HOCHSCHULE HEILBRONN

Bachelor Thesis (SPO NUMMER)

Physik basierter Charaktercontroller mit Unity Machine Learning

Simon Grözinger*

1. August 2024

Eingereicht bei Prof. Dr. Tim Reichert

*205047, sgroezin@stud.hs-heilbronn.de

Inhaltsverzeichnis

Abkürzungsverzeichnis

ABK: ABKÜRZUNG

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

Machine Learning Modelle bieten neue Möglichkeiten den Prozess der Charakter animation zu erleichtern. In der Thesis soll ein Ansatz anhand bestehender Literatur und Beispiele erforscht werden, in dem Spielcharaktere physikalisch mit Rigidbodies und Joints simuliert und mit Hilfe von Machine Learning trainiert werden, um möglichst realistische Bewegung nachahmen zu können.

2 Grundlagen

Dieses Kapitel behandelt die Grundlagen der verwendeten Technologien, Paketen und Unity Komponenten.

2.1 Verstärkendes Lernen

Der Begriff 'Verstärkendes Lernen' beschreibt eine Art von Problemstellung und die dafür geeigneten Problemlösungsmethoden im Bereich des maschinellen Lernens. Die grundlegenden Bestandteile einer Trainingsumgebung sind der Agent und die Umgebung. Die Umgebung kann sich unabhängig vom Agenten verändern, jedoch hat der Agent durch seine Aktionen Einfluss auf die Umgebung.

In vielerlei Hinsicht ist dieser Prozess mit dem Lernvorgang von Menschen vergleichbar. Ein Baby lernt das Krabbeln ohne direkte Anweisungen. Es bewegt sich und agiert in der Umgebung und beobachtet, wie diese auf sein Verhalten reagiert. Der daraus resultierende eigene Gefühlszustand und externe Einflüsse werden als Rückmeldung evaluiert. Durch diese Rückmeldung wird das Verhalten entweder antrainiert oder abtrainiert. Auf dieselbe Art lernt der Agent beim verstärkenden Lernen in jedem Zustand, die Aktion auszuführen, um die Belohnung zu maximieren. Die Belohnungen können dabei positiv oder negativ sein. Im Fall des Babys sind die Belohnungen Faktoren wie Schmerz, Hunger, Müdigkeit, gestillte Neugier oder Lob von Mitmenschen. Der Agent hingegen erhält eine numerische Belohnung.[3]

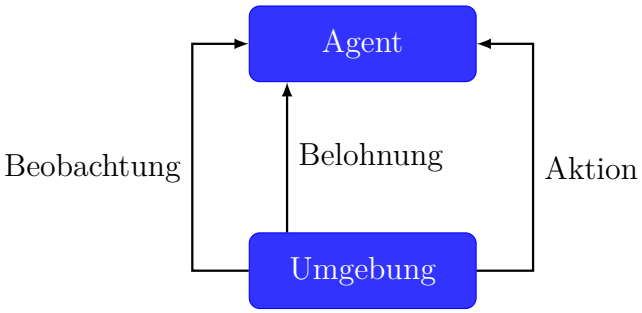

Abbildung 2.1: Verstärkendes Lernen Ablauf

Die Abbildung 2.1 zeigt die Verbindungen zwischen dem Agenten und der Umgebung. Der Agent erhält als Input einen Zustand oder häufig einen Teilzustand der Umgebung und reagiert darauf mit einer Aktion. Dieser Zyklus kann je nach Problem in unterschiedlichen Intervallen durchlaufen werden. Bei kontinuierlichen Kontrollproblemen werden Aktionen meist in regelmäßigen Intervallen angefragt. Bei Problemen mit einem festgelegten Ablauf kann dieser Vorgang jedoch auch nur in einer bestimmten Phase stattfinden.

2.2 ML-Agents

Das Unity ML-Agents Toolkit ist ein Open-Source-Projekt, welches maschinelle Lernalgorithmen und Funktionen für die Verwendung mit der Spieleumgebung Unity implementiert. Es beinhaltet Komponenten um eine Unityumgebung als Umgebung für verstärkendes Lernen zu konfigurieren.[1]

2.2.1 Aufbau

Das Toolkit ist in zwei Teile unterteilt (siehe Abbildung 2.2). Für die Unity-Integration ist das Paket com.unity.ml-agents aus dem Unity Asset Store zuständig. Das eigentliche Training mit den maschinellen Lernalgorithmen findet jedoch in einer separaten Python-Umgebung statt. Für die Kommunikation zwischen den beiden Bereichen verwendet das ML-Agents Toolkit eine gRPC-Netzwerkkommunikation.[1]

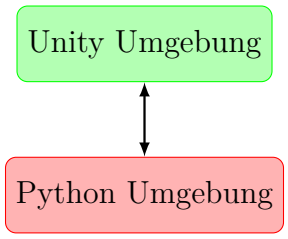

Abbildung 2.2: Unity ML-Agents Aufbau

Um eine Szene in Unity für das verstärkende Lernen zu nutzen, muss die Szene mindestens einen Agenten beinhalten. Jeder Agent referenziert ein Verhalten. Ein Verhalten kann eins von drei verschiedenen Modi verwenden. In Abbildung 2.3 werden drei Agenten mit den unterschiedlichen Verhaltens Modi dargestellt.

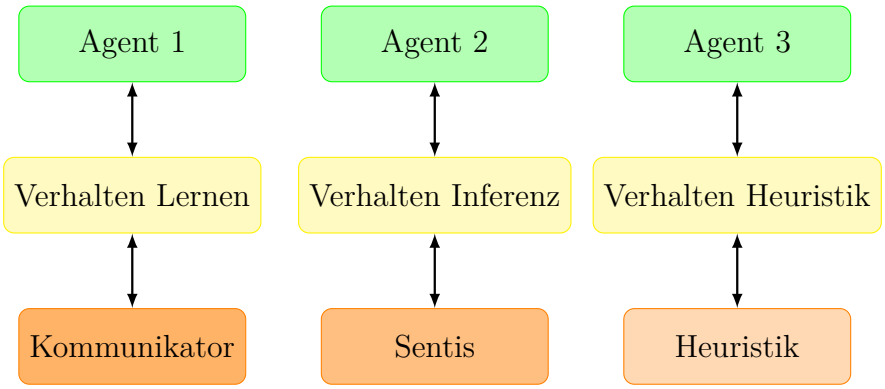

Abbildung 2.3: Unity ML-Agents Aufbau Unity Umgebung

Das Verhalten bildet die Zuweisung von Beobachtung auf eine Aktion in der Unity Umgebung ab. Im Lernmodus nutzt es den Kommunikator, um in der Python Umgebung basierend auf der Beobachtung und der aktuellen Strategie eine Aktion auszuwählen. Im Inferenzmodus wird ein bereits trainiertes Modell mit dem Unity Sentis-Paket ausgeführt. Der Heuristikmodus wird meist zum Testen oder zum Aufzeichnen von Demonstrationen für das Imitationslernen verwendet. Die Heuristik verwendet fest kodierte Anweisungen, um beispielsweise die Aktionen über Tastatureingaben zu steuern.

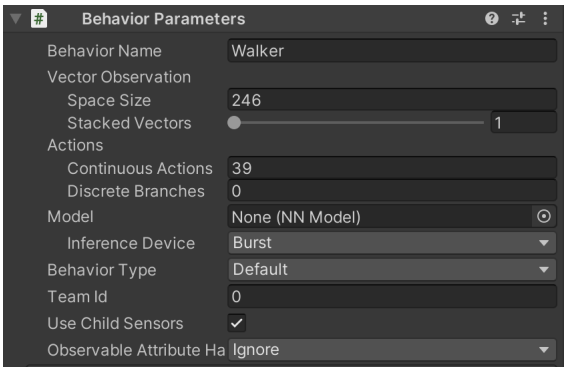

Abbildung 2.4: Unity ML-Agents Verhalten Parameter Komponente

- Behaviour Name: Name des Verhaltens / wird in Trainer Konfiguration referenziert
- Space Size: Anzahl an Beobachtungen / Inputknoten für NN
- Continuous Actions: Anzahl an Aktionen / Outputknoten von NN
- Model: Referenz auf bereits trainiertes Modell zur Verwendung in Inferenz
- Behaviour Type: Lernmodus Default = Lernen, Heuristic, Inferenz

Die Agent-Komponente bildet die Grundlage für alle Implementierungen. Sie bietet abstrakte Funktionen für die Initialisierung, den Start einer Episode, das Erfassen des Zustands der Umgebung sowie das Ausführen von Aktionen. Durch die Implementierung dieser Funktionen können unterschiedlichste Agenten entwickelt und trainiert werden. Die Beobachtungen des Agenten können auf zwei Arten erstellt werden. Beobachtungen basierend auf Raycasts sowie Kamerabildern werden mit separaten Komponenten erstellt. Beobachtungen aus Zahlenwerten sowie Vektoren und Quaternionen können jedoch auch direkt über die Beobachtungs-Funktion im Agenten der Beobachtung angehängt werden.

Abbildung 2.5: Unity ML-Agents Agenten Komponente

Abbildung 2.5 zeigt die Basiskomponente des Agenten. Ohne das Überschreiben der Funktionen ist die Agentenklasse jedoch ohne Funktion. Die genauen Methoden zur Implementierung eigener Agentenklassen werden im folgenden Abschnitt behandelt. Das einzige Feld zur Konfiguration ist "Max Step", welches die maximale Anzahl der Schritte innerhalb einer Episode festlegt.

```
1 public override void CollectObservations(VectorSensor sensor)
2 {
3     sensor.AddObservation(floatObservation);
4 }
5
6 public override void OnActionReceived(ActionBuffers actionBuffers)
7 {
8     var continuousActions = actionBuffers.ContinuousActions;
9     movement.x += continuousActions[0]
```

```
10      movement.y += continuousActions[1]
11  }
12
13  public virtual void FixedUpdate()
14  {
15      AddReward(floatReward);
16  }
```

Listing 2.1: Agent Funktionen

In der CollectObservations-Methode wird festgelegt, welche Daten dem Agent für das Training bereitgestellt werden (siehe Listing 2.1 Zeile 1-3). CollectObservations wird für jede angefragte Entscheidung ausgeführt und das Ergebnis an das NN-Modell oder den Python Trainer übergeben.

Wenn eine Entscheidung angefragt wurde und das NN-Modell ein Ergebnis liefert, wird dieses hier von numerischen Werten in Aktionen umgewandelt. In Listing 2.1 Zeile 6-11 wird gezeigt, wie die Aktion in X- und Y-Bewegung umgesetzt wird.

Im Beispielcode in Listing 2.1 Zeile 13-16 wird eine Belohnung in jedem FixedUpdate vergeben, und zwar über die AddReward Methode, die auch Teil der Agentenkomponente ist. Die Belohnung kann aber an jeder Stelle im Code vergeben werden, der Code dient hier nur als ein Beispiel.

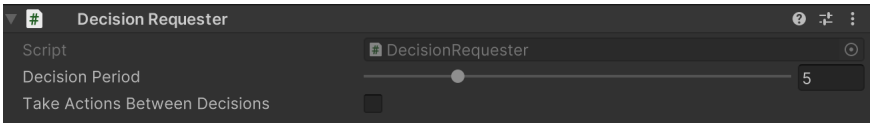

Abbildung 2.6: Unity ML-Agents Entscheidung Anfragen Komponente

Die Komponente in Abbildung 2.6 fragt in regelmäßigen Abständen Entscheidungen an. Das bedeutet, es wird eine Beobachtung erstellt und darauf basierend eine Aktion über das Verhalten ausgewählt. Die “Decision Period“ gibt an, in welchem Intervall der Agent eine Entscheidung treffen soll. Das Kontrollkästchen “Take Actions Between Decisions“ gibt an, ob der Agent die ausgewählte Aktion wiederholen soll, bis die nächste Aktion ausgewählt wurde.

