



**Bachelor Thesis (SPO NUMMER)**

**Physik basierter Charaktercontroller  
mit Unity Machine Learning**

Simon Grözinger\*

13. August 2024

Eingereicht bei Prof. Dr. Tim Reichert

\*205047, [sgroezin@stud.hs-heilbronn.de](mailto:sgroezin@stud.hs-heilbronn.de)

# Inhaltsverzeichnis

<b>Abkürzungsverzeichnis</b>	<b>III</b>
<b>Abbildungsverzeichnis</b>	<b>IV</b>
<b>Tabellenverzeichnis</b>	<b>V</b>
<b>Listings</b>	<b>VI</b>
<b>1 Einleitung</b>	<b>1</b>
<b>2 Grundlagen</b>	<b>2</b>
2.1 Verstärkendes Lernen . . . . .	2
2.2 ML-Agents . . . . .	3
2.2.1 Aufbau . . . . .	3
2.2.2 Training . . . . .	5
2.2.3 Auswertung . . . . .	7
2.3 Unity Physik . . . . .	8
<b>3 Analyse Walker Demo</b>	<b>12</b>
3.1 Lernumgebung . . . . .	12
3.2 Training . . . . .	15
3.3 Auswertung . . . . .	17
<b>4 Umsetzung Charaktercontroller</b>	<b>20</b>
4.1 Nutzersteuerung . . . . .	20
4.1.1 Anforderungen . . . . .	20
4.1.2 First- und Thirdperson-Steuerung . . . . .	20
4.1.3 Top-Down-Steuerung . . . . .	21
4.2 Zusätzliche Bewegungsabläufe . . . . .	22
4.2.1 Anforderungen . . . . .	22
4.2.2 Separate Bewegungsabläufe . . . . .	22
4.2.3 360 Grad Blickziel . . . . .	30
4.2.4 4 Laufrichtungen . . . . .	30
4.3 Unterschiedliche Charakter . . . . .	36
4.3.1 Anforderungen . . . . .	36
4.3.2 Anpassungen . . . . .	36
4.3.3 Einrichtung . . . . .	38
4.3.4 Auswertung . . . . .	41
4.4 Gangbild anpassungen . . . . .	41
4.4.1 Belohnung für Beinwechsel . . . . .	42
4.4.2 Belohnung für Energieminimierung . . . . .	43
4.4.3 Imitationslernen . . . . .	44
<b>5 Fazit</b>	<b>45</b>
<b>Literaturverzeichnis</b>	<b>46</b>

# Abkürzungsverzeichnis

**ABK:** ABKÜRZUNG

# Abbildungsverzeichnis

2.1	Verstärkendes Lernen Ablauf . . . . .	2
2.2	Unity ML-Agents Aufbau . . . . .	3
2.3	Unity ML-Agents Aufbau Unity Umgebung . . . . .	3
2.4	Unity ML-Agents Verhalten Parameter Komponente . . . . .	4
2.5	Unity ML-Agents Agenten Komponente . . . . .	4
2.6	Unity ML-Agents Entscheidung Anfragen Komponente . . . . .	5
2.7	Unity ML-Agents Aufbau Python Umgebung . . . . .	6
2.8	Tensorboard Ansicht . . . . .	7
2.9	Unity ML-Agents Physik Festkörper . . . . .	8
2.10	Unity ML-Agents Physik Kollisionskomponenten . . . . .	9
2.11	Unity ML-Agents Physik Charakter vereinfacht mit Kollisionskomponenten	9
2.12	Unity ML-Agents Physik Gelenk . . . . .	10
3.1	Walker-Demo Umgebung . . . . .	12
3.2	Walker-Demo Läufer . . . . .	13
3.3	Gelenk Motor Steuerung . . . . .	14
3.4	Agent Konfiguration . . . . .	14
3.5	Walker Demo Match Velocity Belohnungsfunktion . . . . .	16
3.6	Walker Demo Look At Target Belohnungsfunktion . . . . .	17
3.7	Walker Demo Training Graphen . . . . .	18
3.8	Walker Demo Analyse Gangbild . . . . .	19
4.1	Neue Sigmoid Geschwindigkeit Belohnungsfunktion . . . . .	23
4.2	Versuch 4 Training Graphen . . . . .	24
4.3	Vergleich von Lauftraining mit Demo Belohnungsfunktion gegen DeepMimic Belohnungsfunktion . . . . .	25
4.4	Vergleich der Demo Belohnungsfunktion unter verschiedenen Zielgeschwindigkeiten . . . . .	25
4.5	Vergleich Demo gegen Belohnungsfunktion mit 0.1 Limit . . . . .	26
4.6	Versuch 5 Training Graphen . . . . .	27
4.7	Versuch 6 Training Graphen . . . . .	29
4.8	Versuch 8 Training Graphen . . . . .	33
4.9	Versuch 9 Training Graphen . . . . .	35
4.10	Mixamo Charakter Y Bot . . . . .	39
4.11	Körperteilkomponente . . . . .	40
4.12	Walker Agentkomponente . . . . .	40
4.13	Mixamo Versuch 10 Gangbild . . . . .	41
4.14	Beinwechsel Belohnung . . . . .	42
4.15	Mixamo Versuch 11 Gangbild . . . . .	42
4.16	Energiespar Belohnung . . . . .	43
4.17	Mixamo Versuch 12 Gangbild . . . . .	43

# Tabellenverzeichnis

3.1	Walker Agent Körperteile . . . . .	13
3.2	Walker Agent Beobachtung . . . . .	15
3.3	Walker Agent Körperteil Beobachtung . . . . .	15
3.4	Walker Agent Aktion . . . . .	16
4.1	Mixamo Charakter Körperteile . . . . .	41

# Listings

2.1	Agent Funktionen . . . . .	4
2.2	Trainer Konfigurationsdatei . . . . .	6
4.1	Nutzersteuerung für First- und Thirdperson . . . . .	20
4.2	Nutzersteuerung für Top-Down . . . . .	21
4.3	Blickrichtung Enum und Belohnung . . . . .	28
4.4	Laufrichtung Modell wechseln . . . . .	30
4.5	Laufrichtung zufällig zum Start und beim erreichen von Ziel . . . . .	31
4.6	Ausschnitt Angepasstes Walker Agent Skript . . . . .	37

# 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 nachzuhahmen 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.[\[3\]](#)

