Hochschule Kempten
University of Applied Sciences

Bachelorarbeit im Studiengang Informatik – Game Engineering

Implementierung des Tagesablaufs eines Kämpfers mit Utility AI

Jonas Erkert

Kurzbeschreibung

Utility AI verspricht, durch die Beachtung vieler verschiedener Überlegungen, bei der Entscheidungsfindung die optimale Auswahl eines Verhaltens in jeder Situation. Diese Architektur ist durch den Einsatz mathematischer Funktionen und durch Normalisierung von Eingabewerten in der Lage, verschiedene Informationstypen zu vergleichen und den relativen Nutzen eines Verhaltens zu bewerten. Ziel dieser Bachelorarbeit ist die Demonstration einer KI, welche durch Utility AI gesteuert wird, dabei einem realistischen Tagesablauf nachgeht und in einer Gefahrensituation angemessen reagieren soll.

Aufgabensteller/Prüfer:	Prof. Dr. Christoph Bichlmeier
Arbeit vorgelegt am:	16.09.2019
Durchgeführt an der:	Fakultät für Informatik
Anschrift des Verfassers:	-
Email:	-

Inhaltsverzeichnis

Vorwort

Aufmerksam auf das Thema Utility AI wurde ich durch einen Blog-Post mit dem Namen *Journey into Utility AI with Unreal Engine 4* des Spiele-Entwicklers Tom Looman [1]. In diesem Blog-Post wird das Thema sowie eine mögliche Implementierung in der Unreal Engine 4 in wenigen Abschnitten beschrieben. Durch weitere Recherchen bezüglich des Utility AI Systems entwickelte sich dadurch schließlich der Wunsch, eine Bachelorarbeit zu diesem Thema zu erstellen.

Ein besonderer Dank geht an Professor Dr. Bichlmeier, der eine Ausarbeitung des Themas der Bachelorarbeit ermöglichte und durch regelmäßige Rückmeldungen die Auslegung der Arbeit betreut und vorangetrieben hat.

Außerdem möchte ich mich bei meinen Eltern, Peter und Tamara und meiner Schwester Lena für ihre Unterstützung bedanken.

1 Einleitung

Methoden, wie Behavior Trees und State Machines, werden in der Spieleindustrie seit geraumer Zeit verwendet, um Künstliche Intelligenz[1] in Spielen zu implementieren. Während die Stärken der beiden Systeme dabei in der klaren Definition von Verhaltensabläufen liegen, ist die Auswahl eines Verhaltens, passend zum Kontext der Spielwelt, limitiert. Der Ansatz der Utility AI versucht dieses Problem zu lösen. Anstatt Entscheidungen zu treffen, welche in Behavior Trees oder in State Machines meist binär sind, berücksichtigt die Utility AI mehrere Einflüsse aus der Spielwelt und berechnet dadurch einen Wert, der die Güte eines Verhaltens bewertet.
Zur Implementierung des Utility AI Systems wurde die Unreal Engine 4 gewählt, da diese viele Funktionen beinhaltet, den Tagesablauf eines Kämpfers zu implementieren. Die Engine stellt einen Editor zur Verfügung, mit welchem die Umgebung der KI, ein Level im Spiel, erstellt werden kann. Der Editor bietet außerdem die Möglichkeit die Logik eines Spiels durch das Blueprint-System visuell zu implementieren. Die Programmierschnittstelle der Engine stellt dagegen verschiedene mathematische Funktionen bereit, welche für Berechnungen des Utility AI Systems verwendet werden können.
Da ein Behavior Tree standardmäßig in der Engine zur Erstellung von KI verwendet wird, bietet sich der Vergleich dieser Methode zum Utility AI System an.

1.1 Methoden

Die Implementierung des Utility AI Systems wurde in der Unreal Engine 4 Version 4.22, welche von Epic Games[2] entwickelt wurde, umgesetzt. Epic Games nahm im Zeitraum der Erstellung dieser Arbeit verschiedene Fehlerbehebungen an der Engine vor, wodurch die Version zu 4.22.3 aktualisiert wurde.
Neben der Unreal Engine 4 wurde das 3D Modellierungsprogramm Blender genutzt, um die benötigten 3D Modelle zu erstellen. Mit Hilfe von Blender wurden ein Schwert, zwei verschiedene Schilde, ein Dolch, ein Bogen, ein Pfeil und ein Beutel als Ausrüstung für den Spieler und die KI-Charaktere modelliert. Für das Spiele-Level wurden Kletterpflanzen und ein Busch-Modell angefertigt. Ein Kristall, welcher das Spielziel des Levels darstellt, wurde ebenfalls mit Blender erstellt. Um die benötigten Texturen der 3D Modelle zu erzeugen, wurde die Software GIMP und Substance Painter verwendet. Zur Erstellung des Terrains wurde World Machine Basic und Gaia eingesetzt. Diese Software kann prozedurales Terrain generieren und Erosion simulieren, um schließlich eine geeignete Heightmap und Normalmap zur Nutzung in der Engine exportieren zu können. Zur Programmierung des C++ Source Codes wurde Microsoft Visual Studio 17 und das Plugin Visual Assist X genutzt. Das Plugin erleichtert die Arbeit mit der Unreal Engine 4 API durch zusätzliche Funktionalität und einer speziellen Unterstützung für die Unreal Engine 4.

[1] Künstliche Intelligenz, häufig auch KI genannt. Eng.: Artificial Intelligence, kurz AI.

[2] Epic Games Inc. ist der Entwickler der Unreal Engine 4 - https://www.epicgames.com/site/en-US/home

Zusätzliche Assets, welche im Level verwendet werden, sind über den Epic Games Launcher verfügbar. Assets des *Infinity Blade Grass Lands* Packets wurden zur Erstellung von Gebäuden im Level genutzt. Das *Water Plane* Packet wurde verwendet, um die Oberfläche eines Sees zu erstellen. Zur Erstellung von Gras und Blumen wurden Assets aus dem *Interactive Open World Foliage* Packet genutzt. Texturen aus dem *Kite Demo* Packet wurden zur Texturierung des Terrains verwendet. Um die Charaktere mit den verschiedenen benötigten Animationen auszustatten wurden die Internetseite Mixamo.com und das *MCO Mocap Basic* Packet verwendet.
Um die Implementierung und die Ziele der Arbeit klar zu definieren, wurde der Issue-Tracker[3] YouTrack von JetBrains genutzt. Mit Hilfe des Issue-Trackers können anstehende Aufgaben in unterschiedliche Kategorien eingeteilt werden. Diese Kategorien definieren den Bearbeitungsstand einer Aufgabe. Beispielsweise werden anstehende Aufgaben in die Kategorie *Open* eingeteilt, während erfolgreich implementierte Aufgaben der Kategorie *Closed* zugewiesen werden. Außerdem kann einer Aufgabe eine Priorität und ein Typ, beispielsweise programmieren oder Level Design, zugewiesen werden. Um die Spielziele, die Fähigkeiten der Charaktere und die verschiedenen Verhaltensabläufe der KI zu implementieren, wurde ein Game Design Dokument erstellt. Als Versionsverwaltungssystem (VCS[4]) wurde Git verwendet und als Serverdienst wurde dabei die Seite GitHub genutzt.
Zur Entwicklung wurde der Intel Core i9-9900k genutzt, welcher ein zügiges Kompilieren des C++ Source Codes ermöglichte. Eine nVidia GTX 970 Grafikkarte sorgte für eine ausreichende Leistung zur Darstellung der Szene im Viewport der Unreal Engine 4.
Um einen KI Kämpfer darzustellen ist, neben dem Utility AI System, ein Kampf- und Ausrüstungssystem implementiert. Für die Arbeit wurde ein Level erstellt, welches vier KI-Charaktere beinhaltet, welche einem Tagesablauf nachgehen. Durch den Einbau eines Missionsziels ist der Spieler dazu angehalten mit der KI zu interagieren. Das Level enthält sinnvoll verteilte Ressourcen, wie Tische zum Essen und Betten zum Schlafen. Während der Entwicklung wurde ein minimales Level verwendet, um das Verhalten und andere Funktionalitäten ungehindert zu testen.
Als Grundlage des Utility AI Graphen, auf den in Abschnitt 3.1 eingegangen wird, dient das Unreal Engine 4 Plugin *Generic Graph* [2]. Dieses Plugin wurde ausgebaut, um durch eine Graph-Struktur die Utility AI definieren zu können.

[3] Ein Issue-Tracker ist Teil eines Projektmanagement-Systems. Durch den Einsatz einer solchen Software kann ein Überblick über die Aufgaben eines Projekts behalten werden.

[4] Engl.: Version Control System

2 Utility AI

Das Utility AI System wurde erstmals im Jahr 2010 von Dave Mark auf der Game Developers Conference vorgestellt. Utility AI, in Deutsch in etwa *Künstliche Intelligenz der Nützlichkeit*, bewertet verschiedene Verhalten abhängig vom Kontext in der Spielwelt. Dieses System, von Mark auch *Infinite Axis Utility System* genannt, beschreibt den relativen Nutzen einer Aktion, beziehungsweise eines Verhaltens. Die Utility AI erlaubt den Vergleich zwischen unterschiedlichen Konzepten und Informationstypen. So ist die Utility AI beispielsweise in der Lage, die Gefahrensituation einer KI mit einer Distanz zwischen bestimmten Objekten zu vergleichen. Damit kann die Auswahl eines Verhaltens zwischen konkurrierenden Interessen getroffen werden. Eine Aktion wird durch einen Utility Score bewertet. Dieser beschreibt, wie stark das Verlangen nach einem bestimmten Verhalten ausfällt. Der Utility Score kann daher je nach Personalität eines Charakters oder Kontext einer Situation unterschiedlich ausfallen.
Während das Utility AI System als eine eigenständige Architektur zur Erstellung von KI genutzt werden kann, besteht außerdem die Möglichkeit einer Kombination mit anderen Architekturen, wie beispielsweise einer Kombination mit Behavior Trees [3].
Das Utility AI System analysiert kontinuierlich alle möglichen Optionen, welche der KI zur Verfügung stehen und beurteilt diese dann aufgrund der definierten Faktoren. Die Optionen werden dabei nicht nacheinander beurteilt, stattdessen werden alle Optionen miteinander verglichen. Die Option, welche am besten geeignet und passend für die aktuelle Situation ist, wird schließlich ausgewählt [4]. Die Berechnungen, welche dafür nötig sind, sollen jedoch nicht in jedem Frame durchgeführt werden, sondern in einem definierten Zeitintervall. Im besten Fall wird eine Berechnung durch das Auslösen eines Events gestartet [5]. Um den Rechenaufwand, welcher durch alle möglichen Kombinationen von Eingaben und Informationen aus der Spielwelt entsteht, in einem angemessenen Rahmen zu halten, dabei jedoch trotzdem eine nachvollziehbare Auswahl zu treffen, ist ein wichtiges Ziel des Utility AI Systems [6].
Das Utility AI System erkennt, dass Entscheidungen selten binär sind und versucht die Komplexität der Kombination vieler verschiedener analoger Informationen zu verknüpfen, um einen finalen Entschluss zu treffen. Die Erarbeitung der benötigten Informationen und wie sich diese Informationen in logischer Art und Weise vergleichen und bewerten lassen, ist ein großer Teil der Herausforderung bei der Erstellung einer Utility AI [4].

2.1 Actions

Im Utility AI System wird eine Reihe von Actions[5] definiert, welche der KI zur Auswahl zur Verfügung stehen. Eine Action kann dabei eine unbegrenzte Anzahl an Considerations besitzen (siehe Abbildung 1). Die Nutzung einer beliebigen Anzahl von Considerations verleiht daher dem Utility AI System die alternative Bezeichnung *Infinite Axis Utility System*. Die Considerations einer Action stellen dabei die Überlegungen dar, welche zu einer Auswahl der Action führen können. Der

[5] Dt.: Aktionen

Nutzen, beziehungsweise die Güte einer Action wird im Utility AI System durch Werte (Score) im Intervall von 0 bis 1 beschrieben. Besitzt eine Action einen Score von 1, so ist diese passend für die aktuelle Situation. Eine Action mit dem Score von 0 sollte im aktuellen Kontext dagegen in keinem Fall in Betracht gezogen werden. Werte zwischen 0 und 1 geben einen relativen Nutzen für die aktuelle Situation an. Der berechnete Action Score kann durch den normalisierten Wertebereich einfach mit anderen Actions verglichen werden. Zur Berechnung eines Action Scores werden die normalisierten Considerations der Action multipliziert. Das Produkt definiert dadurch den Action Score. Ein Action Score von 1 kann durch die Multiplikation nur dann berechnet werden, wenn alle Considerations ebenfalls einen Score von 1 besitzen. Für den Fall, dass eine Consideration einen Score von 0 erhält, entsteht das Produkt von 0 und die Action sollte damit nicht in Betracht gezogen werden [3].

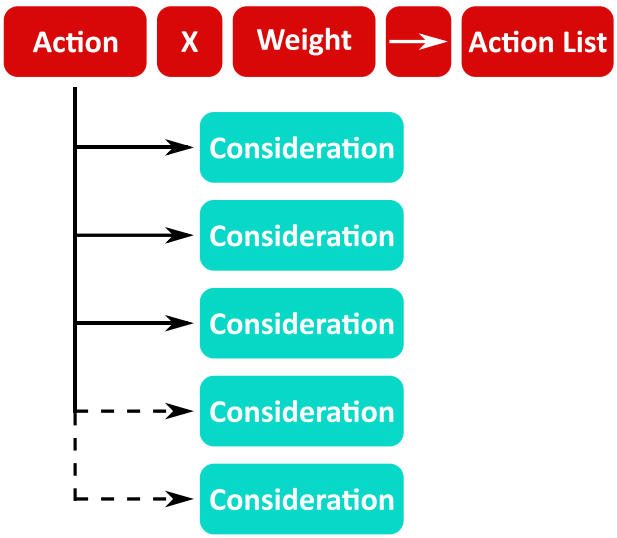

Abbildung 1: Eine Action wird durch die verknüpften Considerations definiert. Eine Gewichtung kann den berechneten Action Score verändern (Abbildung modifiziert aus der GDC 2015 Präsentation [5]).

Soll eine Action bevorzugt werden, kann der berechnete Action Score mit einer Gewichtung[6] multipliziert werden. Dadurch kann ein Vielfaches des Action Scores erzeugt werden. Nach der Nutzung einer Gewichtung befindet sich der Action Score für gewöhnlich nicht mehr im Intervall von 0 bis 1 [6]. Nach der Berechnung des Actions Scores wird dieser Wert einer Liste hinzugefügt, welche die Scores aller Actions enthält, die sogenannte Action-List. Von dieser Action-List wird im nächsten Schritt durch eine Auswahlmethode eine Action ausgewählt.

2.1.1 Auswahlverfahren

Sind alle Actions einer Utility AI in der Action-List gesammelt, so kann eine zum Kontext des Spiels passende Action ausgewählt werden. Die KI stellt mit der Auswahl fest, welche Action von allen möglichen zur Verfügung stehenden Actions, im Moment der Auswertung die höchste Priorität besitzt [4].

[6] Engl.: Weight

Die Selektion kann durch unterschiedliche Auswahlverfahren stattfinden. Im einfachsten Fall wird dabei die Action mit dem höchsten Score ausgewählt. Hierbei muss dann lediglich das oberste Element in der sortierten Action-List entnommen werden. Diese Selektionsmethode wird *Highest Selection* genannt und ist für viele Spiele ausreichend, beispielsweise sollte eine Schach-KI in jedem Fall den Zug mit dem höchsten Score auswählen. Für andere Spiele kann die Auswahl des höchsten Scores in einer bestimmten Situation roboterhaft und eintönig wirken, da ein spezielles Verhalten, welches zu dieser Situation passt, immer ausgewählt wird. Ein anderes Auswahlverfahren, welches die Action Scores als eine Gewichtung betrachtet und nach dieser Gewichtung ein zufälliges Element auswählt, kann bei dem zuvor genannten Problem Abhilfe schaffen. Dieser sogenannte *Weighted Random* Algorithmus wird in Abschnitt 3.2 ausführlich beschrieben. In den meisten Fällen trifft dieses Auswahlsystem die richtige Entscheidung, wobei jedoch eine die geringe Wahrscheinlichkeit besteht, dass eine unpassende Action ausgewählt wird. Ein weiterer Ansatz ist die zufällige Auswahl der höchsten Elemente in einer definierten Reichweite. Dazu können beispielsweise aus der Action-List die höchsten fünf Elemente in Betracht gezogen werden, wobei eine Zufallszahl dann eines dieser Elemente auswählt. Eine Definition der Reichweite in Prozent, zum Beispiel 10 % im Bereich der am höchsten bewerteten Action, ist ebenfalls möglich. Diese Auswahlmethode hat den Nachteil, dass die Scores der Actions nicht in Betracht gezogen werden und große Unterschiede zwischen den Scores bestehen können. Dadurch kann bei dieser Selektionsmethode ebenfalls eine unpassende Auswahl getroffen werden. Eine Kombination der *Weighted Random* Methode und dem Auswahlverfahren, welches nur die höchsten Elemente in Betracht zieht, verringert die Wahrscheinlichkeit zur Selektion einer unpassenden Action [6].

2.1.1.1 Bucketing

Durch die Kombination der beiden soeben beschriebenen Auswahlverfahren wird die Wahrscheinlichkeit, eine unpassende Action auszuwählen, reduziert. Diese Kombination kann jedoch ebenso zur Ausführung eines unangebrachten Verhaltens führen. Eine Wache sollte beispielsweise, während sie vom Spieler angegriffen wird, zu keinem Zeitpunkt in Betracht ziehen, essen zu gehen. Lediglich Actions, welche in Relation zum Kampf stehen, sollten ausgewählt werden können. Wird dieses Verhalten nicht bereits durch den Einsatz von Triggern[7] erreicht, auf die in Abschnitt 2.2 eingegangen wird, ist der Einsatz von Bucketing[8], auch Dual Utility AI genannt, angebracht. Alle Actions werden dabei in verschiedene Kategorien von Buckets eingeteilt. Jeder dieser Buckets bekommt anschließend eine Gewichtung zugewiesen. Ein Bucket mit höherer Priorität, also höherer Gewichtung, wird immer zuerst verarbeitet. Ist in einem Bucket mit hoher Priorität eine beliebige Action gültig, so wird diese den Actions des Bucket bevorzugt, welche eine niedrigere Priorität besitzen. Durch Bucketing können Actions, welche beispielsweise Kampfverhalten definieren, in eine hoch priorisierte Kategorie eingeteilt werden. Diese werden

[7] Dt.: Auslöser

[8] Bucketing, von engl.: bucket, dt.: Eimer

damit im Spiel Actions bevorzugt, die passive Verhalten wie Essen oder Schlafen auslösen. Abhängig von der Spielsituation ist es möglich, die Gewichtung der Buckets zu verändern und dadurch neue Prioritäten festzulegen [6].

2.2 Considerations

Considerations[9] besitzen im Utility AI System die Aufgabe, Eingaben und Informationen aus der Spielwelt in Entscheidungen zur Auswahl einer Action umzuwandeln. Eine Action kann dabei beliebig viele Considerations besitzen. Considerations beschreiben durch einen Wert, welcher normalisiert ist, die Güte einer Überlegung im aktuellen Kontext der Spielwelt. Ein Consideration Score von 1 bedeutet dabei, dass die Überlegung der KI, im Moment der Auswertung, besonders wichtig ist und damit die Auswahl der verknüpften Action wahrscheinlicher wird. Considerations, welche die KI in diesem Moment vernachlässigen kann, besitzen dagegen einen Consideration Score, welcher gegen 0 verläuft. Besitzt eine Consideration einen Score von 0, wird damit auch die verknüpfte Action von der Auswahl ausgeschlossen. Da der Action Score durch die Multiplikation aller Considerations berechnet wird, bedeutet eine Überlegung mit dem Wert von 0 ein Produkt von 0, auch wenn die anderen Considerations der Action einen sehr hohen Score besitzen. Diese Art von Consideration wird als Trigger bezeichnet, da, ähnlich zu booleschen Variablen, eine komplette Action aktiviert oder deaktiviert werden kann.

Eine Consideration besitzt verschiedene Komponenten, welche zur Berechnung des Scores und der Beschreibung der Überlegung benötigt werden. Dargestellt werden einige dieser Komponenten in Abbildung 2. Die erste Komponente definiert den Namen und die Beschreibung einer Consideration. Damit wird der Nutzen und die Funktion der Consideration definiert. Die nächste wichtige Komponente ist der Eingabewert (Input). Durch den Input wird bestimmt, welche Information eine Consideration verarbeitet. Response Curve Parameter, auf welche in Abschnitt 2.2.3 eingegangen wird, werden zur korrekten Auswertung des Eingabewerts benötigt. Zusätzliche Parameter wie Tags[10] und Bookends können ebenfalls in einer Consideration definiert werden [5]. Bookends werden zur Normalisierung der Eingabewerte benötigt. Auf die Normalisierung und Bookends wird in Abschnitt 2.2.2 genauer eingegangen.

[9] Dt.: Überlegungen

[10] Dt.: Etikett, wird zur Markierung spezieller Objekte in Spielen verwendet.

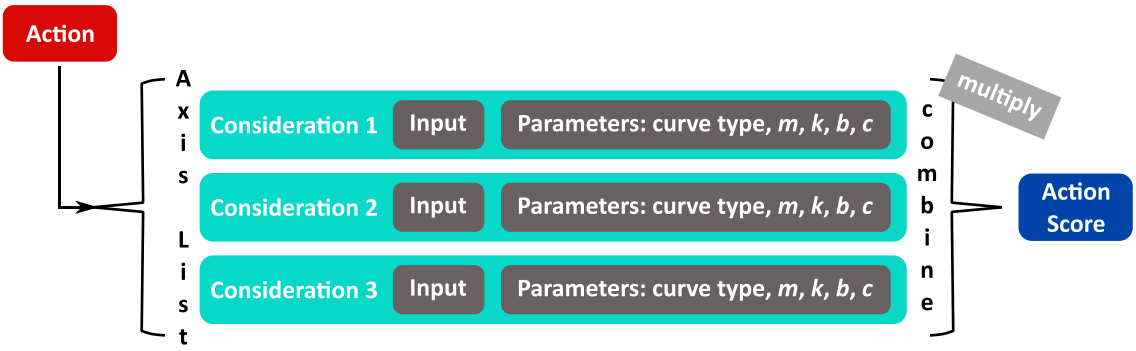

Abbildung 2: Darstellung einer Action, welche 3 verschiedene Considerations besitzt. Abgebildet sind außerdem die verschiedenen Komponenten und die Berechnung des Action Scores durch Multiplikation (Abbildung modifiziert aus der GDC 2013 Präsentation [3]).

2.2.1 Eingabewerte

Eingabewerte, auch Input genannt, werden zur Berechnung des Consideration Scores benötigt. Eine Response Curve wandelt den Input aus der Spielwelt in normalisierte Werte um, damit ein Vergleich der Informationen möglich ist. Eingabewerte können verschiedene Informationen umfassen, beispielsweise Daten des KI- oder Spieler-Charakters. Während eine Consideration beispielsweise die Lebenspunkte der KI bewertet, betrachtet eine andere Consideration die Lebenspunkte des Spielers, um den Consideration Score zu berechnen. Neben Charakter-Daten können auch Informationen aus der Spielwelt, wie zum Beispiel die Distanz zwischen Objekten oder die verstrichene Zeit zwischen zwei Ereignissen, als Eingabewerte definiert werden [5]. Diese Werte werden von den Response Curve Funktionen als *x*-Wert verwendet, um den Score, den *y*-Wert der Funktion, zu berechnen [3]. Eingabewerte benötigen eine Parametrisierung zur korrekten Auswertung. Damit der Input normalisiert werden kann, muss ein Minimum und ein Maximum des Eingabewertes angegeben werden. Soll beispielsweise eine Distanz normalisiert werden, so kann als Minimum der Wert 0 und als Maximum der Wert 500 angegeben werden.

2.2.2 Normalisierung

Die Eingabewerte einer Consideration müssen vor der Nutzung einer Response Curve normalisiert werden, da die Response Curves einen normalisierten Wertebereich abbilden. Die *x*-Achse der Response Curve definiert ein Intervall zwischen 0 und 1, oder zwischen 0 % und 100 %. Eingabewerte sind teilweise bereits standardmäßig normalisiert. Zum Beispiel werden die Lebenspunkte in Spielen häufig in einem Bereich von 0 bis 100 angegeben und können damit ohne Aufwand in das Intervall von 0 bis 1 umgewandelt werden. Jedoch besitzt die Spielwelt eine Reihe an Eingabewerten, welche nicht standardmäßig normalisiert sind. In diesem Fall muss ein Minimum und ein Maximum der Eingabeparameter angegeben werden, so genannte Bookends[11]. Die definierten Bookends können anschließend normalisiert werden. Das minimale Bookend bestimmt dabei den Wert, welcher zur Berechnung des Consideration Scores durch die Response Curve am wenigsten interessant ist. Das maximale Bookend dagegen definiert den

[11] Dt.: Buchstütze, Abschluss

Wert, der im Kontext des Spiels am wichtigsten ist. Alle Eingabewerte, welche sich unter oder über den Bookends befinden, werden durch Clamping[12] auf das minimale, beziehungsweise auf das maximale Bookend gesetzt. Als Beispiel kann die Anzahl von Gegnern angeführt werden, welche eine KI angreifen. Als minimales Bookend wird 1 Gegner angegeben, als maximales Bookend 5 Gegner. Daher macht es kein Unterschied, ob die KI von 6 oder 15 Gegnern attackiert wird, da die Anzahl auf das maximale Bookend von 5 geclamped wird [5].