2.2.2 Training

Beim Starten der Python-Trainingsumgebung mit dem Befehl “mlagents-learn“ wird zu Beginn eine Instanz der Python-API erstellt. Die Python-API ist eine Schnittstelle für die Interaktion mit Unity ML-Agents-Umgebungen. Sobald die Konfigurationsparameter von der Unity-Instanz an die Python-Umgebung übertragen wurden, wird basierend darauf ein Python-Trainer erstellt. Über die Python-API kann der Python-Trainer auf Beobachtungen zugreifen, Aktionen ausführen und anhand der Belohnungssignale, die das Ergebnis der Aktionen bewerten, die Gewichtung der neuronalen Netze anpassen, um das Verhalten des Agenten zu optimieren. Dieser Prozess ermöglicht es, durch wiederholtes Training und Anpassung des Modells intelligente Agenten zu entwickeln, die komplexe Aufgaben bewältigen können.

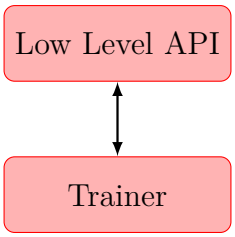

Abbildung 2.7: Unity ML-Agents Aufbau Python Umgebung

Die Trainingskonfigurationsdatei (siehe Listing 2.2) enthält mehrere Teile. Der Hyperparameter-Teil (Zeile 5-13) umfasst die Hyperparameter des Maschinellen Lernalgorithmus, welche die Lernrate, Batchgröße und andere wichtige Parameter für den Lernprozess festlegen. Danach folgt der Abschnitt network_settings (Zeile 14-18), der die Konfiguration des neuronalen Netzes festlegt. Anschließend werden im Bereich reward_signals (Zeile 19-22) die Konfigurationen für die Belohnungssignale festgelegt, die für die Bewertung der Aktionen des Agenten entscheidend sind. In (Zeile 23-27) werden die Frequenz für die Speicherung der Daten sowie der länge des Trainings festgelegt. Ganz am Ende der Konfigurationsdatei (Zeile 28-29) befinden sich noch Umgebungsparameter, die erweitert und während des Trainings ausgelesen werden können, um die Flexibilität und Anpassungsfähigkeit des Trainingsprozesses zu erhöhen.

```
1  {
2  behaviors:
3    Walker:
4      trainer_type: ppo
5      hyperparameters:
6        batch_size: 2048
7        buffer_size: 20480
8        learning_rate: 0.0003
9        beta: 0.005
10       epsilon: 0.2
11       lambd: 0.95
12       num_epoch: 3
13       learning_rate_schedule: linear
14     network_settings:
15       normalize: true
16       hidden_units: 256
17       num_layers: 3
18       vis_encode_type: simple
19     reward_signals:
20       extrinsic:
21         gamma: 0.995
22         strength: 1.0
23     keep_checkpoints: 5
24     checkpoint_interval: 5000000
25     max_steps: 30000000
26     time_horizon: 1000
27     summary_freq: 30000
28 environment_parameters:
29   environment_count: 100.0
30 }
```

Listing 2.2: Trainer Konfigurationsdatei

2.2.3 Auswertung

Um das laufende Training oder bereits abgeschlossene Trainingseinheit zu bewerten oder zu vergleichen, nutzt Unity ML-Agents Tensorboard. Tensorboard visualisiert die Metriken des Trainings in Zeitgraphen (siehe Abbildung 2.8). Der wichtigste Graph ist die gesammelte Belohnung, die ein Maß für den Erfolg des Agenten darstellt. Für Implementierungen mit frühem Stoppen ist die erreichte Episodenlänge ebenfalls sehr aussagekräftig, da sie anzeigt, wie lange der Agent in der Umgebung bestehen kann. Unter der Rubrik “Policy“ finden sich auch die Graphen, welche den Verlauf der Hyperparameter darstellen. Die linke Seitenleiste listet alle im aktuellen Verzeichnis gespeicherten Trainingseinheiten auf. Darüber können Trainingsets ausgewählt und anschließend in den Graphen durch unterschiedlich farbige Linien verglichen werden. Diese Visualisierungen ermöglichen eine detaillierte Analyse und den Vergleich verschiedener Trainingsläufe, was zur Optimierung des Trainingsprozesses beiträgt.

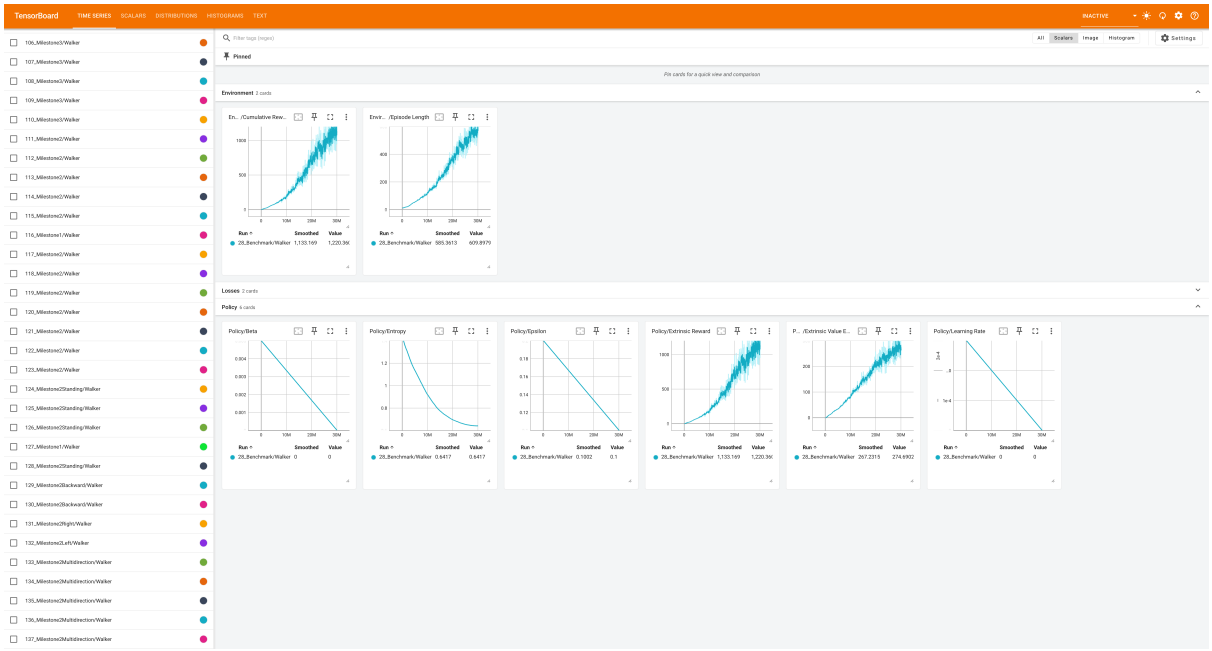

Abbildung 2.8: Tensorboard Ansicht

Nicht immer geben die vorgefertigten Graphen alle relevanten Informationen wieder. Um die erfassten Daten und damit die Graphen zu erweitern, bietet Unity ML-Agents über die Statistikrekorder die “Statistik Hinzufügen“ Funktion. Neu hinzugefügte Werte werden über die Episode und alle Umgebungen aggregiert. Als Aggregationsmethode kann zwischen Duchschnitt und letzten Wert entschieden werden. In Tensorboard werden die neuen Statistiken anschließend dargestellt.

Die letzte Instanz der Auswertung ist das Abspielen des trainierten Modells in der Unity-Umgebung. In den meisten Fällen ist die grafische Darstellung das zuverlässigste Medium, um das trainierte Modell zu bewerten, da sie ermöglicht, das Verhalten des Agenten in Echtzeit und in seiner tatsächlichen Umgebung zu beobachten. Dies bietet wertvolle Einblicke in die

Effektivität und Robustheit des Modells, die durch numerische Metriken allein nicht erfasst werden können.

2.3 Unity Physik

Unitys eingebaute Physik-Engine ermöglicht die realistische Berechnung von Kollisionen, Schwerkraft und anderen Kräften, was Entwicklern hilft, immersive und interaktive Umgebungen zu schaffen.

Die Festkörperkomponente (Rigidbody) erlaubt es, 3D-Objekte als nicht verformbare Einheiten innerhalb dieses Systems zu simulieren. Dies ist entscheidend für die Entwicklung realistischer physikalischer Interaktionen, wie z. B. das Bewegen von Objekten, die auf Kräfte, Drehmomente und Kollisionen reagieren.

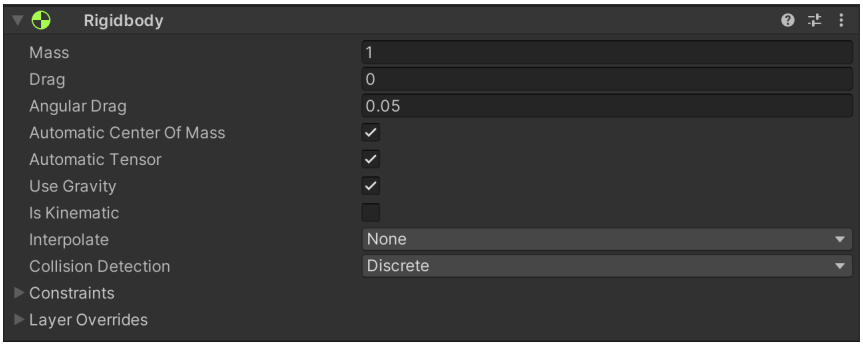

Abbildung 2.9: Unity ML-Agents Physik Festkörper

- Mass: gibt das Gewicht des Körpers an
- Drag: definiert den Geschwindigkeitsverlust eines Körpers in Bewegung durch Reibung, Luftwiederstand
- Angular Drag: definiert den Geschwindigkeitsverlust eines Körpers für Rotationsbewegung
- Collision Detection: legt fest wie Kollisionen berechnet werden (Akkurat/Leistung)

Um Kollisionen zwischen Objekten zu berechnen benötigen diese zusätzlich eine Kollisionskomponente. Komplexe 3D-Modelle können in der Kollisionsberechnung jedoch in ihrer direkten Form rechenintensiv sein. Zur Optimierung werden diese Modelle vereinfacht, indem sie durch geometrische Formen wie Kugeln, Kapseln oder Boxen dargestellt werden. Abbildung 2.10 zeigt die Unterschiedlichen Kollisionskomponenten in Unity. Abbildung 2.11 zeigt wie die Kollisionkomponenten (gelbe Wireframes) genutzt werden um die Körperteile eines komplexen 3D Modells vereinfacht abzubilden.