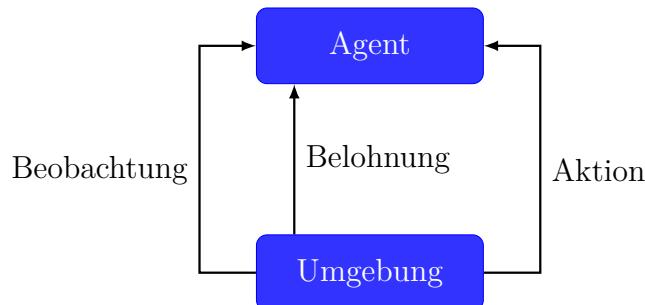


Abbildung 2.1: Verstärkendes Lernen Ablauf

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.[\[1\]](#)

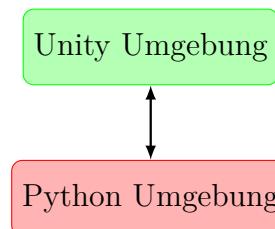


Abbildung 2.2: Unity ML-Agents Aufbau

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 Modis dargestellt.

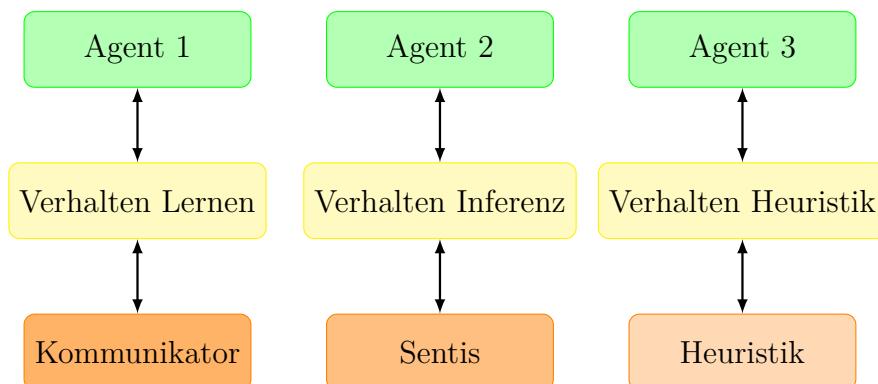


Abbildung 2.3: Unity ML-Agents Aufbau Unity Umgebung

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.

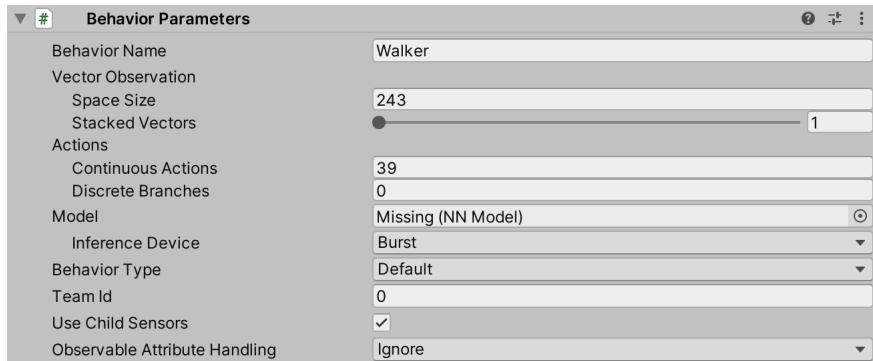


Abbildung 2.4: Unity ML-Agents Verhalten Parameter Komponente

- 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.

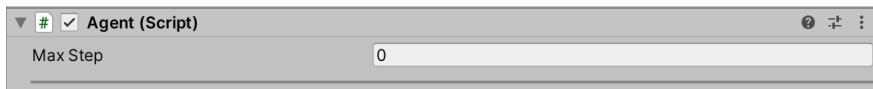


Abbildung 2.5: Unity ML-Agents Agenten Komponente

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.

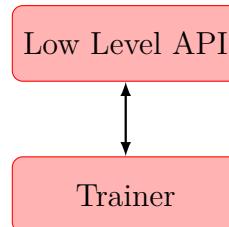


Abbildung 2.7: Unity ML-Agents Aufbau Python Umgebung

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 ausgeweitet 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 Trainingssets 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.

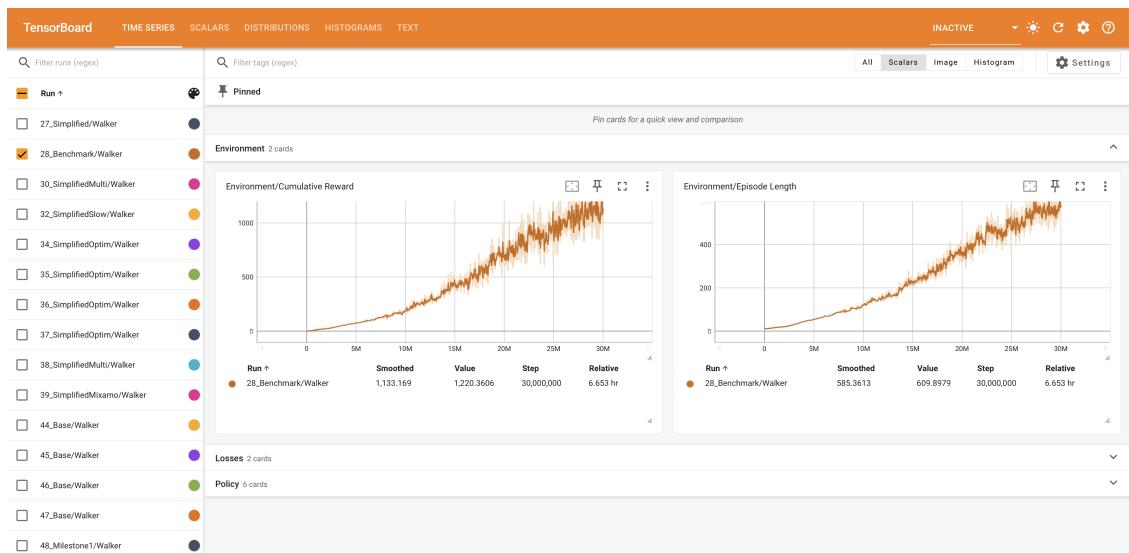


Abbildung 2.8: Tensorboard Ansicht

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 Durchschnitt 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.