2.2.3 Response Curves

Response Curves[13] sind ein zentraler Bestandteil zur Auswertung der bereitgestellten Informationen zu einem Utility Score. Sie verwerten einen konkreten Wert in einen, zu einer Consideration zugehörigen, abstrakten Wert. Im Kontext der Utility AI dienen Response Curves zur mathematischen Repräsentation der Beziehung zwischen Informationen der Spielwelt und möglichen Verhalten. Beispielhaft wäre dabei die Beziehung zwischen Distanz und Gefahr. Befindet sich der Spieler in einer bestimmten Reichweite der KI, so kann durch eine Response Curve der berechnete Score zu einer definierten Distanz definiert werden. Response Curves sind zur Beschreibung dieser Beziehungen geeignet, da sie einfach zu erstellen und zu visualisieren sind. Soll aus einem Eingabewert der Consideration Score berechnet werden, müssen zunächst verschiedene Argumente weitergereicht werden. Neben dem Eingabewert *x* benötigt die Auswertung außerdem den Typ der zu verwendenden Funktion und die Parameter *m*, *k*, *b* und *c*. Auf diese Parameter wird in den folgenden Abschnitten eingegangen. Anschließend findet ein Clamping der Eingabewerte statt. Dabei werden alle Werte, die über dem definierten Bereich liegen, auf den maximalen Wert dieses Bereichs gesetzt. Nach dem Clamping kann der Consideration Score anhand des Eingabeparameters und der verwendeten Kurve berechnet werden. Bei dem berechneten Score muss unter Umständen erneut ein Clamping des Wertes stattfinden [3]. In den folgenden Abschnitten sollen nun die verschiedenen Kurventypen beschrieben werden.

2.2.3.1 Lineare Kurve

Die lineare Kurve ist eine Gerade mit einer konstanten Steigung. Eine lineare Funktion kann damit als direkter Multiplikator von Eingabe-Werten verwendet werden.

Anstatt die gewöhnliche Geradengleichung zu verwenden, wird in dieser Arbeit eine abgeänderte Form der linearen Response Curve eingesetzt, welche weitreichender parametrisiert und angepasst werden kann. Demnach wird die Geradengleichung

$$f(x) = mx + b \tag{2.1}$$

zu

$$f(x) = m(x - c)^k + b \tag{2.2}$$

erweitert. Dabei beschreibt *m* die Steigung, *k* den Exponenten der Kurve, *b* die vertikale Verschiebung und *c* die horizontale Verschiebung im Koordinatensystem. Abbildung 3 zeigt eine lineare Funktion, welche eine Steigung von 1 besitzt [3].

---

[12] Dt.: Einklemmen, Festklemmen

[13] Dt.: Reaktionskurve

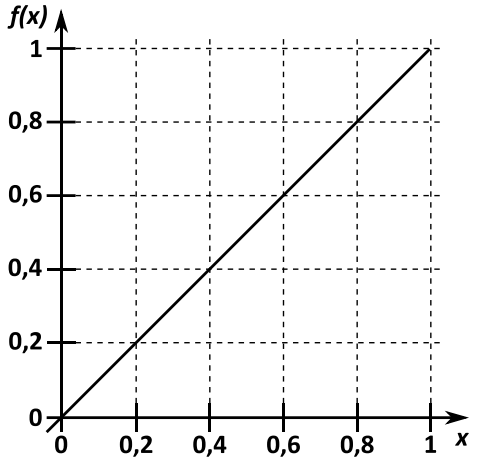

Abbildung 3: Darstellung einer linearen Response Curve mit der Funktion *f(x) = x*.

2.2.3.2 Quadratische Kurve

Eine quadratische Kurve beginnt im Intervall von 0 bis 1 mit einer leichten Steigung, die anschließend schnell gegen den Wert 1 ansteigt. Die Gleichung (2.2), welche zur Definition einer linearen Kurve beschrieben wurde, kommt ebenfalls zur Definition einer quadratischen Kurve zum Einsatz. Die Variable *k*, welche den Exponenten der Funktion beschreibt, muss hierbei jedoch den Wert 2 annehmen, damit die Funktion eine quadratische Kurve abbilden kann [3]. Soll die Kurve rotiert werden und ein starker Anstieg der Kurve zu Beginn und ein langsamer Anstieg am Ende der Kurve erzielt werden, so muss sich der Wertebereich von *k* zwischen 0 und 1 befinden [6]. Abbildung 4 zeigt eine quadratische Funktion mit der Gleichung

$$f(x) = 1(x - 0)^2 + 0 \tag{2.3}$$

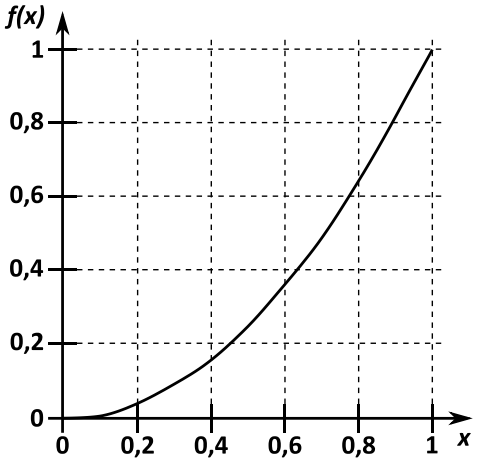

Abbildung 4: Darstellung einer quadratischen Funktion, vgl. Gleichung (2.3), im Wertebereich von 0 bis 1.

2.2.3.3 Logistische Kurve

Eine logistische Kurve zeichnet sich durch einen langsamen Start und ein langsames Ende einer Kurve aus, wobei die Mitte der Kurve dagegen einen schnellen Anstieg besitzt.
Hierbei sollte folgende Formel

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.4}$$

zu einer Formel, welche durch die Variablen *m*, *k*, *b* und *c* stark parametrisiert werden kann,

$$f(x) = k \cdot \frac{1}{1 + 1000em^{-jx+c}} + b \tag{2.5}$$

umgestellt werden. Die Variable *m* gibt die Steigung der Kurve am Wendepunkt an. Eine vertikale Verschiebung der Kurve kann durch den Parameter *b* definiert werden, die horizontale Verschiebung durch die Variable *c*. Die vertikale Größe der Kurve wird von *k* bestimmt [3]. Durch den zusätzlichen Parameter *j* kann die horizontale Stauchung der Kurve angegeben werden. Abbildung 5 zeigt eine logistische Funktion, welche durch

$$f(x) = 1 \cdot \frac{1}{1 + 1000 \cdot 2e^{-25x+2}} + 0 \tag{2.6}$$

parametrisiert wurde.

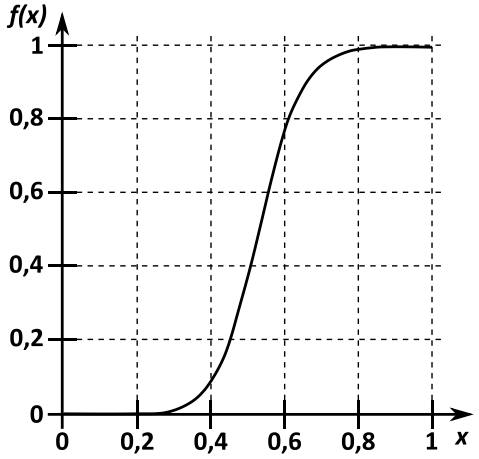

Abbildung 5: Darstellung einer logistischen Kurve, vgl. Gleichung (2.6), im Wertebereich von 0 bis 1.

2.2.3.4 Stückweise lineare Kurve

Durch die Verwendung der in den Abschnitten 2.2.3.1 bis 2.2.3.3 beschriebenen Kurventypen können bereits viele Anwendungsfälle abgedeckt werden. Unter bestimmten Umständen kann aber eine mathematische Kurve nicht ausreichen, um spezifische Eingabewert abbilden zu können. Wird eine sehr präzise, individuelle Repräsentation einer Kurve benötigt, so kann diese durch Abschnitte aus linearen Funktionen definiert werden, wie in Abbildung 6 dargestellt [6].

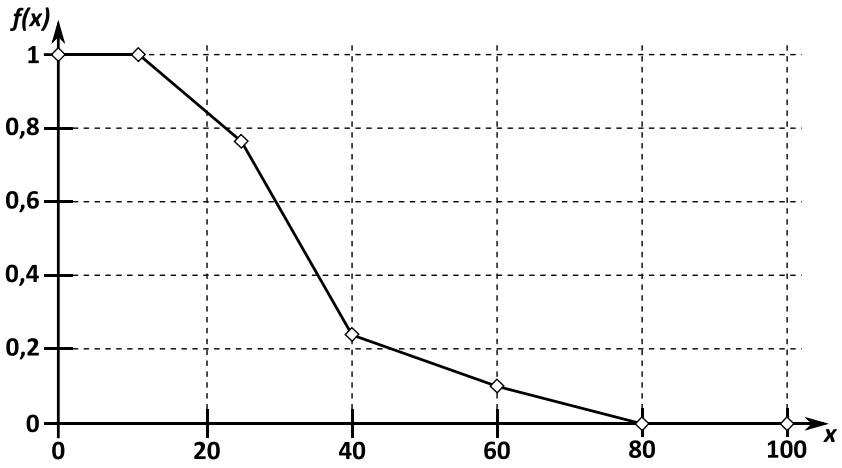

Abbildung 6: Darstellung einer benutzerdefinierten Kurve, bestehend aus Abschnitten linearer Funktionen (modifiziertes Schaubild aus Game AI Pro [6]) .

2.3 Decisions

Wurde durch ein Auswahlverfahren eine passende Action für die KI ausgewählt, kann anschließend eine definierte Decision[14] ausgeführt werden. Decisions im Utility AI System sind mit einer Funktion im Code des Spiels verknüpft. Diese Funktion führt eine Fähigkeit aus, lässt beispielsweise die KI zu einer bestimmten Stelle laufen, spielt eine Animation ab oder startet anderweitige Logik, beispielsweise ein spezielles Skript, in der Spielwelt. Die Ausführung der Decision ist spezifisch zum Spiel und nicht relevant in Bezug auf das Utility AI System.
Eine Decision besitzt verschiedene Eigenschaften. Zum einen muss ein Decision-Identifikator angegeben werden, mit welchem definiert werden kann, welches Verhalten eine Decision ausführen soll. Zum anderen kann ein Decision Parameter angegeben werden, um bestimmte Einstellungen im ausgeführten Verhalten zu konfigurieren, wie beispielsweise der Name des Skripts. Außerdem muss die Decision eine Verknüpfung zur KI aufweisen, welche das Verhalten ausführen soll. Alternativ kann ebenfalls eine Verknüpfung zum Objekt angegeben werden, mit dem die KI bei der Ausführung der Decision interagieren soll, zum Beispiel ein Charakter oder ein Gerät, welches die KI benutzen soll [5].

2.3.1 Compensation Factor

Werden die Werte von verschiedenen normalisierten Considerations multipliziert, so sinkt der Wert der Action. Besitzt eine Consideration beispielsweise den Wert von 1 und eine Andere den Wert von 0,9, wird dadurch ein Action Score von 0,9 berechnet. Besitzt die Action jedoch zwei weitere Considerations, welche ebenfalls einen Consideration Score von 0,9 erhalten haben, so sinkt dadurch der Action Score. Das Produkt der ersten beiden Considerations wird demnach mit dem Score der dritten Consideration multipliziert, der neue Action Score ist damit 0,81. Wird anschließend die vierte Consideration miteinberechnet, so beträgt das Produkt damit 0,729. Actions, welche viele Considerations mit einem hohen Score besitzen, verlieren durch diesen Effekt stark an Wert und werden dadurch benachteiligt. Da das Utility AI System eine unbegrenzte Anzahl von möglichen Considerations zulässt, ist dieser Effekt zur Berechnung eines korrekten Action Scores unerwünscht.
Um der Reduzierung von Action Scores entgegenzuwirken, kann ein Compensation Factor[15] miteinbezogen werden. Der Compensation Factor wird für jede Consideration berechnet und sorgt für einen Ausgleich des berechneten Scores, wobei dieser die Anzahl an Considerations, welche eine Action besitzt, beachtet. Abbildung 7 zeigt die Formel zur Berechnung des Compensation Factors und die Nutzung der Formel mit 6 Considerations und einem Consideration Score von 0,9 [5].

[14] Dt.: Entscheidung

[15] Dt.: Ausgleichsfaktor

ModificationFactor = 1 - (1 / NumberConsiderations)
MakeUpValue = (1 - Score) * ModificationFactor
FinalConsiderationScore = Score + (MakeUpValue * Score)

ModificationFactor = 1 - (1 / 6) = 0,833
MakeUpValue = (1 – 0,9) * 0,833 = 0,083
FinalConsiderationScore = 0,9 + (0,083 * 0,9) = 0,975

Abbildung 7: Berechnung des Compensation Factors für eine Consideration. Dargestellt sind die Formeln des Compensation Factors und die beispielhafte Berechnung eines Consideration Scores.

Der Wert der Variable *Score* entspricht bei der Berechnung des Compensation Factors dem Wert des originalen Action Scores. In Abbildung 8 ist zu erkennen, dass der Action Score einen geringeren Werteverlust bei der Nutzung vieler Considerations aufweist, für den Fall, dass der Compensation Factor miteinbezogen wird.

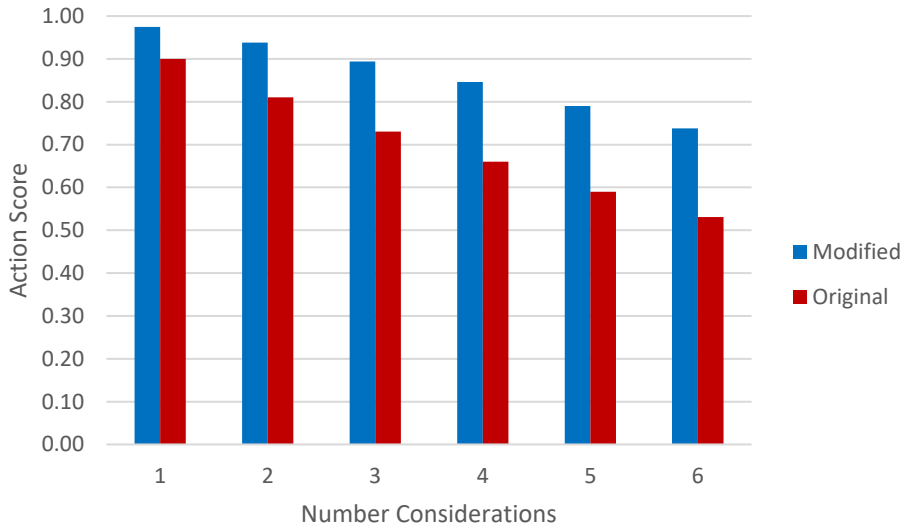

Abbildung 8: Vergleich zwischen Action Scores mit (Modified) und ohne (Original) Berechnung des Compensation Factors (Abbildung modifiziert aus [5]).

2.3.2 Inertia

Zu einem bestimmten Zeitpunkt im Spiel können verschiedene Actions einen sehr ähnlichen Score berechnen. Da eine Entscheidung, je nach Konfiguration, mehrmals in der Sekunde ausgewählt wird, kann dies zu einem schnellen Wechsel von Verhalten führen. Durch das Hinzufügen von Inertia[16] zum Utility AI System kann ein oszillierendes Verhalten vermieden werden. Wird Inertia einer Action hinzugefügt, so erhält die zuletzt ausgewählte Action ein zusätzliches Gewicht. Dadurch bekommt die aktuelle Action einen höheren Score. Das schnelle Wechseln von Verhalten kann durch den entstandenen Größenunterschied der Scores verhindert werden. Die Inertia kann dabei entweder auf die Dauer des Verhaltens angewendet werden oder auf ein

[16] Dt.: Trägheit

definiertes Zeitintervall. Die eingesetzte Gewichtung kann anschließend mit sofortiger Wirkung zurückgesetzt werden, wobei eine schrittweise Reduktion ebenfalls möglich ist [6].
Ein anderer Ansatz zur Vermeidung von schnell wechselndem Verhalten ist das Hinzufügen eines Cooldowns[17] zu einer Action. Wird eine Action ausgewählt, so erhält diese nach der Ausführung des Verhaltens einen geringen Action Score. Dies steht im Gegensatz zur Inertia, denn anstatt ein ausgewähltes Verhalten zu bevorzugen wird hierbei ein Verhalten nach einmaliger Ausführung für eine gewisse Zeit benachteiligt [3].

2.4 Vor- und Nachteile von Utility AI

Durch die Berücksichtigung mehrerer Überlegungen bei der Auswahl eines Verhaltens können binäre Entscheidungen, wie sie oft in KI-Architekturen wie Behavior Trees oder State Machines getroffen werden, vermieden werden. Bei den beiden genannten Architekturen wird zu einem Zeitpunkt häufig lediglich eine Überlegung berücksichtigt [3]. Trifft die Überlegung nicht zu, wird zu einer nächsten übergegangen. Dies geschieht in einer vorab festgelegten Reihenfolge [4]. Entscheidungen werden dabei aufgrund von willkürlichen und statischen Schwellenwerten getroffen. Beispielsweise wird ein Verhalten nicht gewechselt, wenn sich der Spieler in einem Abstand von unter 30 Metern zur KI befindet. Stattdessen wird die Distanz fließend bewertet und eine entsprechende Reaktion kann durch die Nutzung mathematischer Funktionen exakt angepasst werden. Das Utility AI System passt sich besser an im Spiel auftretende Spezialfälle an, als andere Architekturen, wie Behavior Trees, da selbst bei schwierigen Entscheidungen stets das optimale Verhalten gewählt wird und nicht das nächstbeste Verhalten, welches ein bestimmtes Kriterium erfüllt. Zusätzliche Verhalten können durch die Nutzung von Actions und Considerations ohne großen Aufwand hinzugefügt werden, ohne dass dabei eine Umstrukturierung der bestehenden Definition der KI erfolgen muss. Durch die Architektur des Utility AI Systems werden die benötigten Informationen zur Auswahl und Ausführung eines Verhaltens schnell und effizient berechnet. Durch die eingesetzten Werkzeuge zur Erstellung der Utility AI kann der Designer einfach die Definition dieser visualisieren, an gewünschtes Verhalten anpassen und optimieren.
Ein Nachteil des Utility AI Systems ist die eingeschränkte Fähigkeit, Entscheidungsfindungen zu debuggen. Durch die statische Definition von Verhalten und die Übergänge dieser Zustände kann eine Verhaltensänderung in Behavior Trees und State Machines einfach nachvollzogen werden. Da die Auswahl im Utility AI System jedoch dynamisch und nicht durch eine vorgefertigte Reihenfolge getroffen wird, sind Wechsel von Verhalten schwieriger nachzuvollziehen. Hierbei muss dann auf alle Überlegungen, welche zur Auswahl eines Verhaltens führen, eingegangen werden [3].

[17] Dt.: Abkühlen

2.5 Optimierung

Wird das Utility AI System von Dutzenden oder sogar hunderten Charakteren genutzt, muss eine hohe Anzahl von Actions und Considerations ausgewertet und berechnet werden. Damit die Utility AI dennoch performant bleibt, können Optimierungen vorgenommen werden. Actions werden vorab nach ihrer Gewichtung sortiert. Die Action mit der höchsten Gewichtung ist dabei an der ersten Position der Liste. Considerations, welche mit einer Action verknüpft sind, werden ebenfalls sortiert. Dabei wird jedoch nicht nach einer Gewichtung sortiert, sondern nach der Wahrscheinlichkeit, den Action Score zu reduzieren. Diese können Considerations mit einer Trigger-Funktionalität sein, welche ganze Actions deaktivieren können. Verändert eine Consideration den Action Score nur leicht, wird diese nahe dem Ende aller möglichen Considerations angeordnet. Auch Überlegungen, welche eine hohe Rechenleistung erfordern, werden am Ende positioniert. Bei der Auswertung und Auswahl einer Action können durch die Sortierung diejenigen Actions ausgeschlossen werden, welche unter keinen Umständen verwendet werden sollen. Diese Actions müssen daher nicht weiter ausgewertet werden. Häufig genügt dabei lediglich die Gewichtung, um eine Action auszuschließen. Dadurch werden, durch wenig Aufwand, die Actions, welche eine hohe Wahrscheinlichkeit besitzen tatsächlich ausgewählt zu werden, in Betracht gezogen [5].

3 Implementierung der Utility AI in die Unreal Engine 4

Um das Utility AI System in die Unreal Engine 4 zu implementieren, werden der Engine zwei Kernkomponenten, der Utility AI Graph und der Utility AI Controller, hinzugefügt. Während der Graph für die Definition und Parametrisierung von Actions, Considerations und Decisions zuständig ist, hat der Utility AI Controller die Aufgabe, die Utility AI auszuwerten und das selektierte Verhalten einer KI auszuführen. Die Implementierung findet in der Form eines Plugins statt. Im Gegensatz zu einer projektspezifischen Implementierung hat ein Plugin den Vorteil, wiederverwendbar zu sein. Durch das Einfügen des Plugin Ordners in den Root Ordner eines Projekts, kann das Plugin einfach einem neuen Projekt hinzugefügt werden. Nach einer Aktivierung des Plugins steht der Utility AI Graph als neuer Asset-Typ zur Verfügung. Außerdem kann nun der Utility AI Controller als Elternklasse bei der Erstellung eines neuen AI Controllers gewählt werden. Neben dem Utility AI Graph und der Utility AI Controller Klasse sind im Plugin ebenfalls häufig genutzte Response Curves enthalten, um die Erstellung einer Utility AI zu beschleunigen. Eine Herausforderung bei der Implementierung der Utility AI ist die Verknüpfung zwischen dem C++ Code und dem Blueprint Visual Scripting System[18] der Unreal Engine 4. Da in der Auswertung des Utility AI viele Schleifen verwendet werden, wurde diese rechenintensive Logik in C++ geschrieben. Komplexe mathematische Berechnungen sind nach Angaben von Epic Games ebenfalls performanter in C++ als in Blueprint [8]. Das Blueprint System hat dagegen die Vorteile einer schnelleren Erstellung und Iteration von Logik. Daher wird dieses System zum Erlangen von Informationen aus der Spielwelt und zur Implementierung der Verhaltensabläufe genutzt, da die Logik dieser Komponenten häufig angepasst werden muss. Außerdem ist der Zugriff auf Objekte und Parameter in der Spielwelt durch Blueprint weniger umständlich als durch C++. Die Verknüpfung zwischen den in C++ implementierten und den in Blueprint erstellten Funktionen spielt dabei eine wichtige Rolle. Auf diese Verknüpfung wird in Abschnitt 3.2 näher eingegangen. Durch den Einsatz des Macros *UFUNCTION* im C++ Source Code, generiert die Engine, je nach eingesetztem Schlüsselwort, verschiedene Node[19]-Typen. Um eine Funktion als Node in Blueprint zur Verfügung zu stellen, muss das Schlüsselwort *BlueprintCallable* genutzt werden. Besitzt die Funktion ein Argument, so wird der entsprechende Datentyp als Eingabe-Pin des Nodes generiert. Für bestimmte Funktionen ist es sinnvoll, keine native C++ Implementierung zur Verfügung zu stellen, sondern die Logik durch Blueprint zu implementieren. Dies kann durch die Nutzung des Schlüsselworts *BlueprintImplementableEvent* erreicht werden. Um in C++ erstellte Variablen in Blueprint nutzbar zu machen, können durch das Macro UPROPERTY ebenfalls spezielle

[18] Das Blueprint Visual Scripting, häufig Blueprint(s) oder kurz BP genannt, ist ein vollständiges Gameplay-Scripting-System der Unreal Engine 4. Durch die Nutzung bestimmter Knoten, welche miteinander verknüpft werden können, kann eine Logik direkt im Unreal Engine 4 Editor erstellt werden [7].

[19] Dt.: Knoten

Schlüsselwörter genutzt werden und somit den Zugriff auf diese Variablen definieren. Das Schlüsselwort *BlueprintReadWrite* erlaubt beispielsweise das Speichern und Lesen einer Variablen. Durch die Nutzung von *Category* können Funktionen und Variablen in eine Kategorie eingeteilt werden. Auch die Definition eines maximalen und minimalen Wertes für Slider im Editor sind durch *meta*-Schlüsselwörter, siehe Abbildung 9, möglich. In der Implementierung dieser Bachelorarbeit wurden weitere Schlüsselwörter genutzt, auf die jedoch nicht weiter eingegangen werden soll. Abbildung 9 zeigt die Nutzung der speziellen Macros und Schlüsselwörter, welche zur Generierung der benötigten Nodes verwendet wurden.

```
/**
 * Define here how the consideration score evaluation is implemented.
 *
 * @param     ConsiderationName The name of the current consideration node.
 * @see       SetConsiderationScores().
 */
UFUNCTION(BlueprintCallable, BlueprintImplementableEvent, Category = "UtilityAI")
void ImplementConsiderationScores(FName ConsiderationName);


UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category = "UtilityAINode|ActionNode",
          meta = (ClampMin = "0.0", ClampMax = "100.0"))
float Weight;
```

Abbildung 9: Dargestellt sind die Macros UFUNCTION und UPROPERTY mit verschiedenen Schlüsselwörtern, welche die Darstellung im Blueprint Visual Scripting definieren.

Die Generierung der Nodes berücksichtigt außerdem die Kommentare, welche im C++ Source Code erstellt wurden. Dadurch wird im Unreal Engine 4 Editor ein Tooltip erzeugt, welcher die in den Kommentaren beschriebene Funktionalität des Nodes anzeigt und Hilfe zur Nutzung von Pins eines Nodes zur Verfügung stellt.
Die Implementierung des Utility AI Controllers wurde zunächst in Blueprint umgesetzt, bevor die Logik in C++ übertragen wurde. Dadurch konnte in kurzer Zeit eine funktionierende und grundlegende Auswertung der Utility AI erstellt und die wichtigsten Funktionen ermittelt werden.