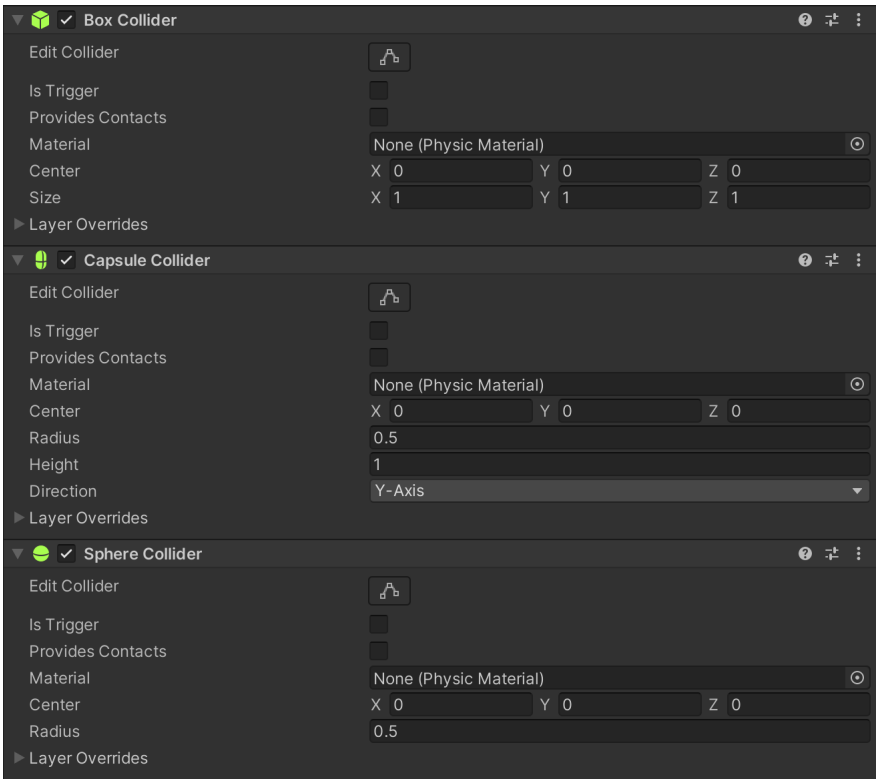

Abbildung 2.10: Unity ML-Agents Physik Kollisionskomponenten

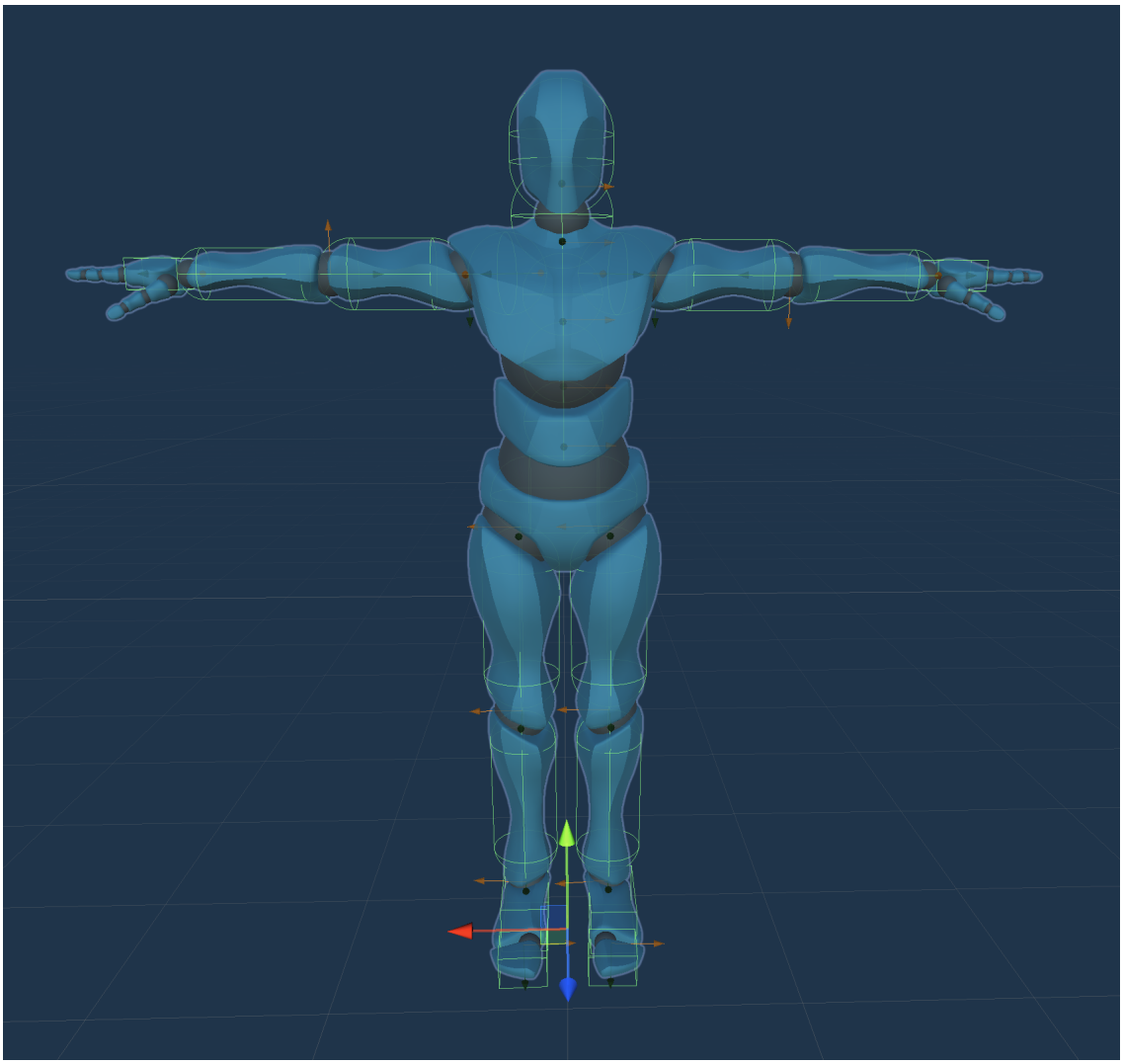

Abbildung 2.11: Unity ML-Agents Physik Charakter vereinfacht mit Kollisionskomponenten

Festkörper können mit Gelenken zu komplexeren Körperstrukturen verbunden werden. Die Konfigurierbare Gelenkkomponente (Configurable Joint) ermöglicht die Simulation von Gelenken mit freier Bewegung und Rotation auf allen drei Achsen. Dies ist wesentlich, um realistische Animationen und Interaktionen in Softwaresimulationen zu erzeugen. Im Kontext dieser Arbeit wird das Gelenk auf Rotation beschränkt und als kugelförmiges Gelenk verwendet. Die Gelenke einer humanoiden Figur können somit vereinfacht aber ausreichend genau simuliert werden.

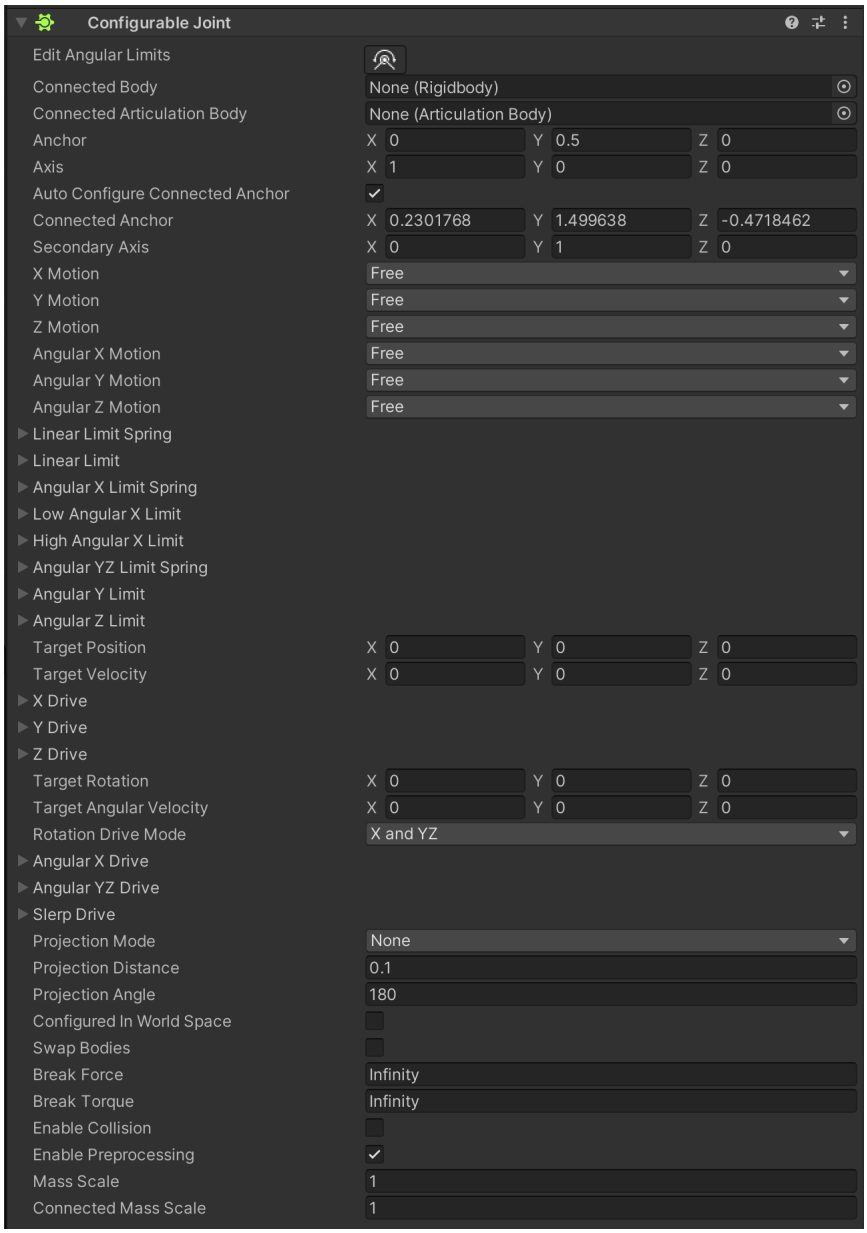

Abbildung 2.12: Unity ML-Agents Physik Gelenk

- Connected Body: bestimmt, mit welchem Körper das Gelenke verbunden ist
- Anchor: legt fest, an welchem Punkt die Verbindung zum verbundenen Körper besteht
- Axis: legt die Hauptbewegungs- und Rotationsachse fest
- Secondary Axis: legt die sekundäre Beweungs- und Rotationsachse fest
- Angular X Y Z Motion: bestimmt, ob das Gelenk Rotation zwischen den Körpern auf der X Y Z Achse zulässt
- Target Position: bestimmt das Ziel, zu welchem das Gelenk sich bewegen soll
- Angular X Y Z Limit: ermöglicht das Festlegen von Winkellimits für die Rotationsbewegungen

- X Y Z und Slerp Drive: bestimmen die Stärke der Federkraft welche das Gelenk in die Zielposition bewegt

3 Analyse

Zusätzlich zu den maschinellen Lernkomponenten stellt Unity auch Demonstrationsumgebungen bereit, in denen verschiedene Lösungen für gängige Verstärkungslernprobleme implementiert sind. In der Walker-Demo wird ein physisch simulierter Charakter darauf trainiert, zu einem Zielwürfel zu laufen. Sie implementiert bereits einige grundlegende Steuerungsmechanismen, die erforderlich sind, um einen Charakter in einer dynamischen Umgebung zu bewegen und zu kontrollieren. Aus diesem Grund wird in dieser Arbeit die Walker-Demo als Basis für die Entwicklung genutzt.