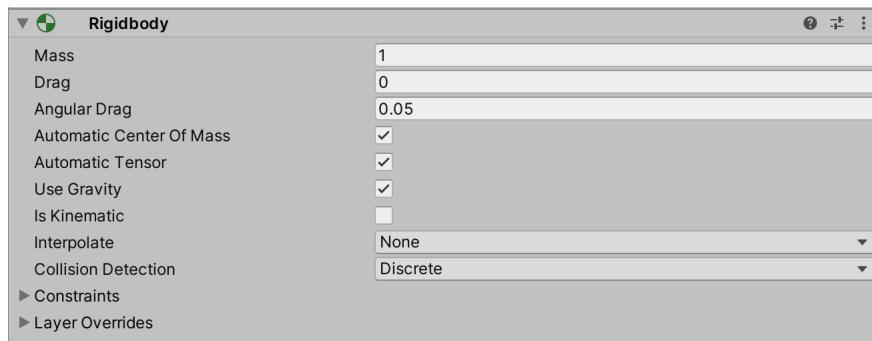


Abbildung 2.9: Unity ML-Agents Physik Festkörper

- Mass: gibt das Gewicht des Körpers an
- Drag: definiert den Geschwindigkeitsverlust eines Körpers in Bewegung durch Reibung, Luftwiderstand
- 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 Kollisionskomponenten (gelbe Wireframes) genutzt werden um die Körperteile eines komplexen 3D Modells vereinfacht abzubilden.

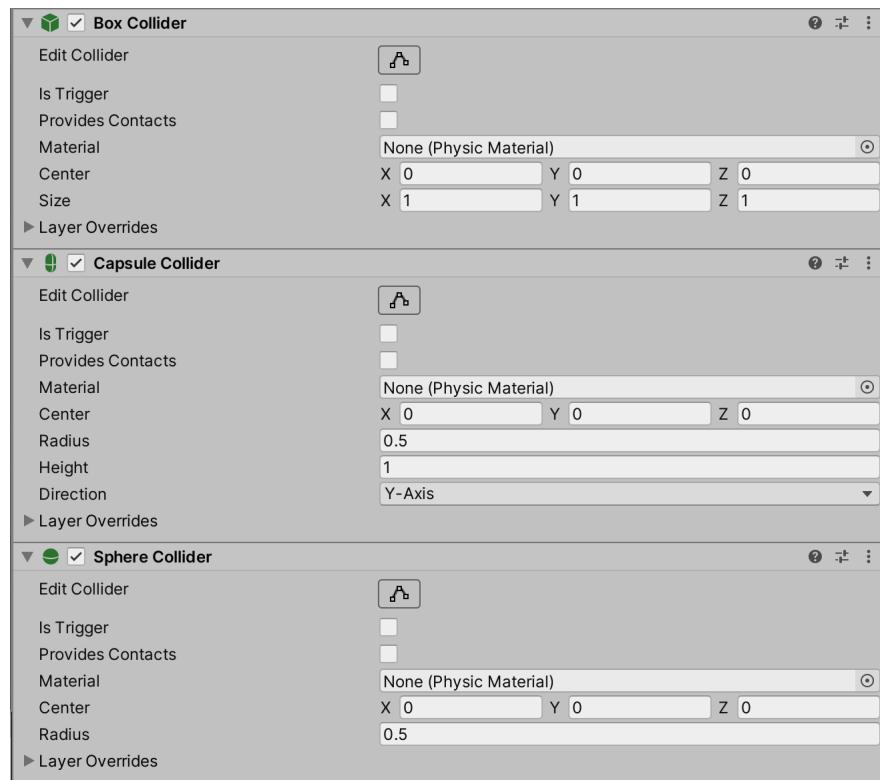


Abbildung 2.10: Unity ML-Agents Physik Kollisionskomponenten

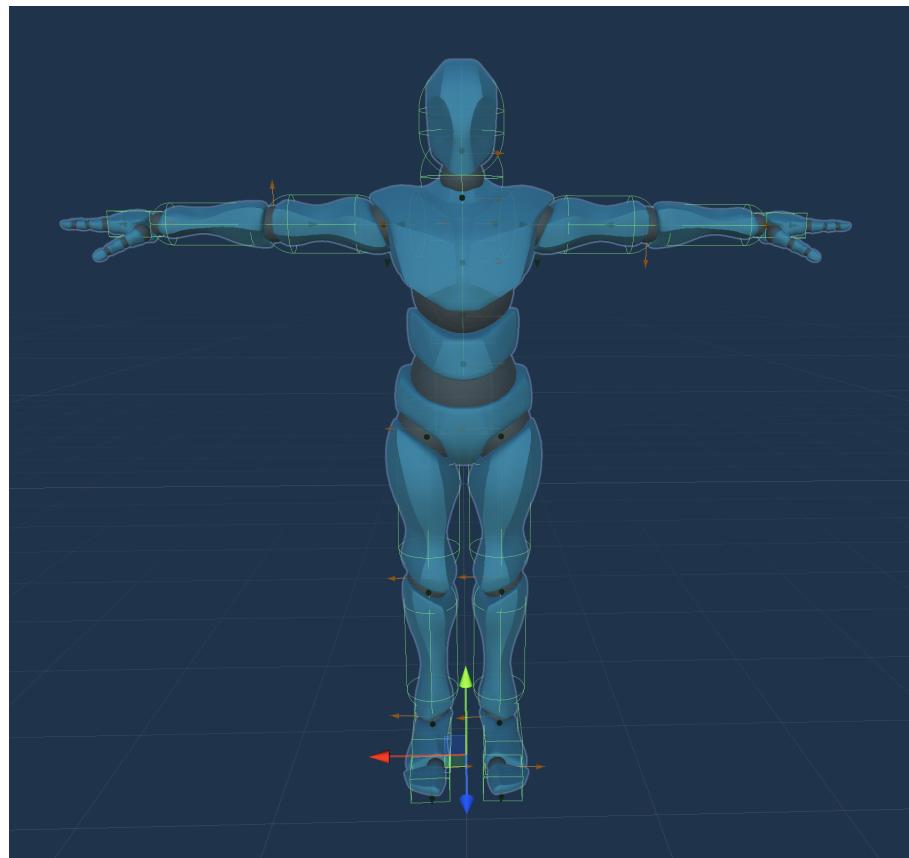


Abbildung 2.11: Unity ML-Agents Physik Charakter vereinfacht mit Kollisionskomponenten

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.