3.1 Utility AI Graph

Die erste Kernkomponente besteht aus dem Utility AI Graphen. Dieser Graph wird benötigt, um Utility Actions, Considerations und Decisions für die Utility AI zu definieren. Diese Bestandteile der Utility AI werden als Nodes im Graphen dargestellt. Der Editor bietet eine visuelle Oberfläche, um Actions, Considerations und Decisions in einer Baumstruktur anzuordnen. Die Variablen der Nodes können dabei über das *Property Panel*, welches im Graph Editor integriert ist, parametrisiert werden. Um den Utility AI Graph nutzen zu können, muss das Asset im Content Browser[20] erstellt werden. Dabei wurde durch das Installieren des Plugins die neue Kategorie *UtilityAI*

[20] Durch den Content Browser können Dateien in der Unreal Engine 4 strukturiert, editiert und erstellt werden.

zur Auswahlliste hinzugefügt (siehe Abbildung 10). Dadurch kann dann das Utility AI Asset erstellt werden.

Abbildung 10: Darstellung des Content Browsers, über welchen das Utility AI Asset eingefügt werden kann**.**

Das Asset besteht einerseits aus dem Editor, welcher eine visuelle Darstellung und die Bearbeitung der Nodes erlaubt, andererseits liegt dem Utility AI System die Klasse *UtilityAINode* zugrunde, durch welche die Parameter der Action-, Consideration- und Decision-Nodes gespeichert werden. Eine Enumeration in der *UtilityAINode* Klasse definiert den Node-Typ und damit auch dessen Funktion in der Utility AI.

Nachfolgend soll nun der Arbeitsablauf zum Erstellen der Verhaltensauswahl durch den Utility AI Graph Editor beschrieben werden.

Zur korrekten Auswertung der Utility AI muss im Utility AI Graph ein sogenannter Root[21]-Node existieren. Durch einen Rechtsklick in den Graphen kann die Palette aus möglichen Actions aufgerufen werden. Um einen Utility AI Node zum Editor hinzuzufügen, muss mit einem Doppelklick der Eintrag *UtilityAINode* ausgewählt werden. Damit der erste eingefügte Node den Root-Node repräsentiert, muss dieser über das *Property Panel*, welches sich standardmäßig auf der rechten Seite des Graph Editors befindet, konfiguriert werden. Die Variablen des Nodes werden in diesem *Property Panel* in Kategorien unter dem Abschnitt *Utility AINode* eingeteilt, um eine erhöhte Bedienbarkeit zu bieten. Im zweiten Abschnitt *UtilityAI Node Editor* kann die Darstellung von Nodes im Graph Editor des Nodes angepasst werden, wobei der Name und die Farbe des Nodes definiert werden können (siehe Abbildung 11). Um eine übersichtliche Darstellung der Node-Typen zu erzeugen, wurde ein Farbschema für die Nodes erstellt. Im gewählten Farbschema dieser Bachelorarbeit wird schwarz für Root-Nodes verwendet, die Farbe rot für Action-Nodes, während Consideration-Nodes die Farbe blau erhalten und Decision-Nodes die Farbe grün zugewiesen wird.

[21] Dt.: Wurzel

Abbildung 11: Konfigurations-Möglichkeiten eines Nodes zur Darstellung im Utility AI Graph Editor.

Zur Differenzierung der Node-Typen in der Auswertung der Utility AI muss die Art des Nodes festgelegt werden. Dabei stehen die Optionen Root-Node, Action-Node, Consideration-Node und Decision-Node zur Auswahl (siehe Abbildung 12). Da der erste eingefügte Node einen Root-Node repräsentieren soll, muss die gleichnamige Auswahl getroffen werden. Für die Auswertung der Nodes ist es notwendig, den *Node Name* in dem Abschnitt *Utility AI Node* auszufüllen. Die Trennung des im Editor angezeigten *Node Title* und des zur Berechnung verwendeten *Node Name* als zusätzlicher Parameter ist nötig, da das Editor Modul beim Packaging[22] des Spiels nicht eingebunden wird und damit nicht zur Auswertung der Utility AI während des Spiels zur Verfügung steht. Damit keine Fehler auftreten und die Übersichtlichkeit gewährleistet ist, sollten die Namen beider Einträge übereinstimmen. Für jeden Node kann außerdem optional eine Beschreibung angegeben werden, welche die Funktion eines Nodes und der konfigurierten Parameter erläutert und damit zum besseren Verständnis der erstellten Utility AI beiträgt.

Abbildung 12: Die abgebildeten Parameter sind nicht spezifisch zum Node-Typ und müssen für jeden erstellten Node konfiguriert werden.

Ausgehend vom erstellten Root-Node kann anschließend ein Action-Node zum Graph hinzugefügt werden. Von der dunkel gefärbten Fläche unterhalb eines Nodes kann durch das Halten der linken Maustaste eine neue Verbindung erzeugt werden. Wird die Maustaste losgelassen, so erscheint der bereits erwähnte *Utility AI Node* Dialog zum Erstellen eines neuen Nodes.

[22] Beim Packaging (dt.: Verpackung) wird der Source Code eines Projekts kompiliert. Die verwendeten Bestandteile werden in ein Format konvertiert, welches von der Zielplattform genutzt werden kann. Der kompilierte Code und die Bestandteile werden anschließend in Dateien gebündelt [9].

Nach der Erstellung kann der Name der neuen Action definiert, rot als Farbe ausgewählt und als Node-Typ *Action Node* festgelegt werden. Unter der Kategorie *ActionNode* können alle relevanten Einstellungen dieses Node-Typs getroffen werden. Durch die Auswahl der Option *DisableAction* kann der Action-Node deaktiviert werden. Soll jedoch eine Gewichtung für die Action genutzt werden, sollte die Option UseWeight ausgewählt werden. Im Feld *Weight* wird der Wert der Gewichtung definiert. Dabei sind Werte zwischen 0 und 100 zulässig, wobei jedoch, durch einen Mausklick auf das Feld, der Wert ebenfalls auf eine beliebige Zahl angepasst werden kann. Ausgehend vom soeben erstellten Action-Node können der Action nun beliebig viele Considerations angehängt werden. Nach dem Erstellen eines Consideration-Nodes muss der Node-Typ *Consideration Node* für eine korrekte Auswertung im Utility AI Controller definiert werden. Im Abschnitt *Consideration Node* (siehe Abbildung 13) kann eine Vielzahl von Parametern konfiguriert werden, die entscheidend zur Bewertung der Consideration sind. Die Variable *Response Curve* benötigt ein Asset vom Typ *FloatCurve*. Das Utility AI Plugin stellt normalisierte lineare, logistische und quadratische Kurven zur Verfügung, da diese eine häufige Verwendung als Response Curve finden. Über den Content Browser können allerdings auch angepasste Response Curves erstellt werden. Mit der Option *Is Normalized Curve* kann angegeben werden, ob sich der Wertebereich der Response Curve in einem Intervall von 0 bis 1 befindet oder ob ein abweichender Wertebereich vorliegt. Diese Einstellung, auf welche in Abschnitt 2.1.2 eingegangen wird, spielt in der Berechnung des Scores eine wichtige Rolle.
Durch die nächsten beiden Parameter kann das minimale und maximale Bookend der Consideration definiert werden, wobei durch die Auswahl *Invert Score* der berechnete Score der Consideration invertiert werden kann. Definiert die Response Curve jedoch beispielsweise ein tageszeitabhängiges Verhalten, so kann der Eingabewert, den die Consideration verarbeiten soll, um einen zufälligen Wert, durch die Option *Random Shift*, verschoben werden. Die *Random Shift Range* gibt dabei den maximalen Zufallswert an. Dieser Wert wird bei jedem verwendeten Utility AI Graphen neu generiert, wodurch eine Variation der verwendeten Consideration für jede KI entsteht. Die *Random Shift Range* sollte dabei im Intervall der Response Curve liegen, damit der Eingabewert der Consideration weiterhin im Wertebereich der Kurve liegt.

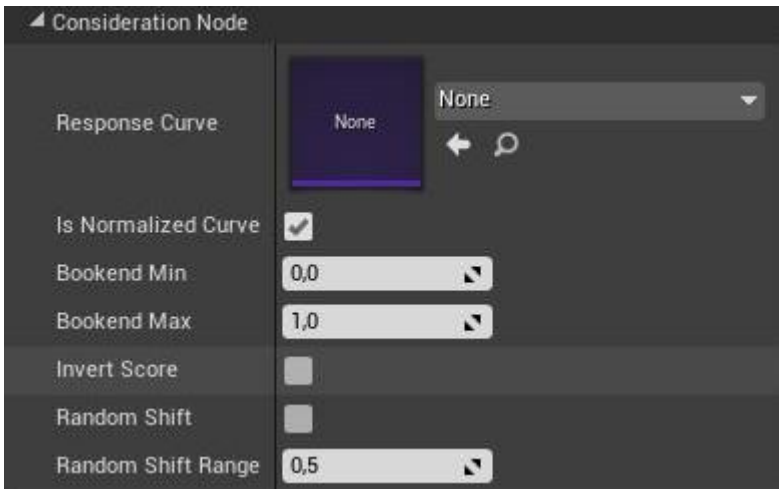

Abbildung 13: Auflistung der Parameter, welche für einen Consideration-Node definiert werden können.

Nachdem alle nötigen Considerations einer Action definiert wurden, muss vom Action-Node ausgehend ein Decision-Node angelegt werden. Im Decision-Node wird der Name der Decision und das Farbschema angepasst. Weitere Einstellungen sind in diesem Node nicht vorhanden, da nur der Name benötigt wird, um im Utility AI Controller zwischen den verschiedenen Decisions zu unterscheiden. Durch das Hinzufügen des Decision-Nodes ist die Definition eines ersten möglichen Verhaltens der Utility AI komplett. In Abbildung 14 ist die Darstellung aller benötigten Nodes im Utility AI Graph Editor abgebildet.

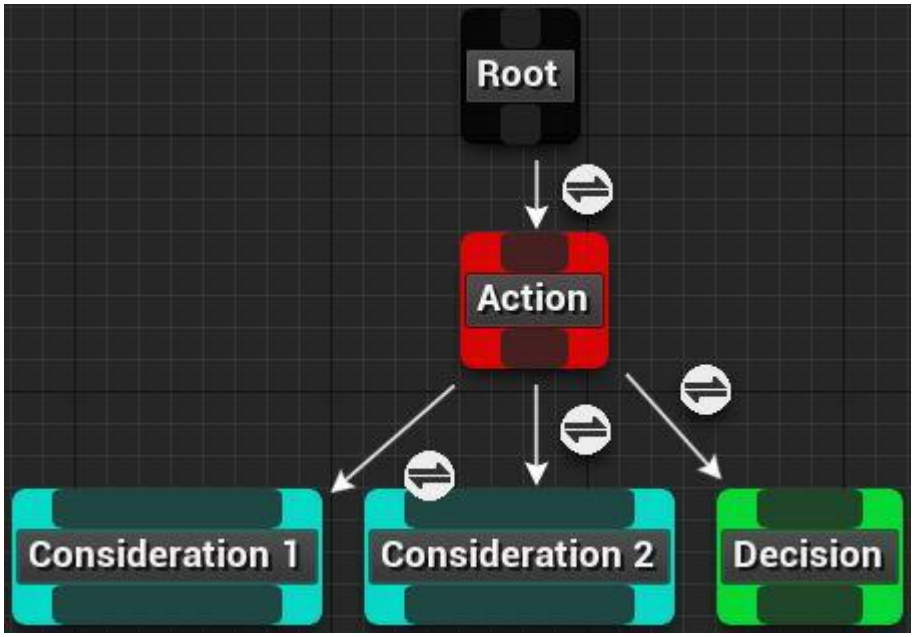

Abbildung 14: Durch Root-, Action-, Consideration- und Decision-Nodes kann die Utility AI definiert werden, wobei der Einsatz von Farben für die verschiedenen Node-Typen eine einfache Unterscheidung zulässt.

Die letzte Kategorie, welche sich im *Utility AINode* Panel befindet, ist *Debug*. Die drei nicht editierbaren Felder, zu sehen in Abbildung 15, zeigen die Werte der Action, der Consideration und des *Random Shifts* an. Je nachdem welcher Node-Typ dabei ausgewählt wurde, besitzen die Werte den aktuellen Score, welcher sich, wenn die Utility AI im Spiel ausgeführt wird, in Echtzeit aktualisiert. Ein Action-Node zeigt damit den Action Score an, während die Felder des Consideration Scores und des *Random Shifts* auf deren Standardwerten verbleiben, da der Node-Typ diese Variablen nicht verändert. In Abschnitt 3.5 wird auf weitere Möglichkeiten eingegangen, die Utility AI zu debuggen.

Abbildung 15: Die Kategorie Debug, welche die Scores der verschiedenen Node-Typen anzeigt.

Abbildung 16 zeigt den Utility AI Graph, welcher im Rahmen dieser Bachelorarbeit erstellt wurde. Der Graph enthält 11 Action- und Decision-Nodes, wobei sich 18 verschiedene Considerations auf die einzelnen Actions auswirken.

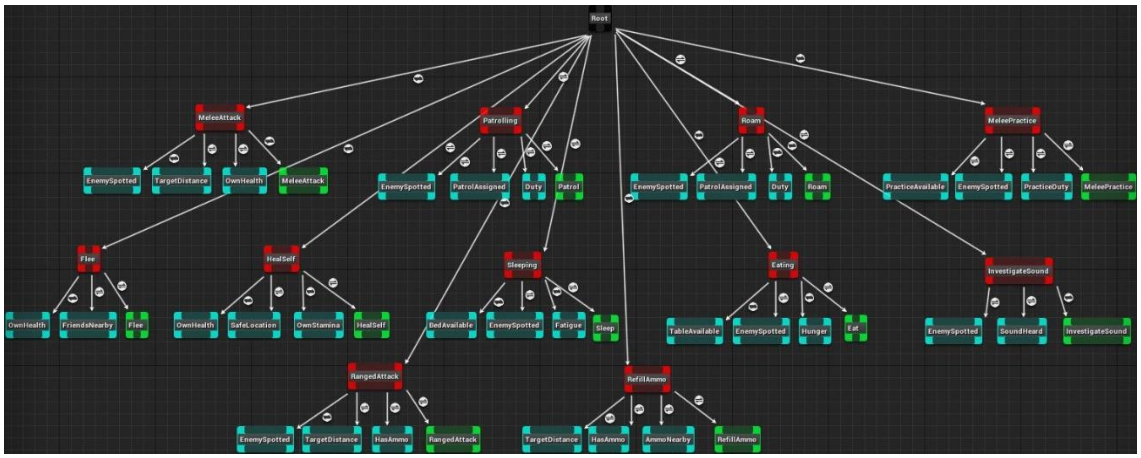

Abbildung 16: Übersicht des kompletten Utility AI Graphen, welcher im Rahmen dieser Bachelorarbeit erstellt wurde.

3.2 Utility AI Controller

Die zweite Kernkomponente des Utility AI Systems ist die Implementierung der Auswertung und das Ausführen von Verhalten. Diese Implementierung findet in einer abgeleiteten Klasse des AI Controllers der Unreal Engine 4 statt.

Der AI Controller ist darauf ausgelegt, auf Eingaben aus der Spielwelt und aus der Umgebung der KI zu reagieren. Er besitzt die Aufgabe die Spielwelt zu beobachten, Entscheidungen zu treffen und entsprechend zu reagieren. Diese Aufgaben soll der AI Controller dabei selbstständig ausführen, also ohne Eingaben des Spielers [10]. Für diese Selbständigkeit sorgen Behavior Trees[23], welche in der Engine integriert werden können. Um einen zugewiesenen Behavior Tree zu instanziieren und auszuführen, stellt der AI Controller die Funktion *RunBehaviourTree* zur Verfügung.

Um den Utility AI Graph zu initialisieren, auszuwerten und Informationen aus der Spielwelt zu erlangen und bereitzustellen, wurde eine C++ Klasse erstellt, die vom AI Controller abgeleitet ist.

Damit die benötigten Informationen der Spielwelt unkompliziert bereitgestellt werden können, wird die in C++ erstellte Klasse als Blueprint-Klasse abgeleitet. Damit besitzt diese Klasse Zugriff auf die in der Elternklasse implementierten Funktionen und kann dadurch mithilfe des Blueprint Visual Scriptings die Logik zur Berechnung der Utility AI und zur Implementierung des Verhaltens erstellen.

Der Utility AI Controller besitzt sechs grundlegende Funktionen, die zur Erstellung einer KI, welche durch das Utility AI System gesteuert wird, benötigt werden. Diese Funktionen besitzen im

[23] Dt.: Verhaltens-Bäume

Sourcecode die Namen *InitializeUtilityAI*, *RunUtilityAI*, *ImplementConsiderations*, *SetConsiderationScore*, *DecisionChanged* und *ImplementDecisions*. Der Ablauf der gesamten Auswertung und die Ausführung des Verhaltens ist im Flussdiagramm in Abbildung 17 dargestellt. Die Funktionsweise und die Abhängigkeiten des C++ Codes zur Blueprint-Implementierung wird in diesem Abschnitt ausführlicher erläutert.

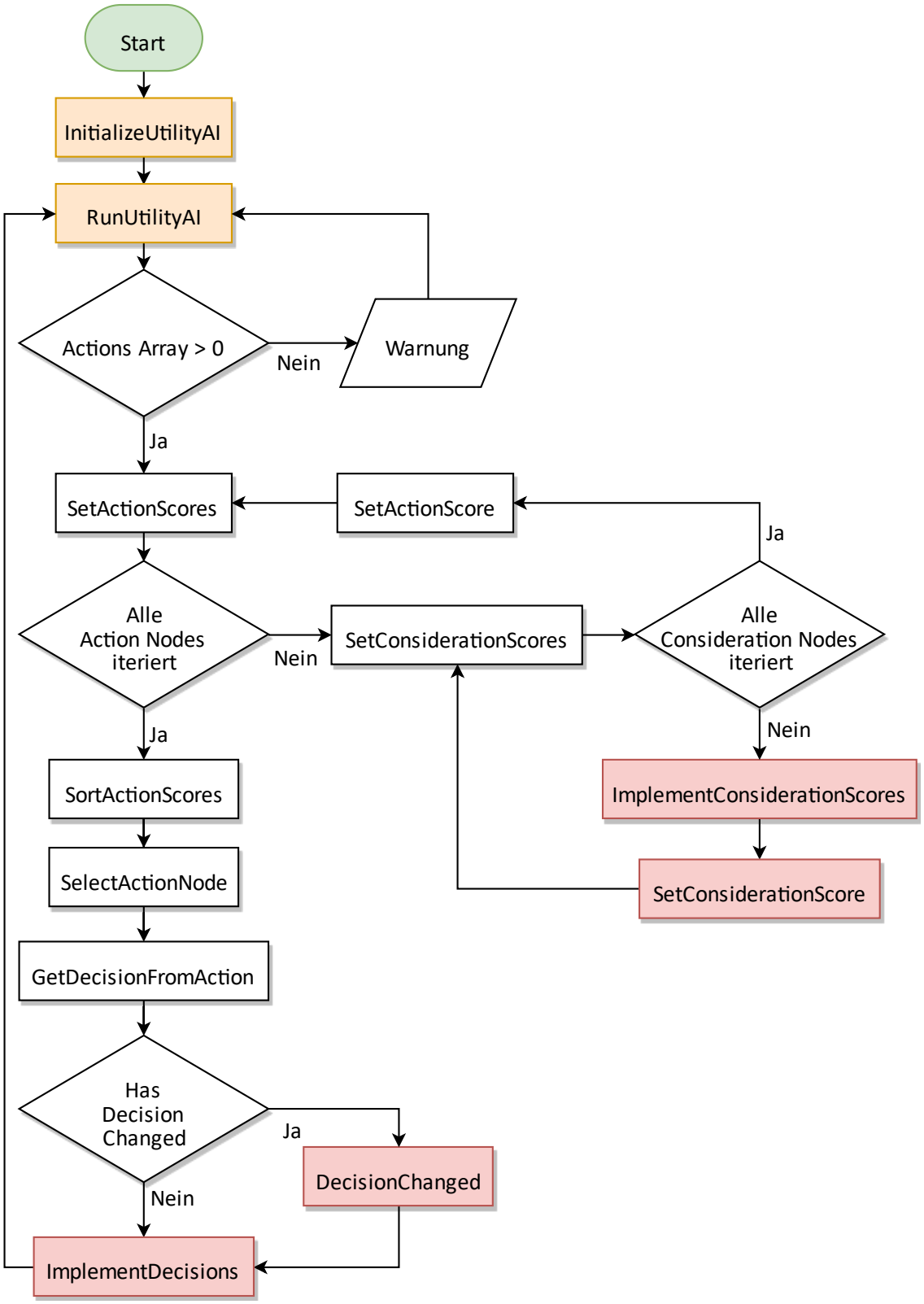

Abbildung 17: Darstellung des Flussdiagramms, welches den Ablauf der Auswertung und der Ausführung der Utility AI visualisiert. Die rot markierten Funktionen sind dabei in der Blueprint-Klasse implementiert, die orange markierten Funktionen werden durch die Blueprint-Klasse aufgerufen.

Zur Reduktion der Rechenleistung während der Laufzeit des Spiels werden durch die Funktion *InitializeUtilityAI,* ausgehend vom Root-Node des Utility AI Graphen, alle angehängten Kind-Knoten über eine Schleife in einen Array aus Action-Nodes gespeichert. Da der Graph zur Laufzeit des Spiels nicht geändert wird, kann diese Funktion durch *BeginPlay* aufgerufen werden. In der Unreal Engine wird das Event *BeginPlay* einmalig beim Start des Levels ausgeführt. Das Array aus Action-Nodes wird für verschiedene Funktionen, welche später in diesem Abschnitt beschrieben werden, benötigt. Als Argument fordert die Funktion *InitializeUtilityAI* ein Utility AI Graph Asset, welches für die Speicherung von Scores und die Definition der Action-, Consideration- und Decision-Nodes zuständig ist. Das Asset kann, wie in Abbildung 18 dargestellt ist, im Blueprint-Node über eine Dropdown-Liste ausgewählt werden. Dabei wird bereits nach dem Utility AI Asset-Typ für eine einfache Auswahl des Assets gefiltert.

Abbildung 18: Darstellung des *InitializeUtilityAI* Blueprint-Nodes, der die Action-Nodes des Utility AI Graphen speichert und ein Graph-Asset als Übergabeparameter benötigt.

Die Funktion *RunUtilityAI* ist von zentraler Bedeutung für die Ausführung der Utility AI. Nachdem der Graph durch *InitializeUtilityAI* initialisiert ist, ruft ein Timer den Blueprint-Node der Funktion *RunUtilityAI* in einem definierten Zeitintervall auf. Diese Intervalldauer entscheidet, wie oft der Utility Graph ausgewertet wird und damit auch, wie schnell die KI auf die Spielwelt reagieren kann. Das Intervall ist in der eigenen Implementierung auf 0,1 Sekunden gesetzt. Dadurch wird der Node *RunUtilityAI* und damit die Berechnung der Utility Scores nicht zu häufig im laufenden Spiel aufgerufen und die benötigte Rechenleistung wird damit reduziert. Jedoch ist die durch die gewählte Intervalldauer verursachte zeitliche Unterbrechung der Auswertung kurz genug, um eine fließende Auswahl und Ausführung des Verhaltens beizubehalten, beziehungsweise zu erzeugen. *RunUtilityAI* führt bei einem Aufruf zuerst eine Überprüfung durch, ob das Array aus Action-Nodes bereits gefüllt wurde, da ohne die Action-Nodes auch kein Verhalten ausgewählt werden kann. Liegt ein leeres Array vor, so wird ein Null-Pointer als ausgewählte Action zurückgegeben und im Output-Log der Unreal Engine darauf hingewiesen, dass die Funktion *InitializeUtilityAI* in *BeginPlay* nicht aufgerufen wurde. Anschließend wird die Funktion *SetActionScores* aufgerufen, welche für die Berechnung der Action Scores aller Action-Nodes verantwortlich ist.

Dabei wird das zuvor angelegte Array der Actions über eine Schleife iteriert und für jeden Consideration-Node, der eine Action besitzt, wird der Consideration Score berechnet. Nachdem die Action Scores berechnet wurden, werden diese der Größe nach sortiert und in das bestehende *ActionScores* Array eingespeichert. Die Selektionsmethode bestimmt daraufhin, welche der Actions ausgewählt und welches Verhalten damit ausgeführt werden soll, da jede Action eine Decision besitzt, welche das Verhalten der KI definiert.
Im nachfolgenden Schritt wird überprüft, ob sich die neu berechnete Action von der in der vorherigen Ausführung gewählten Action unterscheidet. Hat sich die ausgewählte Action geändert, so wird die Funktion *DecisionChanged* aufgerufen. Da diese mit dem Schlüsselwort *BlueprintImplementableEvent* markiert ist, wird die Funktion in der Blueprint Klasse des Utility *AI Controllers* als Event ausgelöst. Durch Blueprint-Nodes wird die Logik implementiert, die bei einer Änderung der Action ausgeführt werden muss, damit das Verhalten der KI korrekt ablaufen kann.
Die Selektionsmethode kann, wie in Abbildung 19 zu sehen, über die Option *SelectionMethod* durch eine Dropdown-Liste ausgewählt werden. Neben der Auswahl des höchsten Scores *Highest* stehen in der Liste noch die Optionen der gewichteten Auswahl W*eighted Random* und der zufälligen Wahl aus den höchsten *n* Elementen *Random Top N* zur Verfügung. Der Integer *Top N* wird in der Funktion nur verarbeitet, wenn *Random Top N* zur Entscheidungsfindung einer Action gewählt wird. Auf diese Tatsache wird durch einen Tooltip hingewiesen, wodurch der Benutzer der Nodes die C++ Implementierung nicht kennen muss.