In diesem Kapitel wird daher die Walker-Demo analysiert, um in den folgenden Kapiteln darauf aufzubauen. Es wird untersucht wie die Szene und der Charakter aufgebaut sind, welche Beobachtungen als Eingabe verwendet werden und wie Ausgabe des neuronalen Netz in der Umgebung ausgeführt werden. Anschließend wird der Ablauf und die verwendete Belohnungsfunktion analysiert.

3.1 Szenenaufbau

Die Szene besteht aus einem quadratischen Spielfeld mit einem Boden und Wänden, die der Charakter nicht verlassen kann (siehe Abbildung 3.1). Diese Begrenzungen dienen dazu, die Bewegung des Charakters zu kontrollieren und sicherzustellen, dass die Lernumgebung konsistent bleibt. Die Umgebung umfasst weiterhin den Läufer und das Ziel, zu dem der Läufer lernt zu laufen.

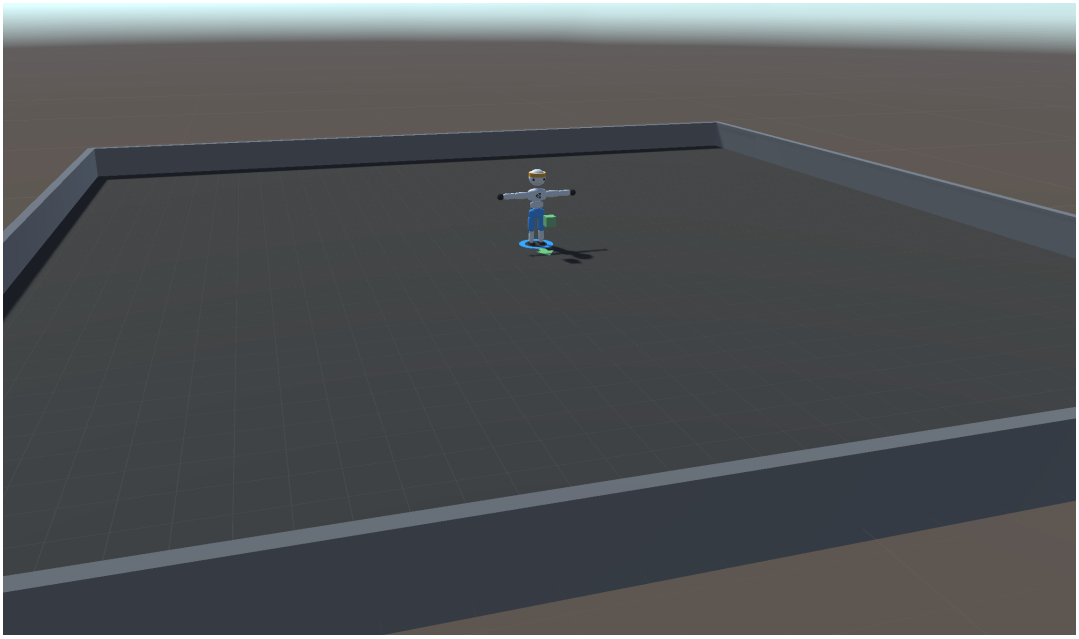

Abbildung 3.1: Walker-Demo Szenenaufbau

3.2 Läufer

Der Körper des Läufers ist sorgfältig mit verschiedenen geometrischen Formen aufgebaut: 11 Kapseln, drei Kugeln und zwei Quadern. Jede dieser Formen verfügt über eine Festkörper- und eine Kollisions-Physikkomponente, die eine realistische Interaktion innerhalb der Szene ermöglichen. Die Gelenke zwischen den Körperteilen werden als Kugelgelenke simuliert, um eine flexible und natürliche Bewegung zu gewährleisten.

Die genaue Physikkonfiguration der Körperteile wird in der Tabelle 3.1 veranschaulicht. Diese Konfiguration spielt eine zentrale Rolle, da sie die Art und Weise bestimmt, wie der Läufer lernt, auf das Ziel zuzulaufen, und dabei die physischen Einschränkungen und Möglichkeiten berücksichtigt.

Körperteil	**Verbundenes Körperteil**	**Gewicht**	**Winkellimits**	**Form**
Hüfte	-	15kg	-	Kapsel
Wirbelsäule	Hüfte	10kg	x(-20,20) y(-20,20) z(-15,15)	Kapsel
Oberkörper	Wirbelsäule	8kg	x(-20,20) y(-20,20) z(-15,15)	Kapsel
Kopf	Oberkörper	6kg	x(-30,10) y(-20,20)	Kugel
Oberarm LR	Oberkörper	je 4kg	x(-60,120) y(-100,100)	Kapsel
Unterarm LR	Oberarm	je 3kg	x(0,160)	Kapsel
Hand LR	Unterarm	je 2kg	-	Kugel
Oberschenkel LR	Hüfte	je 14kg	x(-90,60) y(-40,40)	Kapsel
Unterschenkel LR	Oberschenkel	je 7kg	x(0,120)	Kapsel
Fuß LR	Unterschenkel	je 5kg	x(-20,20 y(-20,20) z(-20,20)	Quader

Tabelle 3.1: Walker Agent Körperteile

Der Läufer wird über die Gelenk Motor Steuerung (Joint Drive Controller) gesteuert. Das Walker Agent Skript registriert die Körperteile bei der Initialisierung in der Gelenk Motor Steuerung, wodurch eine effektive Schnittstelle zur Kontrolle der Gelenke geschaffen wird. Anschließend können über die Gelenk Motor Steuerung die Zielrotationen sowie die Maximale Kraft des Gelenks festgelegt werden, und somit der Läufer gesteuert werden. Die Gelenk Motor Einstellungen (Joint Drive Settings) siehe Abbildung 3.3 bestimmen die Stärke mit welcher die Gelenke in die Zielstellung bewegt werden.

Abbildung 3.2: Gelenk Motor Steuerung

- Max Joint Spring: Bestimmt den Drehmoment mit welchem das Gelenk in die Zielposition rotiert wird.
- Joint Dampen: Verringert den Drehmoment proportional zur Differenz zwischen aktueller Geschwindigkeit und der Zielgeschwindigkeit. Verringert Schwingungen.
- Max Joint Force Limit: Gibt die maximale Kraft des Gelenks an (verhindert zu schnelle Bewegung bei großer Abweichung).

3.3 Agent

Das Walker Agent Skript, definiert den Läufer als Agent für das maschinelle Lernen. In Abbildung 3.3 wird die Agentenkomponente im Inspektor gezeigt. Diese Komponente ist entscheidend für die Konfiguration der Lernumgebung des Läufers. Um die Komponente zu nutzen, müssen hier die Körperteile des Walkers referenziert werden. Diese Referenzierung ermöglicht es, spezifische Bewegungen und Interaktionen der einzelnen Körperteile zu steuern, was für das Training des Agenten entscheidend ist. Zusätzlich kann eine Zielgeschwindigkeit festgelegt werde und ob die Geschwindigkeit variieren soll während dem Training. Die Geschwindigkeit während dem Training zu variieren hilft dem Agent sein Verhalten besser an Umgebungsveränderungen anzupassen. Als letztes muss auch das Zielobjekt referenziert werden.

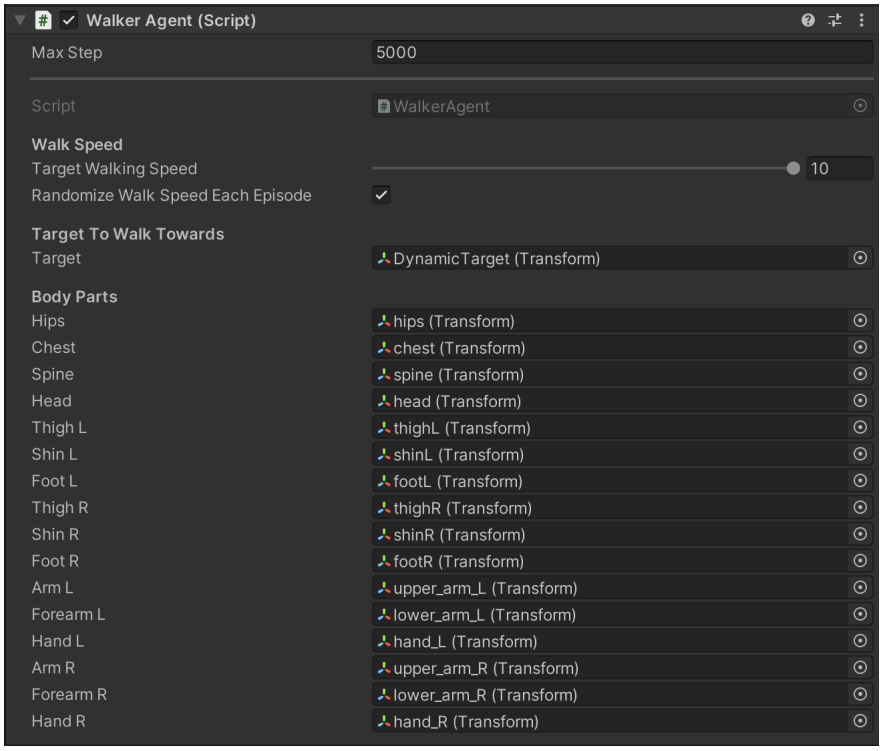

Abbildung 3.3: Agent Konfiguration

Die Beobachtung des Agenten wird in Tabelle 3.2 dargestellt. Für jedes Körperteil wird die Beobachtung aus Tabelle 3.3 dem Zustand angefügt. Die Beobachtungen müssen den Zustand des Läufers und der Umgebung im Bezug auf das Trainingsziel genau darstellen. Nur so kann der Agent die Situation verstehen und das Ziel erreichen.

ID	Beobachtung	Anmerkung
1	Abweichung Durchschnittsgeschwindigkeit von Zielgeschwindigkeit	
2	Durchschnittsgeschwindigkeit	
3	Zielgeschwindigkeit	
4	Abweichung Hüftrotation von Zielrotation	
5	Abweichung Kopfrotation von Zielrotation	
6	Zielposition	
7	Körperteil Beobachtungen	Beobachtung aus Tabelle 3.3 für jedes Körperteil

Tabelle 3.2: Walker Agent Beobachtung

ID	Beobachtung	Anmerkung
1	Bodenkontakt	
2	Geschwindigkeit	
3	Rotationsgeschwindigkeit	
4	Position relativ zur Hüfte	
5	LokaleRotation	Fehlt für Hüfte und Hände
6	Gelenkstärke	Fehlt für Hüfte und Hände

Tabelle 3.3: Walker Agent Körperteil Beobachtung

Das Format einer Aktion besteht aus den in Tabelle 3.4 aufgeführten Feldern für jedes Körperteil des Läufers, ausgenommen der Hüfte und Hände. Jedes Körperteil wird somit separat bewegt, um die Bewegungen zu optimieren und schlussendlich das Gleichgewicht zu halten und das Fortbewegen zu erlernen.

Die Hüfte ist das zentrale Körperteil woran alle weiteren Körperteile mit Gelenken direkt oder indirekt anknüpfen. Aufgrund dieser zentralen Rolle wird die Hüftbeugung über das Gelenk des verbundenen Körpers gesteuert.

Da die Hände kaum Relevanz für das laufen haben, sind in der Demo fest mit dem Unterarm verbunden und brauchen daher nicht gesteuert werden.