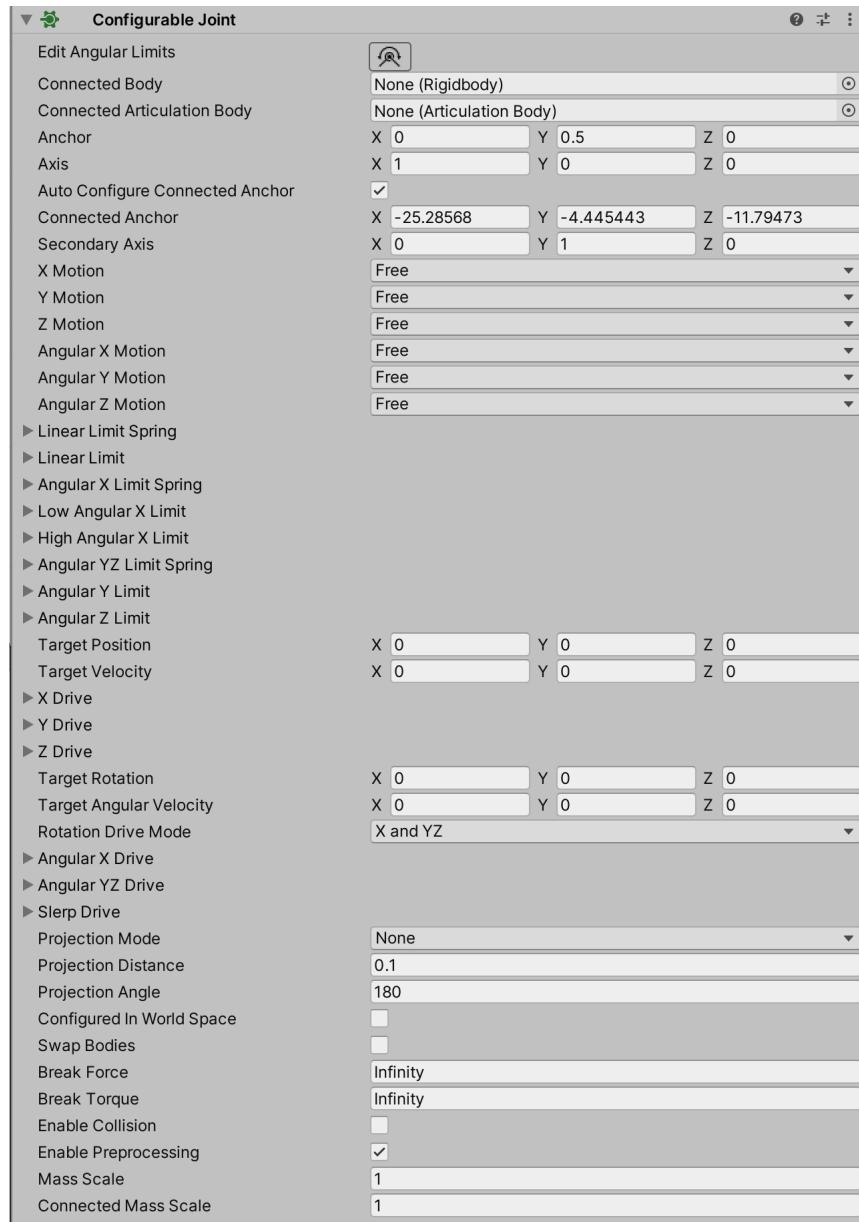


Abbildung 2.12: Unity ML-Agents Physik Gelenk

- Connected Body: bestimmt, mit welchem Körper das Gelenk 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 Bewegungs- 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 Walker Demo

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. Die Demo implementiert bereits einige grundlegende Steuerungsmechanismen, die erforderlich sind, um einen Charakter in einer Umgebung zu bewegen. Aus diesem Grund wird in dieser Arbeit die Walker-Demo als Basis für die Entwicklung genutzt.

Im folgenden Kapitel wird daher die Walker-Demo analysiert, um in den weiteren Kapiteln darauf aufzubauen. Es wird untersucht, wie die Lernumgebung aufgebaut ist. Anschließend werden der Ablauf und die Komponenten für das verstärkende Lernen analysiert. Zum Abschluss werden das Trainingsergebnis und die Bewegungsabläufe der Demo analysiert.

## 3.1 Lernumgebung

Die Umgebung besteht aus einem quadratischen Spielfeld mit einem Boden und vier 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.

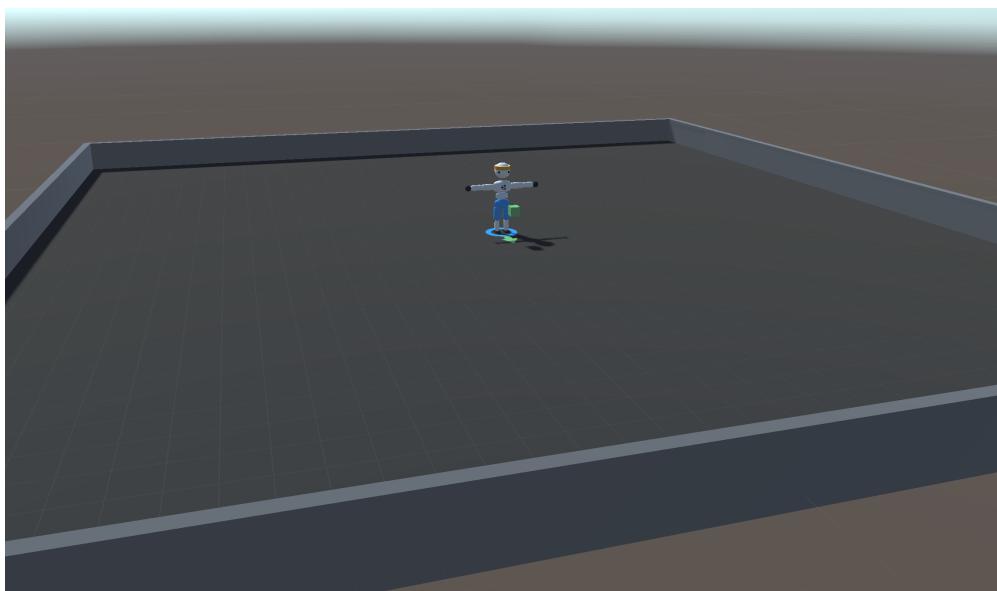


Abbildung 3.1: Walker-Demo Umgebung

Der Läufer besteht aus einfachen geometrischen Körpern. Insgesamt 11 Kapseln, drei Kugeln und zwei Quadern, von welchen jede über eine Festkörper- und eine Kollisions-Physikkomponente verfügt. 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 die gesetzten Freiheiten sowie Einschränkungen beeinflussen, wie der Läufer lernt, auf das Ziel zuzulaufen.