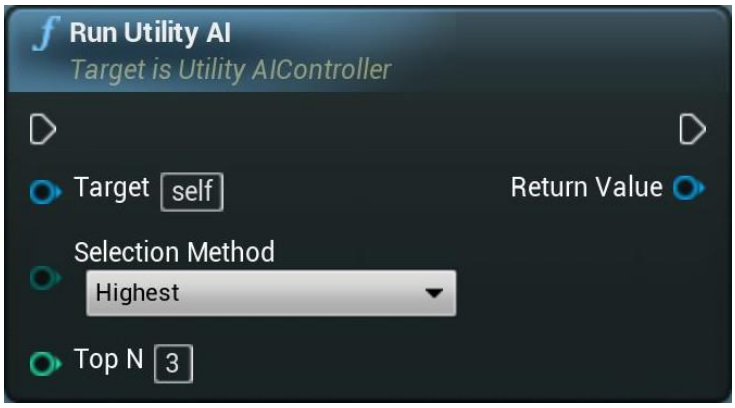

Abbildung 19: Abbildung des Blueprint-Nodes *RunUtilityAI*, welcher durch die gleichnamige Funktion in C++ generiert wird. Dadurch können Parameter zur Selektionsmethode konfiguriert werden.

RunUtilityAI gibt bei einer erfolgreichen Ausführung eine Instanz des ausgewählten Action-Nodes zurück. Dieser Rückgabewert wird während dem Vorgang des Debuggings zur Anzeige des Werts der ausgewählten Action verwendet.
Die Funktion *SetConsiderationScores* wird für jeden Action-Node im Utility Graph ausgeführt. Da eine Action mehrere Considerations haben kann, wird in dieser Funktion durch alle angehängten Considerations des aktuellen Action-Nodes iteriert und die Funktion *ImplementConsideration Scores* aufgerufen. Dabei wird der Name der aktuellen Consideration als Übergabeparameter weitergegeben. Auf diesen kann, wie in Abbildung 20 dargestellt, über den lila gefärbten Pin mit

dem Namen *ConsiderationName* zugegriffen werden. *ImplementConsiderationScores* besitzt keine native C++ Implementierung. Die Logik, welche durch den Aufruf der Funktion ausgelöst wird, ist in der abgeleiteten Blueprint Klasse erstellt, da die Funktion mit dem Schlüsselwort *BlueprintImplementableEvent* markiert ist. Über einen Switch-Node, der wie ein Switch Statement in C++ funktioniert, kann zwischen den verschiedenen Considerations unterschieden und die benötigten Informationen aus der Spielwelt abgefragt werden. Da der Utility AI Controller und der Utility AI Graph zwei voneinander unabhängige Assets sind, muss sichergestellt werden, dass die Namen im Switch-Node exakt den Namen der Consideration-Nodes im Utility AI Graph entsprechen. Wird der Name einer Consideration nicht erkannt, so wird in der Engine eine Information diesbezüglich angezeigt.

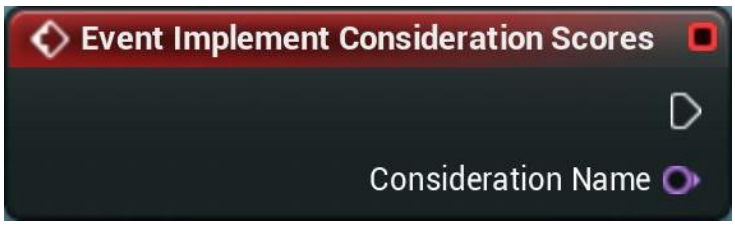

Abbildung 20: Darstellung der Funktion *ImplementConsiderationScores*, welche für jeden Consideration-Node im Utility AI Graph aufgerufen wird.

Wird der korrekte Name und damit der Consideration-Node ausgewählt, so kann der Consideration Score durch die Funktion *SetConsiderationScore* berechnet werden. Dazu muss zuerst aus der Spielwelt die benötigte Information gesammelt werden. Die Art der Informationen unterscheidet sich zum Teil stark, je nachdem welches Verhalten die Considerations beeinflussen sollen. Außerdem kann die Komplexität der Eingabewerte je nach Consideration drastisch voneinander abweichen.

Beispielsweise wird durch eine Consideration vergleichsweise niedriger Komplexität lediglich die Lebenspunkte der gesteuerten KI abgefragt. Eine Consideration relativ hoher Komplexität muss jedoch zum Beispiel die Anzahl von bestimmten Objekten im Level in Erfahrung bringen und für jedes dieser Objekte muss zusätzlich eine Entfernung zur KI berechnet werden. Im einfachsten Fall kann jedoch ein Eingabewert aus einer booleschen Variablen bezogen werden, die beispielsweise entweder 0 oder 1 ist.

Der Eingabewert wird an den grünen Pin mit dem Namen *ValueToEvaluate* (siehe Abbildung 21) weitergegeben, wodurch die in C++ implementierte Funktion daraus den Consideration Score berechnen kann.

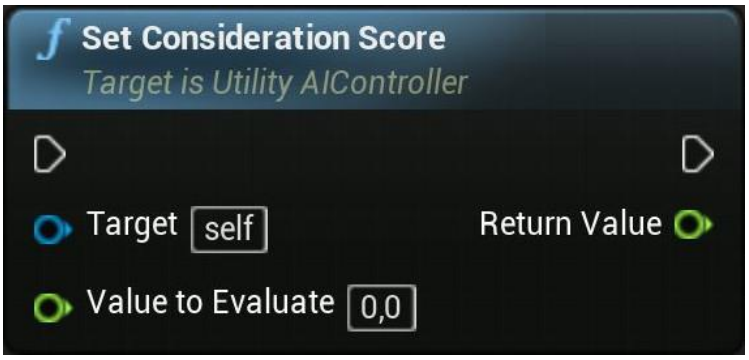

Abbildung 21: Darstellung des Blueprint-Nodes *SetConsiderationScore*, welcher für die Umwandlung von Eingaben aus der Spielwelt in einen Consideration Score verantwortlich ist.

Sobald die Funktion einen Eingabewert zugewiesen bekommt, kann sie diesen in einen Utility Score umwandeln. Die zur Berechnung notwendigen Einstellungen bezieht die Funktion aus dem aktuellen Consideration-Node, welcher ausgewertet werden soll. Zu Beginn der Funktion werden daher das maximale und das minimale Bookend und die zugewiesene Response Curve zwischengespeichert. Anschließend wird überprüft, ob der Consideration eine Response Curve zugewiesen wurde, da diese von zentraler Bedeutung während der Auswertung ist. Fehlt diese, so wird eine Warnung im Editor angezeigt und weitere Berechnungen finden mit einem Wert von 0 statt.

Wird die Option *RandomShift* im Utility AI Graph Editor ausgewählt, so wird im nächsten Schritt, unter Beachtung der angegebenen *RandomShiftRange,* eine zufällige Gleitkommazahl berechnet. Anschließend wird dieser Wert auf den übergebenen Wert *ValueToEvaluate* aufaddiert. Überschreitet diese Summe das angegebene maximale Bookend, so wird der überschreitende Wert auf das minimale Bookend addiert. Dadurch wird vermieden, dass die maximale Verschiebung auf der *x*-Achse den Wertebereich der Response Curve überschreitet. Als Beispiel kann dabei die Action *Eat*, auf welche in Kapitel 3.3.6 eingegangen wird, herangezogen werden. Dessen Consideration *Hunger* besitzt eine Response Curve, die in einem Wertebereich von 0 bis 24 das Hungergefühl definiert. Soll aus der Tageszeit von 20 Uhr das Hungergefühl berechnet werden und eine *RandomShiftRange* von 5 wurde angegeben, so kann damit eine maximale Abweichung von bis zu 1 Uhr entstehen.

Unabhängig davon, ob eine zufällige Verschiebung aufaddiert wurde, wird im nächsten Schritt zur Berechnung des Consideration Scores im aktuellen Node überprüft, ob die Response Curve normalisiert ist oder nicht. Ist diese normalisiert, so wird der Eingabewert auf das minimale und das maximale Bookend im normalisierten Wertebereich von 0 bis 1, durch die von der Unreal Engine 4 zur Verfügung gestellten Funktion *MapRangeClamped,* neu zugeordnet (siehe Abbildung 22). Durch die Response Curve kann der entsprechende Consideration Score aus dem Wert der *x*-Achse, dem *ValueToEvaluate,* abgelesen werden. Das Clamping ist notwendig, da durch das Editieren von Kurven im Curve Editor der Engine geringe negative Werte oder Werte größer als 1 vorkommen können. Wird im Utility AI Graph Editor angegeben, dass keine normalisierte Kurve verwendet wird, so wird davon ausgegangen, dass sich der *ValueToEvaluate* ebenfalls nicht im Wertebereich zwischen 0 und 1 befindet und stattdessen im Intervall der Bookends

liegt. Daher muss der Wert nicht neu zugeordnet werden und der Consideration Score kann direkt aus der Response Curve berechnet werden. Da die Unreal Engine 4 einen speziellen Editor zum Editieren von Kurven bereitstellt, wurde auf die Nutzung von mathematischen Funktionen, auf die in Abschnitt 2.2.3 eingegangen wurde, zur Definition von Response Curves verzichtet. Durch den sogenannten Curve Editor können stückweise beliebige Funktionstypen per Spline definiert werden. Dadurch muss kein geeigneter Parameter der mathematischen Gleichungen gesucht werden, stattdessen kann die Biegung einer Kurve per Hand angepasst werden. Sogenannte Keys definieren den Anfang und das Ende einer Kurve im Unreal Engine 4 Curve Editor.

```
// If the response curve is normalized...
if (CurrentConsiderationNode->bIsNormalizedCustomCurve)
{
	// ... the ValueToEvaluate can be normalized/remapped
	float NormalizedValue = UKismetMathLibrary::MapRangeClamped(
		ValueToEvaluate, MinBookend, MaxBookend, 0.f, 1.f);

	// The remapped value is then fed into the response curve
	// Clamping because some curves might accidentally return values < 0 or > 1
	CalculatedConsiderationScore = FMath::Clamp(
		ResponseCurve->GetFloatValue(NormalizedValue), 0.f, 1.f);
}
else
{
	// If the custom curve isn't normalized, the ValueToEvaluate is probably
	// also not in range 0-1, but matching the custom curves bookends
	// Therefore the value from the curve doesn't need to be normalized/
	// remapped anymore
	// Clamping because some curves might accidentally return values < 0 or > 1
	CalculatedConsiderationScore = FMath::Clamp(
		ResponseCurve->GetFloatValue(ValueToEvaluate), 0.f, 1.f);
}
```

Abbildung 22: Auswertung des Consideration Scores durch die Response Curve.

Für den Fall, dass im Utility AI Graph Editor die Option *InvertScore* ausgewählt wurde, kann der berechnete Score invertiert werden. Da der Score normalisiert ist, kann der invertierte Score durch die Subtraktion von 1 berechnet werden. Anschließend wird der berechnete Score im Node selbst gespeichert, damit dieser im Graph Editor für jeden Consideration-Node betrachtet werden kann und somit eine Möglichkeit bietet, die Utility AI zu Debuggen. Auf die Debugging-Möglichkeiten wird in Abschnitt 3.5 näher eingegangen. Das *ConsiderationScores* Array speichert den berechneten Score ab, da dieser für die Berechnung des Action Scores benötigt wird. Im letzten Schritt gibt *SetConsiderationScore* den zuvor berechneten Consideration Score als Rückgabewert der Funktion zurück, welcher ebenfalls für das Debugging genutzt wird.
Sind alle Consideration Scores einer Action berechnet, so kann schließlich der Action Score selbst durch die Funktion *SetActionScore* festgelegt werden. Dieser wird berechnet, indem das Array aus Consideration Scores, welches durch die Funktion *SetConsiderationScore* gefüllt wurde, iteriert wird und jeder eingespeicherte Wert dabei auf das Produkt der vorherigen Iteration multipliziert wird. Sind alle Consideration Scores multipliziert, wird durch den aktuellen Action-Node in Erfahrung gebracht, ob dieser durch Auswählen von *UseWeight* im Utility AI Graph Editor eine Gewichtung besitzt und mit welchem Wert diese Gewichtung konfiguriert wurde. Soll der Action

Score eine Gewichtung nutzen, so wird diese auf das Produkt des im vorherigen Schritt berechneten Wertes multipliziert. Da diese Action beispielsweise zum Debuggen deaktiviert werden kann, wird im nächsten Schritt überprüft, ob diese Option beim aktuellen Action-Node ausgewählt wurde. Zur Deaktivierung der Action kann der bisher berechnete Score auf 0 gesetzt werden und wird damit nicht in der Auswahl der Actions miteinbezogen. Um den berechneten Score für die spätere Selektion und für das Debugging verwenden zu können, wird dieser im Node selbst gespeichert. Damit kann auf den Score jeder Action zugegriffen und über den Utility AI Graph Editor eingesehen werden. Im letzten Schritt der Funktion *SetActionScore* wird der Array aus Consideration Scores geleert, damit die Berechnung der nächsten Action nicht von anderen Considerations beeinflusst wird. Wurden nach dem Abschluss der Schleife in der Funktion *SetActionScores* alle Actions berechnet, so müssen diese erst sortiert werden, damit die Selektion einer Action ausgeführt werden kann. Die Sortierung der Scores übernimmt dabei die Funktion *SortActionScores*. Zuerst wird sichergestellt, dass das Array aus Action Scores geleert ist, damit keine Scores aus einem vorherigen Durchlauf in der Sortierung berücksichtigt werden. Anschließend wird das Array mit den zuvor im Action-Node gespeicherten Scores gefüllt. Das Array wird dann in absteigender Reihenfolge der Werte, der höchste Score dabei an erster Stelle, sortiert. In *RunUtilityAI* kann nun eine Auswahl der Action und damit das Verhalten der KI getroffen werden. Die Funktion *SelectActionNode* unterscheidet zunächst zwischen den verschiedenen Selektionsmethoden. Wurde *Highest* als Selektionsmethode ausgewählt, so kann das erste Element aus dem Action Scores Array ausgewählt werden. Bei der Auswahl des Action Scores durch *Weighted Random* wird der Score als eine Gewichtung zur Auswahl eines zufälligen Elements miteinbezogen. Umso höher der Score der Action dabei ist, desto größer ist die Wahrscheinlichkeit, dass die Action durch den Algorithmus ausgewählt wird.

Abbildung 23 zeigt den *Weighted Random* Algorithmus. Dabei werden zuerst die Scores aller Actions addiert. Anschließend wird eine Zufallszahl im Bereich von 0 bis 1 generiert und mit der Summe der Actions Scores multipliziert. Dadurch kann der maximale Zufallswert der Summe der Action Scores entsprechen. Durch eine Schleife werden anschließend die einzelnen Action Scores durch die Summe aller Action Scores geteilt. Das Ergebnis wird in der Variablen *Accumulation* gespeichert. In jedem Schleifendurchlauf wird geprüft, ob die generierte Zufallszahl kleiner als der, in *Accumulation* gespeicherte Wert, ist. Trifft diese Bedingung zu, so wird der aktuell evaluierte Action Score ausgewählt und die Schleife abgebrochen.

```
float Total = 0.f;
for (auto ActionNode : ActionNodes)
{
	Total += ActionNode->ActionScore;
}

float Choice = (FMath::Rand() * (1.f / RAND_MAX)) * Total;
float Accumulation = 0.f;

for (auto ActionScore : ActionScores)
{
	Accumulation += ActionScore / Total;

	if (Choice < Accumulation)
	{
		ScoreToFind = ActionScore;
		break;
	}
}
```

Abbildung 23: Darstellung des *Weighted Random* Algorithmus, welcher einen Action Score nach Gewichtung zufällig auswählt.

Anhand Abbildung 24 soll der Algorithmus durch ein Beispiel genauer erklärt werden. In diesem Beispiel wurde angenommen, dass die generierte Zufallszahl Choice einen Wert von 0,6 besitzt. Die drei Actions Essen, Training und Schlafen sind in dieser Reihenfolge im Action Score Array gespeichert und besitzen die Action Scores von 0,7, 0,4 und 0,2. Durch die Addition der Scores ergibt sich eine Summe von 1,3. Im ersten Durchlauf der Schleife wird nun der Score der Action Essen durch diese Summe geteilt, wodurch der Quotient von 0,5 berechnet wird. Da dieser Wert kleiner als der Zufallswert ist, wird der nächste Schleifendurchlauf gestartet. In diesem wird nun der Action Score des zweiten Array Elements, des Trainings, durch die Summe der Scores geteilt und auf den bestehenden Wert der *Accumulation* Variable addiert. Da dieser mit 0,8 größer als der Zufallswert ist, wird die aktuelle Action Training ausgewählt und der Algorithmus beendet.

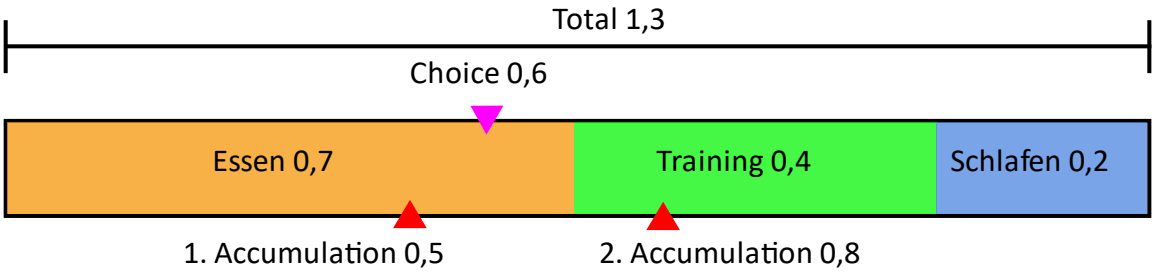

Abbildung 24: Beispielhafte Scores drei verschiedener Actions, welche im Algorithmus als Gewichtung genutzt werden.

Der beschriebene Algorithmus wurde von David Graham zur Demonstration des Utility AI Systems implementiert und in der eigenen Umsetzung geringfügig angepasst, damit der Algorithmus in der Unreal Engine 4 ausgeführt werden kann [11].
Bei der letzten Auswahlmethode, *Random Top N*, wird eine Zufallszahl mit einem maximalen Wert der angegebenen Reichweite generiert (siehe Abbildung 25). Die generierte Zufallszahl

stellt dabei den Index des Action Score Arrays dar, aus dem das entsprechende Element ausgewählt wird. Sollte der Nutzer durch *TopN* eine größere Reichweite angegeben haben, als Actions existieren, so wird das höchste Element ausgewählt und eine Warnung angezeigt.

```
// Get a random action in the defined range
int32 RandomIndex = FMath::RandRange(0, TopN - 1);
if (RandomIndex < ActionScores.Num())
{
	ScoreToFind = ActionScores[RandomIndex];
}
else
{
	// If there are no actions available,
	// just return the first (highest scoring) element
	ScoreToFind = ActionScores[0];
	UE_LOG(LogTemp, Warning, TEXT("TopN higher than available actions!"));
}
```

Abbildung 25: Auswahl einer zufälligen Action in definierter Reichweite des Action Arrays.

Durch den ausgewählten Action Score kann nun die zugehörige Action gefunden werden. Die Funktion *GetDecisionFromAction* liefert eine Instanz des Decision-Nodes, welcher dem ausgewählten Action-Node zugehörig ist.
Um angemessen auf eine Verhaltensänderung reagieren zu können, wird überprüft, ob sich die ausgewählte Decision seit der letzten Ausführung von *RunUtilityAI* geändert hat. Wurde eine neue Auswahl getroffen, so wird die Funktion *DecisionChanged* (siehe Abbildung 26) aufgerufen, welche in der Blueprint Klasse implementiert werden muss. Die Funktion übergibt dabei eine Instanz des neu selektierten Decision-Nodes.

Abbildung 26: Darstellung des Blueprint-Nodes, welcher im AI Controller aufgerufen wird, wenn sich die ausgewählte Action ändert.

Der Aufruf der Funktion *ImplementDecisions*, durch welche im AI Controller das Verhalten der KI definiert und ausgeführt wird, erfolgt. Da der Name der Decision der Funktion übergeben wird (siehe Abbildung 27), kann durch einen Switch-Node zwischen den Namen der verschiedenen Decisions unterschieden und das korrekte Verhalten ausgewählt werden.

Abbildung 27: Darstellung der Funktion *ImplementDecisions*, welche aufgerufen, wenn die zur Action zugehörige Decision ausgeführt werden soll.

3.3 Verhalten der KI

Vorteilhaft an der Utility AI ist die Auswahl und Abwägung vieler verschiedener Verhalten. Daher werden im Rahmen dieser Bachelorarbeit möglichst viele verschiedene Verhaltensabläufe implementiert. Aufgrund der Zielsetzung dieser Arbeit, den Tagesablauf eines Kämpfers zu realisieren, sind neben passivem Verhalten ebenfalls offensive Handlungsmöglichkeiten erstellt.
In diesem Abschnitt soll nun der Ablauf der einzelnen Verhalten beschrieben werden. Außerdem wird die Definition und Parametrisierung von Action- und Consideration-Nodes im Utility AI Graph beschrieben.

3.3.1 Schwertkampf

Der Schwertkampf ist das erste offensive Verhalten der KI. Wird dieses Verhalten der KI als geeignete Action im aktuellen Kontext ausgewählt, so rüstet die KI ihr Schwert und Schild aus und läuft auf den Spieler zu, um diesen anzugreifen. Ist der Spieler in Reichweite, so wird der Nahkampfangriff ausgeführt. Die KI schlägt dabei mit dem Schwert auf den Spieler ein. Dadurch verliert dieser Lebenspunkte, das heißt der Spieler wird verletzt. Damit der Spieler eine Chance hat, den Schlägen auszuweichen und die KI dem Spieler nicht zu schnell Schaden zufügen kann, wird der Nahkampfangriff alle 1,5 Sekunden ausgeführt.
Im Utility AI Graph wird dieser Verhaltensablauf durch die Action *MeleeAttack* definiert. Sie besteht aus drei verschiedenen Considerations.
Die erste Überlegung, die zur Verhaltensfindung genutzt wird, bestimmt, ob die KI einen Gegner im Perception System registriert hat, also ob sie den Spieler gesichtet hat. Diese Überlegung dient dem *MeleeAttack* Trigger, wie in Abschnitt 2.2 beschrieben wurde, da entweder ein Score von 0 oder 1 vergeben wird. Dadurch soll vermieden werden, dass diese Action überhaupt in die Liste der möglichen Actions aufgenommen wird, wenn überhaupt kein Gegner von der KI erkannt wurde. Durch den Boolean Datentyp kann entschieden werden, ob die KI momentan den Spieler gesichtet hat oder nicht. Dadurch kann der entsprechende Score von 0, das heißt kein Spieler wurde gesichtet, oder 1, das heißt der Spieler wurde gesichtet, gesetzt werden.
Genutzt wird eine lineare Response Curve, da der Wert dieser Kurve bei einem *x*-Achsenwert von 0 auch garantiert einen Utility Score von 0 zurückgibt und analog auch bei einem *x*-Achsenwert von 1 auch einen Score von 1 liefert. Diese Consideration wird im Graph als *EnemySpotted* bezeichnet.

Die zweite Überlegung ermittelt die aktuellen Lebenspunkte der KI als Kontext. Im Graphen wird diese Consideration *OwnHealth* genannt. Als Response Kurve wird eine logistische Kurve ausgewählt, welche die Lebenspunkte der KI bewertet. Diese Consideration führt dazu, dass die KI bei wenigen Lebenspunkten einen Nahkampfangriff überdenkt, da dieser unter Umständen zu viel Schaden in kurzer Zeit und damit zum Tod der KI führen kann.
Eine logistische Funktion bietet sich für die Berechnung des Utility Scores an, da bei mehr als 50 Lebenspunkten der Utility Score hoch, mit einem Wert größer 0,5, ausfällt. Besitzt die KI jedoch weniger als die Hälfte der Lebenspunkte, so wird ein geringer Score berechnet, der durch die logistische Kurve schnell auf 0 abfällt. Da jeder Charakter maximal 100 Lebenspunkte besitzen kann und bei 0 Lebenspunkten stirbt, werden die Bookends dieser Response Curve auf 0 des minimalen und auf 100 des maximalen Bookends gesetzt.
Die letzte Überlegung, die im Utility AI Graphen zur Bewertung dieses Verhaltens getroffen wird, berechnet die Distanz der KI zum Spieler. Für diese Consideration wird eine spezielle Response Curve angelegt (siehe Abbildung 28), die zur Bewertung der Entfernung verwendet wird. Sie ist normalisiert, wodurch die Entfernung in Prozent bewertet wird. Außerdem ist diese unabhängig von den Werten der Bookends, welche unter Umständen vom Ersteller der KI oft geändert werden, um diese Überlegung anzupassen. Das minimale Bookend ist definiert als 0 und das maximale Bookend als 1200. Bei einer niedrigen Entfernung soll diese Überlegung einen hohen Utility Score erhalten. Sobald jedoch mehr als 25 % der Entfernung überschritten werden, nimmt die Bewertung des Scores rapide ab. Bei einer Distanz von mehr als 1200 Einheiten wird der Utility Score geclamped und somit immer 0 sein. Diese zentrale Überlegung ist verantwortlich dafür, ob die KI im Rahmen der Action in den Schwertkampf- oder in den Fernkampfmodus wechselt.
Eine weitere mögliche Consideration ist die Bewertung der Lebenspunkte des Spielers. Sind diese niedrig, besteht die Möglichkeit, dass der Utility Score höher ausfällt und die KI das Risiko eingeht, in den Nahkampf zu wechseln, um dem Spieler in kurzer Zeit viel Schaden zuzufügen und einen möglichen Todesstoß zu versetzen.
Außerdem könnte die KI abwägen, wie viele andere KI Charaktere sich bereits im Fernkampfmodus befinden. Schießen beispielsweise bereits mehrere KIs auf den Spieler, so sollte die KI in Erwägung ziehen, in den Nahkampfmodus überzugehen, damit nicht zu viele Pfeile auf einmal auf den Spieler geschossen werden und der Kampfmodus interessanter gestaltet wird.
Eine weitere mögliche Überlegung bestünde darin, den Ablauf des Verhaltens von der Anzahl der sich in diesem Verhalten befindlichen KIs abhängig zu machen. In dem in diesem Abschnitt beschriebenen Ablauf wird nicht beachtet, wie viele Gegner den Spieler auf einmal angreifen. Unter Umständen führt dies dazu, dass alle KIs auf einmal zuschlagen und der Spieler damit sofort stirbt. Dieser Aspekt sollte ebenfalls beachtet und der Nahkampf nur ausgeführt werden, wenn keine andere KI mit ihrem Schwert zuschlägt.