ID	Beobachtung	Anmerkung
1	Rotationswinkel X	Nur wenn Körperteil X Rotation beweglich ist
2	Rotationswinkel Y	Nur wenn Körperteil Y Rotation beweglich ist
3	Rotationswinkel Z	Nur wenn Körperteil Z Rotation beweglich ist
4	Gelenkstärke	

Tabelle 3.4: Walker Agent Aktion

Die Belohnungsfunktion enthält zwei Komponenten. Zum einen wird die Differenz der Bewegung in Zielrichtung zwischen momentaner Bewegung und Zielbewegung durch die Funktion R_V bewertet. Somit wird der Läufer dazu motiviert effizient auf das Ziel zuzusteuern,

indem Geschwindigkeit und Richtung optimiert werden. Zum Anderen wird die Abweichung zwischen momentaner Blickrichtung und der Zielrichtung in R_L berechnet. Diese Komponente stellt sicher das der Läufer sich vorwärts geradeaus auf das Ziel bewegt. Die Belohnung ergibt sich am ende durch die Multiplikation beider Teilterme. Die Verwendung der Multiplikation hat zur Folge das die Belohnung gleichermaßen von beiden Teiltermen abhängig ist und es somit notwendig ist, beide Teile gleichzeitig zu optimieren. Als Ergebnis lernt der Läufer gleichermaßen die Ausrichtung als auch die Bewegung in Zielrichtung.

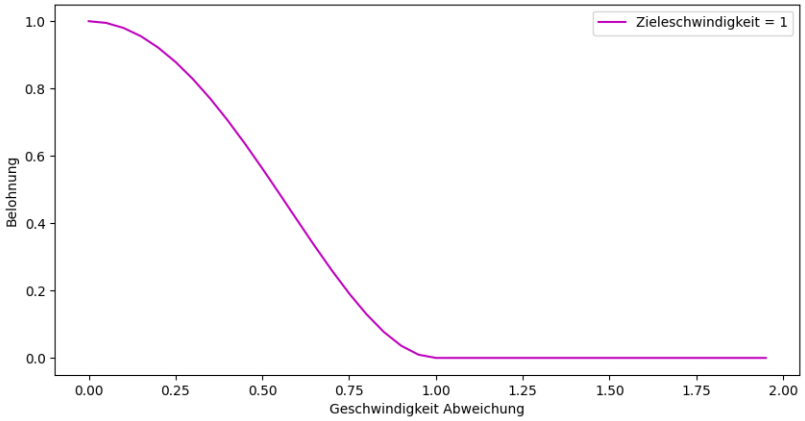

Abbildung 3.4: Walker Demo Match Velocity Belohnungsfunktion

$$V_\delta = Clip(|\vec{Geschwindigkeit} - \vec{Zielgeschwindigkeit}|, 0, |\vec{Zielgeschwindigkeit}|)$$
$$R_V = (1 - (V_\delta / |\vec{Zielgeschwindigkeit}|)^2)^2$$

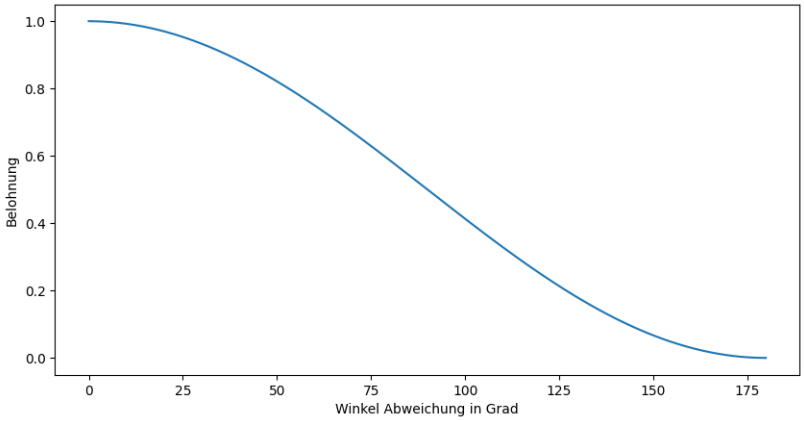

Abbildung 3.5: Walker Demo Look At Target Belohnungsfunktion

$$R_L = (\vec{Zielrichtung} \cdot \vec{Blickrichtung}) + 1) \cdot 0.5$$
$$R = R_V \cdot R_L$$

Initialisierung bzw Episodereset erklären Orientation Object erklären Zielsetzung erklären Rewardfrequenz erwähnen

4 Charaktercontroller

Dieses Kapitel geht auf die Anforderungen eines Charaktercontrollers so wie die Entwicklung innerhalb dieser Arbeit ein. Dabei werden die verschiedenen Ansätze und Ihre Implementierung in Prototypen aufgezeigt.

4.1 Nutzersteuerung

Um von einem Charaktercontroller sprechen zu können muss der Agent über Nutzerinput gesteuert werden können. Mit diesem Gedanke wurde die ersten Anpassungen der Walker Demo implementiert um das Ziel zur Laufzeit über Tastatureingabe zu bewegen.

4.1.1 Versuch 1

Das Ziel soll zur Laufzeit durch Tastatureingabe des Nutzers gesteuert werden können. Um das zu umzusetzen wird die Tastatureingabe eingelesen und darauf basierend das Ziel relativ zur Hüfte des Walkers gesetzt. Über die Vorwärtsrichtung und Rechte-Richtung multipliziert mit dem Input wird die Position relativ zur Hüfte berechnet (siehe Listing 4.1).

```
1 void FixedUpdate()
2 {
3     //Einlesen Tastatur Input
4     float inputHor = Input.GetAxis("Horizontal");
5     float inputVert = Input.GetAxis("Vertical");
6
7     //Setzen der Zielposition
8     transform.position = root.position + root.forward * inputVert +
          root.right * inputHor;
9 }
```

Listing 4.1: Nutzersteuerung erster Prototyp

Diese implementierung ermöglicht grundsätzlich das der Läufer über die Tastatureingabe gesteuert werden kann, hat aber noch einige Probleme. Das positionieren des Ziels relativ zur Hüfte hat als Konsequenz das jede Bewegung der Hüfte Einfluss auf die Laufrichtung hat, wodurch sich der Läufer nicht stabil steuern lässt.

4.1.2 Versuch 2

Um die Probleme aus 4.1.1 zu beheben wird in folgendem Versuch das Ziel relativ zu den Weltachsen gesetzt. In Unity gibt die Vector3 Klasse mit den Feldern forward den Vektor (0,0,1) und mit right den Vektor (1,0,0) in Weltkoordinaten an.

```
1 //Setzen der Zielposition
2 transform.position = root.position + Vector3.forward * inputVert +
      Vector3.right * inputHor;
```

Listing 4.2: Nutzersteuerung berechnung mit Weltachsen

Durch die Nutzung der Weltachsen anstatt der Hüftrotationsachsen (siehe Listing 4.2) kann das Problem behoben werden. Es tritt dadurch jedoch ein weiteres Problem auf. Bei Verwendung der Weltachsen ist die Steuerung des Walkers aus Spielersicht nicht mehr intuitiv, da der Input je nach Rotation des Walkers einen anderen Einfluss hat.

4.1.3 Versuch 3

Die Lösung ist eine Kombination aus den Ideen der ersten beiden Versuche. Die Laufrichtung soll weiterhin relativ zum Läufer bestimmen werden gleichermaßen aber von der wechselhaften Bewegung des Läufers entkoppeln sein. Das hinzufügen einer separaten Rotationskomponente für die Bestimmung der Blickrichtung erfüllt hier die Kriterien.
Zu beginn wird die Blickrichtung mit der Vorwärtskomponente der Hüftrotation gleichgesetzt. Ausgehend von der Startrichtung wird dann über horizontalen Mausinput die Richtung angepasst (siehe Listing 4.3).

```
1  void Start()
2  {
3      //Root Position als Startposition festhalten
4      startForward = root.forward;
5      startRight = root.right;
6  }
7  void FixedUpdate()
8  {
9      //Einlesen Tastatur Input
10     float inputHor = Input.GetAxis("Horizontal");
11     float inputVert = Input.GetAxis("Vertical");
12
13     //Einlesen Maus Input
14     float mouseX = Input.GetAxis("Mouse X");
15     rotAngle += mouseX;
16
17     //Berechnung der Rotation
18     Quaternion rotation = Quaternion.AngleAxis(rotAngle,
           rotationAxis);
19
20     //Anwendung der Rotation auf Richtungsvektoren
21     Vector3 directionForward = rotation * startForward;
22     Vector3 directionRight = rotation * startRight;
23
24     //Setzen der Zielposition
25     transform.position = root.position + directionForward *
           inputVert + directionRight * inputHor;
26 }
```

Listing 4.3: Erweiterung der Nutzersteuerung mit separater Blickrichtung

Das Ergebnis ermöglicht die Steuerung des Läufers als Spielecharakter, mit einer gewohnten Steuerung aus schon bestehenden Spieletiteln und ist kompatibel mit einer Drittenperson Ansicht als auch mit der Erstenperson Ansicht.

4.2 Modell Anpassungen

Das trainierte Modell der Walker Demo beherrscht jedoch nur die Fortbewegung in Blickrichtung. Der Läufer ist auch nicht darauf trainiert stehen zu bleiben. Das resultiert darin das der Läufer fällt sobald der Nutzer keinen Tastaturinput gibt. Dieses Kapitel beschäftigt sich mit den Einschränkungen der Walker-Demonstration im Bezug auf unterschiedliche Bewegungsrichtungen.

Im ersten Schritt wird getestet wie der Walker angepasst werden kann um die fehlenden Bewegungsabläufe in separaten Modellen zu erlernen.

4.2.1 Versuch 4

Versuch 4 behandelt die Bewegung auf der Stelle stehen. Für das stehenbleiben wird die Zielgeschwindigkeit auf 0 gesetzt während das Ziel auf der Startposition befindet. Die Belohnungsfunktion der Demo, wird ab jetzt Demo Belohnungsfunktion genannt. Die Demo Belohnungsfunktion hat das Problem das durch die Zielgeschwindigkeit geteilt wird, was bei einer Zielgeschwindigkeit von 0 zu Mathematischen Fehlern führt. Um das zu vermeiden wurde das trainieren mit einer anderen Belohnungsfunktion getestet.

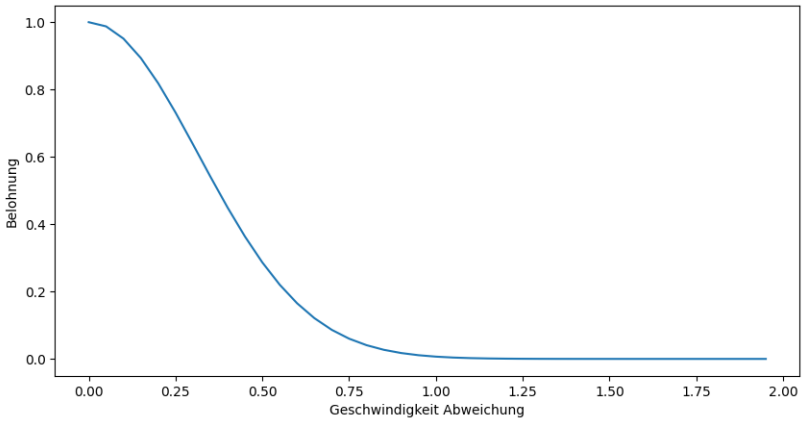

Abbildung 4.1: DeepMimic Match Velocity Belohnungsfunktion

Die neue Belohnungsfunktion ist inspiriert von den Belohnungsfunktionen des Papers "DeepMimic: Example-Guided Deep Reinforcement Learning of Physics-Based Character Skills".[2] Die Belohnungsfunktion aus Abbildung 4.1 wird daher ab hier DeepMimic Belohnungsfunktion genannt.

