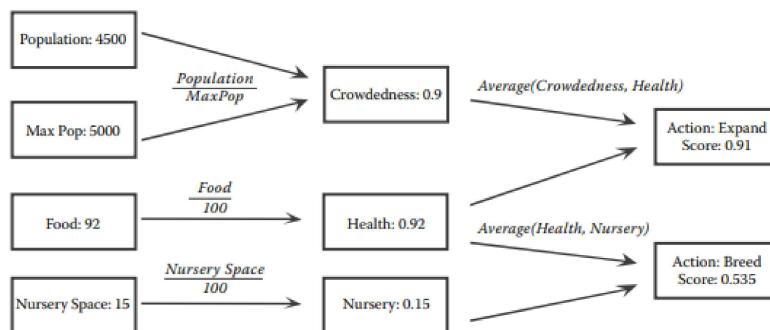


Abbildung 2.2: Entscheidungsfaktor [Gra13] , S.116

## Berechnung von Utility

Bisher wurde besprochen wie der Utilityscore berechnet wird.

David Rez Graham<sup>2</sup> beschreibt wie zwei verschiedene Programmierer verschiedene Funktionen entwickeln können um einen Utilityscore zu berechnen. Er weiß darauf hin ,dass man verstehen muss welchen Einfluss eine Eingabe auf eine Ausgabe haben kann.

Als Beispiel nennt er eine Kolonie die Essen sammelt. Berechnet man den Utilityscore des Bedürfnisses Essen zu sammeln indem man das verfügbare Essen durch das maximal lagerbare Essen teilt ist dies kein guter Weg.

Dadurch erhält man eine Lineare Funktion. Dies macht wenig Sinn da eine Kolonie kein Interesse daran haben sollte Essen zu sammeln wenn das Lager für das Essen fast gefüllt ist.

Um dieses Problem zu umgehen erwähnt er mögliche Funktionen die man nutzen kann und erläutert dessen Eigenschaften.

<sup>2</sup> [Gra13], S.117

Die lineare Funktion<sup>3</sup>:

$$U = \frac{x}{m} \quad (2.2)$$

- U ist der UtilityScore.
- x ist die Eingabe.
- m ist der maximale Wert der Eingabe.

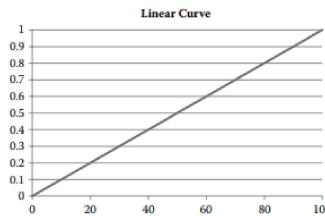


Abbildung 2.3: Lineare Funktion [Gra13], S.117

Wird für eine lineare Ausgabe genutzt.

Die Quadratisch Funktion<sup>4</sup>:

$$U = \left(\frac{x}{m}\right)^k \quad (2.3)$$

- U ist der UtilityScore.
- x ist die Eingabe.
- m ist der maximale Wert.
- k ist ein quadratischer Wert den man Vorgeben kann.

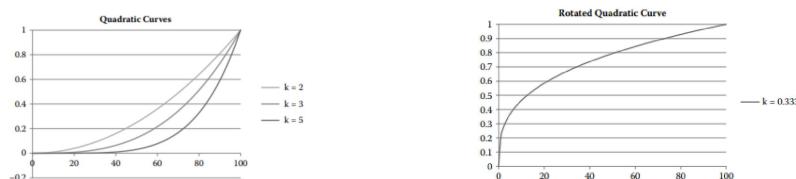


Abbildung 2.4: Quadratisch Funktionen. Hohes k links , geringes k rechts [Gra13], S.118

<sup>3</sup> [Gra13], S.117

<sup>4</sup> [Gra13], S.118

Hierbei hat  $k$  eine starke Auswirkung auf den berechneten Utilityscore. Ein hoher Wert von  $k$  hat einen geringen Einfluss auf die Ausgabe, wenn die Eingabe klein ist. Sie hat aber einen großen Einfluss wenn die Eingabe groß ist. Ein geringer Wert von  $k$  hat einen hohen Einfluss auf die Ausgabe, wenn die Eingabe klein. Sie flacht aber ab sobald die Eingabe größer wird.

Die Logistische Funktion<sup>5</sup> :

$$U = \left( \frac{1}{1 + e^{-x}} \right) \quad (2.4)$$

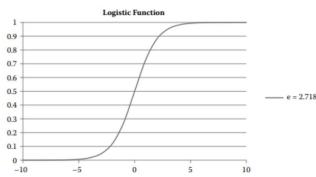


Abbildung 2.5: Logistische Funktion [Gra13], S.119

Bei dieser Funktion wird die natürliche Zahl genutzt um diese Kurve zu erzeugen. Die Eingabegrenzen der Kurve sind -10 und 10. Es wird darauf hingewiesen, dass man diese Funktion nicht nutzen sollte wenn man Eingaben außerhalb der Eingabegrenzen nutzt. Der Grund ist, dass die Ausgabe dann entweder 0 oder 1 wäre. Alternativ kann man einen festen anderen Wert statt der natürlichen Zahl nutzen. Ein hoher alternativer Wert führt dazu, dass die Kurve ausgeprägter und steiler ist und ein niedriger Wert führt dazu, dass die Kurve abflacht.

Die teilweise Lineare Funktion<sup>6</sup> :

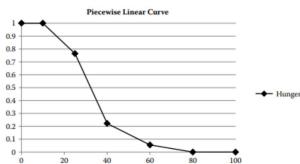


Abbildung 2.6: Teilweise Lineare Funktion [Gra13], S.120

Dies ist eine selbst designte Kurve. Die Idee ist es durch vorgegebene Punkten eine Funktion zu bilden. Der große Vorteil ist das dadurch Designer konkret vorgeben können wie der Utilityscore berechnet wird.

<sup>5</sup> [Gra13], S.119

<sup>6</sup> [Gra13], S.120

### 2.0.3 Pathfinding

Pathfinding oder zu deutsch "Wegfindung" wird genutzt um einen Weg von einem Start zu einem Endpunkt zu finden. Dies wird durch ein Suchalgorithmus realisiert. Dabei sollen Hindernisse vermieden werden.

Der A\* Navigationsalgorithmus ist ein beliebter Suchalgorithmus in 2D Spielen.

Ross Graham , Hugh McCabe und Stephen Sheridan betonen :

"A\* (pronounced a-star) is a directed algorithm, meaning that it does not blindly search for a path (like a rat in a maze) [Matthews02]. Instead it assesses the best direction to explore, sometimes backtracking to try alternatives. This means that A\* will not only find a path between two points (if one exists!) but it will find the shortest path if one exists and do so relatively quickly."<sup>7</sup>

Der A\* Navigationsalgorithmus nutzt ein Gitter aus Navigationspunkten der vorher definiert sein muss. Berechnet man einen Weg so hat man einen Start und einen Endknoten .Dabei hat jeder Knoten auf dem Grid sogenannte G , H und F Kosten. Diese sind vom Start und Endknoten abhängig.

Die jeweiligen Kosten werden wie folgt berechnet <sup>8</sup> :

- G Kosten sind die Kosten vom Startknoten zum betrachteten Knoten .
- H Kosten sind die Kosten von dem betrachteten Knoten bis zum Endknoten. Meist wird die direkte Distanz zwischen den Knoten als Wert betrachtet.
- F ist die Summe aus G und H Kosten.

Der Algorithmus betrachtet zunächst alle Nachbarknoten des Startknotens und sortiert die aus die nicht erreichbar sind. Zum Beispiel Knoten die auf Wänden liegen. Dann wird der Knoten mit den niedrigsten F Kosten gewählt. Dies wird wiederholt bis man den Zielknoten erreicht hat. Die genutzten Knoten repräsentieren den Weg von Start zum Zielknoten.

7	6	5	6	7	8	9	10	11		19	20	21	22
6	5	4	5	6	7	8	9	10		18	19	20	21
5	4	3	4	5	6	7	8	9		17	18	19	20
4	3	2	3	4	5	6	7	8		16	17	18	19
3	2	1	2	3	4	5	6	7		15	16	17	18
2	1	0	1	2	3	4	5	6		14	15	16	17
3	2	1	2	3	4	5	6	7		13	14	15	16
4	3	2	3	4	5	6	7	8		12	13	14	15
5	4	3	4	5	6	7	8	9	10	11	12	13	14
6	5	4	5	6	7	8	9	10	11	12	13	14	15

Abbildung 2.7: A\* Algorithmus berechneter Weg [Nic17]

<sup>7</sup> [GMS03], S.64

<sup>8</sup> [GMS03], S.64

---

**Algorithmus 1:** A\* Navigationsalgorithmus in Pseudocode [GMS03], S.65

---

Let P = starting point.

Assign f, g and h values to P.

Add P to the Open list. At this point, P is the only node on the Open list.

Let B = the best node from the Open list (i.e. the node that has the lowest fvalue).

**if** B is the goal node **then**

    quit – a path has been found

**end if**

**if** The Open list is empty **then**

    quit – a path cannot be found

**end if**

Let C = a valid node connected to B.

a. Assign f, g, and h values to C.

**if** C is on the Open or Closed list **then**

**if** Path is more efficient **then**

        update the path.

**end if**

**end if**

Repeat step 5 for all valid children of B.

Repeat from step 4.

---

## 2.0.4 Unity

Unity ist eine von UnityTechnologie entwickelte Engine zur Spiele Entwicklung für verschiedenen Plattformen.

### Komponenten und Gameobjects

Unity nutzt ein Komponentensystem. Gameobjects in Unity haben zusätzlich zu ihrer Beschreibung eine Transform Komponente oder Rect Transform Komponente für das User Interface und Komponenten die man diesem Gameobject optional hinzufügen kann. Jedes Gameobject enthalten mindestens diese Komponenten Dabei sind die häufigst genutzten Komponenten:

Allgemein:

- Animator : Wird genutzt um Animationen durchzuführen.
- Script : Hiermit kann man mit der Programmiersprache C# eigene Komponenten erstellen. Diese Scripte erben von Monobehaviour.
- AudioListener : Wird genutzt um Sound hörbar zu machen.
- AudioSource : Wird genutzt um Sounds abzuspielen.