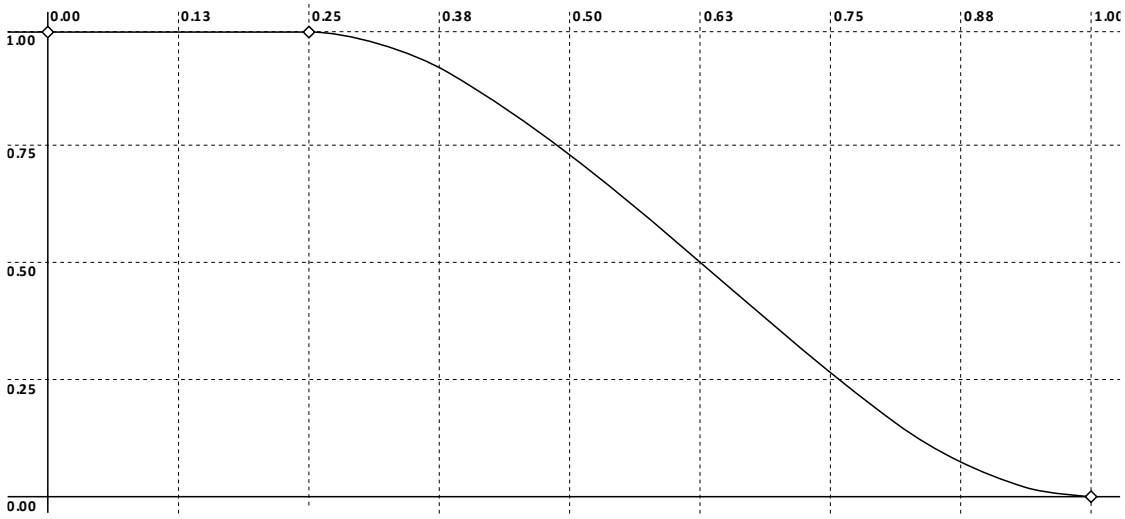

Abbildung 28: Kurve zum Scoring der Entfernung zwischen KI und Spieler. Durch das Zuordnen der Bookends auf das Intervall von 0 bis 1 wird, für bis zu 25 % der maximalen Entfernung, ein hoher Score vergeben.

3.3.2 Fernkampf

Das zweite offensive Verhalten der KI ist der Fernkampf mit einem Pfeil und Bogen. Wird dieses Verhalten durch eine der Selektionsmethoden ausgewählt, nimmt die KI den Bogen in die Hand. Ist die KI dabei weiter entfernt als eine definierte Distanz, so bewegt sie sich auf den Spieler zu und richtet sich dann zum Spieler aus.

Ist die KI in Reichweite, fängt sie an mit dem Bogen zu zielen. Bevor sie schießt, wartet die KI eine kurze Zeit, um den Schuss aufzuladen und dem Spieler damit mehr Schaden zuzufügen. Anschließend schießt die KI einen Pfeil auf den Spieler. Durch die vorherige Drehung der KI, fliegt der Pfeil in die Richtung des Spielers. Da die Ausrichtung jedoch nicht perfekt zur Position des Spielers angepasst wird, fliegt der abgeschossene Pfeil in manchen Fällen am Spieler vorbei.

Bis zum nächsten Abschuss eines Pfeils wartet die KI erneut eine gewisse Zeit, damit der Spieler nicht zu häufig getroffen wird und sich der KI annähern kann, ohne durch die Fernkampfangriffe der KI zu sterben.

Um dieses Verhalten, welches im Utility Graph als *RangedAttack* bezeichnet wird, in Betracht zu ziehen, muss die KI, ähnlich zum Schwertkampf, zunächst einen Gegner gesehen haben, bevor sie den Bogen ausrüstet und auf den Spieler schießt. Daher ist die Consideration *TargetSpotted* analog zur Consideration, welche der Action MeleeAttack anhängt, konfiguriert.

Die Anzahl der bereits verschossenen Pfeile und deren verbleibende Anzahl spielt bei der Auswahl des Verhaltens eine große Rolle. Sollten der KI keine Pfeile mehr zur Verfügung stehen und diese damit auch nicht mehr auf den Spieler schießen können, so sollte keinesfalls in den Fernkampfmodus gewechselt werden. Diese Consideration wird im Graphen als *HasAmmo* bezeichnet und besitzt eine spezielle Response Curve. Der Wertebereich der Kurve ist auf die maximale Anzahl der Pfeile angepasst, die ein Charakter im Inventar besitzen kann. Durch das Anpassen der Kurve kann der Utility Score bei wenigen Pfeilen sehr genau definiert werden. Da die KI erst beim Aufbrauchen der letzten fünf Pfeile in Betracht ziehen sollte, auf den Fernkampfmodus zu

verzichten, kann eine logistische Kurve dafür verwendet werden, diesen schnellen Abbau des Scores nachzubilden. Bei mehr als fünf Pfeilen im Inventar, betrachtet die KI ihre Anzahl an Pfeilen als ausreichend. Daher wird bei 5 bis 20 Pfeilen der Utility Score von 1 vergeben.
Die letzte getroffene Consideration, die auch im Nahkampfangriff als Überlegung in die Verhaltensfindung mit einfließt, ist die Distanz von der KI zum Spieler, die sogenannte *TargetDistance*. Die Consideration nutzt eine normalisierte Response Curve, wobei im Gegensatz zum Nahkampfangiff ein niedriger Score für nahe Entfernungen und ein hoher Score für weitere Entfernungen vergeben wird. Durch eine logistische Funktion steigt der Score bis zu 80 % der maximalen Entfernung an. Danach liegt der Utility Score bei einem Wert von 1. Das maximale Bookend ist, wie beim Nahkampfangriff, auf 1200 Einheiten festgelegt.
Eine weitere mögliche Consideration ist das mMteinbeziehen der sich bereits im Nahkampf befindlichen KIs. Damit der Spieler nicht vollständig von Gegnern umringt ist und damit eine Flucht aus dem Kreis von KIs nicht mehr möglich ist, könnte die Anzahl der KIs in den verschiedenen Kampfmodi durch eine solche Überlegung ausgeglichener verteilt werden.

3.3.3 Pfeile auffüllen

Das Fernkampfverhalten schließt bereits die Anzahl der zur Verfügung stehenden Pfeilen mit ein. Jedoch sollte die KI nach dem Verbrauch aller Pfeile in der Lage sein, ihren Vorrat wieder aufzufüllen, damit das Verhalten des Fernkampfangriffs nicht wegfällt.
Wird das Verhalten ausgeführt, so sucht die KI nach den im Level platzierten Köchern, also nach Orten, an denen sie den Pfeilvorrat auffüllen kann.
Im Utility Graph wird diese Action als *RefillAmmo* bezeichnet. Zur Auswahl werden drei Considerations mit einbezogen.
Bei diesem Verhalten wird zunächst die Entfernung zum Spieler in Betracht gezogen. Durch die *TargetDistance* wird sichergestellt, dass diese Action nicht ausgewählt wird, falls sich der Spieler in der Nähe der KI befinden sollte. Diese Consideration dient dazu, dass sich die KI nicht zu einem anderen Ort bewegt, wenn der Spieler in der Nähe ist und die KI den Schwertkampf bevorzugen sollte.
Die nächste Überlegung, *AmmoNearby* iteriert über eine Schleife sämtliche Ressourcen im Level, die der KI einen Pfeilnachschub bieten können. Dabei wird der Köcher ausgewählt, welcher die geringste Distanz zur KI besitzt. Diese Distanz wird über eine normalisierte und lineare Funktion bewertet, damit bei einer größeren Entfernung ein geringerer Utility Score berechnet wird. Das maximale Bookend von 6000 Einheiten definiert dabei die maximale Entfernung, bevor alle Köcher mit höherer Distanz auf einen Wert von 0 geclamped werden.
Da dieses Verhalten ausgeführt werden soll, wenn die KI ihren Pfeilvorrat fast oder vollständig aufgebraucht hat, muss durch die Überlegung *HasAmmo* ein entsprechender Utility Score berechnet werden. Die Parametrisierung dieses Utility Scores erfolgt entsprechend der des Fernkampfmodus. Damit ein hoher Score, der zur Auswahl dieses Verhaltens führen soll, gebildet werden kann, wird der Score invertiert. Dadurch geht der Score bei einer geringen Anzahl an

Pfeilen gegen 1. Bei mehr als 5 vorhandenen Pfeilen liegt die Bewertung der Consideration dagegen bei 0.

3.3.4 Patrouillieren

Das Verhalten, welches die KI für gewöhnlich den größten Teil des Tages ausführt, ist das Patrouillieren. Der KI kann dabei über den Editor eine zuvor angelegte Route zugewiesen werden. Diese Route wird über eine Spline repräsentiert und durch einen Level Designer verlegt. Die Punkte der Spline bilden dabei die Wegabschnitte im Level. Bei einer Ausführung des Verhaltens folgt die KI diesen Wegpunkten. Ist sie am letzten Splinepunkt angekommen, wird die Route von deren Startpunkt erneut abgelaufen. Wird ein anderes Verhalten, wie zum Beispiel der Fernkampf ausgewählt und zu einem späteren Zeitpunkt wieder beendet, so wird der zuletzt genutzte Wegpunkt wieder aufgesucht, auch wenn andere Wegpunkte näher an der Position der KI liegen.
Im Graphen wird dieses Verhalten durch die Action *Patrolling* definiert und besitzt drei verschiedene Considerations.
Hat die KI einen Gegner gesichtet, sollte sie durch die Consideration *TargetSpotted* nicht darüber nachdenken, weitere Wegpunkte abzulaufen und stattdessen den Spieler angreifen. Da deshalb der Action Score auf 0 gesetzt werden soll, besitzt diese Consideration eine lineare Kurve, die über die *InvertScore* Checkbox invertiert wurde.
Je nachdem, ob der KI eine Route zugewiesen wurde, vergibt die Consideration *PatrolAssigned* einen Score von 0 oder 1. Wurde keine solche Route zugewiesen, so kann die KI keinen Wegpunkten folgen. Somit wird der Utility Score auf 0 gesetzt, um eine Auswahl der Action zu vermeiden. Wie bei allen anderen Considerations, welche eine Trigger-Funktionalität besitzen, hat auch diese eine lineare Response Curve, um bei den Werten von 0 und 1 entlang der *x*-Achse den entsprechenden Utility Score zu erhalten.
Die letzte Consideration, welche die KI trifft, ist das Pflichtgefühl eine Aufgabe auszuführen, im Utility Graphen als *Duty* bezeichnet. Der Score ist dabei konstant definiert und wird nicht durch die Spielwelt beeinflusst. Durch den Einsatz einer normalisierten linearen Kurve kann der konstante Wert direkt in einen Utility Score von 0,8 umgewandelt werden.
Durch den Einsatz eines konstanten Scores, der bei einem vergleichsweise hohen Wert liegt, kann eine Schwelle erzeugt werden, welche andere Verhalten, wie beispielsweise Essen oder Schlafen erst überschreiten müssen, bevor sie ausgewählt werden. Diese Schwelle funktioniert jedoch nur mit der Selektionsmethode des höchsten Scores. Bei der Auswahl nach *Weighted Random* oder *Random Top N* wird, durch die Art der Selektion, unter Umständen auch ein Verhalten mit niedrigerem Score ausgewählt.

3.3.5 Umherwandern

Das Verhalten des Umherwanderns besitzt ähnliche Überlegungen wie das Verhalten des Patrouillierens. Anstatt jedoch Wegpunkten nachzulaufen, wird bei der Ausführung des Verhaltens ein zufälliger Punkt im Level ausgewählt und zu dieser Position gelaufen. Der Punkt wird dabei stets innerhalb einer bestimmten Reichweite gesucht, damit die KI sich nicht zu weit über

das Level bewegt. Das Verhalten dient dazu, ungeahnte Laufwege der KI zu erzeugen. Anstatt lediglich eine statische Route abzulaufen, wird dadurch Abwechslung und ein interessanter Spielablauf erzeugt.
Der Action Name des Verhaltens im Graph ist *Roam*. Bei dieser Action wird erneut durch die Consideration *EnemySpotted* die Auswahl der Action ausgeschlossen, sollte die KI einen Gegner gesehen haben.
PatrolAssigned ist parametrisiert, wie im Verhalten des Patrouillierens. Wird der KI jedoch keine Route über den Editor zugewiesen, so soll diese Action bevorzugt werden. Der Score der Consideration wird daher invertiert.
Dieses Verhalten besitzt ebenfalls den konstanten Duty-Score, da dieses eine Alternative zum Patrouillieren darstellt und den Großteil des Tages ausgeführt werden soll.

3.3.6 Essen

Da der Tagesablauf eines Kämpfers erstellt werden soll, ist das Verhalten Essen wichtig für eine überzeugende Darstellung der Intelligenz einer KI.
Wird durch eine der Selektionsmethoden das Verhalten ausgewählt, so wird ein freier Tisch im Level ausgesucht und belegt. Sobald die KI am Essplatz angekommen ist, wird eine Animation abgespielt, bei welcher der Charakter trinkt, um die Auswahl des Verhaltens zu veranschaulichen.
Die Action *Eating* besitzt im Graphen die drei Considerations *EnemySpotted*, *TableAvailable* und *Hunger* und führt bei der Auswahl der Action die Decision mit dem Namen *Eat* aus.
Die erste Consideration dient als Trigger zum Ausschluss des Verhaltens, wenn sich der Spieler im Sichtfeld der KI befindet.
Da die maximale Anzahl der Tische im Level begrenzt ist, also unter Umständen weniger Tische zur Verfügung stehen als KI Charaktere im Level aktiv sind, wird durch die Consideration *TableAvailable* überprüft, ob diese Ressource zur Verfügung steht. Sind ausreichend Tische vorhanden, so wird durch eine lineare Response Curve der Utility Score von 1 vergeben. Sind alle Tische bereits besetzt, wird die Action durch einen Utility Score von 0 ausgeschlossen.
Hunger wurde eine speziell angepasste Response Curve zugewiesen, welche das Hungerbedürfnis über den Tag hinweg abbildet. Die in Abbildung 29 dargestellte Kurve zeigt einen logistischen Anstieg des Hungergefühls bis 8 Uhr. Um diese Uhrzeit erreicht der Consideration Score den Wert 1. Nachdem der Utility Score circa 40 Minuten auf diesem Niveau gehalten wird, sinkt der Wert anschließend abrupt auf einen niedrigen Wert von 0,3 ab. Dies soll die Sättigung der KI nach dem Ausführen des Verhaltens abbilden. Ein zweiter Utility Score von 1 wird um 18 Uhr erreicht. Über Nacht fällt das Hungergefühl der KI auf einen Wert von 0, bevor dieses am nächsten Tag erneut ansteigt.
Damit nicht alle KIs gleichzeitig zum Esstisch laufen und ihre Wachposten aufgeben, wird für dieses Verhalten die Option des *RandomShifts* eingeführt. Dadurch können alle berechneten Utility Scores der *Hunger* Response Curve entlang der *x*-Achse verschoben werden, sodass nicht

mehr alle KIs auf einmal um 8 Uhr ein Hungergefühl vom Wert 1 verspüren. Die Option *RandomShift* wird daher im Utility AI Graph Editor ausgewählt und die *RandomShiftRange* wird auf 4 Stunden gesetzt, damit die Abweichung von der typischen realen Essenszeit gering ausfällt.
Ein anderer Ansatz zur Darstellung des Hungergefühls ist die Einführung eines internen Hunger-Bedürfnisses durch einen Wertebereich von 0 bis 1. Dieses könnte bei jeder KI mit einem anderen Wert initialisiert werden. Erreicht dieses Hungerbedürfnis den Wert von 1, so könnte ein entsprechender Utility Score ebenfalls diesen Score erhalten und damit zu einer Auswahl des Verhaltens führen. Durch den Einsatz einer speziellen Response Curve, wie in Abbildung 29 dargestellt, kann jedoch ein genauer Zeitpunkt für das Hungergefühl definiert und editiert werden. Dass der Unreal Engine 4 Curve Editor diese Möglichkeit besitzt, ist eine seiner Stärken. Durch den Einsatz von Keys zur Definition der Punkte im Graph kann eine präzise Darstellung des Hungergefühls erzeugt werden. Diese Kurve durch eine mathematische Funktion zu beschreiben ist nur bedingt möglich und nicht intuitiv.

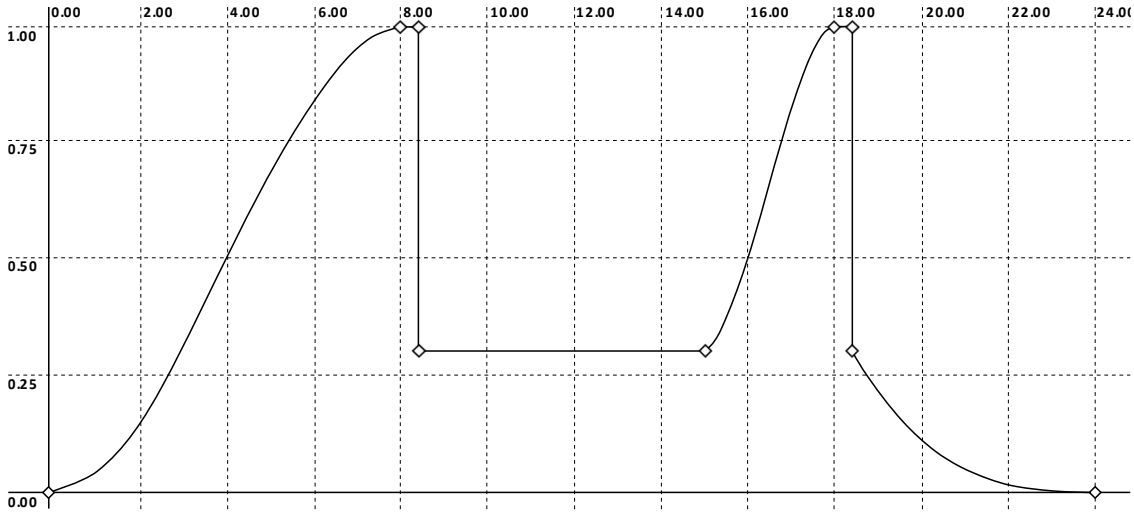

Abbildung 29: Das Hungergefühl steigt um die Frühstücks- und Abendessenszeit an, sinkt nachts dagegen stark ab und befindet sich über den Mittag auf einem niedrigen Wert.

3.3.7 Schlafen

Neben dem Essen ist das Schlafen ebenfalls ein Grundbedürfnis, weshalb dieses im Tagesablauf der KI berücksichtigt ist. Bei der Auswahl dieses Verhaltens sucht die KI nach den Positionen der Betten im Level, läuft zu diesen und legt sich für eine gewisse Zeit schlafen. Während die KI schläft wird die Sight-Perception[24] des Charakters deaktiviert, die KI kann den Spieler also nicht mehr wahrnehmen.
Im Utility AI Graph wird das Verhalten durch die Action *Sleeping* definiert und besitzt, wie das Verhalten Essen, drei Considerations. Bei der Wahl des Verhaltens wird die Decision mit dem Namen *Sleep* ausgelöst.

[24] Dt.: Sichtwahrnehmung. Befindet sich ein Objekt in einem definierten Sicht-Winkel der KI, so wird dieses Objekt erkannt und einer Liste aus kürzlich erkannten Objekten hinzugefügt.

Die Consideration *EnemySpotted* garantiert auch bei diesem Verhalten, dass die KI sich nicht schlafen legt oder zum Schlafplatz läuft, wenn sie den Spieler gesehen hat.
Ähnlich wie *TableAvailable*, sucht die *BedAvailable* Consideration nach freien Betten im Level. Sind alle Betten bereits durch andere KIs besetzt, wird ein Score von 0 berechnet, womit diese Action ebenfalls einen Utility Score von 0 erhält und damit nicht in Betrachtung gezogen wird.
Die letzte Überlegung, die getroffen wird, trägt im Graphen den Namen *Fatigue*. Eine Response Curve mit dem Intervall von 0 bis 24 definiert, wann die KI ein großes Bedürfnis nach Schlaf verspürt. Abbildung 30 zeigt die verwendete Response Curve mit den verwendeten Kurvenabschnitten, die zum Schlafverhalten der KI führen. Durch das Verschieben von Keys entlang der *x*-Achse kann die Schlafdauer schnell an das Game Design angepasst werden, damit die KI den gewünschten Zeitraum schläft. In diesem Fall wird die Kurve so konfiguriert, dass die KI von 22 Uhr bis 6 Uhr einen Utility Score von 1 erhält, also in diesem Zeitraum ein hohes Bedürfnis nach Schlaf empfindet. Nach 6 Uhr fällt das Bedürfnis durch eine lineare Kurve schnell auf 0 ab, bevor ab 16 Uhr der Score durch eine logistische Kurve wieder ansteigt. Diese Consideration besitzt ebenfalls einen *RandomShift* mit einer maximalen Reichweite von 4 Stunden. Die Wahrscheinlichkeit, dass alle KIs ihre Wachposten auf einmal verlassen, schlafen gehen und damit völlig angreifbar sind, wird dadurch geringgehalten.
Anstatt das Schlafverhalten über eine Response Curve zu definieren, könnte auch bei diesem Verhalten ein Bedürfnis erstellt werden, das bei jeder KI mit einem zufälligen Wert initialisiert wird und über ein bestimmtes Intervall linear ansteigt. Durch diesen Ansatz könnten sich auch andere Verhalten auf das Schlafbedürfnis auswirken. Wird zum Beispiel Schwertkampf trainiert, so kann währenddessen das Bedürfnis nach Schlaf stark steigen. Solche dynamischen Reaktionen sind durch die vorab definierten Response Curves nicht möglich. Zusätzliche Funktionalität zum aufaddieren von Scores zu der Response Curve müsste hierbei hinzugefügt werden.

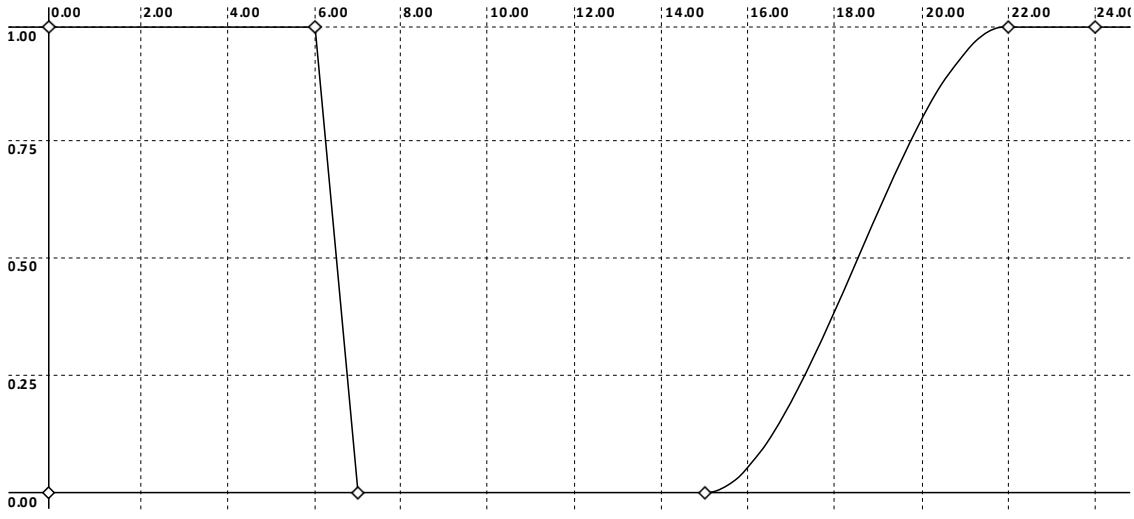

Abbildung 30: Durch diese Response Curve wird das Schlafgefühl definiert. Dieses ist über den Tag hinweg gering, bevor dieses abends ansteigt und morgens schnell abnimmt.

3.3.8 Training

Damit die KI neben Essen, Schlafen und Patrouillieren noch weitere Aufgaben am Tag ausführen kann, wurde das Verhalten des Nahkampftrainings eingeführt. Dabei sucht sich die KI unbesetzte Trainingspuppen im Level, läuft zu diesen hin und rüstet ein Schwert und ein Schild aus. In bestimmten Abständen wird ein Nahkampfangriff ausgeführt, wodurch das Schwertkampftraining dargestellt werden soll.

Die Action *MeleePractice* besitzt drei verschiedene Considerations, die gemeinsam den Utility Score bilden.

EnemySpotted stellt, wie bei den anderen passiven Verhalten, sicher, dass die KI nicht trainieren geht, wenn sich ein Gegner im Sichtbereich aufhält.

Die Consideration *PracticeAvailable* dagegen bewirkt einen Ausschluss der Action, sollte keine Trainingspuppe zur Verfügung stehen, da von dieser Trainingsressource ebenfalls eine begrenzte Anzahl im Level existiert.