Abbildung 4.2: Versuch 4 Traininggraphen

Abbildung 4.3: Versuch4 Training Belohnungsgraphen

Der Walker konnte mit der DeepMimic Belohnungsfunktion lernen auf der Stelle zu stehen. Abbildung 4.2 zeigt wie die zurück gelegte Distanz um 0 herum pendelt, während die Episodenlänge die maximale Länge von 1000 erreicht hat. Die Belohnungen sind auch nahezu maximal ausgereizt siehe Abbildung 4.3.

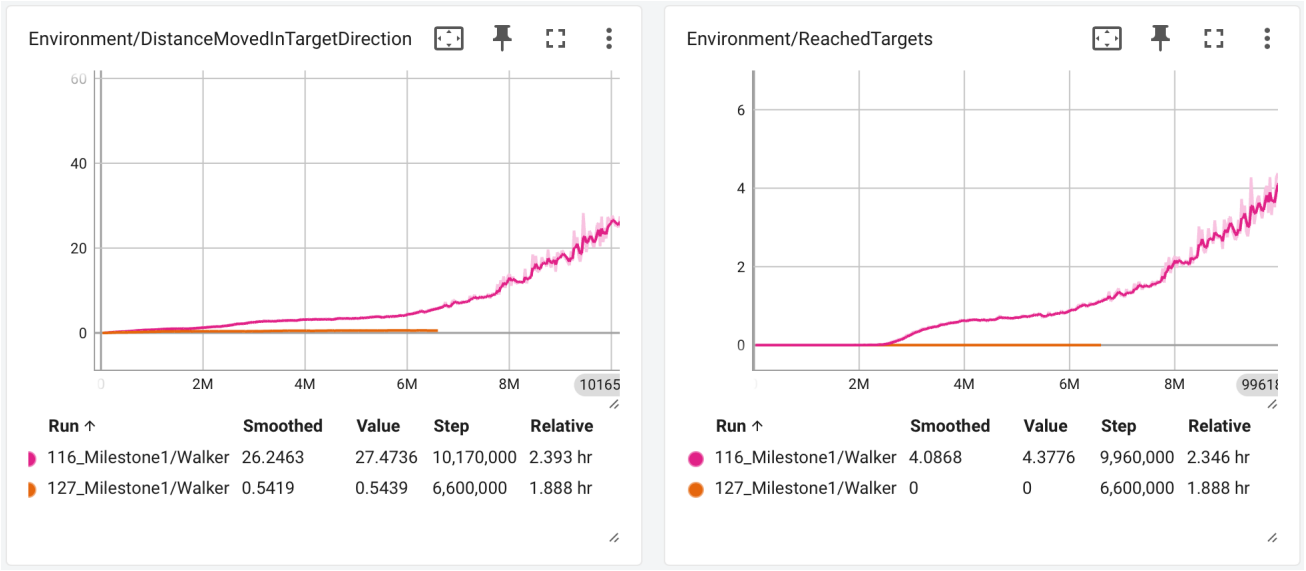

Abbildung 4.4: Vergleich von Lauftraining mit Demo Belohnungsfunktion gegen DeepMimic Belohnungsfunktion

Mit zufälliger Zielgeschwindigkeit zu einem Ziel zu laufen wie im Ursprünglichen Verhalten konnte damit jedoch nicht zufriedenstellend erlernt werden. Die Abbildung 4.4 zeigt mit der orangenen Linie die Leistung der DeepMimic Belohnungsfunktion und mit der pinken Linie die Leistung der Demo Belohnungsfunktion. Nachfolgender Vergleich der Belohnungsfunktionen zeigt das die Ursprüngliche Belohnungsfunktion durch das Teilen mit der Zielgeschwindigkeit die Sensitivität der Funktion je nach Zielgeschwindigkeit beeinflusst. Daraus folgt das bei steigender Zielgeschwindigkeit eine größere Abweichung der Geschwindigkeit geduldet wird (siehe Abbildung 4.5). Diese Anpassung verbessert die Generalisierung zwischen den wechselnden Geschwindigkeiten um ein vielfaches.

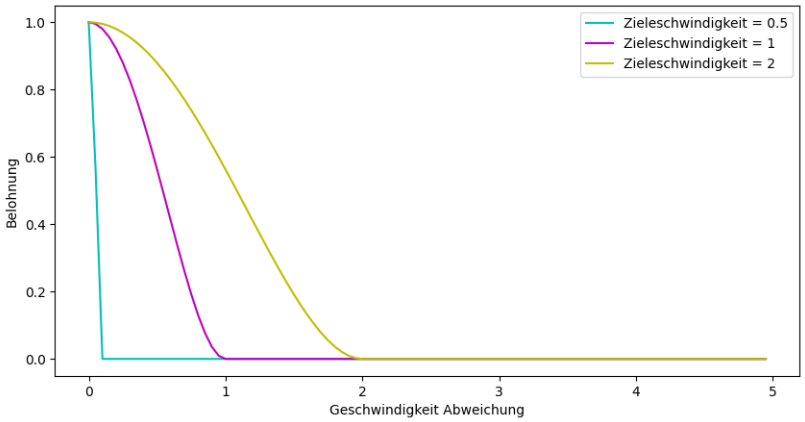

Abbildung 4.5: Vergleich der Demo Belohnungsfunktion unter verschiedenen Zielgeschwindigkeiten

4.2.2 Versuch 5

Mit dieser Erkenntnis wurde eine neue Anpassung untersucht. In der folgenden Anpassung blieb die Belohnungsfunktion weitestgehend Unverändert. Lediglich das obere Limit ab wel-

chem die Funktion eine Belohnung von 0 annimmt, wurde auf ein minimum von 0.1 beschränkt. Somit konnte sicher gestellt werden das im Bereich der normalen Fortbewegung keine Veränderung auftritt. Mit der Demo Belohnungsfunktion konnten nur Annäherungen an eine Zielgeschwindigkeit von 0 genutzt werden. Bei einer Annäherung von 0.000001 ist das Spektrum an akzeptablen Geschwindigkeiten bevor die Belohnung 0 ist nahezu unerreichbar (siehe Abbildung 4.6). Mit dem Limit von 0.1 ist der Bereich der Belohnungsfunktion > 0 groß genug, sodass der Läufer durch ausprobieren Belohnungen über 0 erreichen kann. Somit kann der Läufer die Belohnung optimieren.

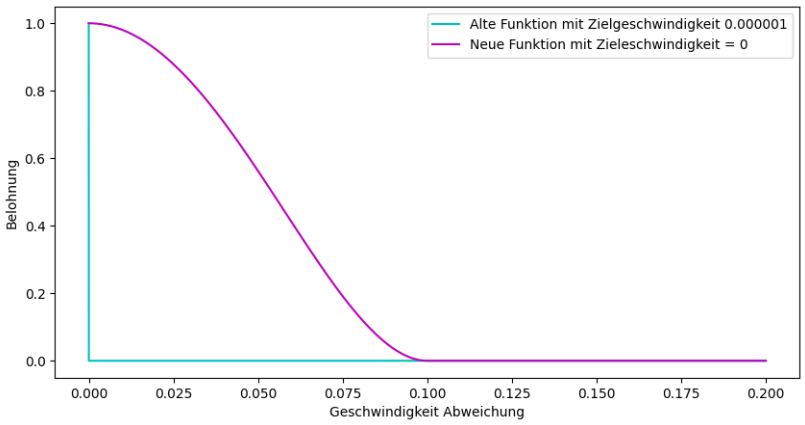

Abbildung 4.6: Vergleich Demo gegen Belohnungsfunktion mit 0.1 Limit

$$V_\delta = Clip(|Geschwindigkeit - Zielgeschwindigkeit|, 0, |Zielgeschwindigkeit|)$$
$$V_\delta = Clip(|Geschwindigkeit - Zielgeschwindigkeit|, 0, max(0.1, |Zielgeschwindigkeit|))$$

Das auf einer Stelle stehen hat der Läufer damit in einem separaten Training auch erlernt. Abbildung 4.7 zeigt das der Läufer die maximale Episoden Länge von 1000 erreicht hat ohne zu fallen. Die bewegte Distanz hat sich auch 0 angenähert.

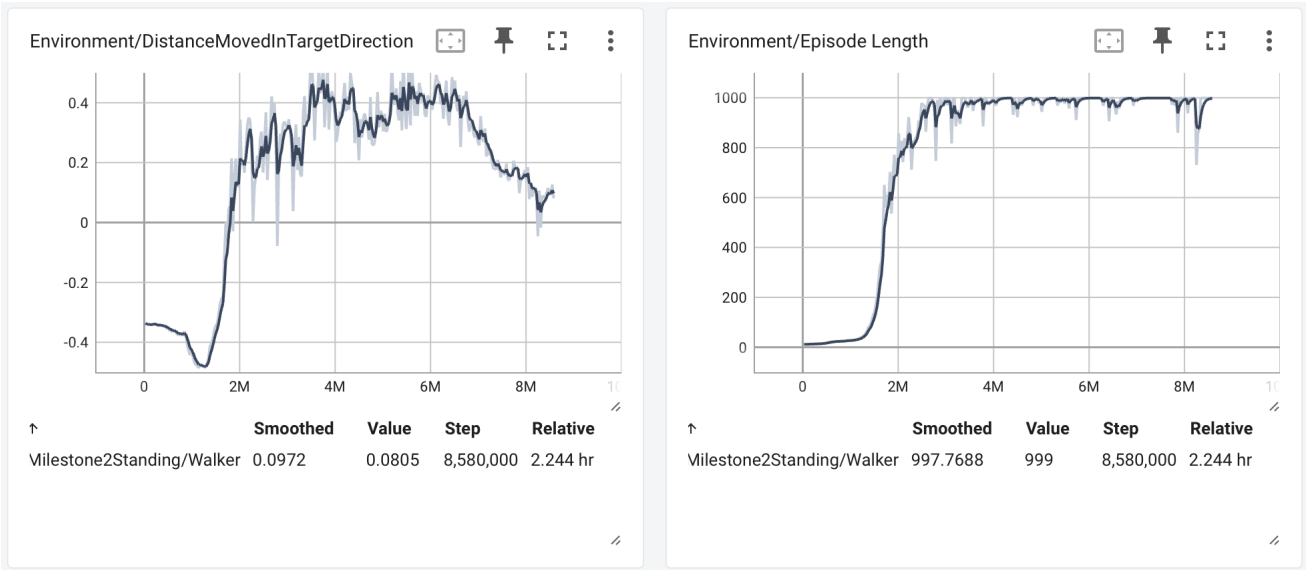

Abbildung 4.7: Versuch 5 Traininggraphen

Die Belohnungen wurden auch weitestgehend optimiert siehe Abbildung 4.8.