Abbildung 3.2: Walker-Demo Läufer

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

Das Walker Agent Skript definiert den Läufer als Agent für das maschinelle Lernen. In Abbildung 3.4 wird die Agentenkomponente im Inspektor gezeigt. Diese Komponente ist entscheidend für die Konfiguration des Läufers. Um die Komponente zu nutzen, müssen hier die Körperteile des Walkers referenziert werden. 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. Die Gelenk-Motor-Einstellungen

(Joint Drive Settings) siehe Abbildung 3.3 bestimmen die Stärke, mit welcher die Gelenke in die Zielstellung bewegt werden. Die Zielgeschwindigkeit kann manuell festgelegt werden oder während des Trainings in einem festgelegten Bereich variieren. Die Geschwindigkeit während des Trainings zu variieren hilft dem Agenten sein Verhalten besser an Umgebungsveränderungen anzupassen. Als Letztes muss auch das Zielobjekt referenziert werden.

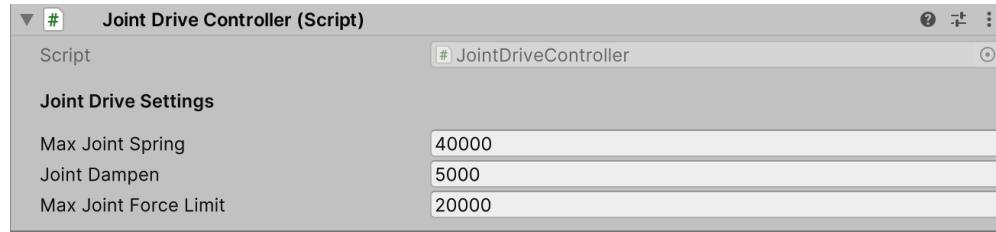


Abbildung 3.3: Gelenk Motor Steuerung

- 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. Dadurch verringert es Schwingungen.
- Max Joint Force Limit: Gibt die maximale Kraft des Gelenks an (verhindert zu schnelle Bewegung bei großer Abweichung).

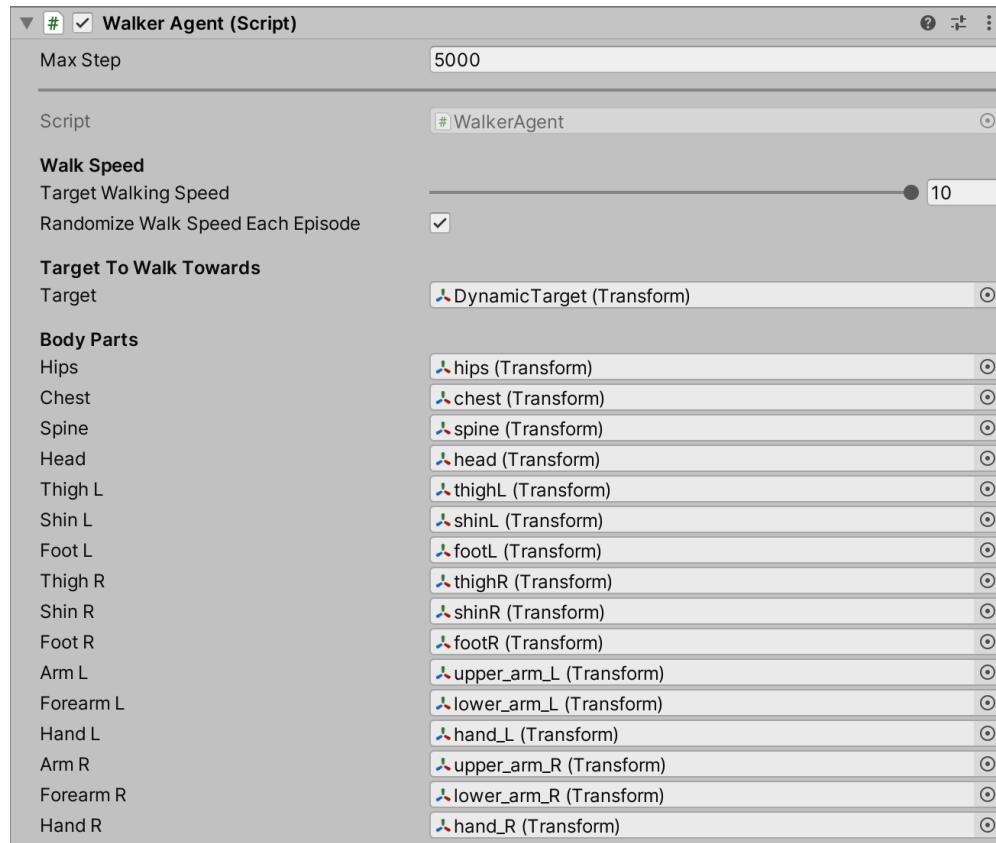


Abbildung 3.4: Agent Konfiguration

## 3.2 Training

Zu Beginn werden die Körperteile in der Gelenk-Motor-Steuerung initialisiert und das Ziel auf eine zufällige Position gesetzt. Darauf folgend beginnt die Simulation der Trainingsepisoden. Hierfür werden alle Körperteile in ihre Startposition und die Rotation des Läufers um die Y Achse zufällig gesetzt. Die zufällige Rotation hilft dabei, das Verhalten des Läufers flexibel zu gestalten. Es wird weiterhin eine zufällige Zielgeschwindigkeit gewählt, um zusätzliche Stabilität zu gewährleisten.

Sind die Vorbereitungen getroffen, beginnt die Simulation mit einer Updatefrequenz von 75 Hz für die Phsikkalkulation. Der Agent fragt jedes fünfte Physikupdate eine Entscheidung an. Bei der Simulationsfrequenz von 75 Hz ergibt das eine Frequenz von 15 Anfragen pro Sekunde. Der Grund dafür, dass die Anfragen nur jedes fünfte Update angefragt werden ist, dass der Agent durch diese Einschränkung seine Bewegungen genauer wählen muss. Es kann so verhindert werden, dass der Agent zu hastige und ruckartige Bewegungswechsel lernt. Sobald eine Entscheidung angefragt ist, erfasst der Agent den Zustand der Umgebung. Dieser wird anschließend im referenzierten Verhalten ausgewertet und eine Aktion ausgewählt.