Die letzte Überlegung, im Utility Graph *PracticeDuty* genannt, definiert durch eine individuelle Response Curve (siehe Abbildung 31) das Pflichtgefühl der KI, einmal am Tag dieses Verhalten ausführen zu wollen. Das Pflichtgefühl ist dabei um 13 Uhr auf dem höchsten Stand, wobei der Anstieg und das Abfallen des Scores durch eine logistische Kurve beschrieben wird. Da begrenzte Trainingspuppen im Level platziert sind und unter Umständen mehrere KIs auf einmal dieses Verhalten in Betracht ziehen, wird in dieser Consideration eine zufällige Verschiebung von maximal 12 Stunden zugewiesen. Da anders als beim Essen und Schlafen das Bedürfnis des Verhaltens nicht zwingend von der Tageszeit abhängig ist, wurde daher eine größere Reichweite der *RandomShiftRange* gewählt.

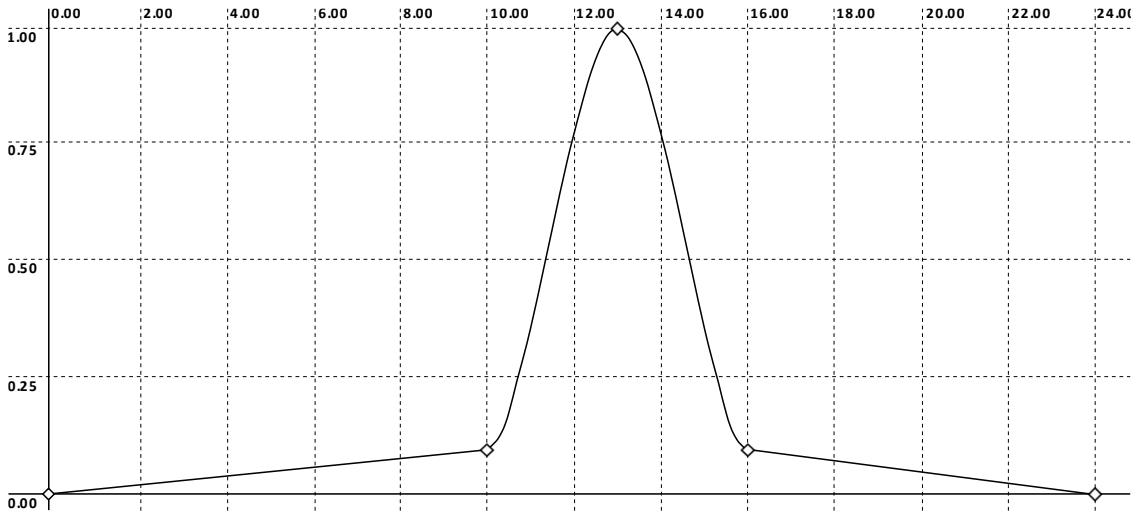

Abbildung 31: Um 13 Uhr ist der Utility Score des Schwertkampftrainings am höchsten. Morgens und abends ist der Score auf einem niedrigen Niveau.

3.3.9 Fliehen

In bestimmten Fällen sollte die KI die Möglichkeit besitzen, vor dem Spieler zu fliehen, anstatt in jedem Fall bis zum Tod zu kämpfen. Dazu sucht sich die KI eine sichere Position, damit sie besser vor Angriffen des Spielers geschützt ist.

Wählt eine Selektionsmethode dieses Verhalten aus, so wird eine EQS Query ausgeführt, um einen sicheren Ort zu finden, zu dem die KI fliehen kann. Auf das EQS System wird in Abschnitt 5.2 eingegangen. Die eingesetzte EQS Query prüft die Entfernung der KI zum Spieler, ob sich Deckung in der Nähe befindet und an welchen Positionen Sichtkontakt zum Spieler besteht.
Sobald die KI dann eine passende Position ausgewählt hat, läuft sie zu diesem Ort und markiert diesen als eine sichere Position. Die Consideration des Verhaltens *Heal* benötigt diese Markierung, auf welches in Abschnitt 3.3.10 eingegangen wird.
Im Utility AI Graph wird das Verhalten durch die Action mit dem Namen *Flee* definiert. Die Action besitzt drei Considerations, welche die Auswahlkriterien des Verhaltens definieren. Diese Action wird durch den Einsatz eines Gewichts von 1,1 vor Verhalten wie Nahkampf oder Fernkampf bevorzugt.
Die erste Consideration *OwnHealth* bewertet das Leben der KI und ist, wie bei der Consideration von *MeleeAttack*, parametrisiert. Da jedoch bei niedrigen Lebenspunkten ein hoher Score erzielt werden soll, ist das Ergebnis der Response Curve invertiert, damit die KI bei kritischer Verwundung flieht.
FriendsNearby wird als weiteres Auswahlkriterium definiert. Durch diese Überlegung wird überprüft, ob und wie viele befreundete Charaktere sich in der Nähe der KI befinden. Die Consideration wird durch eine normalisierte, individuell angelegte Response Curve abgebildet und beachtet dabei bis zu 5 freundliche KIs. Der Utility Score fällt dabei logistisch von 5 auf 0 ab. Da bei mehr als 5 Kameraden ein niedriger Utility Score entstehen soll, verläuft diese Kurve von einem Utility Score von 1 bis 0. Sollten sich mehr als 5 befreundete KIs in der Nähe aufhalten, wird der Score auf 0 geclamped. Durch diese Consideration wird die KI dazu verleitet, dieses Verhalten stärker in Betracht zu ziehen, wenn sie allein gegen den Spieler kämpfen muss oder der Spieler bereits einige andere KIs ausgeschaltet hat.

3.3.10 Heilen

Wird das Verhalten Heilen ausgewählt, wurde die KI durch einen Kampf mit dem Spieler bereits verwundet und muss sich selbst heilen, damit sie nicht stirbt. Dabei verbraucht die KI einen Großteil der ihr zur Verfügung stehenden Ausdauer, um ihre Lebenspunkte zu regenerieren. Damit der Spieler erkennen kann, dass sich die KI heilt, wird ein Partikelsystem erzeugt, welches die Heilung visualisieren soll. Da dieses Verhalten als Auswahlkriterium eine Variable enthält, die nur von *Flee* gesetzt werden kann, ist sie die direkte Konsequenz des Fliehen Verhaltens.
Diese Action mit dem Namen *HealSelf* bezieht die eigene Gesundheit als Überlegung mit ein. Da *OwnHealth* bei wenigen Lebenspunkten einen hohen Score erzielen soll, wird die verwendete normalisierte, logistische Response Curve invertiert. Dadurch wird mit sinkender Gesundheit die Auswahl dieser Action immer wahrscheinlicher. Damit das Verhalten in allen Fällen Vorrang hat, wird dem Action-Node ein Weight von 3 zugewiesen.
Da die KI Ausdauer benötigt, um sich heilen zu können, bewertet die Consideration *OwnStamina* den aktuellen Ausdauerzustand der KI. Die logistische Response Curve, die durch das minimale

Bookend von 1 und das maximale Bookend von 100 beschrieben wird, liefert bei voller Ausdauer einen Utility Score von 1.
Die letzte Consideration, die im Utility Graph definiert wurde, wird *SafeLocation* genannt. Diese wird beim Erreichen einer sicheren Position durch *Flee* auf den Score von 1 gesetzt. Durch diese Überlegung kann sich die KI nur heilen, wenn sie sich in großer Entfernung zum Spieler befindet, damit nicht die Gefahr entsteht, bei der Ausführung des Verhaltens direkt wieder vom Spieler verletzt zu werden.

3.3.11 Geräusch erforschen

Der Spieler kann durch das Drücken einer Taste ein Geräusch erzeugen. Nimmt die KI ein solches Geräusch wahr, so läuft sie zur Geräuschquelle und wartet dort für eine kurze Zeit.
Die Action *IvestigateSound* besitzt zwei Considerations, *EnemySpotted* und *SoundHeard*. Die erste der beiden Kriterien stellt sicher, dass die KI dem Geräusch nicht nachgeht, wenn sie den Spieler bereits gesehen hat.
SoundHeard dagegen vergibt einen Utility-Score von 1, wenn die Perception-Komponente der KI ein Geräusch registriert hat. Ansonsten besitzt diese Consideration einen Score von 0. Damit der entsprechende Wert aus der Response Curve berechnet werden kann, wird eine normalisierte, lineare Kurve verwendet.

3.4 Ansteuerung mehrerer Charaktere

Damit die in der Unreal Engine 4 implementierte Utility AI auf mehrere KI Charaktere angewendet werden kann, müssen verschiedene Komponenten in der Engine angepasst werden. Eine Zuweisung desselben Utility AI Controllers zu mehr als einem KI Charakter ist nicht möglich, da der AI Controller selbst nicht instanziiert wird. Da im Utility AI Controller die Ausführung des Verhaltens implementiert ist, würde die mehrfache Zuweisung der Klasse zu einem Schwarmverhalten führen. Nimmt beispielsweise einer der KI Charaktere ein Geräusch wahr, würden damit alle KIs das Verhalten *Geräusch erforschen* ausführen. Außerdem wäre dadurch die Auswertung des Utility AI Graphen fehlerhaft, da im AI Controller das Utility AI Graph Asset referenziert und nicht instanziiert würde. Da im Graphen die Utility Scores gespeichert werden, würden sich die verschiedenen zugewiesenen AI Controller gegenseitig die zuvor berechneten Werte überschreiben. Verschiedene Eingabewerte, wie zum Beispiel die Anzahl der Lebenspunkte einer KI, sind unter Umständen unterschiedlich, wodurch verschiedene Consideration Scores berechnet würden. Dies würde jedoch zu einem sprunghaften Wechsel der Consideration Scores und Action Scores führen, wodurch ständig neue Verhalten selektiert würden.
Damit diese Probleme nicht entstehen, muss vom Utility AI Controller für jede KI, welche angesteuert werden soll, eine neue Klasse abgeleitet werden. In der abgeleiteten Klasse werden anschließend die Funktionen *BeginPlay* und *Tick* der Eltern-Klasse aufgerufen, damit die Auswertung des Utility AI Graphen und die Ausführung des Verhaltens stattfinden kann. Durch die Erstellung abgeleiteter Klassen lassen sich Änderungen, beispielsweise im Ablauf eines Verhaltens, zentral in einer Klasse vornehmen, anstatt diese in mehreren Klassen zu aktualisieren. Da auch der Utility AI Graph zur Speicherung genutzt wird, muss das Utility AI Asset dupliziert werden,

um dieses den verschiedenen Instanzen der AI Controller zuweisen zu können. Diese Implementierung ist nicht optimal, da für jede KI im Level zwei verschiedene Assets dupliziert und zugewiesen werden müssen.

3.5 Debugging

Für den Behavior Tree stehen in der Unreal Engine 4 zahlreiche Methoden zur Verfügung, um ein erstelltes Verhalten einer KI zu debuggen. Ist der Behavior Tree im Graph Editor geöffnet, so wird der ausgeführte Branch und Node visuell hervorgehoben. Dadurch lässt sich schnell erkennen, welcher Teilbaum des Behavior Trees und damit welches Verhalten von der KI ausgeführt wird. In Abbildung 32 ist ein Ausschnitt des im Rahmen dieser Bachelorarbeit erstellten Behavior Trees abgebildet. Die KI führt zum Zeitpunkt der Aufnahme einen Fernkampfangriff aus und wartet, bevor sie auf den Spieler zielt.

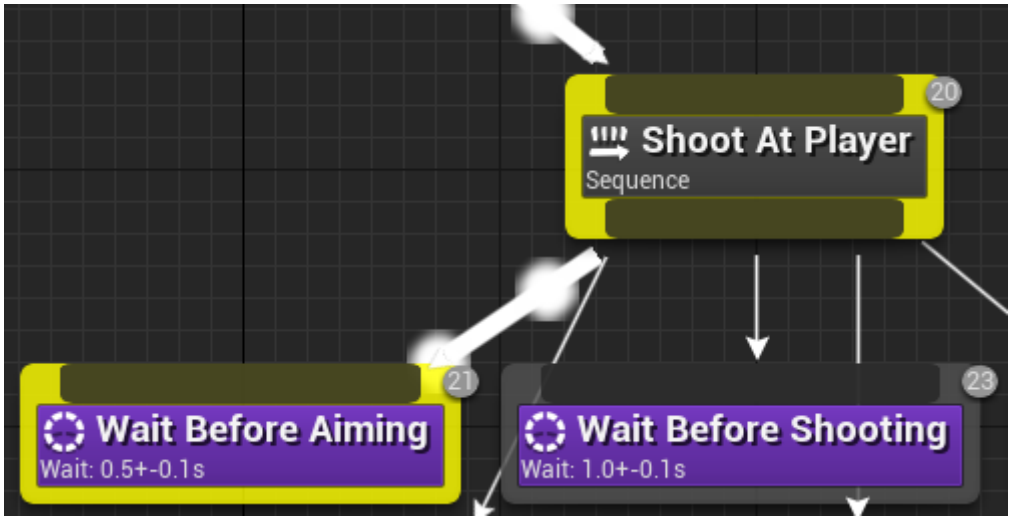

Abbildung 32: Visualisierung des ausgeführten Behavior Trees, wobei Nodes, welche gelb markiert sind, durch die KI ausgeführt werden.

Neben der visuellen Darstellung im Behavior Tree Graph Editor kann das ausgeführte Verhalten außerdem über den Gameplay Debugger der Engine analysiert werden. Der Gameplay Debugger dient dazu, Informationen der KI zur Laufzeit des Spiels bereitzustellen. So können Daten vom Environment Querying System, worauf in Abschnitt 4.2 eingegangen wird, dem AI Controller und dem Perception-System angezeigt werden. Bezüglich des Behavior Trees werden dabei unter anderem die durchlaufenen Sequence- und Selector-Nodes und die momentan ausgeführte Aufgabe angezeigt. Neben Informationen zum Behavior Tree selbst, werden außerdem die Schlüssel und Werte des zugewiesenen Blackboards aufgelistet.

Im Fall des Utility AI Graphen sollte zum Debuggen jeder Consideration Score, aber auch jeder Action Score, ersichtlich sein, da diese Werte entscheidend bei der Auswahl des Verhaltens sind. Dies wird in der Implementierung auf zwei verschiedene Arten ermöglicht. Einerseits durch eine

Anzeige aller Action Scores als Progress Bars[25] und die Darstellung der Action sowie der Consideration Scores im Utility AI Graph Editor. Über den Utility AI Controller besteht außerdem die Möglichkeit, durch das Ausführen spezieller Debug-Nodes, die Scores als Text im Viewport anzuzeigen.

Um in den Debug Modus des Utility AI Systems zu wechseln, wird die Tab-Taste gedrückt. Dadurch wechselt die Steuerung zu einer Flugkamera, mit der zu jedem beliebigen Punkt im Level navigiert werden kann. Außerdem wird beim Wechsel in den Debug Modus das User Interface geändert. So werden anstatt einer Lebenspunkte- und Ausdaueranzeige relevante Informationen zur Utility AI angezeigt, wie beispielsweise die Anzeige der Action Scores.

Im Folgenden soll auf die Visualisierung der Action Scores durch Progress Bars eingegangen werden.

Durch die Anzahl der verschiedenen Verhalten, welche durch das Utility AI System implementiert wurden, ist es möglich, eine Anzeige zu erstellen, welche alle Action Scores visualisieren kann. Um eine intuitive Darstellung der Werte zu ermöglichen, wurden daher Progress Bars verwendet, da sich dadurch die Größenverhältnisse der Scores ablesen lassen. Deshalb wurde eine Anzeige der wichtigsten Informationen direkt in den Viewport des Spielers integriert. Diese befindet sich in der rechten oberen Ecke des Bildschirms.

Der Progress Bar hat in der Unreal Engine 4 standardmäßig einen Wertebereich von 0 bis 1, womit diese sich sehr gut für die Darstellung der Scores eignet, da durch die Berechnungen der Utility AI die Werte bereits normalisiert werden. Neben den Progress Bars wird in der Debug Ansicht außerdem der Namen der ausgewählten Decision angezeigt (siehe Abbildung 33), da dieser je nach Selektionsmethode nicht immer dem höchsten Score entspricht.

Durch das Bestimmen einer Gewichtung für einen Action-Node kann ein Score entstehen, welcher sich nicht mehr im normalisierten Wertebereich befindet, also größer als 1 ist. In diesem Fall wird der Progress Bar auf den Wert von 1 geclamped. Damit der Score trotzdem in Erfahrung gebracht werden kann, wird der Score der ausgewählten Decision ebenfalls abgebildet.

Da jede KI über einen eigenen Graphen, eigene Scores und ein eigenes Verhalten verfügt, müssen diese Informationen getrennt ersichtlich sein. Um eine ausgewählte KI zu analysieren, muss das Modell des KI-Charakters fokussiert werden, damit die Debug Anzeige aktualisiert werden kann. Damit der Charakter korrekt fokussiert werden kann, sollte sich die Kamera in der Nähe des Charakters befinden.

[25] Dt.: Fortschrittsanzeige

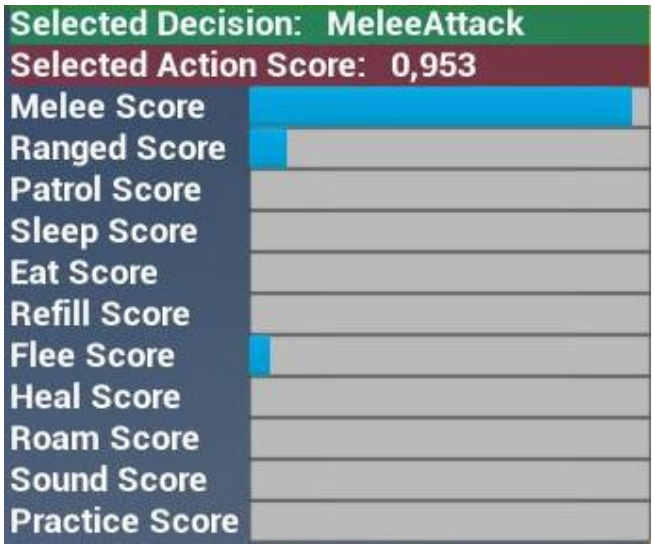

Abbildung 33: Die Werte der verschiedenen Actions werden durch Progress Bars dargestellt. Der grün hinterlege Text zeigt die ausgewählte Decision an und der rot hinterlegte Text den Action Score der ausgewählten Decision.

Neben den Action Scores und den ausgewählten Actions sind außerdem andere Informationen zur Findung möglicher Fehler relevant. Daher wird im oberen linken Bildschirmrand der Name der Blueprint Instanz der KI im Level angezeigt, wobei sich dahinter die Farbe des Charakters zur einfachen Differenzierung befindet. Außerdem wird der Name des verwendeten AI Controllers, des zugewiesenen Utility AI Graphen und die Anzahl der Pfeile angezeigt. Aufgrund der Existenz von Verhalten, welche auf die Tageszeit angewiesen sind, wird außerdem die aktuelle Uhrzeit im Level angezeigt.

Hat eine Action einen Score, welcher nicht den Überlegungen des KI-Erstellers entspricht und daher zu einer fehlerhaften Auswahl eines Verhaltens führt, so müssen die Scores der verschiedenen Considerations ersichtlich sein. Sollte beispielsweise ein Action Score einen niedrigen Wert besitzen, obwohl dieser, dem Kontext des Spiels entsprechend, einen hohen Wert haben sollte, so können die Consideration Scores über die Kategorie Debug des Utility AI Graph Editors eingesehen werden.

4 Behavior Trees

In diesem Kapitel soll das Konzept der Utility AI mit der Methodik der Behavior Trees verglichen werden.

Behavior Trees werden heutzutage immer beliebter in der Spieleindustrie, weshalb sie weit verbreitet und bereits ein wichtiger Standard in vielen Game Engines sind. So verwendete sie beispielsweise der Entwickler *Bungie* in der *Halo* Serie und auch der deutsche Spieleentwickler *Crytek* machte sich diese Art KI zu erstellen in *Crysis 2* zunutze. Durch ihre Struktur und Regeln bieten Behavior Trees einen pragmatischen Ansatz, um Entscheidungen zu treffen und Verhalten abzubilden [12].

Eine KI zu erstellen kann durch verschiedene Methoden erfolgen. Dafür sind unter anderem Fuzzy Logic, Goal Oriented Behavior und State Machines geeignet. Einen Ansatz zu finden, welchen das Team in kurzer Zeit erfassen, ist bei der Erstellung einer KI entscheidend. Die Entwicklung, das Design und die Struktur zur Erstellung von Behavior Trees ist ausführlich im Internet sowie in Literatur dokumentiert.

Frei zugängliche Game Engines, wie beispielsweise die Unreal Engine 4, bieten eine graphische Oberfläche zum Erstellen, Editieren und Debuggen von Behavior Trees [12].

In Abschnitt 4.1 wird im Detail auf die Implementierung von Behavior Trees in der Unreal Engine 4 eingegangen. Weiterhin wird eine beispielhafte Implementierung einer KI mit dieser Methode vorgestellt. Die *CryEngine* verwendet ebenfalls Behavior Trees zur Erstellung von KI. Hier steht jedoch keine graphische Oberfläche zur Verfügung, weshalb die Struktur über XML definiert wird [13].

Behavior Trees bieten im Allgemeinen einen Ausgleich zwischen Implementierbarkeit und Komplexität. Durch die Baumstruktur können komplexe Verhaltensabläufe und die Auswahl von Entscheidungen übersichtlich dargestellt werden. Damit ergibt sich auch für neue Teammitglieder ein gewisses Maß an Zugänglichkeit für bereits erstellte Bäume. Die Erstellung eines Werkzeugs zur Auswertung und Kreation von Behavior Trees ist für erfahrene Programmierer unkompliziert und spart damit wertvolle Zeit im Entwicklungsprozess [6].

Behavior Trees bestehen aus sogenannten Branches[26] und Nodes, welche verschiedene Aufgaben im Kontrollfluss des Baumes besitzen oder direkt ein Verhalten der KI ausführen können. Dabei werden bei der Erstellung dieser Struktur Aufgaben entsprechend ihrer Priorität angeordnet, wobei diese von links nach rechts abnimmt. Die Priorität gibt dabei an, welches Verhalten von der KI bevorzugt wird.

Behavior Trees haben jedoch auch Nachteile. Die Implementierung eines Verhaltens einer KI, welche keine eindeutige und statische Reihenfolge von Verhaltensabläufen besitzt, ist mit dieser Struktur zeitaufwändig und komplex. Dadurch würde eine KI, welche immer in derselben Reihenfolge essen, schlafen und patrouillieren gehen würde, sich aber nicht durch äußere Einflüsse beeinflussen lässt, sehr statisch und ohne Intelligenz wirken.

[26] Dt.: Äste

Die strikten Regeln zur Priorisierung führen einerseits zu verständlichen und klar strukturierten Graphen, andererseits kann die Nutzung einer Priorität jedoch auch limitierend sein. In manchen Fällen könnte ein Verhalten, beispielsweise abhängig vom Kontext, eine unterschiedliche Priorisierung erfordern. Um ein solches Verhalten abzubilden, werden Teilbäume dupliziert, damit die verschiedenen Kontexte und Prioritäten nachgebildet werden können. Auch wenn diese Teilbäume durch die Implementierung instanziiert oder referenziert werden können, ist diese Methode ineffizient, führt zu einem unübersichtlichen Graphen und ist damit lediglich mit viel Zeitaufwand wartbar. Außerdem können Entscheidungen, deren Antwort sich nicht intuitiv durch ja oder nein beantworten lassen, zu komplexen Strukturen beim Erstellen des Behavior Trees führen. Entscheidungen sind selten binär, im besten Fall sollte eine breite Menge an Kombinationen von möglichen Eingaben beachtet werden. Zu diesen Eingaben zählen die Einflüsse der Spielwelt, ebenso wie die inneren Bedürfnisse der KI [6].

4.1 Behavior Trees in der Unreal Engine 4

In diesem Abschnitt soll auf die Unterschiede zwischen typischen Behavior Trees und der Funktionsweise der Implementierung in der Unreal Engine 4 eingegangen werden.

Im Gegensatz zu traditionellen Behavior Trees, deren Evaluation in einem Tick, also in definierten Abständen stattfindet, liegt bei der Auswertung in der Unreal Engine ein Event basierter Ansatz vor. Wird ein Node des Behavior Trees aufgerufen, so wird der Baum nicht neu ausgewertet, bis die zugewiesene Aufgabe abgeschlossen ist oder bis bestimmte Bedingungen im Baum geändert werden.

Die Implementierung des Behavior Trees in der Unreal Engine 4, welche einen Graph Editor, also eine visuelle Oberfläche zum Erstellen von Verhalten anbietet, ist eng mit dem Unreal Engine 4 Blackboard System verknüpft.

Ein Blackboard dient der KI als Informationsspeicher. In diesem Speicher können, neben Werten, die lediglich zur Steuerung des Ablaufs von Teilbäumen im Graphen nötig sind, auch äußere Einflüsse der Spielwelt, sowie Kontexte und innere Werte der KI, wie beispielsweise Lebenspunkte und Ausdauer, abgefragt und abgelegt werden.

Diese Werte werden als Schlüsselwert-Paar gespeichert. Dazu muss der Datentyp, welcher ein einfacher *Integer* oder ein komplexes Objekt sein kann, ausgewählt und ein eindeutig identifizierbarer Name vergeben werden. Der Wert kann in einfachen Fällen, wie zum Beispiel für die Datentypen *String* oder *Float*, schon vorab vergeben oder während der Laufzeit durch Funktionen verändert werden.

Durch das Blackboard System ist der Behavior Tree dynamisch und flexibel. Die Einträge im Blackboard bieten die Möglichkeit die verschiedenen Node-Typen des Behavior Trees zu parametrisieren.