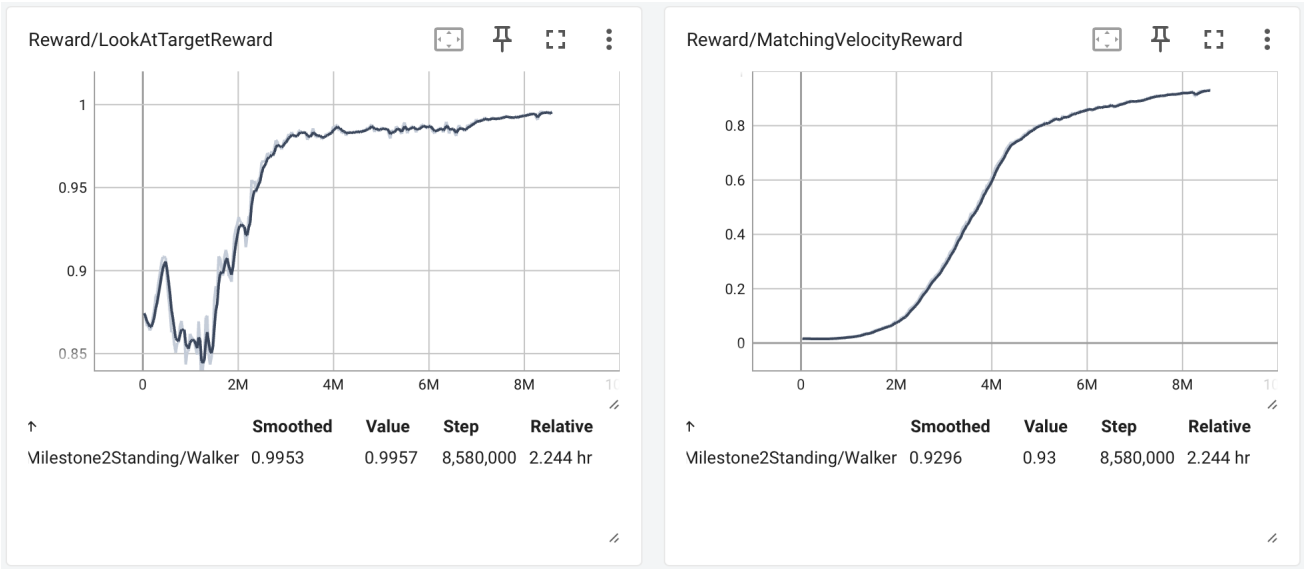

Abbildung 4.8: Versuch 5 Training Belohnungsgraphen

4.2.3 Versuch 6

Der folgende Versuch untersucht das Laufen in unterschiedliche Richtungen relativ zur Blickrichtung. Um das zu realisieren wurde dem Agent ein enum mit der Laufrichtung hinzugefügt. Die Blickrichtung Belohnungsfunktion wird relativ zur Zielrichtung berechnet. Bei Vorwärtsbewegung ist die Blickrichtung gerade aus. Bei Seitlicher Bewegung ist die Blickrichtung gespiegelt zur Laufrichtung. Läuft der Läufer seitlich rechts ist die Blickrichtung links zum ziel und anders herum. Beim rückwärts gehen ist die Blickrichtung entgegen der Zielrichtung. Die Implementierung ist in 4.4 zu sehen.

```
1  public enum Direction
2  {
3      Forward,
4      Right,
5      Left,
6      Backward,
7  }
8
9  public override void FixedUpdate()
10 {
11     ...
12     var headForward = head.forward;
13     headForward.y = 0;
14     Vector3 lookDirection = cubeForward;
15     switch (direction)
16     {
17         case Direction.Right:
18             lookDirection = -walkOrientationCube.transform.right;
19             break;
```

```
20          case Direction.Left:
21              lookDirection = walkOrientationCube.transform.right;
22              break;
23          case Direction.Backward:
24              lookDirection = -walkOrientationCube.transform.forward;
25              break;
26      }
27      ...
28  }
```

Listing 4.4: Laufrichtung Enum, Beobachtung und Belohnung

Das gehen in Zielrichtung wurde durch die Änderungen nicht beeinflusst. Separate Trainings zu den drei anderen Laufrichtungen waren erfolgreich.

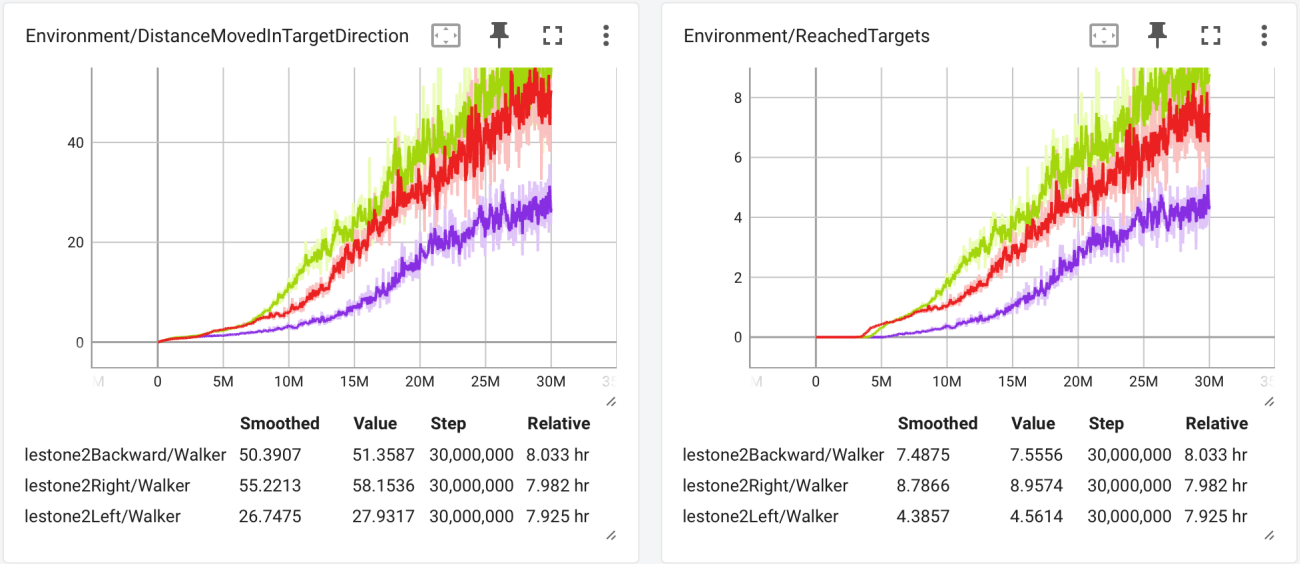

Abbildung 4.9: Versuch 6 Trainninggraphen

Abbildung 4.9 zeigt die zurückgelegte Distanz und die Anzahl an erreichten Zielen in einer Trainingsepisode. Die Ergebnisse der 3 Laufrichtungen (rot = Rückwärts, gelb = rechts, blau = links) sind alle vergleichbar mit den Ergebnissen der Demo. Die Abweichung der Laufrichtung Links ist vermutlich der zufällligen Natur des Trainings anzurechnen.

4.2.4 Versuch 7

Die Charaktersteuerung benötigt je nach Tastatureingabe eine der vier Bewegungsrichtungen. Der Unity ML-Agents Agent enthält eine Funktion zum wechseln des verwendeten Modells. Mit dieser Funktion wird in folgender Implementierung zwischen den Modellen aus Versuch 6 gewechselt um alle Bewegungsrichtungen mit einer Steuerung abzudecken. Zu Erwarten ist das die Bewegung in die einzelnen Richtungen funktioniert, der Läufer aber beim Wechsel zwischen den Modellen das Gleichgewicht nicht halten kann.

```
1  public override void FixedUpdate() {
2      ...
3      agent.targetWalkingSpeed = 5f;
```

```
4       if (inputVert != 0) //Tastatur Input Vor oder Zurück
5       {
6           // Vorwärts
7           if (inputVert > 0)
8           {
9               agent.SetModel("Walker", modelForward);
10          }
11          else // Zurück
12          {
13              agent.SetModel("Walker", modelBackward);
14          }
15      }
16      else if (inputHor != 0) // Links oder Rechts
17      {
18          if (inputHor > 0) // Rechts
19          {
20              agent.SetModel("Walker", modelRight);
21          }
22          else // Links
23          {
24              agent.SetModel("Walker", modelLeft);
25          }
26      }
27      else //kein Input -> Auf der Stelle stehen
28      {
29          agent.targetWalkingSpeed = 0f;
30          agent.SetModel("Walker", modelStanding);
31      }
32      ...
33  }
```

Listing 4.5: Laufrichtung Modell wechseln

Wie angenommen funktioniert das Bewegen in eine konstante Richtung gut. Beim Wechsel zu einem anderen Modell fällt der Läufer ohne Ausnahme.

4.2.5 Versuch 8

In Versuch 8 wird die Möglichkeit geprüft, alle Bewegungsrichtungen in einem Modell anzulernen. Dafür wird zum Start eine zufällige Bewegungsrichtung für jeden Läufer ausgewählt, mit dem Ziel das die Läufer direkt mit unterschiedliche Bewegungsrichtungen trainieren. Das gleichzeitige trainieren mit mehreren Läufern und unterschiedlichen Gehrichtungen soll das erlernen einer generell gültigen Strategie fördern. Die Gehrichtung wechselt beim erreichen eines Ziels, damit soll erreicht werden das der Läufer das aktuelle Ziel mit ausgewählter Bewegungsrichtung vollständig erlernt. Durch das öftere erreichen von Zielen im Verlauf des Trainings wird aber auch gleichzeitig jede beliebige Kombination angelernt. Als Ausgleich in der Komplexität wird die Zielgeschwindigkeit für das ganze Training festgesetzt.

```
1   public Direction direction = Direction.Forward;
2   Direction[] directions;
3
4   public override void Initialize()
```

```
5  {
6      ...
7      directions = (Direction[])Enum.GetValues(typeof(Direction));
8      SetRandomWalkDirection();
9      onTouchedTarget.AddListener(SetRandomWalkDirection);
10 }
11
12 public void SetRandomWalkDirection()
13 {
14     direction = directions[Random.Range(0, directions.Length)];
15 }
16
17 public override void CollectObservations(VectorSensor sensor)
18     {
19         ...
20         sensor.AddObservation((float)direction);
21     }
```

Listing 4.6: Laufrichtung zufällig zum Start und beim erreichen von Ziel

Codeausschnitt 4.6 erstellt beim Initialisieren des Agenten ein Array mit allen Werten, welche das Richtungs-Enum zulässt. Die Funktion SetRandomWalkDirection wählt eine zufällige Richtung aus und setzt diese für den Agenten. Die Funktion wird zu Beginn in Initialize aufgerufe. Zusätzlich wird die Methode mit einem Listener auf das onTouchedTarget des Agenten registriert. Die Methode wird somit bei jedem berühren eines Ziels ausgeführt. Das der Agent während dem Training sowie nach dem Training zwischen den Laufrichtungen entscheiden kann, bekommt er einen Zahlenwert repräsentativ für die Richtung in der Beobachtung angehängt.

Der Läufer lernt unter diesen Bedingungen sehr langsam und das Training stagniert. Ab ca. 20 millionen Trainingsschritten fängt der Läufer an regelmäßig Ziele zu erreichen (siehe Abbildung 4.10). Durch das häufige erreichen von Zielen steigt aber auch die Anzahl der Ziel- und Laufrichtungswechsel. Aus diesem Grund brechen die Belohnungen ein und der Fortschritt stagniert (siehe Abbildung 4.11).