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, eine passende Aktion auswählen und gleichermaßen sein Verhalten optimieren.

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 sie 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

Nach dem Erhalten der Aktion werden über die Gelenk-Motor-Steuerung die Zielrotationen, sowie die maximale Kraft des Gelenks festgelegt, und somit der Läufer gesteuert.

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, dass 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, dass 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.

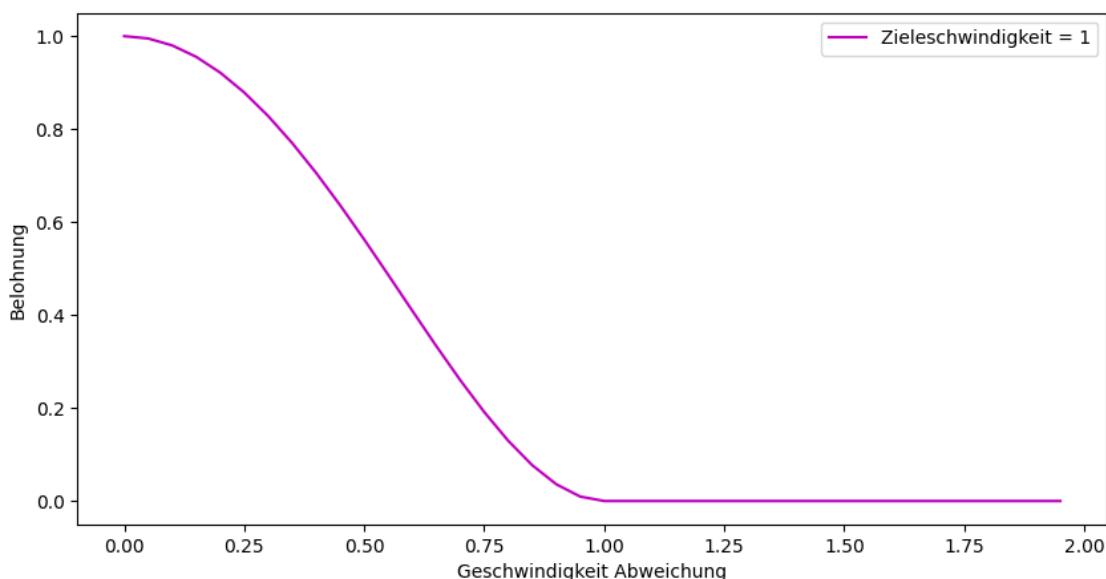


Abbildung 3.5: Walker Demo Match Velocity Belohnungsfunktion

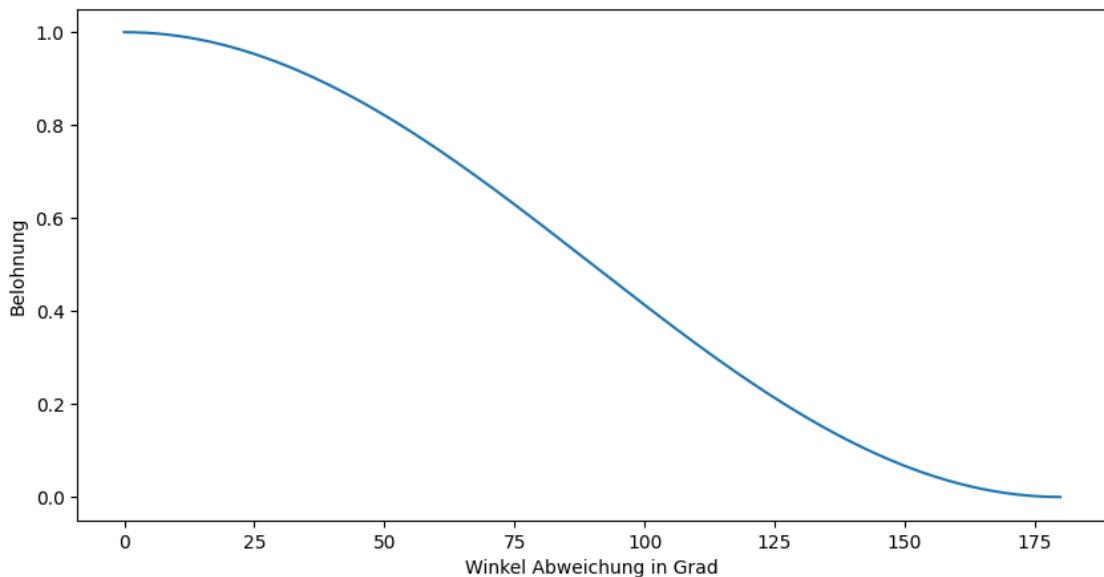


Abbildung 3.6: Walker Demo Look At Target Belohnungsfunktion

Die Belohnung wird in jedem Physikupdate neu berechnet und dem Agenten hinzugefügt. Die Belohnung wird für den Zeitraum zwischen zwei Entscheidungen aufsummiert. Bevor die nächste Entscheidung getroffen wird, wird das Tupel aus Beobachtung, Aktion und erhaltener Belohnung im Trainingspuffer gespeichert. Hat der Puffer genug Informationen gespeichert, beginnt ein Lernprozess, in welchem die zuvor gespeicherten Tupel evaluiert werden. Es wird der PPO Algorithmus auf Teilbatches ausgeführt und so schrittweise das Verhalten angepasst.

Erreicht der Läufer ein Ziel, wird dieses an eine neue zufällige Position in der Umgebung bewegt.

Die Trainingsepisode läuft solange, bis entweder 5000 Schritte erreicht sind oder der Läufer fällt. Wenn ein Körperteil des Läufers, ausgenommen den Füßen und Schienbeinen, den Boden berührt, wird die Trainingsepisode sofort beendet. Diese Technik nennt man “frühes Stoppen“. Fällt der Läufer, benötigt es eine sehr komplexe Reihenfolge an Aktionen, um zurück auf die Beine zu kommen. Das frühe Stoppen ermöglicht dem Läufer, weniger Zeit für das Lernen irrelevanter Bewegungsabläufe zu verlieren. Ist die Episode zu Ende, wird sofort eine neue gestartet.