Für 3D:

- Mesh Filter : Renderd einen Mesh.
- MeshRenderer : Nimmt die geometry des Meshfilters und Renderd es bei der Position der Transformations Componente.
- Collider (Box Collider , Sphere Collider, Capsule Collider) : Wird genutzt um Collisionen festzustellen.
- Rigidbody : Wird zur physikalischen Symulation genutzt.

Für 2D:

- SpriteRenderer : Rendert einen Sprite.
- Rigidbody 2D : Wird zur Physikalischen Symulation im 2D Raum genutzt
- Collider2D (BoxCollider2D,CircleCollider2D,CapsuleCollider2D) : Wird genutzt um Kollisionen festzustellen im 2D Raum.

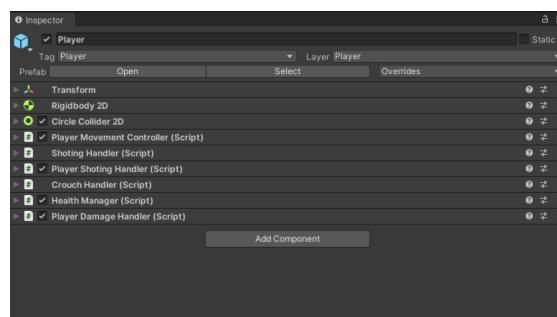


Abbildung 2.8: Komponenten eines Gameobjects in Unity

Man kann Gameobjects zu einem Prefab machen. Prefabs sind hierbei Assets die als Schablone für andere Gameobjects dienen. Dieses kann man dann in der Scene oder während dem Spiel instanziieren.[Uni22b]

## Nutzerinterface

Der Aufbau des Benutzerinterfaces besteht aus Fenstern. Die wichtigsten Fenster sind das Scenenfenster, die Scenenhierarchie, einem Gamefenster, dem Inspector und einem Projektfenster.

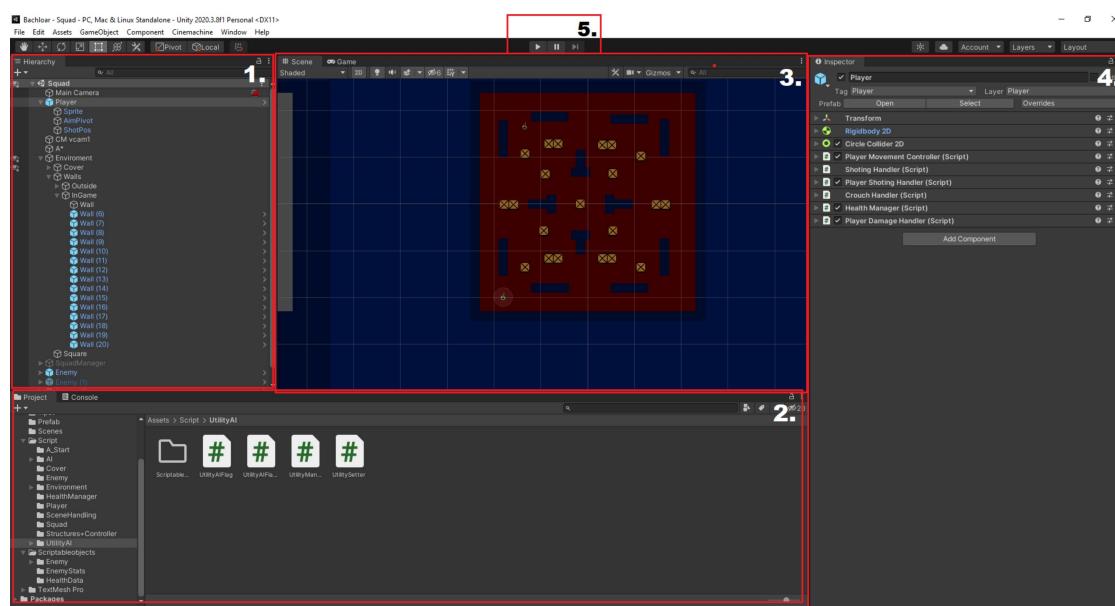


Abbildung 2.9: Standart Nutzerinterface in Unity

- 1 Scenenhierarchie : In diesem Fenster werden Gameobjects der Scene in einer Baumstruktur aufgelistet.
- 2 Projekt Ordner : Hier werden Assets des gesamten Projekts dargestellt.
- 3 Scenen Fenster : Hier werden alle Gameobjects der Scene über eine Scenenkamera gerendert und dargestellt.
- 4 Inspektor : Hier werden Informationen über das mit der Maus ausgewählte Gameobject oder Asset angezeigt.
- 5 Play und Stop Button : Hiermit kann man das Spiel starten und Stoppen. Beim Starten des Spiels wird im Spielfenster das Spiel abgespielt.

## Spiele Assets

Unter Spiele Assets oder einfach Assets genannt versteht man vor allem Graphik für Charaktere, Umgebung, Benutzer Interface sowie Hintergrundmusik, Spezialeffekte und Sound Effekte. [Uni20]

Assets sind :

- C# Scripte
- Modelle
- Materialien
- Texturen
- Sounds
- Sprites
- Shader
- Prefabs
- RenderPipeline Asset

Jede Art von Assets die Außerhalb von Unity erstellt wurden haben einen eigenen Importer.

## Render Pipeline

Seit der Unity Version 2019 gibt es 3 wichtige Renderpipelines die von Unity gestellt werden.[Uni22c]

- Die Build In Render Pipeline: Dies ist die alte Renderpipeline von Unity.
- Die Universal Render Pipeline (URP) : Wird genutzt um das Spiel leicht auf verschiedenen Plattformen darstellen zu können.
- Die High Definition Render Pipeline (HDRP) : Wird genutzt um möglichst realistische Grafik darzustellen.
- Scriptable Render Pipeline (SRP) : Hiermit kann man seine eigene Renderpipeline erstellen. Die URP und die HDRP sind Scriptable Render Pipelines die von Unity erstellt wurden.

Die Renderpipeline ist dafür verantwortlich wie das Spiel gerendert wird. Dies wird in 3 Schritten durchgeführt. [Uni22a]

- Der erste Schritt ist das Culling. Es wird eine Liste erstellt, die nur von Gameobjectsn befüllt wird, die von der Kamera gesehen werden.
- Im zweite Schritt werden die Objekte abhängig der Beleuchtung und anderen Eigenschaften gerendert und das Ergebnis wird in den Pixel Buffer geschrieben.
- Im dritte Schritt wird Post-processing hinzugefügt. Damit wird Bloom und Depth-of-Field auf das Bild angewendet. Das Ergebnis wird zum Bildschirm gesendet und dargestellt.

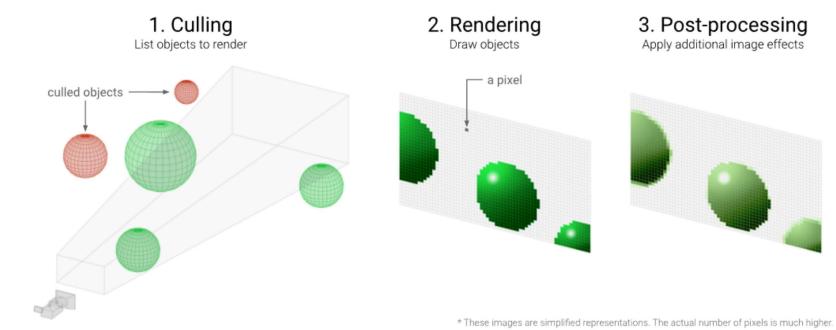


Abbildung 2.10: Renderpipeline Schritte

## Scriptableobjects

Scriptablobjekte sind Daten Container.[Scr22]

- Sie werden genutzt um Daten zu speichern während man sich im Editor befindet.
- Speicher Daten als ein Asset im Projekt um diese während dem Spiel zu nutzen.

Diese werden erstellt durch ein Script das von ScriptableObject erbt. Darin kann man öffentliche Felder deklarieren die im Inspektor angezeigt werden können.

# 3

---

## Umsetzung

### 3.0.1 Vorgehensweise

Die Squad KI wurde innerhalb der Unity Engine umgesetzt. Der erste Schritt war es, einen Spielerkontroller und die dazugehörigen Kameraeigenschaften zu erstellen. Weil es eine Möglichkeit geben muss, mit dem Spiel und somit mit der KI interagieren zu können. Darüber hinaus wurden allgemeine Systeme implementiert wie:

- Ein Schießsystem mit visuellen Feedback für den Spieler und die Squadmitglieder.
- Ein Healthsystem damit der Spieler und die Squadmitglieder Schaden bekommen können.
- Ein simples Deckungssystem sodass sich der Spieler und die Squadmitglieder hinter vorgegebener Deckung hinknien können und somit keinen Schaden bekommen können.

Im nächsten Schritt wurde der A\* Navigationsalgorithmus eingeführt. Danach wurde die State Machine mithilfe des Scriptableobject System umgesetzt. Darüber hinaus wurde dann ein Utilitysystem entwickelt, welches dann in die State Machine eingebettet wurde. Das Utilitysystem ist zusätzlich dafür verantwortlich, in welchen State die KI wechselt.

Nun wurde die einzelne Squadmitglied KI geplant und umgesetzt. Da das Squad aus 4 Mitgliedern besteht, müssen diese auch individuell handeln können. Zusätzlich werden dann über eine separate Squad KI Statewechsel durchgeführt, wenn ein bestimmtes Event ausgeführt wird. Die Squad KI wählt dann anhand der Utility welche Mitglieder den State wechseln.

### 3.0.2 Spieler und Allgemeine Systeme

#### Spieler

Die Eingaben des Spielers werden über das neue Eingabesystem von Unity realisiert. Das Bewegen des Spielercharacters erfolgt physikalisch über dessen Rigidbody abhängig von dem Eingabevektors des Spielers.

Die Kamera wurde über das externe Packet Cinemashine realisiert. Diese Kamera bewegt sich abhängig von dem Richtungsvektors zwischen der Position des Spielercharakter und der Position der Mausposition in der Spielwelt. Zusätzlich rotiert sich der Spielcharakters mit diesem Richtungsvektor.