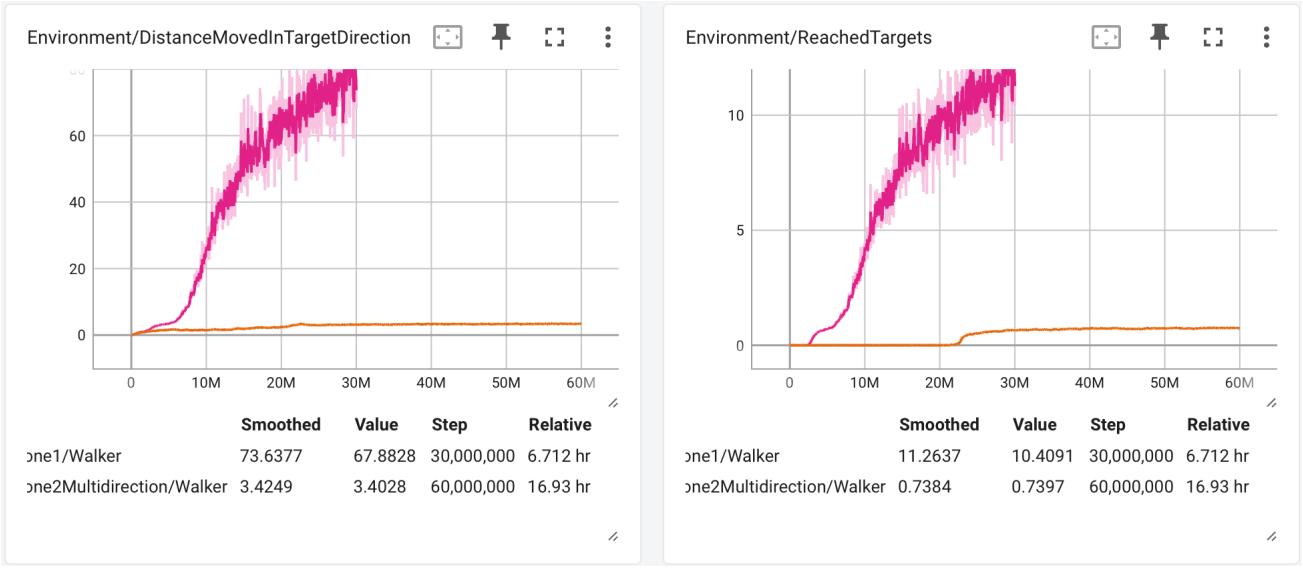

Abbildung 4.10: Versuch 8 Traininggraphen

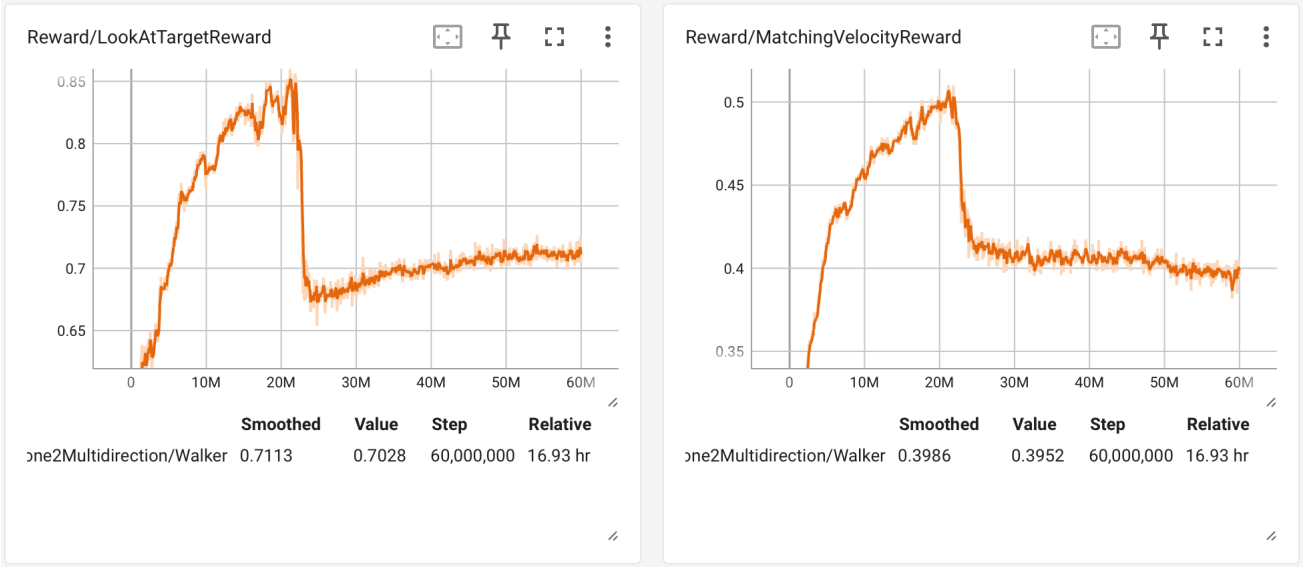

Abbildung 4.11: Versuch 8 Training Belohnungsgraphen

4.2.6 Versuch 9

Um den Richtungswechsel regelmäßiger zu gestalten, wird getestet wie das Training sich verhält wenn die Richtung beim Start jeder neuen Trainingsepisode zufällig gewählt wird.

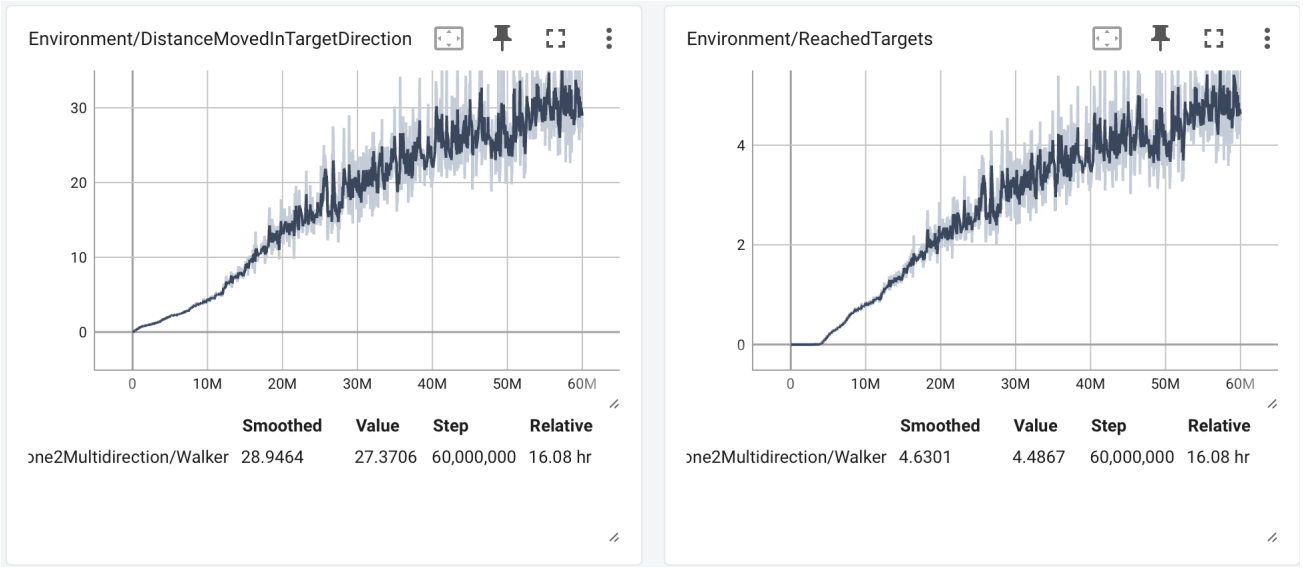

Abbildung 4.12: Versuch 9 Traininggraphen

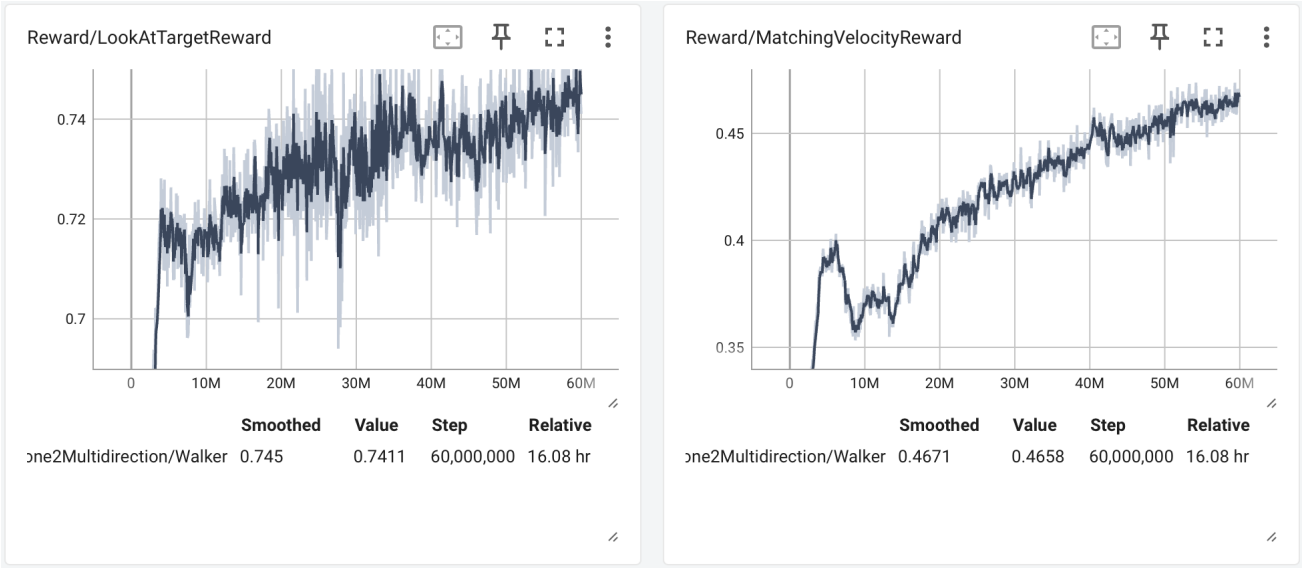

Abbildung 4.13: Versuch 9 Training Belohnungsgraphen

Das gleichbleiben der Laufbewegung über die komplette Trainingsepisode hat zur folge dass das Training stabiler verläuft siehe Abbildung 4.10. Der Nachteil ist jedoch das der Läufer keine Bewegungswechsel lernt. Beim steuern des Läufers ist das wechseln zwischen den Laufrichtungen noch immer ein Problem. Dazu kommt das der in diesem training die Blickrichtungs Belohnung geringer ist als bei vorherigen Trainingseinheiten. Der Läufer lernt die seitwärts Bewegungen nicht richtig sondern nimmt einen Verlust in der Belohnung in kauf für die Steigerung der Episodenlänge.

4.3 Mixamo Charakter

-Konfiguration der Physikkomonenten für mixamo Charakter -Codeanpassung der Konfiguration für mehrere Körperteile -Vereinfachung durch versteifen von einigen extra Gelenken -Gallopiert -Beinwechsel Belohnung um galoppieren zu vermeiden -> funktioniert -Leistungsminimierung Belohnung um laufverhalten natürlicher zu machen -> funktioniert nur arme sind sehr nah und starr am Körper -

5 Fazit

Text

Literaturverzeichnis

[1] Arthur Juliani u. a. "Unity: A general platform for intelligent agents". In: *arXiv preprint arXiv:1809.02627* (2020). URL: https://arxiv.org/pdf/1809.02627.pdf.

[2] Xue Bin Peng u. a. "Deepmimic: Example-guided deep reinforcement learning of physics-based character skills". In: *ACM Transactions On Graphics (TOG)* 37.4 (2018), S. 1–14.

[3] Richard S Sutton und Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.