### 3.3 Auswertung

Der Agent der Walker Demo erlernt im Laufe von 30 Millionen Trainingsschritten ein Verhalten, welches beinahe die Grenze der Episodenlänge erreicht, ohne zu fallen. In Abbildung 3.7a wird die durchschnittlich erreichte Episodenlänge in Anfragen pro Episode dargestellt. Mit 800 Anfragen pro Episode kommt man auf 4000 Trainingsschritte beziehungsweise Physikupdates. Dabei erreicht er eine durchschnittliche Belohnung pro Episode von 1600 (siehe Abbildung 3.7b). Die durchschnittliche Belohnung ist ohne Kontext erstmal nur eine Zahl. Schaut man jedoch genauer, ergibt sich die durchschnittliche Belohnung aus der durchschnittlichen Episodenlänge und der durchschnittlich erreichten Belohnung. Teilt man die durchschnittliche Belohnung mit der Anzahl an Schritten, kommt man auf eine durchschnittliche Belohnung von ca. 0.4 pro Schritt. Die im Verlauf der Arbeit hinzugefügten Statistiken der Belohnungen zeigen eine Aufteilung von 0.9 Belohnung für die Blickrichtung und 0.45 für das

Halten der Zielgeschwindigkeit (siehe Abbildung 3.7c, 3.7d). Eine Blickrichtungsbelohnung von 0.9 ergibt eine Abweichung von durchschnittlich 35 Grad. Die Zielgeschwindigkeitsbelohnung ergibt eine Abweichung von ca. 51%.

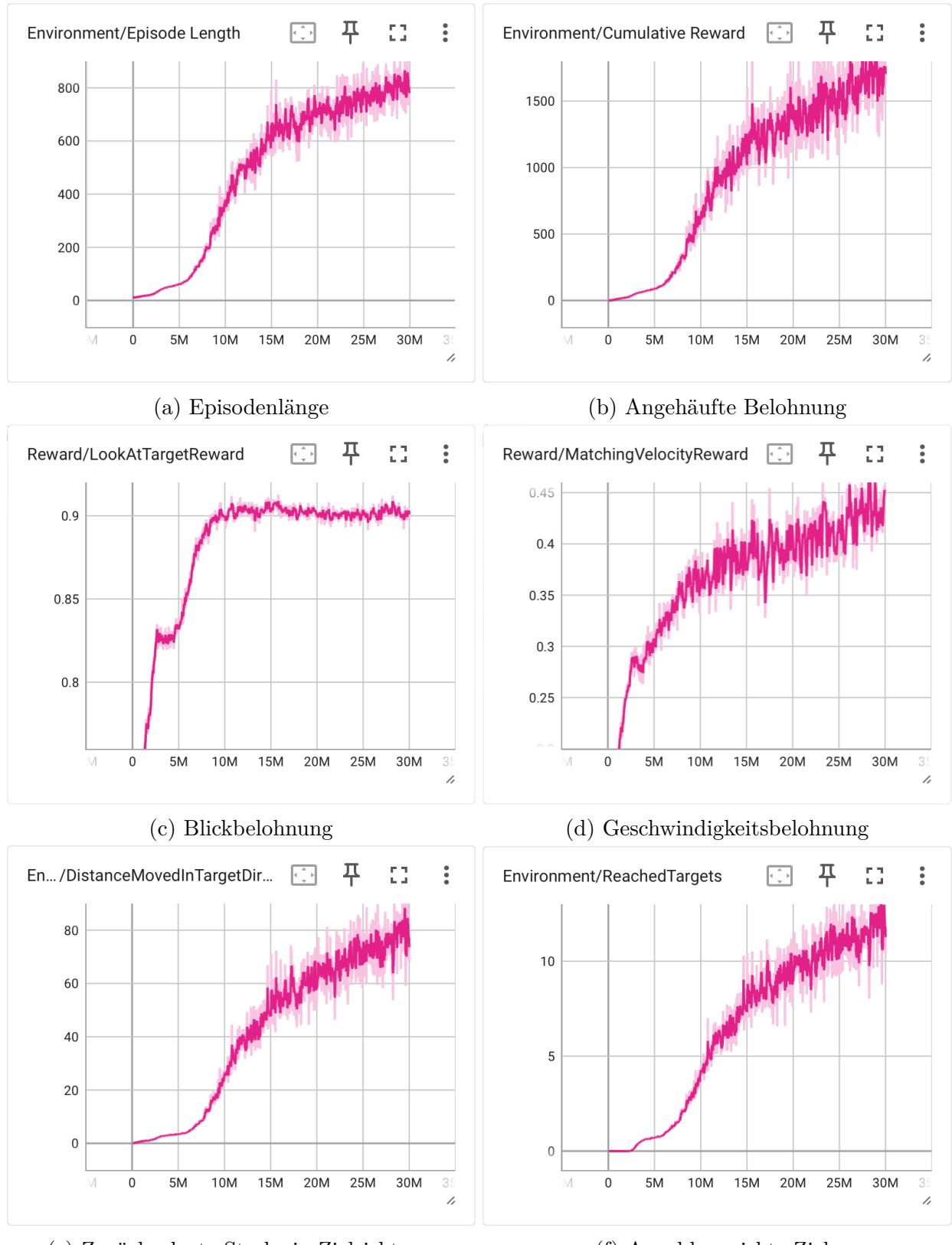


Abbildung 3.7: Walker Demo Training Graphen

Der Läufer ist nicht in der Lage, die Belohnungen pro Zeitschritt maximal auszureißen, sondern steigert die Länge der Episode, indem er die Sturzrisiken minimiert. Dies führt zu steigender Belohnung innerhalb der Episode. Wie in Abbildung 3.7e und 3.7f zu sehen ist, läuft er während einer Episode durchschnittlich eine Distanz von 80 Einheiten und erreicht dabei 10,4 Ziele. Der Läufer lernt sich stabil zum Ziel zu bewegen. In Abbildung 3.8 ist das Gangbild des Läufers abgebildet. Das Gangbild wechselt periodisch das Standbein, setzt den einen Fuß vor den anderen und drückt sich über das Standbein voran. Die Arme jedoch schwingt der Läufer sehr stark um das Gleichgewicht halten zu können. Das Gangbild ist daher nicht so wie von einem zweibeinigen menschenähnlichen Charakter erwartet.