Außerdem ermöglicht die Nutzung eines Blackboards eine schnelle Abfrage von bereits gespeicherten Werten. Somit muss nicht mehrmals im Baum derselbe Wert aus der Spielwelt abgefragt werden, wodurch ebenfalls Rechenleistung reduziert werden kann.

So kann zum Beispiel in einem Teilbaum einmalig die Position des Spielers ermittelt und gespeichert werden. In einem anderen Teilbaum kann nun auf diese Information zugegriffen werden und der Node *MoveTo*, welcher die KI durch Wegfindung zu einem bestimmten Punkt im Level steuert, kann dadurch parametrisiert werden. Damit ist die Möglichkeit einer Entscheidung der KI zwischen Bewegungen hin zu unterschiedlichen Orten gegeben.
In der Unreal Engine 4 werden Bedingungen als so genannte Decorator-Nodes verwendet. Diese können an Selector[27]- oder an Sequence[28]-Nodes angehängt werden. Diese beiden Nodes sind für den Kontrollfluss im Behavior Tree Graphen verantwortlich. In der Engine sind außerdem verschiedene vorgefertigte, oft benötigte Bedingungen verfügbar, welche beispielsweise überprüfen, ob ein Eintrag im Blackboard bereits definiert wurde. Dabei kann jedoch auch ein benutzerdefinierter Decorator-Node erstellt werden. Sollte sich einer der Parameter in einer der Bedingungen ändern, so kann darauf durch Events, die beispielsweise den ganzen Graphen neu evaluieren lassen, reagiert werden. Teilbäume des Behavior Trees mit niedrigerer Priorität oder der eigene Teilbaum können dadurch ebenfalls unterbrochen werden, um auf den neuen Kontext zu reagieren und das Verhalten entsprechend zu ändern [14].

4.2 Environment Querying System

Da die KI, um Entscheidungen zu treffen, zahlreiche Informationen aus der Spielwelt benötigt, ist ein System erforderlich, welches diese Informationen in effizienter Art und Weise zur Verfügung stellen kann.
Sollen beispielsweise in einem bestimmten Umkreis der KI spezielle Objekte gefunden werden, so könnte über eine Schleife im Code festgestellt werden, welche der gesuchten Objekte in der angegeben Reichweite liegen. Während dieser Ansatz für einmalige Anfragen plausibel und unter Umständen auch zu bevorzugen ist, kann dies bei gehäufter Verwendung in verschiedenen Teilen eines Projekts zu Unübersichtlichkeit und schlechter Wartbarkeit führen. Anhand nur eines Kriteriums eine geeignete Antwort zu finden, ist häufig nicht ausreichend, um eine Entscheidung zur Auswahl eines Verhaltens zu treffen.
Für dieses Problem stellt die Unreal Engine 4 das Environment Querying System[29] (EQS) zur Verfügung. Dieses System wird dafür genutzt, räumliche Informationen für die KI während der Laufzeit bereitzustellen.
EQS ist derzeit noch eine experimentelle Komponente der Engine und muss daher über die Editor-Präferenzen unter der Kategorie AI aktiviert werden, bevor dieses genutzt werden kann [15].
Zur Erstellung von EQS Assets stellt die Unreal Engine 4 einen eigenen Editor bereit, mit dessen Hilfe übersichtlich eine solche räumliche Abfrage erstellt werden kann. Diese Assets können gespeichert und wiederverwendet werden. Abbildung 34 zeigt dabei die Darstellungsweise im Unreal Engine 4 Editor.

[27] Dt.: Auswahl

[28] Dt.: Sequenz

[29] Dt.: System zur Abfrage der Umgebung

Die beispielhafte Implementierung durch Behavior Trees und auch die eigene Utility AI nutzen EQS, damit eine optimale Position für eine KI in der Spielwelt gefunden werden kann. EQS Abfragen können außerdem dazu verwendet werden, um die Distanz der KI zu bestimmten Objekten im Level zu ermitteln und so beispielsweise den nächsten Tisch zum Essen für die KI zu finden, da nach verschiedenen Klassen und Objekten gefiltert werden kann.
Behavior Trees stellen standardmäßig Nodes zur Verfügung, welche eine zugewiesene EQS-Query ausführen und die gefundene, abgefragte Position in einen Blackboard-Eintrag speichern [14].
Das Verhalten Fliehen führt nach Auswahl dieses Verhaltens eine EQS-Query aus, durch welches eine Position ermittelt werden soll, zu welcher sich die KI in Sicherheit begeben kann.

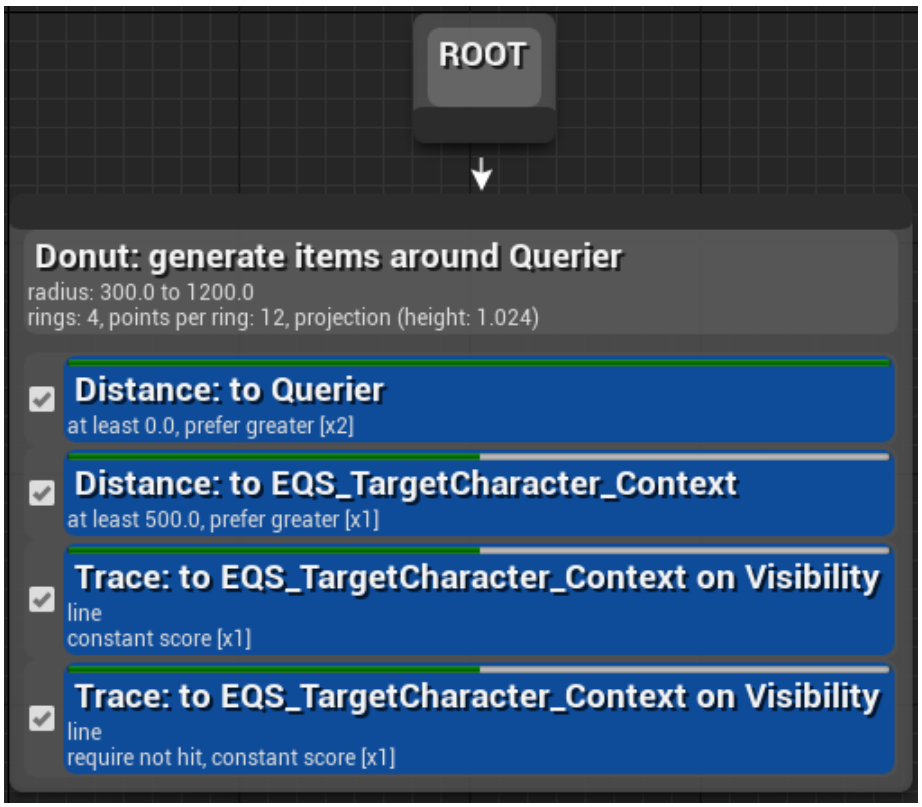

Abbildung 34: Darstellung der EQS-Query der verschiedenen Kriterien zur Ermittlung einer Position zum Fliehen im EQS Editor.

Im EQS System kann eingestellt werden, auf welche Weise die möglichen Positionen um den Querier[30], also die KI, welche das EQS ausführt, generiert werden. Abbildung 34 zeigt dabei die konfigurierten Optionen des erstellten EQS Systems. In diesem Fall werden die Positionen ringartig um die KI in einer Reichweite von 300 bis 1200 Einheiten erstellt.
Die in blau dargestellten Komponenten sind Kriterien, sogenannte Tests und dienen der Ermittlung einer geeigneten Position im Level. Eines dieser Kriterien ermittelt die Distanz zu den generierten Punkten. Durch eine Gewichtung kann der berechnete Score höher priorisiert werden.

[30] Dt.: Abfragequelle

Für den Fall, dass die KI fliehen muss, werden weiter entfernt liegende Positionen höher bewertet. Das gleiche Prinzip wird auch im nächsten Test angewandt. Dabei wird jedoch eine größere Entfernung zum Spieler mit einem höheren Wert bewertet.
Die letzten beiden Tests führen für jede mögliche Position einen Line-Trace zum Spieler aus. Dabei wird ermittelt, ob sich die Position hinter hüfthoher oder in kompletter Deckung befindet. Positionen in kompletter Deckung werden mit einem niedrigeren Score bewertet, da die KI von der neuen Position aus in der Lage sein soll, hinter der Deckung eine Fernkampfwaffe zu verwenden.
Durch die Kombination der Kriterien wird, ähnlich zum Utility AI System, ein Score ermittelt, der die Qualität einer Position repräsentiert. Dieser kann mithilfe eines sogenannten *EQS Testing Pawns* auch im Level visualisiert werden. Neben der Anzeige des berechneten Scores wird durch Farben der Wert dargestellt. Blau markierte Positionen sind als ungeeignet zu interpretieren. Ein Farbverlauf von rot nach grün visualisiert entsprechend niedrige bis hohe Scores.

4.3 Implementierung von Verhalten durch Behavior Trees

Neben der Implementierung einer KI durch das Utility AI System wurde im Rahmen dieser Bachelorarbeit außerdem eine KI durch das Behavior Tree System der Unreal Engine 4 erstellt. Der Behavior Tree definiert dabei jedoch nur zum Teil die in Abschnitt 3.3 beschriebenen Verhalten, da die Implementierung lediglich einer einfachen Demonstration dienen soll.
Die KI, welche dieses Verhalten zugewiesen bekommen hat, prüft zum Start des Levels, ob sich der Spieler im Sicht- oder Hörbereich befindet. Der Service[31] *Select Sensed Player* ist dabei dem ersten Selektor-Node des Behavior Trees zugewiesen. Ein Decorator überprüft anschließend, ob ein Geräusch im Level wahrgenommen wurde. Trifft die Bedingung des Decorators zu, so wird, ähnlich zum Verhalten Geräusch erforschen, zum Ort der Geräuschquelle gelaufen und dort für wenige Sekunden gewartet. Wurde ein Spieler zum Angreifen gefunden, so wird eine Sequenz im Behavior Tree gestartet. Befindet sich die KI außerhalb einer definierten Reichweite von 2000 Einheiten, siehe *Is Further Than X Units* Decorator in Abbildung 35, so bewegt sich die KI in die Richtung der aktuellen Position des Spielers. Ist die KI jedoch bereits innerhalb der definierten Reichweite, so wird eine EQS-Query ausgeführt, um eine geeignete Position zu bestimmen. Dabei wird dieselbe EQS-Query eingesetzt, welche auch für das Verhalten Fliehen verwendet wurde. Nach der Bestimmung einer geeigneten Position bewegt sich die KI zu diesem Ort und wartet dort eine bestimmte Zeit. Befindet sich die KI an einer sicheren Position, zielt und schießt diese mit Pfeil und Bogen auf den Spieler. Für den Fall, dass kein Spieler zum Angreifen gefunden wurde, bestimmt der Task *Find Nearest Player*, welcher ebenfalls eine EQS-Query ausführt, erneut die Position des Spielers und bewegt sich in diese Richtung. Dadurch wird garantiert, dass die KI den Spieler in jedem Fall findet und angreift, auch wenn sich die KI und der Spieler an weit entfernten Orten im Level befinden.

[31] Sogenannte Services in der Unreal Engine 4 führen in einem definierten Zeitintervall individuelle Aufgaben, wie beispielsweise Überprüfungen von Werten, aus [16].

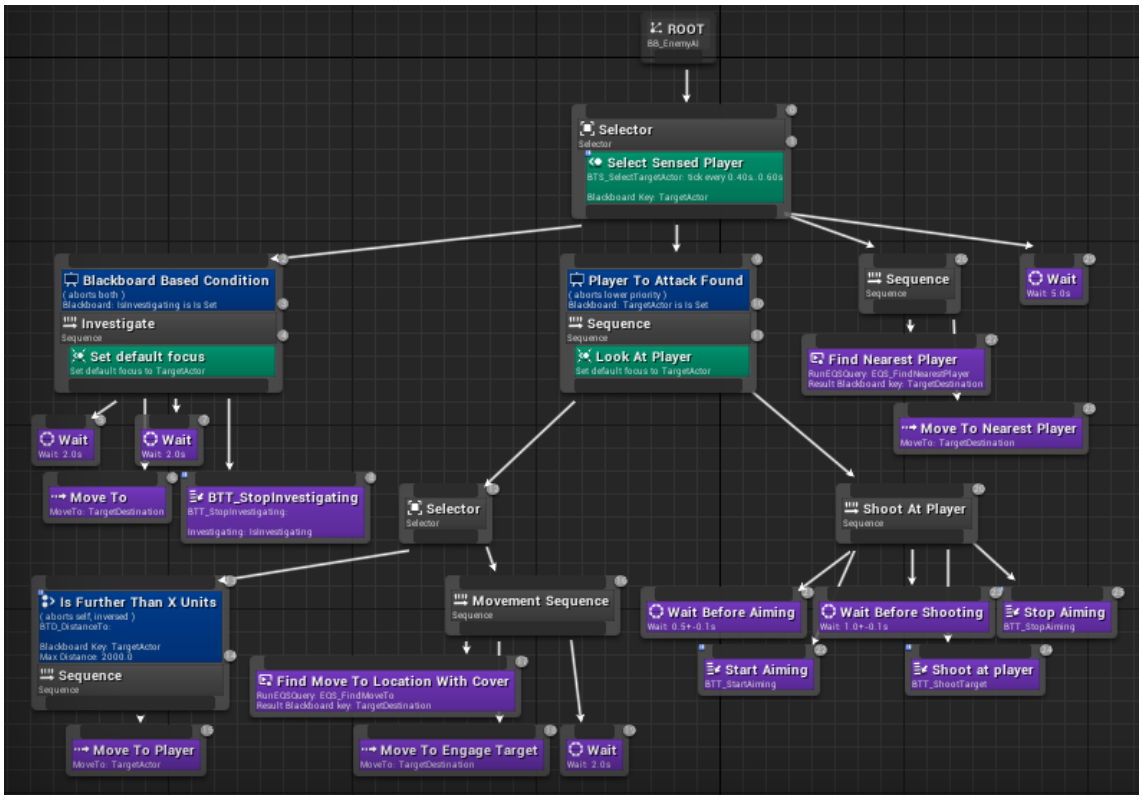

Abbildung 35: Übersicht des implementierten Behavior Trees in der Unreal Engine 4. Implementiert wurden Verhalten, wie Geräusch erforschen und mit dem Bogen schießen.

4.4 Einbindung der Utility AI in Behavior Trees

Mit Behavior Trees ein Verhalten auszuwählen, welches nicht durch eine binäre Entscheidung selektiert werden kann, sondern eine Reihe von Eingaben in Betracht ziehen muss, ist in den meisten Fällen umständlich. Anstatt neue Werkzeuge und Editoren zur Erstellung von Utility AI zu entwerfen und implementieren, kann die flexible Struktur von Behavior Trees dafür verwendet werden, das Utility AI System direkt in diesen einzubinden. Dafür muss keine grundlegende Veränderung der Behavior Tree Architektur vorgenommen werden. Alle Vorteile des Behavior Trees bleiben erhalten und die Vorteile des Utility AI Systems können zusätzlich ergänzt werden. Dieser Abschnitt beschreibt die Komponenten, die zur Integration der Utility AI in Behavior Trees benötigt werden. Da der Behavior Tree auf einer Baumstruktur basiert, bietet sich die Erweiterung durch spezielle Nodes zur Einbindung des Utility AI Systems an. Um eine Utility AI basierte Auswahlmöglichkeit von Verhalten in den Behavior Tree einzubinden, kann eine spezialisierte Variante des Selektors erstellt werden. Dieser Utility Selektor wählt den Kind-Knoten nicht nach einer binären Gültigkeit aus, also ob der Kind-Knoten *Wahr* oder *Falsch* zurückgibt, sondern nach einem berechneten Utility Score. Wird ein Utility Selektor ausgeführt, so stellt dieser zunächst die Utility Scores aller Kind-Knoten fest. Wurden diese Scores gesammelt, so kann, durch eines der in Abschnitt 2.1.1 beschriebenen Auswahlverfahren, der entsprechende Kind-Knoten ausgewählt und das damit verbundene Verhalten ausgeführt werden. Anstatt die Knoten des Utility Selektors in einer statischen Reihenfolge abzuarbeiten, wird damit eine dynamische Selektion erreicht [6].

5 Ausblick und Diskussion

In diesem Kapitel werden die Schwierigkeiten, welche bei der Implementierung der Utility AI in der Unreal Engine 4 entstanden sind, beschrieben. Außerdem wird auf mögliche Optimierungen der Auswertung der Utility AI eingegangen, wodurch der bestehende Utility AI Graph verbessert werden kann, wie die Debugging-Möglichkeiten erweitert werden können und womit das implementierte Verhalten ausgebaut werden kann. Anschließend soll auf eine mögliche Verknüpfung des Utility AI Graphen mit dem Behavior Tree der Unreal Engine 4 eingegangen werden.

5.1 Probleme bei der Implementierung der Utility AI

Um die Utility AI in der Unreal Engine 4 umzusetzen, mussten während der Implementierung verschiedene Probleme gelöst werden.

Eines der Probleme war die Anwendung des implementierten Utility AI Systems auf mehrere Charaktere. Der Utility AI Controller, welcher für die Auswertung des Utility AI Graphen, die Berechnung und Auswahl der Actions und für die Implementierung des Verhaltens verantwortlich ist, wird nicht instanziiert. Das bedeutet, dass ausgeführtes Verhalten für alle KI-Charaktere übernommen wird, welche den Utility AI Controller zugewiesen bekommen haben. Da zur Speicherung der berechneten Action und Consideration Scores dieselbe Instanz des Utility Graphen verwendet wird, überschreiben die verschiedenen verwendeten Utility AI Controller diese berechneten Scores des Utility AI Graphen. Dies führt zur Berechnung verschiedener Consideration Scores, die beispielsweise eine Distanz zum Spieler bewerten. Dadurch entstehen abrupte Verhaltensänderungen. Behavior Trees werden in der Unreal Engine durch eine Ableitung der Klasse *UBrainComponent* instanziiert. Daher wurde eine individuelle Klasse erstellt, die ebenfalls von dieser Komponente abgeleitet wird, die *UUtilityAIComponent*. Die Berechnung der Action und Consideration Scores wurde vom Utility AI Controller in diese abgeleitete Klasse übernommen. Im Utility AI Controller wurde anschließend eine Funktion erstellt, welche beim Start der KI ein neues Objekt der *UUtilityAIComponent* erzeugt. Im Zeitrahmen dieser Bachelorarbeit gelang jedoch keine funktionale Umsetzung dieses Ansatzes, da aus einem unbekannten Grund die Initialisierung, beziehungsweise Ausführung der *UUtilityAIComponent* Klasse nicht stattfand und dadurch keine Auswertung des Utility AI Graphen möglich war.

Um die Nutzung des Utility AI Graph Editors benutzerfreundlicher und effektiver zu gestalten, wurde versucht, die Oberfläche des Editors anzupassen. Dabei sollten beispielsweise Symbole zur Identifikation von genutzten Optionen angezeigt oder verschiedene Node-Typen erzeugt werden können. Auf die genannten möglichen Verbesserungen wird in Abschnitt 5.3 näher eingegangen. Da die Implementierung der nötigen Klassen für solche Anpassungen komplex und zeitaufwändig ist, wurde auf die Modifikation des Utility AI Graph Editors verzichtet. Auch ohne die geplante Anpassung erfüllt der Utility AI Graph Editor seinen Zweck zur Definition von Action-, Consideration- und Decision-Nodes. Andere Komponenten der Utility AI Implementierung wurden daher priorisiert.

5.2 Optimierung der Auswertung

In diesem Abschnitt soll auf die verschiedenen Möglichkeiten eingegangen werden, die Auswertung der Utility AI zu optimieren und die Bedienbarkeit des Utility AI Graph Editors zu verbessern.

Um die Auswertung zu optimieren und Rechenleistung zu reduzieren, können in der eigenen Implementierung an verschiedenen Komponenten Verbesserungen vorgenommen werden. Durch eine Einführung eines Schwellenwerts für Action Scores kann die Geschwindigkeit der Auswahl eines Verhaltens erhöht werden. Action Scores, welche sich unter einem definierten Schwellenwert befinden, müssen nicht sortiert und durch den *Weighted Random* Algorithmus nicht weiter beachtet werden.

Werden Consideration-Nodes im Utility AI Graph Editor explizit in einer der Priorität absteigender Reihenfolge angeordnet, können Considerations, welche eine hohe Rechenleistung benötigen, am Ende einer Action ausgewertet werden. Zum Beispiel sollte aufwändiges Suchen von Objekten im Level eine niedrige Priorität erhalten. Die Priorität kann, wie im Behavior Tree, durch die Anordnung der Nodes von links nach rechts erzeugt werden, wobei sich der Node mit der höchsten Priorität links befindet. Diese Prioritäten können durch eine entsprechende Zahl an den Consideration-Nodes angezeigt werden, wie in Abbildung 36 zu sehen.

In der eigenen Implementierung wird bei jedem Aufruf eines Consideration-Nodes der Eingabewert von Neuem berechnet. Das ist auch bei derselben Consideration der Fall. Dies sollte vermieden werden, da die Eingabewerte nicht verändert werden. Durch die Parametrisierung werden die Eingabewerte der Consideration lediglich unterschiedlich interpretiert. Damit die Eingabewerte und die Auswertung dieser Consideration voneinander getrennt werden kann, muss eine Liste implementiert werden, welche die Logik der Informationsbeschaffung einmalig speichert. Anstatt diese Logik mehrmals aufzurufen, können die benötigten Eingabewerte in einem Auswertungszyklus einmalig berechnet und anschließend mehrmals von verschiedenen Considerations referenziert werden. Die Consideration *OwnHealth* wird zum Beispiel für beide Actions *MeleeAttack* und *Flee* benötigt. Die Consideration wird bei der Berechnung der jeweiligen Action Scores aufgerufen, wobei dadurch zweimal die Lebenspunkte der KI in Erfahrung gebracht werden. Der einzige Unterschied in der Parametrisierung der Nodes ist lediglich die Invertierung des berechneten Scores. Wird die Beschaffung der Informationen nur einmalig ausgeführt und referenziert, so kann dies bei aufwändigen Eingabewerten zu einer Reduzierung der Rechenleistung führen.

5.3 Verbesserungen des Utility AI Graphen

Um das Erstellen einer Utility AI über den Utility AI Graph Editor zu erleichtern, können am User Interface des Editors verschiedene Änderungen vorgenommen werden. In der aktuellen Implementierung wird im Graphen lediglich der Name der Action, Consideration oder Decision angezeigt. Die Nodes könnten zur verbesserten Übersicht die verschiedenen Optionen der Node-Typen als Icons darstellen. Zum Beispiel könnte die Gewichtung einer Action durch ein Gewicht-Symbol seitlich des Nodes dargestellt werden (siehe Abbildung 36). Wird zum Debugging eine

Action deaktiviert, kann darauf im Graphen durch ein Ausrufezeichen aufmerksam gemacht werden.

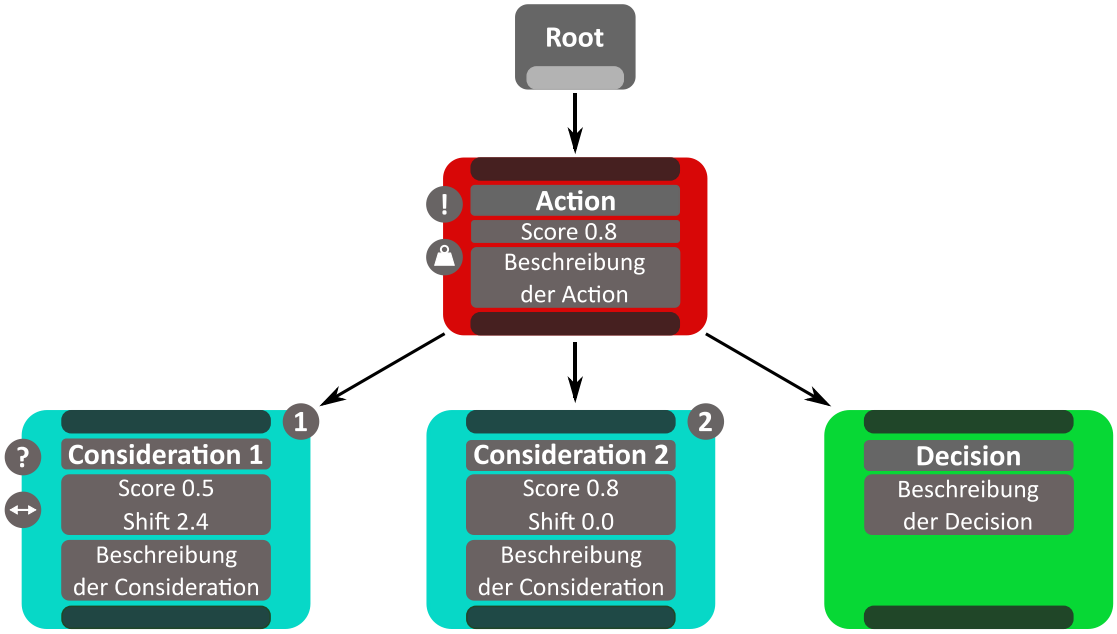

Abbildung 36: Darstellung der verbesserten Action-, Consideration- und Decision-Nodes. Neben dem Namen der Nodes wird eine Beschreibung anzeigt. Symbole zur Darstellung von konfigurierter Gewichtung, Deaktivierung, Invertierung und des *RandomShifts* wurden hinzugefügt.