Das Deckungssystem wird über das Script CrouchHandler realisiert. Das hinknien wird über die Methoden StartCrouching() realisiert und StopCrouching() ist dafür verantwortlich wenn der Charakter wieder aufstehen soll.

Darüber hinaus wird jeweils ein bool gesetzt und ein Event ausgelöst, abhängig davon, ob man sich hinkniet oder ob man aufsteht.

Das Healthsystem wird über das Script Health Manager realisiert. Wenn Schaden zugefügt werden soll, muss die Methode CallculateDamage() ausgeführt werden. Hierbei wird die Menge des Schadens, das TeamFlag und die Transform Komponente der Schadensquelle mitgegeben.

Das TeamFlag ist ein Enum das zwischen Spieler und Gegner unterscheidet. Zusätzlich können andere Scripte an dem OnCallculateDamage Event subscriben um auf den zugefügten Schaden zu reagieren. Das Healthsystem benötigt zusätzlich das ScriptableObject HealthData.

Dies ist ein Datencontainer, welches die Anzahl der Lebenspunkte und das Teamflag des Charakters hält.

Das Schießen wird über das Script Shoting Handler realisiert. Dabei wird in eine gewünschte Richtung ein Raycast durchgeführt mit einer vorgegebenen Länge.

Trifft der Raycast einen Spieler oder einen Gegner, so wird überprüft, ob vor dem Trefferpunkt eine Deckung existiert.

Daraufhin wird überprüft, ob der Spieler oder der Gegner sich hinkniet. Ist dies nicht der Fall, erhält der Charakter Schaden. Das Unterscheiden der Objekte wird über das Tag System in Unity realisiert.

Die Grafiken wurden mit dem Programm Aseprite erstellt.

### 3.0.3 A\* Navigationsalgorithmus

Der A\* Navigationsalgorithmus wurde nach dem Beispiel von Sebastian Lague erstellt. [Seb14]

#### Grid

Das Gitter oder auch Grid wird in der Klasse A\_Star\_Grid dargestellt und die Knotenpunkte oder auch Node in der Klasse A\_Star\_Node. Diese A\*Nodes halten die Daten der Worldposition, die Position im Grid, ob die Node walkable sind, die g und h Kosten der Node, und eine Parent Node. Abhängig von der vorher bestimmten Weltgröße und dem Durchmesser der einzelnen Nodes wird das A\* Star Grid generiert. Hierbei wird über die X Länge und Y Länge der vorher bestimmten Weltgröße iteriert. Hierbei werden für die X und Y Position des Grids eine Node erstellt und über die Kollisionsüberprüfung Physics2D OverlayCircle geprüft, ob die Node auf einem Gameobject liegt, das als Unwalkable oder Cover gelagert wurde. Ist dies der Fall, wird das bool walkable auf false gesetzt. Zusätzlich hat jede Node ein bool nextToUnwalkable. Darüber hinaus wird zusätzlich über alle vorhandenen Nodes iteriert. Ist die Variable walkable auf false gesetzt so wird nextToUnwalkable alle Nachbar Nodes dieser Node auf true gesetzt. Nach dieser Iteration wird nochmals über alle Nodes iteriert und geprüft, ob nextToUnwalkable true ist. Ist dies der Fall, so wird die Variable walkable dieser Nodes auf false gesetzt.

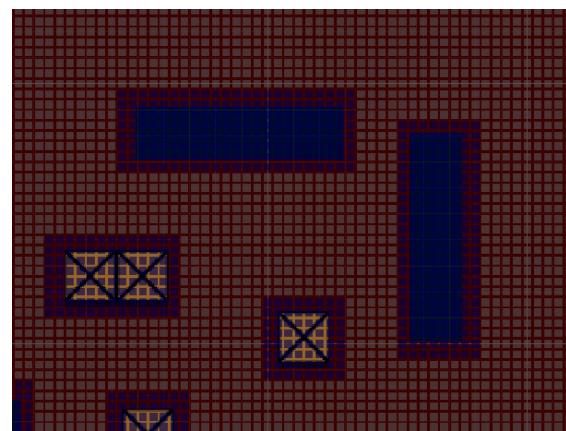


Abbildung 3.1: A\* Navigationsgitter im Spiel

Die weißen Nodes sind walkable true und die blauen Nodes sind walkable false.

## Pfadsuche

Es werden zwei HashSets angelegt. Das openSet hat die Größe des gesamten Grids und closedSet hat die Größe 0. Nun wird eine Liste aller Nachbarn des betrachteten Knotens erstellt.

Der erste betrachtete Knoten ist der Startknoten. Nun wird durch diese Liste iteriert. Ist die Variable walkable des Knoten der Liste false oder der betrachtete Knoten ist gleich des Knotens der Liste so wird diese Iteration übersprungen.

Nun wird der Knoten der Liste die aus Nachbarknoten besteht mit den geringsten Kosten dem HashSet openSet hinzugefügt. Danach wird der betrachteten Knoten der Elternknoten des vorherigen Listenknoten.

Der hinzugefügte Knoten ist nun der zu betrachtende Knoten. Ist der zu betrachtende Knoten der Zielknoten so wurde der Weg gefunden. Im letzten Schritt wird der gefundene Weg umgekehrt indem man ab dem Zielknoten die jeweiligen Elternknoten in ein Array schreibt.

## Bewegung

Die Klasse A\_Star\_Movement ist für die Bewegung anhand des generierten Weges durch die Pfadsuche verantwortlich. Darin wird mit Hilfe des PathRequestManagers ein Weg angefragt. Nachdem der Weg generiert und in einer Vektor 2 Liste festgehalten wurde. Nun wird durch eine Coroutine der Rigidbody des Gameobjectss bewegt.

Dies wird erreicht indem ein Richtungsvektor vom Gameobject zur ersten Position der Liste berechnet wurde und dieser mit der Laufgeschwindigkeit und deltaTime multipliziert. Erreicht das Gameobject diese Position so wird die nächste Position der Liste betrachtet.

Es wird regelmäßig ein neuer Weg angefragt. Damit werden die Positionen der Liste geändert.

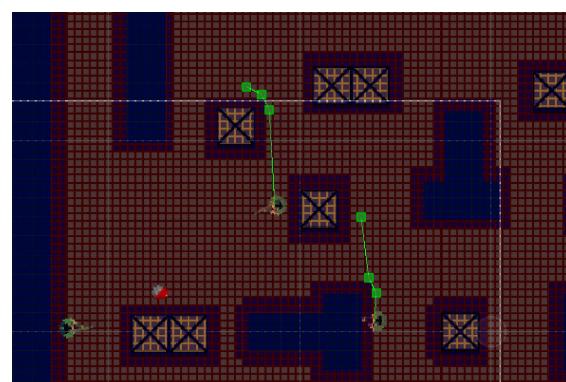


Abbildung 3.2: A\* Bewegung im Spiel

### 3.0.4 StateMachine mit Scriptableobjects

Die Basis für die State Machine ist Matthew Schells vorgestellten Unity Projekt nachempfunden.[Mat17] Diese Basis wurde im Laufe dieser Arbeit erweitert.

Die State Machine wird über das Monobehaviour Script State MachineController gesteuert. Hierbei werden Scriptableobjects der Klasse State dem State Machine-Controller übergeben.

Die Klasse State enthält 2 öffentliche Listen für die ScriptableObjects der Klassen PlugableAction und Decision. PlugableAction und Decision sind abstracte Klassen die von Scriptable Object erben. Jede Methode dieser Abstracten Klassen verlangen als Parameter einen StateMachineController außer die Clone Methoden.

Das Hinzufügen in die Listen der States erfolgt über das Erstellen der Assets der jeweiligen Klassen die von PlugableAction oder Decision erben und das Drag and Dropen in die öffentliche Liste.

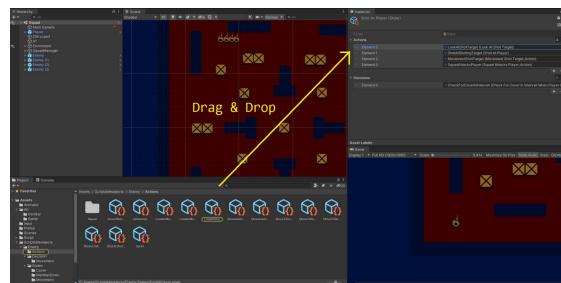


Abbildung 3.3: Drag & Drop einer Action in die Liste des States

### Statewechsel

Der Statewechsel wird über eine Decision ausgelöst. Zunächst wird überprüft, ob eine Instanz des States in der Liste AlreadyUsedStates existiert. Ist dies nicht der Fall, so wird eine Instanz des States und der dazugehörigen PluggableActions und Decisions erstellt und als Klon dieser Liste hinzugefügt und die State Machine wechselt zu dem instanzierten State.

Dies wird über die Klon-Methode der Klassen State, PlugableAction und Decision realisiert. Existiert bereits eine Instanz dieses States so wechselt die Statemachine zu dem Klon des States.

Die Unterscheidung zwischen einem Klon und einem noch nicht genutzten State wird über das Vergleichen der Namen der States realisiert.

Der Grund für diesen Vorgang ist, dass durch das Nutzen von Datencontainern als ausführbaren Code man das Problem hat, dass diese nicht GameObject unabhängig sind. Sodass man beim Ausführen des Codes in einer Klasse, die von `PlugableAction` erbt, darauf achten müsste, keine Variablen zu ändern da sich diese sonst global für alle anderen GameObjects ändern würde, die dieses `ScriptableObject` nutzen würden.

Da es sich bei `ScriptableObject`s um Datencontainer handelt. Zusätzlich benötigt man Referenzen auf Komponenten, die man mithilfe des `ScriptableObject`s manipulieren möchte. Dies erfolgt über sogenannte Referenz Holder Skripte. Diese halten eine Referenz einer Komponente des GameObjects. Zum Beispiel hält der `Animatorholder` eine Referenz der `Animator` Komponente des GameObjects.

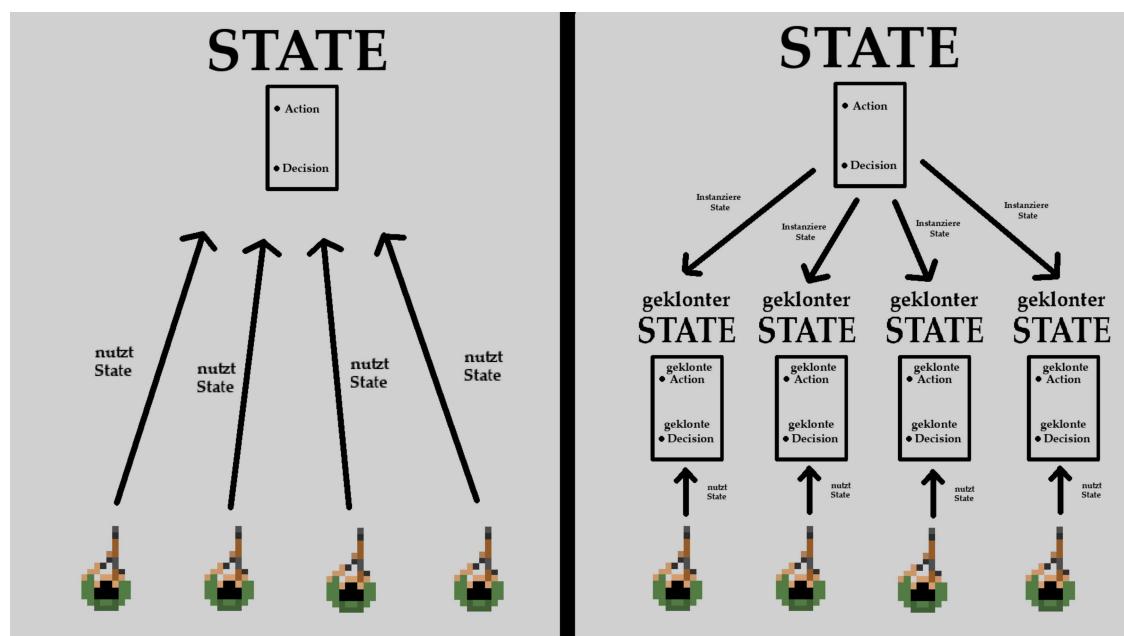


Abbildung 3.4: Vergleich von normale States und geklonnte States

Daraufhin werden die jeweiligen Methoden des alten States und des neu gewechselten States der Scriptableobjects die von Plugableactions und Decions erben ausgeführt.

```

1  public virtual void TransitionToState(State nextState)
2  {
3      if (nextState.stateName != currentState.stateName)
4      {
5          foreach (PluggableAction a in currentState.actions)
6          {
7              a.OnActEnd(this);
8          }
9          foreach (Decision d in currentState.decisions)
10         {
11             d.OnDecideEnd(this);
12         }
13
14         bool stateAllreadyUsed = false;
15         foreach (State s in allreadyUsedStates)
16         {
17             if (nextState.name == s.stateName)
18             {
19                 stateAllreadyUsed = true;
20                 currentState = s;
21                 break;
22             }
23         }
24         if (!stateAllreadyUsed)
25         {
26             currentState = nextState.Clone(this);
27             allreadyUsedStates.Add(currentState);
28         }
29
30         foreach (PluggableAction a in currentState.actions)
31         {
32             a.OnActStart(this);
33         }
34         foreach (Decision d in currentState.decisions)
35         {
36             d.OnDecideStart(this);
37         }
38     }
39 }
```

Listing 3.1: Code zum Statewechsel

## PlugableAction

Klassen die von der Klasse PlugableAction erben können als Scriptableobjects Assets erstellt werden und der PlugableAction Liste beliebig vieler States hinzugefügt werden. Die abstrakten Methoden werden unter folgenden Umständen in der Klasse StateMachineController ausgelöst:

- OnInit : Wird durchgeführt wenn ein Statewechsel durchgeführt und der State noch nicht instanziert wurde in der Methode TransitionToState() in der Klasse StateMachineController.
- OnActStart : Wird beim Wechseln von einem anderen State in diesen durchgeführt in der Methode TransitionToState() in der Klasse StateMachineController.
- OnAct : Diese Methode wird jeden Frame ausgeführt, wenn der State aktiv ist in der Methode UpdateActions() in der Klasse StateMachineController.
- OnActEnd : Wird beim wechseln von dem jetzigen State in einen neuen durchgeführt in der Methode TransitionToState in der Klasse StateMachineController.

Dies gilt für jede Klasse die von PlugableAction erbt und sich in der Liste des jeweiligen States befindet.

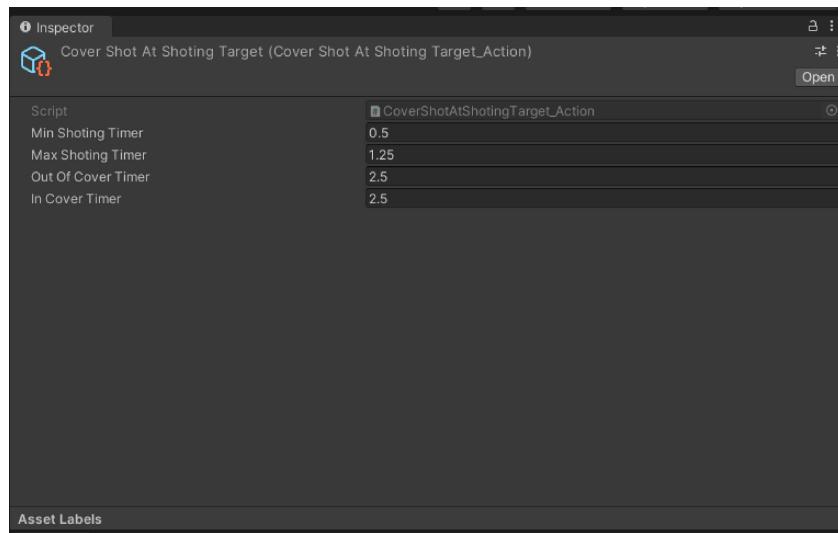


Abbildung 3.5: Inspector eines Scriptableobjects von PlugableAction

## Decision

Klassen die von Decision erben können als Scriptableobjects Assets erstellt werden und der Decision Liste beliebig vieler States hinzugefügt werden. Jede Decision hat öffentliche Felder um ein State oder mehrere mögliche States die wiederum abhängig vom Utility System sind, zum Wechseln hinzuzufügen.

Auf das Utility System wird in den weiteren Schritten eingegangen. Die abstrakten Methoden werden unter folgenden Umständen in der Klasse State MachineController ausgelöst :

- OnDecideInit() : Wird durchgeführt wenn ein Statewechsel durchgeführt und der State noch nicht instanziert wurde.
- OnDecideStart() : Wird beim wechseln von einem alten State in diesen durchgeführt.
- Decide() : Gibt ein bool Wert zurück. In der Methode UpdateDecisions() in der Klasse State MachineController wird jeden Frame überprüft, ob diese Methode true zurückgibt. Ist dies der Fall so wird eine Statewechsel durchgeführt, abhängig davon, ob man das Utility System nutzt oder nicht.
- OnDecideEnd() : Wird beim wechseln von dem jetzigen State in einen neuen durchgeführt.

Dies gilt für jeden Klasse die von Decision erbt und sich in der Liste des jeweiligen States befindet.

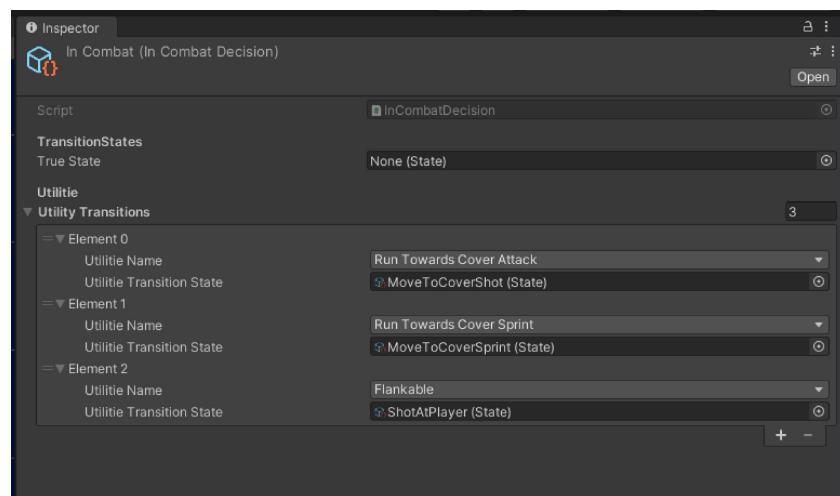


Abbildung 3.6: Inspector eines Scriptableobjects von Decision

### 3.0.5 Squad AI

Jedes Squad Mitglied Gameobject besitzt einen eigenen StateController mit den dazugehörigen ReferenzHoldern zu den jeweiligen States.

Der Squadmanager ist dafür verantwortlich den Wechsel zwischen bestimmten States die nicht von den Squadmitglieder selbst durchgeführt werden können durchzuführen .

Dies erfolgt sowohl durch das Aufrufen bestimmter Events des Squadmanagers als auch durch den Squadmanager selbst. Die Kommunikation zwischen Squadmitglied und Squadmanager wird durch das Script SquadMemberHandler realisiert.

Ich fasse ein Verhalten das durch verschiedene States beschrieben wird als Statepaket zusammen.

Davon ausgenommen ist das Statepaket IsDowned. Dies wird über den individuellen Healthmanager jedes Squadmitglieds ausgelöst.