Die Option *InvertScore* eines Consideration Scores kann durch einen doppelseitigen Pfeil dargestellt werden und *RandomShift* durch ein Fragezeichen. Bei der Einführung einer expliziten Priorisierung der Consideration Scores bestünde die Notwendigkeit die Consideration-Nodes eines Action-Nodes zu nummerieren. Neben der Darstellung wichtiger Parameter, wäre die direkte Anzeige der Beschreibung von Nodes im Graphen sinnvoll. Damit ist auf einen Blick ersichtlich, welche Aufgabe der Node übernehmen soll. Das Anzeigen der Action oder Consideration Scores direkt im Graphen wäre für das Debugging eine große Erleichterung. Anstatt jeden einzelnen Node anklicken und in das *Property Panel* schauen zu müssen, könnten alle Scores in Echtzeit im Graph angezeigt werden. Da jeder Utility AI Graph einen Root-Node benötigt, könnte dieser standardmäßig beim Erstellen eines Utility AI Assets erzeugt werden. Dadurch muss dieser nicht jedes Mal neu erstellt werden.

Neben Verbesserungen in Bezug auf die UI des Graph Editors und Optimierungen der Auswertung der Utility AI fehlt dem Utility AI Graphen in der Implementierung die Wiederverwendbarkeit des Utility AI Graph Assets. Nodes können innerhalb des Graphen mitsamt ihrer Konfiguration dupliziert und auch in einen anderen Utility AI Graphen kopiert werden. Um die Modularität der Nodes zu erhöhen, könnte ein Ansatz ähnlich der Erstellung von Nodes im Behavior Tree Graphen verfolgt werden. In diesem bestehen die Nodes nicht aus generischen Nodes, die durch eine Parametrisierung des Nodes selbst in die verschiedenen Node-Typen eingeteilt werden, sondern werden von einer speziellen Klasse abgeleitet, die den Node-Typ definiert. Wird beispielsweise ein neuer Task-Node benötigt, welcher ein individuelles Verhalten definiert, wird

dieser von der Unreal Engine 4 bereitgestellten Klasse *BTTask_BlueprintBase* abgeleitet. Nachdem der Node erstellt ist, wird dieser automatisch zur Auswahl im Rechtsklick-Menü bei der Erstellung neuer Nodes hinzugefügt, wie in Abbildung 37 dargestellt. In diesem Node würde die Logik, unabhängig von den anderen eingesetzten Nodes und vom AI Controller, implementiert werden. Die Logik zur Beschaffung von Informationen und die Ausführung von Verhalten befinden sich in der aktuellen Implementierung der Utility AI in der Unreal Engine 4 komplett in der Utility AI Controller Klasse. Einerseits kann dadurch auf die Logik in einem zentralen Ort zugegriffen werden, andererseits ist dieser Ansatz wenig modular. Bei einer Erstellung eines anderen Utility AI Controllers mit unterschiedlichem Verhalten müssen Teile der Informationsbeschaffung einer Consideration kopiert und im neuen AI Controller wieder eingebunden werden. Damit dies nicht der Fall ist, könnte die Informationsbeschaffung in die zugehörigen Nodes verlagert werden. Dadurch erlangt der Consideration-Node Unabhängigkeit vom Utility AI Controller. Können Action- oder Consideration-Nodes als eigener Asset-Typ gespeichert werden, würden diese in verschiedenen Utility AI Graphen nutzbar sein. Soll zu einem späteren Zeitpunkt die Logik der Informationsbeschaffung geändert werden, so könnte diese Änderung in dem Asset selbst vorgenommen werden und müsste nicht in jedem Utility AI Controller, welcher diese Consideration verwendet, angepasst werden. Durch das Speichern der Consideration-Nodes als Asset kann eine Bibliothek an verschiedenen Considerations erstellt werden, die für verschiedene Projekte und KIs genutzt werden können.

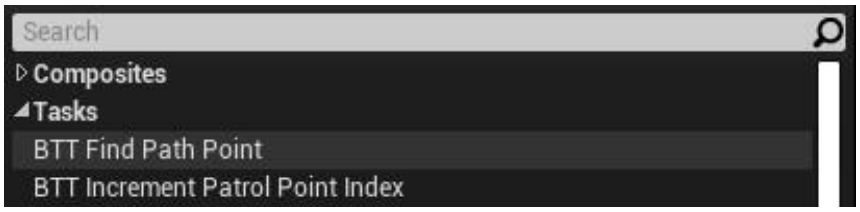

Abbildung 37: Beim Erstellen neuer Nodes werden im Behavior Tree Graph Editor Tasks zur Auswahl angezeigt. Die in der Abbildung zu sehenden Tasks wurden im Rahmen des in Abschnitts 4.3 beschriebenen Verhaltens individuell erstellt.

5.4 Ausbau des Debuggings

Im Rahmen dieser Bachelorarbeit wurden 11 verschiedene Verhalten durch die entsprechende Anzahl von Actions implementiert. Die Anzeige zum Debugging der Action Scores wird über ein spezielle Benutzeroberfläche erstellt. Dafür müssen jedoch die Werte zwischen dem Utility AI Controller und der UI[32] Klasse ausgetauscht werden. Das Bereitstellen der Scores für die UI und das Erstellen neuer UI Elemente bei neuen Actions ist zeitaufwändig und nicht automatisiert. Daher ist eine Implementierung angebracht, welche beim Hinzufügen eines Nodes im Utility AI Graph Editor automatisch eine Liste aller Actions oder Considerations erstellt, welche in Tabellenform angezeigt werden kann. Abbildung 38 zeigt die Tabelle der Scene Rendering Stats, in welcher die verschiedenen Statistiken zum Rendern einer Szene angezeigt werden. Neben den

[32] User-Interface

Werten in Millisekunden werden auch Werte als Balken visualisiert. Die Darstellungsweise in einer solchen Form eignet sich zur Übersicht der gesamten Action oder Consideration Scores.

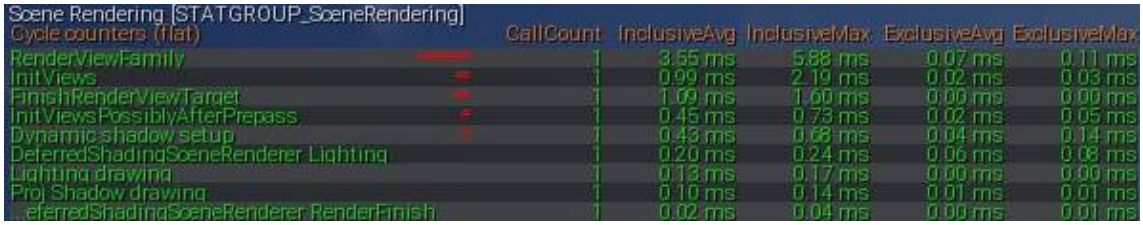

Abbildung 38: Die mögliche Darstellungsweise der Action oder Consideration Scores in Form einer Tabelle. Diese Tabelle bildet Statistiken der gerenderten Szene ab.

5.5 Ausbau des Verhaltens

Um die Auswahl der Verhalten zu verbessern und noch präziser zu gestalten, bietet sich die Einführung weiterer Considerations an. Mögliche Überlegungen, die den verschiedenen Actions hinzugefügt werden können, sind in den verschiedenen implementierten Verhalten des Abschnitts 3.3 beschrieben. Durch den Ausbau der Considerations kann die Auswahl zwischen den verschiedenen Actions noch genauer bewertet werden.

Neben einer Verbesserung der Auswahl der verschiedenen Verhalten können dem Level weitere Verhalten hinzugefügt werden, um die Varianz dieser zu erhöhen und den Tagesablauf der KI interessanter zu gestalten. Ein mögliches neues Verhalten ist das Entfachen von Feuern entlang der Mauern, wenn im Spiel die Sonne untergeht und es im Level dunkel wird. Dabei würde die KI zu den Kohlepfannen laufen und eine Animation ausführen, um ein Feuer zu entfachen. Um die Interaktion zwischen den KI Charakteren zu erhöhen, könnte ein Verhalten hinzugefügt werden, durch welches sich die KI zu zufälligen Zeitpunkten im Level trifft und eine Unterhaltung führt. Neben Essen und Schlafen kann die KI zudem zu einem zufälligen Zeitpunkt das Bedürfnis nach einem Toilettengang verspüren. Dafür läuft die KI zu einem definierten Punkt im Level und führt eine entsprechende Animation aus.

Die Nutzung von Utility AI ist intuitiv, da die Überlegungen, die zur Auswahl eines Verhaltens führen, durch logische und nachvollziehbare Considerations getroffen werden können, anstatt durch verschiedene Regeln, wie sie in einem Behavior Tree existieren, verkompliziert zu werden. Außerdem hat die Erstellung eines Utility AI Graphen, im Gegensatz zum Behavior Tree, wenige Regeln, bei welchem Prioritäten eine entscheidende Rolle besitzen. Selektoren und Sequenzen werden genutzt, um Entscheidungen zu treffen. Um damit komplexe Entscheidungen zu treffen, muss unter Umständen ein großer Teilbaum erstellt werden. Wird ein Decorator in einem Behavior Tree eingesetzt, um Entscheidungen zu treffen, so wird dieser meist individuell angelegt. Dieser eignet sich daher nicht zur Wiederverwendung der Auswahl ähnlicher Entscheidungen, sondern muss dupliziert und abgeändert werden. Soll der KI ein neues Verhalten zugewiesen werden, so muss im Utility AI Graphen lediglich an einer beliebigen Stelle eine neue Action mit den zugehörigen Considerations und dem Decision-Node hinzugefügt werden. Bei einem Behavior Tree müssen dagegen je nach der Priorität, Abbruchkriterien und der Entscheidung, die zur Auswahl des Verhaltens führt, Teile des Behavior Trees neu strukturiert werden. Durch eine Wiederverwendung gespeicherter Consideration-Node Assets kann innerhalb kurzer Zeit eine Action erstellt werden, welche auch komplexe Entscheidungen abbilden können.

Mögliche Folgeprojekte beinhalten eine Implementierung der in diesem Abschnitt vorgeschlagenen Verbesserungen und eine Verknüpfung des Utility AI Graphen mit dem Behavior Tree der Unreal Engine 4. Auf diese Verknüpfung wird in Abschnitt 5.6 genauer eingegangen.

5.6 Verknüpfung des Utility AI Graphen mit Behavior Trees

Anstatt den Behavior Tree, wie in Abschnitt 4.4 beschrieben, durch die Erweiterung spezieller Selektor und Task-Nodes zur Evaluation von Actions und Considerations zu nutzen, kann auch ein alternativer Ansatz verfolgt werden. Während ein Behavior Tree für die Ausführung von Verhalten geeignet ist, liegt die Stärke des Utility AI Systems in der Auswahl eines Verhaltens. Da mit dem Utility AI Graph Editor bereits ein Editor zur Verfügung steht, bietet sich eine Verknüpfung der beiden Systeme an, wodurch die Vorteile beider Methoden genutzt werden können. So kann einerseits die verbesserte Auswahl von Verhalten durch die Utility AI und andererseits eine geordnete und übersichtliche Erstellung von Verhaltensabläufen ohne komplexe Strukturen zum Treffen von Entscheidungen durch einen Behavior Tree kombiniert werden. Durch die Trennung der beiden Aufgabenbereiche entsteht eine bessere Wiederverwendbarkeit, da beide Arten, eine KI zu erstellen, modularer gestaltet werden können. Die Verknüpfung der Utility AI Graphen und des Behavior Trees ermöglicht außerdem eine Integration von neuem Verhalten in kurzer Zeit, da der bestehende Behavior Tree nicht umstrukturiert werden muss.

In diesem Abschnitt soll der theoretische Ansatz zur Verknüpfung des Utility AI Graphen mit dem Behavior Tree der Unreal Engine 4 erläutert werden. Da jeder Utility AI Graph nicht nur die verschiedenen Considerations zur Auswahl einer Entscheidung enthält, sondern auch berechnete Scores speichert, muss jede KI über einen individuellen Utility AI Graph verfügen. Durch die Nutzung des Blackboards kann der Graph als individueller Wert gespeichert werden. Damit der Utility AI Graph korrekt initialisiert werden kann, müsste ein besonderer Task-Node im Behavior Tree erstellt werden, welcher diese Aufgabe übernimmt. Wie in Abbildung 39 dargestellt, könnte dieser Task-Node *BTT_SetupUtilityAI* genannt werden. Da das Setup der Utility AI nur einmal durchgeführt werden soll, besitzt der Task-Node einen Decorator, der eine einmalige Ausführung des Nodes sicherstellt. Dieser Decorator würde prüfen, ob die Variable *IsUAISetup* den Wert *true* oder *false* besitzt. Da die Bedingung dabei invertiert wird, würde der Decorator Node im ersten Durchlauf *true* zurückgeben und lässt dadurch die Ausführung des Task-Nodes zu. Dieser würde, nach dem Setup der Utility AI, den Wert von *IsUAISetup* auf *true* setzen. Bei allen weiteren Ausführungen des Behavior Trees würde der Decorator durch die Invertierung *Failure* zurückgeben. Durch die Nutzung eines Selektor-Nodes könnte der rechte Teilbaum ausgeführt werden. Für die Funktionen zur Auswertung des Utility AI Graphen, *RunUtilityAI*, *ImplementConsiderationScores* und *SetConsiderationScore,* wird der Behavior Tree Service *BTS_RunUtilityAI* erstellt. Im Blackboard würde das Ergebnis des Services, welches der ausgewählten Decision entspricht, gespeichert. Über den Selektor-Node, welcher den Service enthält, könnten anschließend die Teilbäume der im Utility AI Graphen definierten Verhalten ausgeführt werden. In den Kind-Knoten des Selektor-Nodes wird geprüft, welche Decision ausgewählt werden würde. Dafür müsste im Decorator-Node *BTD_CheckDecision* der Name der Decision manuell

eingetragen werden, damit deren Verhalten bei einer Auswahl ausgeführt wird. Als Beispiel wurden dabei, wie in Abbildung 39 dargestellt, die Namen *Eat* und *Sleep* gewählt. Je nachdem, welches Verhalten gewählt und wie dieses im Behavior Tree implementiert wird, können sich die Kind-Knoten des Service-Selektors unterscheiden. Wichtig wäre hierbei lediglich der Abgleich des im Blackboard gespeicherten Namens der Decision. Die in Abbildung 39 dargestellten implementierten Verhalten sind beispielhaft und erfüllen nicht die tatsächliche Logik der beiden Verhaltensabläufe.

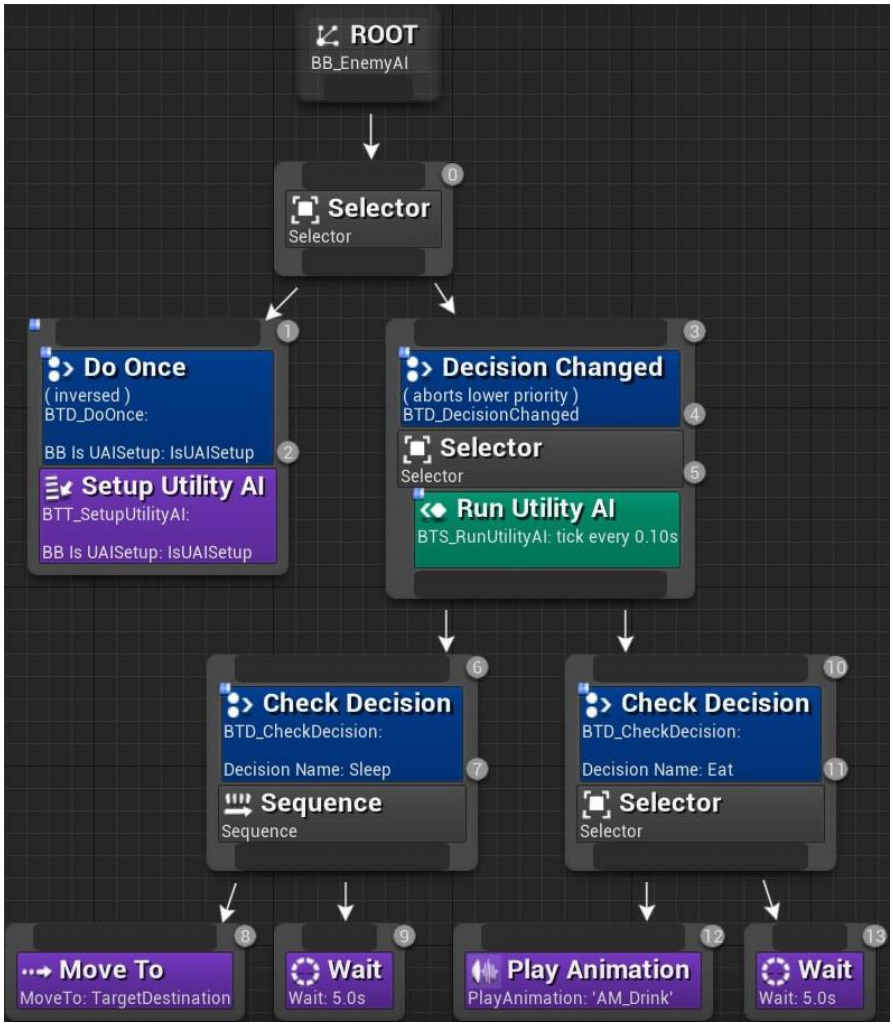

Abbildung 39: Darstellung der möglichen Behavior Tree Struktur, verknüpft mit der Utility AI Graph Auswertung.

Sobald ein neues Verhalten durch den Service ausgewählt werden würde, sollte der Teilbaum der aktuell ausgeführten Decision abgebrochen werden, damit entsprechend auf die Verhaltensänderung reagiert werden könnte. Der Abbruch, in Abbildung 39 *aborts lower priority* genannt, könnte durch den Decorator-Node *BTD_DecisionChanged* ausgelöst werden und würde bei der Änderung einer Decision aktiviert werden.

Durch die Verknüpfung des Utility AI Graphen mit dem Behavior Tree wird die Implementierung und die Auswahl der verschiedenen Verhalten getrennt. Dadurch kann die Erstellung einer KI modularer und übersichtlicher gestaltet werden.

6 Zusammenfassung

Das Utility AI System kann den relativen Nutzen einer Aktion oder eines Verhaltens bewerten. Dabei erlaubt dieses System den Vergleich zwischen unterschiedlichen Konzepten und Informationstypen, die in einer bestimmten Umgebung auftreten können. Die Aktionen werden mithilfe eines Scores, der sich in einem Intervall von 0 bis 1 befindet, bewertet. Besitzt eine Aktion, im Utility AI System Action genannt, einen Score von 1, so ist diese passend zur aktuellen Situation, während ein Score von 0 bedeutet, dass die Action im Moment der Auswahl ungeeignet ist. Das Utility AI System vergleicht kontinuierlich alle verfügbaren Aktionen, die eine KI ausführen kann und wählt, abhängig von der Auswahlmethode, eine optimal zur Situation passende Aktion aus. Mit der benötigten Auswertung soll erreicht werden, die Rechenleistung möglichst gering zu halten und dabei jedoch trotzdem eine plausible Auswahl treffen zu können. Um den Score einer Aktion zu berechnen, werden verschiedene Überlegungen, sogenannte Considerations, getroffen und kombiniert. Diese Considerations beziehen Informationen aus der Spielwelt, um die aktuelle Situation zu beurteilen. Von zentraler Bedeutung der Verwertung dieser Informationen sind Response Curves. Durch diese lässt sich definieren, wie die Werte interpretiert werden sollen. Zum Einsatz kommen dabei unter anderem lineare, quadratische und logistische Funktionen. Wurde eine passende Aktion ausgewählt, so wird durch sogenannte Decisions das mit der Aktion verbundene Verhalten ausgeführt.

Zur Implementierung des Utility AI Systems in der Unreal Engine 4 wurden zwei zentrale Komponenten in der Engine implementiert. Eine dieser Komponenten, der sogenannte Utility AI Graph, ist für die Definition der verschiedenen Actions, Considerations und Decisions verantwortlich. Diese Komponenten können über einen speziellen Editor visuell definiert und strukturiert werden. Der Editor erlaubt außerdem die Parametrisierung benötigter Eigenschaften. Die zweite Komponente, der Utility AI Controller, ist für die Auswertung des Utility AI Graphen, für die Berechnung der Scores und für die Auswahl einer passenden Aktion verantwortlich. Außerdem werden im Utility AI Controller alle möglichen Verhaltensabläufe, welche eine KI ausführen kann, definiert.

Um die Implementierung der Utility AI zu testen und zu demonstrieren, wurde eine KI erstellt, welche zwischen 11 verschiedenen Verhalten eine zur Situation passende Auswahl treffen kann. Damit die verschiedenen Aktionen und Verhalten möglichst optimal bewertet werden können, wurden viele verschiedene Considerations erstellt und individuell parametrisiert.

Um mögliche Fehlerquellen zu finden und auszuschließen, wurden spezielle Anzeigen entwickelt, welche die Scores und sonstige nützliche Informationen darstellen können.

Da Behavior Trees in der Spieleindustrie weit verbreitet sind und in der Unreal Engine 4 zur Erstellung von KI genutzt werden, wurde im Rahmen der Bachelorarbeit das Konzept der Behavior Trees dem Konzept der Utility AI gegenübergestellt. Dabei wird auf eine mögliche Einbindung, aber auch auf eine Verknüpfung der Utility AI in Behavior Trees eingegangen.

Durch die Implementierung des Utility AI Systems wurden die Schwächen und Stärken dessen ausgearbeitet. Verbesserungen bezüglich der Benutzeroberfläche, wie zum Beispiel die Einführung von Symbolen zur Anzeige bestimmter aktiver Parameter, können dabei in Folgeprojekten umgesetzt werden. Auch die Auswertung und Berechnung kann weiter optimiert werden.

7 Literaturverzeichnis

[1] Looman, Tom (2018): Journey into Utility AI with Unreal Engine 4 - Tom Looman. https://www.tomlooman.com/journey-into-utility-ai-ue4/. Abgerufen am 11.09.2019.

[2] jinyuliao: GenericGraph. https://github.com/jinyuliao/GenericGraph. Abgerufen am 15.08.2019.

[3] Mark, Dave (2013): Architecture Tricks: Managing Behaviors in Time, Space, and Depth. Infinite Axis Utility System. https://www.gdcvault.com/play/1018040/Architecture-Tricks-Managing-Behaviors-in. Abgerufen am 12.08.2019.

[4] Mark, Dave, Dill, Kevin (2010): Improving AI Decision Modeling Through Utility Theory. https://www.gdcvault.com/play/1012410/Improving-AI-Decision-Modeling-Through. Abgerufen am 12.08.2019.

[5] Lewis, Mike, Mark, Dave (2015): Building a Better Centaur: AI at Massive Scale. https://www.gdcvault.com/play/1021848/Building-a-Better-Centaur-AI.

[6] Rabin, Steve, Graham, David, Merill, Bill (Hrsg) (2014): Game AI Pro. Collected wisdom of game AI professionals. CRC Press/A K Peters, Boca Raton, Fla.

[7] Epic Games (15.08.2019): Blueprints Visual Scripting. https://docs.unrealengine.com/en-US/Engine/Blueprints/index.html. Abgerufen am 03.09.2019.

[8] Epic Games (15.08.2019): Balancing Blueprint and C++. https://docs.unrealengine.com/en-US/Resources/SampleGames/ARPG/BalancingBlueprintAndCPP/index.html. Abgerufen am 03.09.2019.

[9] Epic Games (15.08.2019): Gameplay Debugger. https://docs.unrealengine.com/en-US/Gameplay/Tools/GameplayDebugger/index.html. Abgerufen am 18.08.2019.

[10] Epic Games (09.08.2019): AIController. https://docs.unrealengine.com/en-US/Gameplay/Framework/Controller/AIController/index.html. Abgerufen am 15.08.2019.

[11] Rabin, Steve, Graham, David (05.11.2017): Game AI Pro- Demo code zu " An Introduction to Utility Theory". Algorithmus in Klasse "AiActor" in Funktion "ChooseNextAction". http://www.gameaipro.com/. Abgerufen am 16.08.2019.

[12] Epic Games (25.07.2019): Behavior Trees. https://docs.unrealengine.com/en-US/Engine/AI/BehaviorTrees/index.html. Abgerufen am 14.08.2019.

[13] Crytek: Modular Behavior Tree - CRYENGINE Programming - Documentation. https://docs.cryengine.com/display/CEPROG/Modular+Behavior+Tree. Abgerufen am 18.08.2019.

[14] Rabin, Steve (2017): Game AI Pro 3. Collected Wisdom of Game AI Professionals. Chapman and Hall/CRC, Milton.

[15] Epic Games (25.07.2019): Environment Query System. https://docs.unrealengine.com/en-US/Engine/AI/EnvironmentQuerySystem/index.html. Abgerufen am 15.08.2019.

[16] Epic Games (25.07.2019): Behavior Tree Node Reference: Services. https://docs.unrealengine.com/en-US/Engine/AI/BehaviorTrees/NodeReference/Services/index.html. Abgerufen am 10.09.2019.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die aus fremden Quellen direkt oder indirekt übernommenen Gedanken als solche kenntlich gemacht habe.
Die Arbeit habe ich bisher keinem anderen Prüfungsamt in gleicher oder vergleichbarer Form vorgelegt. Sie wurde bisher nicht veröffentlicht.

_____ _____

Ort, Datum Unterschrift

Ermächtigung

☐ Hiermit ermächtige ich/wir die Hochschule Kempten zur Veröffentlichung einer Kurzzusammenfassung sowie Bilder/Screenshots und ggf. angefertigte Videos meiner studentischen Arbeit z. B. auf gedruckten Medien oder auf einer Internetseite der Hochschule Kempten zwecks Bewerbung des Bachelorstudiengangs „Game Engineering“ und des Masterstudiengangs „Game Engineering und Visual Computing“.
Dies betrifft insbesondere den Webauftritt der Hochschule Kempten inklusive der Webseite des Zentrums für Computerspiele und Simulation. Die Hochschule Kempten erhält das einfache, unentgeltliche Nutzungsrecht im Sinne der §§ 31 Abs. 2, 32 Abs. 3 Satz 3 Urheberrechtsgesetz (UrhG).

_____ _____

Ort, Datum Unterschrift