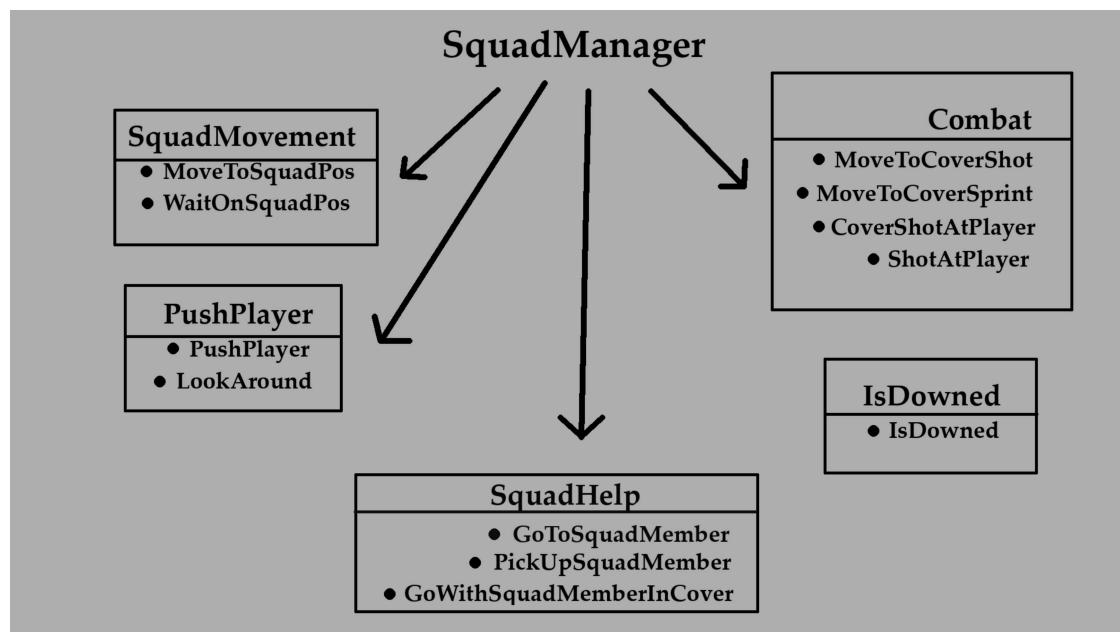


Abbildung 3.7: Darstellung der Statepakete

---

## SquadMovement

In dem Statepaket sind die States : MoveToSquadPos und WaitOnSquadPos verbunden. Das Statepaket ist dafür verantwortlich, alle Squadmitglieder zu vordefinierten Positionen, die vom SquadManager bestimmt werden zu bewegen wenn ein Squadmitglied seine zugewiesene Position erreicht hat, so wartet dieser, bis alle anderen Squadmitglieder ihre zugewiesene Position erreicht haben. Ist dies der Fall, so bestimmt der Squadmanager die nächsten Positionen.

Um die Positionen zu bestimmen, wird der Normalenvektor von einem Elternobjekt, das die Squadpositionen enthält, zum Spieler bestimmt. Im nächsten Schritt wird das Elternobjekt um einen vordefinierten Prozentsatz von 25% in Richtung des Spielers verschoben. Liegt eine Sqaudposition auf einer Position, die ein Squadmitglied nicht erreichen kann, so wird das Elternobjekt bis zu 10-mal zufällig rotiert und nochmals geprüft. Durch die Rotation ändern sich auch die Squadpositionen. Sind die Squadpositionen dennoch nicht gültig, so wird der vordefinierte Prozentsatz um 5% erhöht und das Elternobjekt wird wieder bewegt. Dies wird bis zu 20 mal wiederholt, bis eine gültige Squadposition gefunden wurde. Ist dies dennoch nicht der Fall, so wechseln die Squadmitglieder direkt in das CombatStatepaket.

---

### **Algorithmus 2:** Bestimme Squadposition mit Pseudocode

---

```

for int versuche = 0 ; versuche < 20 ; versuche++ do
    Verringere Abstand zu Spieler um Prozentsatz
    if Squadpositionen nicht gültig then
        for int rotationen = 0 ; rotationen < 10 ; rotationen++ do
            if Squadpositionen sind gültig then
                Bewege Squad und beende die Suche
            end if
            Rotiere Elternobjekt
        end for
    end if
    end for
    if Squadpositionen wurden nach 20 Versuchen nicht gefunden then
        Wechselt jedes Squadmitglied in CombatStatepaket
    end if
```

---



Abbildung 3.8: Darstellung Squadposition Suchalgorithmus

Beschreibung der States :

- MoveToSquadPos : Bewegt das Squadmitglied auf die von dem Squadmanager vorgegebenen Position. Währenddessen schaut dieser auf die zugewiesene Position.
- WaitOnSquadPos : Wartet auf weitere Anweisungen und dreht sich in Richtung des forward Vektors des neu zugewiesenen Punktes.

Der Grund für das Wechseln in das Statepaket ist entweder der Anfang des Spiels, nach dem das Squad im Statepaket PushPlayer war und diesen nicht gefunden hat oder wenn das Squad zu verstreut ist.

Jedes Squadmitglied prüft in der Klasse SquadMemberAreClose den Abstand zu jedem anderen Squadmitglied geringer ist als die Variable distanceToOtherSquadMember. Ist dies bei der Hälfte der verbleibenden Squadmitglieder nicht der Fall, so läuft ein Timer innerhalb des Sqaudmanagers. Läuft dieser ab, so wird in dieses Statepaket gewechselt.

## Combat

Dieses Statepaket enthält die States : MoveToCoverShot , MoveToCoverSprint , ShotAtPlayer und CoverShotAtPlayer. Dieses Squadpacket ist dafür verantwortlich, wie sich das Squadmitglied verhält, wenn sich das Squad plötzlich in einer Kampfsituation befindet.

Dabei suchen die einzelnen Squadmitglieder Deckung in einem gewissen Radius, die am nächsten zum Squadmitglied ist. Wird eine gültige Deckung gefunden, so werden die Deckungspunkte, die einzelne Deckungspositionen repräsentieren, geprüft. Erst wird geprüft, ob die Deckungspunkte schon von anderen Squadmitgliedern besetzt sind und danach, ob sie sich im direkten Schussfeld des Spielers befindet. Dies wird durch einen sehr kurzen Raycast in Richtung des Spielers geprüft. Trifft der Raycast dem Collider des Deckungsobjekts, so ist der Deckungspunkt in Deckung. Wenn eine gültige Deckungsposition gefunden wurde, so bewegt sich das



Abbildung 3.9: Deckungspunkt Überprüfung

Squadmitglied entweder schnell oder langsam in Deckung, bewegt es sich langsam, so schießt es zusätzlich in Richtung des Spielers. Wird der Deckungspunkt erreicht, so schießt das Squadmitglied in zufälligen Abständen auf den Spieler, solang dieser sichtbar ist und duckt sich in Deckung.

Findet das Squadmitglied keine Deckung, so bewegt es sich auf den Spieler schießend zu und führt zufällig einen Seitenschritt aus. Während sich das Squadmitglied auf den Spieler zu bewegt, sucht es in einem vorher definierten Intervall nach einer gültigen Deckung.

---

Beschreibung der States:

- MoveToCoverShot: Bewegt sich auf einen gültigen Deckungspunkt zu und schießt dabei auf den Spieler.
- MoveToCoverSprint: Bewegt sich auf einen gültigen Deckungspunkt schnell zu und schaut dabei auf den Deckungspunkt.
- CoverShotAtPlayer: Während sich das Squadmitglied in Deckung befindet und schießt auf den Spieler wenn möglich und duckt sich hinter der Deckung.
- ShotAtPlayer: Das Sqaudmitglied bewegt sich auf den Spieler zu und schießt auf diesen. In Intervallen führt das Squadmitglied einen Seitenschritt aus. Währenddessen sucht dieser nach einer gültigen Deckung.

Der Grund für das wechseln in das Statepaket ist durch das sichten des Spielers durch ein Squadmitglied oder ein Squadmitglied wird durch den den Spieler angeschossen wird.

## **PushPlayer**

Dieses Statepaket enthält die States : Push Player , Look Around. Wird der Spieler eine gewisse Zeit nicht mehr von jedem Squadmitglied gesehen so läuft ein Timer. Läuft dieser Timer ab, so werden anhand des Utilitywertes der Utility Flankable bis zu zwei Squadmitglieder ausgewählt, die sich auf Positionen, die von dem SquadManager bestimmt wurde, bewegen.

Wenn das Squad nur aus einem Squadmitglied besteht, bewegt sich dieses alleine auf diese Position. Die erste Position wird bestimmt, indem ein zufälliger Einheitsvektor vom Spieler aus mit einem Abstand berechnet wird. Daraufhin wird geprüft, ob ein Squadmitglied diese Position erreichen kann.

Ist dies nicht der Fall, so wird der erste Vorgang wiederholt. Wenn dies auch fehlschlägt, wird die direkte Spielerposition genommen. Wenn die erste Position gültig ist, wird anhand dieser die zweite Position berechnet. Es wird der Richtungsvektor der ersten Position zum Spieler bestimmt.

Der Richtungsvektor wird mit der Position des Spielers addiert. Damit erhält man die gegenüberliegende Position zur ersten Position, die bestimmt wurde.

Zu der neu berechneten Position wird zufällig ein Richtungsvektor addiert und geprüft, ob diese Position eine gültige ist. Ist dies nicht der Fall, so wird in neuer zufälliger Position mit einem neuen Richtungsvektor berechnet.

Wenn dies zu oft fehlschlägt, so wird die Spielerposition als zweite Position genommen.

---

Daraufhin bewegen sich die ausgewählten Squadmitglieder auf die vorgegebenen Positionen. Die anderen Squadmitglieder führen ihre bisherigen States aus.

Wenn die ausgewählten Squadmitglieder die vorgegebenen Positionen erreicht haben, schauen diese sich um und geben dem Squadmanager Bescheid.

Daraufhin wird im SquadManager ein Timer ausgelöst, läuft dieser ab, so wechseln alle Squadmitglieder in das Statepaket Squadmovement.

Beschreibung der States :

- Push Player : Bewegt sich auf eine gültige Position zu und schaut auf diese. Wenn der Spieler gesehen wird dann wechselt jedes Squadmitglied in das Statepaket Combat. Dies wird durch den Squadmanager verursacht.
- Look Around : Wird die Position erreicht so rotiert das Squadmitglied langsam in eine zufällige Richtung bis der Timer des Squadmanagers abgelaufen ist.

Der Grund für das Wechseln in dieses Statepaket ist, dass falls der Spieler nicht mehr von den Squadmitgliedern gesehen wird und dem daraus resultierenden Ablaufen eines Timers im Squadmanager.

## Downed

Dies ist ein Statepaket das aus dem State IsDowned besteht. Hat ein Squadmitglied unter 20% Leben so ist es niedergeschlagen. Dabei wechselt die State Machine durch den Healthmanager und dessen Events in den State IsDowned. Daraufhin lößt das Squadmitglied das Pickup Event des SquadManagers aus.

Fällt das Leben des Squadmitglieds auf 0 so wird das Squadmitglied aus der Liste von Squadmitgliedern des Squadmanagers entfernt. Daraufhin wird ein Dummy Gameobject instanziert das ein liegendes Squadmitglied darstellen soll. Im letzten Schritt wird das Squadmitglied Gameobject gelöscht.

---

## SquadHelp

Dieses Statepaket besteht aus den States GoToSquadMember, PickUpSquadMember , GoWithSquadMember. Nachdem ein Squadmitglied das PickUpEvent des Squadmanagers ausgelöst hat, entscheidet der Squadmanager anhand des Utility Werts der Utility CanGiveMedicalAssistance welches Squadmitglied ausgenommen des zu helfend Squatmitglieds diesem helfen soll. Das Squadmitglied mit dem höchsten Utility Wert wird gewählt.

Dieses bewegt sich auf das niedergeschlagene Squadmitglied zu und schießt dabei auf den Spieler, wenn möglich. Ist das niedergeschlagene Squadmitglied erreicht, so entscheidet das helfende Squadmitglied anhand der Utilitywerte der Utilitys CanPickUpImmediatly und NeedToTakeCover.

Hat CanPickUpImmediatly der höher Wert so wird direkt angefangen das niedergeschlagene Squadmitglied aufzuheben.

Hat jedoch NeedToTaleCover einen höheren Utilitywert zieht das helfende Squadmitglied das niedergeschlagenes Sqaudmitglied in die nächstmögliche Deckung. Wenn keine Deckung gefunden wurde so hebt das helfende Squadmitglied das Niedergeschlagene direkt auf.

Nachdem das niedergeschlagene Squadmitglied aufgehoben wurde, erhält es die Hälfte seines maximalen Lebens wieder. Sowohl das niedergeschlagene als auch das helfende Squadmitglied wechseln in das Statepaket Combat. Beschreibung der States :

- GoToSquadMember : Das helfende Squadmitglied bewegt sich zum niedergeschlagenen Squadmitglied.
- PickUpSquadMember: Das helfende Squadmitglied schaut auf das niedergeschlagene Squadmitglied. Nachdem der Aufhebtimer abgelaufen ist wird dies dem Squadmanager mitgeteilt. Dieser hebt das niedergeschlagenen Squadmitglied wieder auf und setzt das Leben auf die Hälfte des ursprünglichen Lebens des Squadmitglieds.
- GoWithSquadMember: Das helfende Squadmitglied sucht nach einer Deckung und zieht das niedergeschlagene Squadmitglied mit. Daraufhin wechselt das helfende Squadmitglied in den PickUpSquadMember State.

### 3.0.6 Utility System

Das Utility System besteht aus den Scripten Utility Setter und Utility Manager. Das Utility System wird von der State Machine genutzt um in States die Utilitys nutzen zu entscheiden in welchen State gewechselt wird und vom Squad Manager um zu bestimmen welches Squad Mitglied bestimmte Aktionen ausführt.

#### UtilityAIFlags

Die UtilityAIFlag Klasse besteht aus einem Enum das den UtilityName angibt, dem UtilityScore des Flags und die bool Variable isLocked um zu verhindern das die UtilityScore geändert wird. Zusätzlich enthält die Klasse Methoden um den UtilityScore zu manipulieren. Dabei kann der Utilityvalue nicht unter 0 oder über 1 verändert werden.

Die verschiedenen UtilityAIFlags

- Aggressiv
- Flankable
- RunTowardsCoverAttack
- RunTowardsCoverSprint
- NeedFireAssistance
- CanGiveFireAssistance
- CanGiveMedicalAssistance
- CanPickUpImmediately
- NeedToTakeCover

#### UtilityManager

Der UtilityManager ist ein Monobehaviour Script und wird als Komponente dem Gameobject hinzugefügt. Dabei hat der UtilityManager eine öffentliche Liste mit allen möglichen UtilityAIFlags.

Darüberhinaus bestimmt der Manager durch das Übergeben einer Liste von UtilityAIFlags welches dieser Flags den höchsten Utilityscore hat. Dies wird vor allem beim Wechsel der State Machine und von Squadmanager zum bestimmen für geeignete Squadmitglieder.

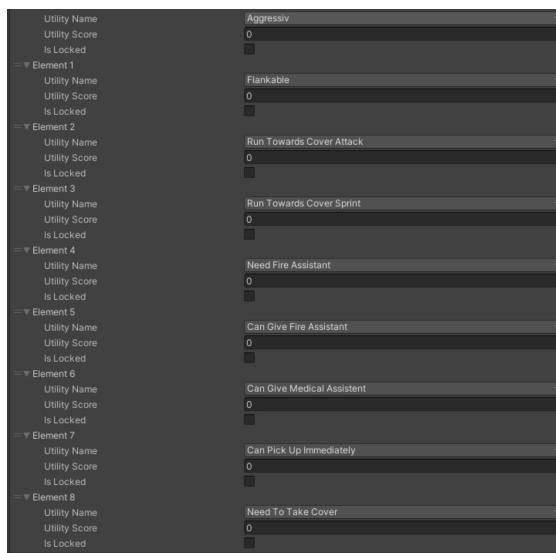


Abbildung 3.10: öffentliche Utilityliste des UtilityManagers in der UnityEngine

```

1  public UtilityAIFlag GetFlagWithMostUtilities(
2      List<UtilityAIFlagName.UilityName> compareFlagNames)
3  {
4      /**
5       * Vergleicht die angegebene Liste compare Flags
6       * mit den utilityAIFlags und gibt die passenden
7       * utilityAIFlags in Form einer Liste zurueck.
8       */
9      List<UtilityAIFlag> choosenUtilityFlags
10     = GetComparedFlags(compareFlagNames);
11     UtilityAIFlag bestFlag = choosenUtilityFlags[0];
12     float bestValue = choosenUtilityFlags[0].utilityScore;
13     if(choosenUtilityFlags.Count > 1)
14     {
15         for(int i = 1; i< choosenUtilityFlags.Count; i++)
16         {
17             if (choosenUtilityFlags[i].utilityScore > bestValue)
18             {
19                 bestValue = choosenUtilityFlags[i].utilityScore;
20                 bestFlag = choosenUtilityFlags[i];
21             }
22         }
23     }
24     return bestFlag;
25 }
```

Listing 3.2: Code zum bestimmen des besten UtilityAIFlags

## Berechnung des UtilityScore

Im UtilityManager wird zusätzlich das Event utilityEvent deklariert. Im Utility-Setter abonnieren Methoden auf dieses Event sodass wenn dieses ausgelöst wird die jeweiligen UtilityScore gesetzt werden.

Innerhalb der Methoden wird der UtilityScore der jeweiligen UtilityFlags individuell berechnet. Um mehr Freiheit bei der Berechnung des Utilitiescores zu haben. Wurde die Animationskurve von Unity genutzt. Damit ist es einem möglich eine Funktion zu generieren die man personalisieren kann.

Der vorherige berechnete Utilityscore ist hierbei der X Wert. Im weiteren Verweise ich darauf mit  $U_x$ . Der Y Wert wird als tatsächlicher Utilityscore in das jeweilige UtilityAIFlag geschrieben.

Aggressiv :

Das UtilityAI Flag ist dafür verantwortlich wie oft ein Squadmitglied schießt und wie selten es sich hinter eine Deckung duckt. Zusätzlich wird bei Kontakt mit dem Spieler entschieden ob das Squadmitglied in den State ShotAtPlayer wechselt. Damit Bewegt sich das Squadmitglied aggressiv auf den Spieler zu.

$$U_x = \left(1 - \frac{cPH}{mPH}\right) \quad (3.1)$$

- $U_x$  = berechneter UtilityScore x Wert.
- cPH = Leben des Spielers zu dem Zeitpunkt
- mPH = Maximales Leben des Spielers

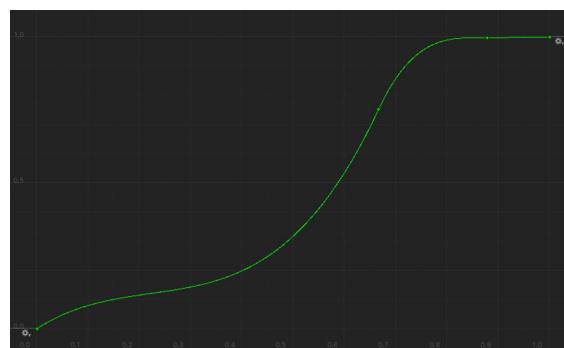


Abbildung 3.11: Utilityscore Funktion : Aggressiv

Durch die Funktion werden die einzelnen Squadmitglieder viel aggressiver wenn der Spieler unter 80% Leben fällt.

Flankable :

Dieses UtilityAIFlag wird genutzt um zu entscheiden welche der Squadmitglieder den Spieler flankieren wenn dieser nicht mehr vom Squad zu sehen ist.

Der Utilityscore wird berechnet indem der Abstand zum Spieler untersucht wird. Dafür sind die Variablen maxFlankDistanceToPlayer und minFlankDistanceToPlayer. Ist das Squadmitglied weiter als die maxFlankDistanceToPlayer entfernt dann wird der UtilityScore auf 1 gesetzt. Ist er weniger als minFlankDistanceToPlayer so ist der UtilityScore 0. Für die Distanz dazwischen wird folgende Formel zur Berechnung des Utilityscors genutzt.



Abbildung 3.12: Utilityscore Funktion : Flankable

$$Ux = -1 * \left( \frac{maxFlankDistanceToPlayer - d}{maxFlankDistanceToPlayer - minFlankDistanceToPlayer} - 1 \right) \quad (3.2)$$

- $Ux$  = berechneter UtilityScore X Wert.
- $d$  = Distanz zum Spieler
- $maxFlankDistanceToPlayer$  = Maximale Distanz zum Spieler
- $minFlankDistanceToPlayer$  = Minimale Distanz zum Spieler



Abbildung 3.13: Utilityscore Funktion : Flankable

---

RunTowardsCoverAttack und RunTowardsCoverSprint :

Wenn das Squad den Spieler bemerkt und in Deckung möchte wird anhand RunTowardsCoverAttack und RunTowardsCoverSprint entschieden auf welche Art das jeweilige Squadmitglied sich in Deckung bewegt. Abhängig davon welcher den höheren Utilityscore hat wird durch RunTowardsCoverSprint in MoveToCoverSprint und durch RunTowardsCoverAttack in MoveToCoverShot gewechselt. Da diese Utilities direkt verglichen werden, werden diese auch ähnlich berechnet. Der Abstand zur nächsten Deckung, sowie ob der Spieler sich selbst in Deckung befindet und der Abstand zum Spieler haben Einfluss auf den Berechneten Utilityscore. Dabei haben diese Einflüsse verschiedene Gewichtungen.

Um den ersten Entscheidungsfaktor  $U_a$  zu berechnen werden die Variablen maxCoverDistance und minCoverDistance genutzt. Ist der Abstand zur nächsten Deckung kleiner als minCoverDistance so ist der Entscheidungsfaktor 1. Ist der Abstand größer als maxCoverDistance so ist der Entscheidungsfaktor von 0. Für alle Werte zwischen den beiden Variablen wird folgende Formel genutzt.

$$U_a = \left( \frac{\maxCoverDistance - d}{\maxCoverDistance - \minCoverDistance} \right) \quad (3.3)$$

- $U_a$  = Entscheidungsfaktor a.
- $\maxCoverDistance$  = Maximaler Abstand die eine Deckung entfernt sein darf.
- $\minCoverDistance$  = Minimaler Abstand zur Deckung.

Der zweite Entscheidungsfaktor  $U_b$  wird abhängig davon ob der Spieler hinter einer Deckung ist oder nicht entweder zum Entscheidungsfaktor von RunTowardsCoverSprint oder RunTowardsCoverAttack hinzugezählt.

Der dritte Entscheidungsfaktor  $U_c$  wird abhängig davon wie weit der Spieler vom Squadmitglied entfernt ist entweder zu RunTowardsCoverAttack oder RunTowardsCoverSprint.

Dabei hat jeder Entscheidungsfaktor eine eigene Gewichtung. Der erste Entscheidungsfaktor hat eine Gewichtung von 50% und der zweite und dritte jeweils von 25%. Daraus ergibt sich diese Formel.

$$U_x = U_a * 0.5 + U_b * 0.25 + U_c * 0.25 \quad (3.4)$$

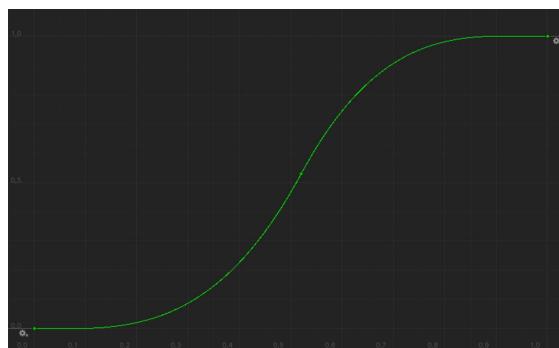


Abbildung 3.14: Utilityscore Funktion : RunTowardsCoverSprint

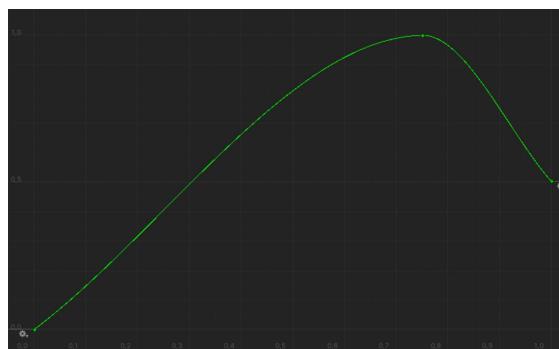


Abbildung 3.15: Utilityscore Funktion : RunTowardsCoverAttack

Die RunTowardCoverAttack Funktion hat ihren Höchspunkt bei Ux gleich 0.75. Der Grund dafür ist das falls Ub und Uc als Entscheidungsfaktor jeweils auf RunTowardsCoverSprint und RunTowardCoverAttack aufgeteilt werden, dann soll RunTowardCoverAttack bevorzugt werden.

NeedFireAssistant :

Dieses UtilityAiFlag sorgt dafür das falls der Utilityscore 1 ist das squadMemberFireAssistantEvent des jeweiligen Squadmitglieds auslöst. Das UtilityAiFlag wird durch das anschauen des Spielers nach ablaufen eines Timerintervals immer wieder um 0.05 erhöht. Wird das Squadmitglied getroffen so erhöht sich der UtilityScore um 0.1. Diese UtilityAiFlag hat keine Funktion da sie nicht genutzt wird um einen State zu wechseln. Sie wird nur genutzt um das jeweilige Event auszulösen.

CanGiveFireAssistance :

Wenn das squadMemberFireAssistantEvent des jeweiligen Squadmitglieds ausgelöst wird so sucht der Squadmanager ein anderes Squadmitglied anhand des Utilityscors von CanGiveFireAssistance. Das UtilityAIFlag Agressiv des Squadmitglied mit dem höchsten Utilitiescore wird auf 1 gesetzt. Der berechnete Utilitiescore  $U_x$  besteht aus 3 Entscheidungsfaktoren  $U_a$ ,  $U_b$  und  $U_c$  mit eigenen Gewichtungen.  $U_a$  wird ist abhängig vom Abstand zum Squadmitglied das um Unterstützung bittet. Dies beschreiben die Variablen minDistanceToSquadMember und maxDistanceToSquadmember. Hierbei wird das selbe vorgehen genutzt wie beim Suchen von nahen Deckungen. Es wird auch die Formel 3.3 genutzt.  $U_b$  ist davon abhängig ob der Spieler von dem Squadmitglied angreifbar ist. Nur wenn dies der Fall ist so wird  $U_b$  und somit dessen Gewicht mit  $U_a$  addiert.

$U_c$  ist abhängig vom Leben des Squadmitglieds dabei wird das übrige Leben durch das maximale Leben des Squadmitglieds geteilt. Das Ergebnis wird mit dem Gewicht multipliziert.

$$U_c = \frac{cPH}{mPH} \quad (3.5)$$

- $U_c$  = Dritter Entscheidungsfaktor ohne Gewicht.
- $cPH$  = Leben des Spielers zu dem Zeitpunkt
- $mPH$  = Maximales Leben des Spielers zu dem Zeitpunkt

$$U_x = U_a * 0.25 + U_b * 0.5 + 0.25 * U_c \quad (3.6)$$

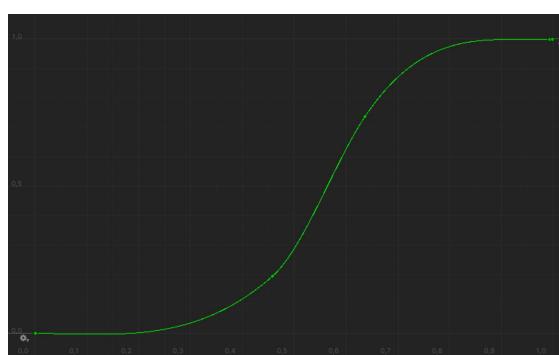


Abbildung 3.16: Utilityscore Funktion : CanGiveFireAssistance

Die Funktion ist so geformt, dass sie den Entscheidungsfaktor  $U_b$  bevorzugt. Da dieser nötig ist um einen hohen Utilitiescore  $U_y$  zu erhalten.

### CanGivePickUpAssistance :

Schießt der Spieler auf ein Squadmitglied und führt damit dazu, dass es unter 20% Leben hat, wechselt es in den State IsDowned. Wenn das Squadmitglied in dieses State wechselt lößt es ein das squadMemberPickUpEvent aus. Damit sucht der Squadmanager abhängig des UtilityAIFlags CanGivePickUpAssistance das beste Sqaudmitglied um dem Niedergeschlagenen zu helfen.

Der Utilityscore ist abhängig von 2 Entscheidungsfaktoren  $U_a$  und  $U_b$  die das gleiche Gewicht von 0.5 haben. Dabei ist  $U_a$  abhängig von dem Abstand vom niedergeschossenen Squadmitglied. Dabei wird das selbe vorgehen genutzt wie bei  $U_a$  der UtilityAIFlags von RunTowardsCoverAttack und RunTowardsCoverSprint.

$U_b$  wird berechnet wir  $U_c$  des UtilityAiFlags CanGiveFireAssistance und ist somit abhängig vom übrigen Leben des Squadmitglieds.

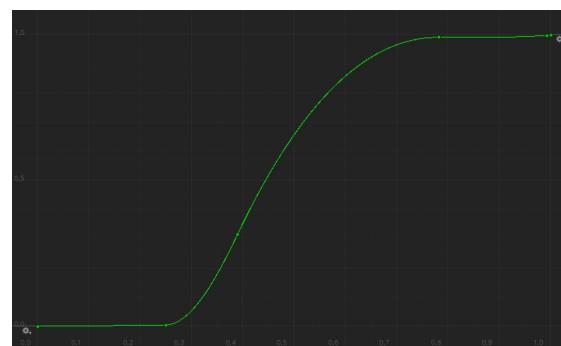


Abbildung 3.17: Utilityscore Funktion : CanGivePickUpAssistance

Die Funktion dient als Grenze für den Unityscore  $U_x$ . Ist dieser kleiner als 0.25 so ist der eigentliche Utilityscore  $U_y$  0. Der Squadmanager ignoriert Squadmitglieder mit einem Utilityscore  $U_y$  von 0.

### CanPickUpImmediatly und NeedToTakeCover :

Erreicht das Squadmitglied ein niedergeschossenes Squadmitglied so wird entschieden ob das niedergeschlagene Mitglied erst in Deckung gezogen werden muss oder nicht. Dies wird entschieden indem geprüft wird ob sich das niedergeschossene Squadmitglied in Deckung befindet oder nicht. Ist das niedergeschossene Squadmitglied in Deckung so ist der Utilityscore von CanPickUpImmediatly 1 und NeedToTakeCover 0. Ist es nicht in Deckung so ist der Utilityscore von CanPickUpImmediatly 0 und NeedToTakeCover 1. Da es nur 2 mögliche Werte gibt wird keine Funktion genutzt.

# 4

---

## Zusammenfassung und Ausblick

### 4.1 Zusammenfassung

In dieser Arbeit wurde mit Hilfe von Unity eine Prototyp eines Top Down Shooters umgesetzt dessen Gegner als Squad gemeinsam gegen den Spieler kämpfen. Die einzelnen Squadmitglieder sowie der Spieler können aufeinander schießen und sich hinter Deckungen ducken. Darüberhinaus wurde ein Utilitysystem implementiert. Dieses bestimmt den Utilityscore einzelner Utilities. Diese werden verwendet um bei einem Statewechsel in den State zu wechseln mit dem höchsten Utilityscore.

Die Squad KI wird durch den Squadmanager realisiert. Dieser koordiniert die einzelnen Squadmitglieder und reagiert auf Ereignisse wie das Niederschießen eines Squadmitgliedes oder das Verschwinden des Spielers aus der Sicht des Squads. Der Squadmanager entscheidet anhand der Utilitiescores der Squadmitglieder welches ein Statewechsel durchführt und somit eine Aktion durchführt.

Darüberhinaus wurde die State Machine mit Hilfe von Scriptableobjects umgesetzt. Ein State sowie die Aktionen als auch die Entscheidungen zum Statewechsel bestehen aus einem ScriptableObject. Dadurch lassen sich neue States sowie dessen Statewechsel leicht zusammensetzen ohne neuen Code schreiben zu müssen.

Eine große Herausforderung während der Arbeit war das Integrieren des Utilitiesystems sowie die Einstellung der jeweiligen Gewichtungen sowie das Anpassen der Funktionen.

## 4.2 Weiterentwicklung

Eine Weiterentwicklungs möglichkeit wäre die Implementierung verschiedener Gegnertypen, wie es im Kapitel Einführung beschrieben wurde, um das Spielerlebnis für den Spieler abwechslungsreich zu gestalten. Zusätzlich könnten diese vom Squadmanager unterschieden werden und je nach Event andere Aufgaben zu erhalten. Interessant wäre zusätzlich die Betrachtung der Entscheidungsfaktoren dieser Gegnertypen, um diese auf Situationen anders reagieren lassen zu können.

Ein weiterer Punkt wäre die Erweiterungen der Entscheidungsfaktoren und das Ändern der Funktionen der jeweiligen Utilitys, um diese optimal Nutzen zu können.

Interessant wäre auch, dem Spieler und den Squadmitglieder verschiedene Waffen zu geben. Je nach Waffe könnten diese anders reagieren. Die Entscheidungsfaktoren der Utilitys der einzelnen Squadmitglieder könnten davon abhängig sein, welche Waffe sie nutzen.

---

## Literaturverzeichnis

- Bev13. BEVILACQUA, FERNANDO: *Finite-state machines: theory and implementation.* <http://gamedevelopment.tutsplus.com/tutorials/finite-statemachines-theory-and-implementation--gamedev-11867>, 2013. letzter Aufruf 11.02.2022.
- Chi13. CHIOSSI, CLARISSA, 2013.
- Dam22. DAMIAN MEIER: *Battlefield 2042: Singleplayer und Kampagnen - alle Infos.* [https://praxistipps.chip.de/battlefield-2042-singleplayer-und-kampagnen-alle-infos\\_140960](https://praxistipps.chip.de/battlefield-2042-singleplayer-und-kampagnen-alle-infos_140960), 2022. letzter Aufruf 11.02.2022.
- Fab10. FABIAN SIEGISMUND: *Battlefield: Bad Company 2 im Test - Multiplayer-Kracher von DICE.* [https://www.gamestar.de/artikel/battlefield-bad-company-2-im-test-multiplayer-kracher-von-dice\\_2313005.html](https://www.gamestar.de/artikel/battlefield-bad-company-2-im-test-multiplayer-kracher-von-dice_2313005.html), 2010. letzter Aufruf 11.02.2022.
- GMS03. GRAHAM, ROSS, HUGH McCABE und STEPHEN SHERIDAN: *Pathfinding in computer games.* ITB Journal, 8:57–81, 2003.
- Gra13. GRAHAM, DAVID REZ: *Game AI pro: collected wisdom of game AI professionals, S.113 - 126 ,Section 2: Architecture, Kapitel 9 : An Introduction to Utility Theory.* CRC Press, 2013.
- Hei08. HEIKO KLINGE: *Warum Künstliche Intelligenz (KI) in Spielen stagniert.* [https://www.tecchannel.de/a/warum-kuenstliche-intelligenz-ki-in-spielen-stagniert\\_1744817](https://www.tecchannel.de/a/warum-kuenstliche-intelligenz-ki-in-spielen-stagniert_1744817), 2008. letzter Aufruf 12.02.2022.
- Iri20. IRINA MORITZ: *Bericht zeigt, wie viele Menschen weltweit Games zocken. Spoiler: Es sind viele.* <https://mein-mmo.de/bericht-zeigt-wie-viele-menschen-weltweit-games-zocken-spoiler-es-sind-viele>, 2020. letzter Aufruf 11.02.2022.
- Mat17. MATTHEW SCHELL: *Pluggable AI With Scriptable Objects Episode 1-10.* [https://www.youtube.com/watch?v=cHUXh5biQMg&list=PLX2vGYjWbI0ROSj\\_B0\\_eir\\_VkHrEkd4pi](https://www.youtube.com/watch?v=cHUXh5biQMg&list=PLX2vGYjWbI0ROSj_B0_eir_VkHrEkd4pi), 2017. letzter Aufruf 12.02.2022.
- Mat20. MATHIAS DIETRICH: *Doom Eternal begeistert mehr als doppelt so viele Spieler als beim Vorgänger.* <https://praxistipps.chip.de/>

- battlefield-2042-singleplayer-und-kampagnen-alle-infos\_140960, 2020. letzter Aufruf 11.02.2022.
- NEL22. NELSON CHITTY: *Top 25 Best Weapons & Guns in Enter the Gungeon.* <https://www.fandomspot.com/enter-the-gungeon-best-weapons/>, 2022. letzter Aufruf 13.02.2022.
- Nic17. NICHOLAS SWIFT: *Director AI for Balancing In-Game Experiences — AI 101.* <https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2>, 2017. letzter Aufruf 12.02.2022.
- RHW12. RIVERA, GABRIEL, KENNETH HULLETT und JIM WHITEHEAD: *Enemy NPC design patterns in shooter games.* In: *Proceedings of the First Workshop on Design Patterns in Games*, Seiten 1–8, 2012.
- Sco06. SCOTTIE THENERD: *Top 10 Lists: The Top 10 Squad-based A.I. In FPS Games.* 2006. letzter Aufruf 11.02.2022.
- Scr22. SCRIPTABLEOBJECT: *Render pipelines.* <https://docs.unity3d.com/Manual/class-ScriptableObject.html>, 2022. letzter Aufruf 11.02.2022.
- Seb14. SEBASTION LAGUE: *A\* Pathfinding E01-E10.* [https://www.youtube.com/watch?v=-L-WgKMFuhE&list=PLft\\_AvWsXl0cq5Umv3pMC9SPnKjfp9eGW](https://www.youtube.com/watch?v=-L-WgKMFuhE&list=PLft_AvWsXl0cq5Umv3pMC9SPnKjfp9eGW), 2014. letzter Aufruf 11.02.2022.
- Ste19. STEFAN KREML: *Starcraft 2: Verbesserte DeepMind-KI schlägt 99,8 % der menschlichen Spieler.* <https://www.heise.de/newsticker/meldung/Starcraft-2-Verbesserte-DeepMind-KI-schlaegt-99-8-der-menschlichen-Spieler.html>, 2019. letzter Aufruf 12.02.2022.
- Tom16. TOMMY THOMSON: *The AI of Alien: Isolation — AI and Games.* <https://youtu.be/Nt1XmiDwxhY?t=303>, 2016. letzter Aufruf 11.02.2022.
- Tom20. TOMMY THOMSON: *Director AI for Balancing In-Game Experiences — AI 101.* <https://www.youtube.com/watch?v=Mnt5zxb8W0Y>, 2020. letzter Aufruf 11.02.2022.
- Tom21. TOMMY THOMSON: *How Utility AI Helps NPCs Decide What To Do Next — AI 101.* <https://www.youtube.com/watch?v=p3Jbp2cZg3Q>, 2021. letzter Aufruf 11.02.2022.
- Uni20. UNITY TECHNOLOGIES: *General game development terms.* <https://unity.com/how-to/beginner/game-development-terms>, 2020. letzter Aufruf 11.02.2022.
- Uni22a. UNITY TECHNOLOGIES: *Choosing and configuring a render pipeline and lighting solution.* <https://docs.unity3d.com/Manual/BestPracticeLightingPipelines.html>, 2022. letzter Aufruf 11.02.2022.
- Uni22b. UNITY TECHNOLOGIES: *Prefabs.* <https://docs.unity3d.com/Manual/Prefabs.html>, 2022. letzter Aufruf 11.02.2022.
- Uni22c. UNITY TECHNOLOGIES: *Render pipelines.* <https://docs.unity3d.com/Manual/render-pipelines-overview.html>, 2022. letzter Aufruf 11.02.2022.

- Uwe18. UWE SCHICK: *Was ist künstliche Intelligenz?* <https://news.sap.com/germany/2018/03/was-ist-kuenstliche-intelligenz/>, 2018. letzter Aufruf 12.02.2022.

# A

---

## Selbstständigkeitserklärung



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

- Diese Arbeit wurde als Gruppenarbeit angefertigt. Meinen Anteil habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Namen der Mitverfasser:

Meine eigene Leistung ist:

---

15.02.2022

Datum

---

Unterschrift der Kandidatin/des Kandidaten