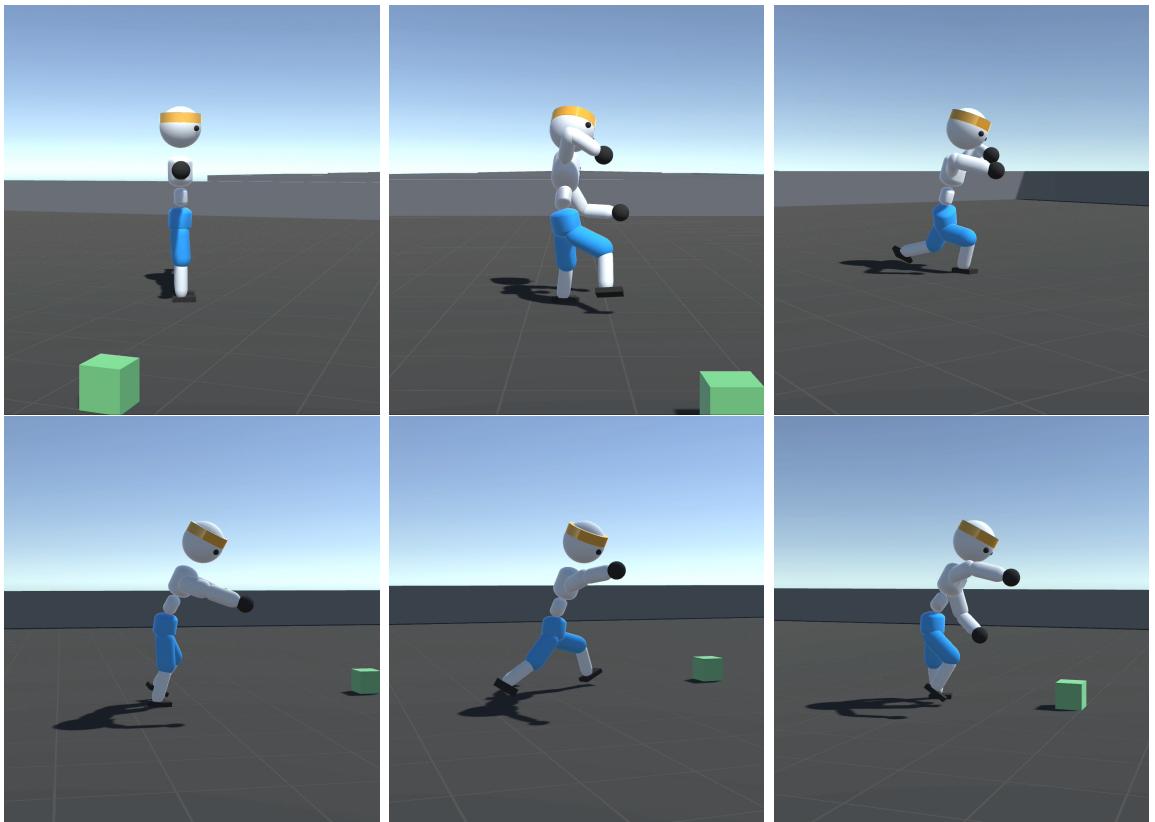


Abbildung 3.8: Walker Demo Analyse Gangbild

# 4 Umsetzung Charaktercontroller

Folgender Abschnitt geht auf die Anforderungen eines Charaktercontrollers sowie die Entwicklung innerhalb dieser Arbeit ein. Dabei werden verschiedene Ansätze getestet, implementiert und evaluiert. Die Versuche beinhalten jeweils Planung, Umsetzung und Auswertung.

## 4.1 Nutzersteuerung

### 4.1.1 Anforderungen

Um von einem Charaktercontroller sprechen zu können, muss der Agent über Benutzereingaben gesteuert werden. Je nach Spielgenre erfolgt die Steuerung des Charakters unterschiedlich. Für eine intuitive Steuerung wird der Charakter in den meisten Spielen relativ zur Kameraansicht bewegt. Bei First- oder Third-Person-Spielen kann die Kameraansicht zusätzlich über Mausbewegungen gedreht werden. Andere Titel mit einer Top-Down-Ansicht haben hingegen eine feste Kameraansicht. In diesen Spielen wird die Ausrichtung des Charakters daher durch die Mausposition bestimmt. Um den Walker-Agenten zu steuern, muss das Ziel des Läufers zur Laufzeit je nach Benutzereingabe bewegt werden.

In den folgenden Abschnitten wurden beide Ansätze in einem Charaktercontroller-Skript zur Steuerung des Walker-Agents implementiert.

### 4.1.2 First- und Thirdperson-Steuerung

Die First- oder Third-Person-Steuerung liest zunächst die Tastatureingaben sowie die Mausbewegung auf der Y-Achse ein. Anschließend wird der aktuelle Rotationswinkel basierend auf der Mausbewegung angepasst. Dabei wird die Distanz, die die Maus seit dem letzten Update zurückgelegt hat, nach links als negativer Wert und nach rechts als positiver Wert auf den aktuellen Rotationswinkel addiert. Die Rotation wird durch eine Quaternion bestimmt, die um die vertikale Weltachse gedreht wird. Um die Richtungsvektoren zu berechnen, wird die zuvor berechnete Rotation auf die Basisvektoren angewendet. Zum Schluss werden die Richtungsvektoren mit den Tastatureingaben multipliziert. Mit den Tastatureingaben W und S wird der Input in vertikaler Richtung im Bereich von -1 bis 1 angegeben. Mit den Tastatureingaben A und D wird gleichermaßen die horizontale Richtung angegeben. Abschließend wird das Ziel an die errechnete Position gesetzt, wodurch der Läufer in die angegebene Richtung läuft.

```

1 public virtual void FixedUpdate()
2 {
3     //Einlesen Tastatur Input
4     float inputHor = Input.GetAxis("Horizontal");
5     float inputVert = Input.GetAxis("Vertical");
6
7     Vector3 position;
8
9     //Einlesen Maus Input
10    float mouseX = Input.GetAxis("Mouse X");
11    rotAngle += mouseX;
12

```

```

13     //Berechnung der Rotation
14     rotation = Quaternion.AngleAxis(rotAngle, Vector3.up);
15
16     //Anwendung der Rotation auf Richtungsvektoren
17     Vector3 directionForward = rotation * Vector3.forward;
18     Vector3 directionRight = rotation * Vector3.right;
19
20     //Position berechnen
21     position = root.position + directionForward * inputVert +
22                 directionRight * inputHor;
22
23     //Setzen der Zielposition
24     target.position = position;
25 }
```

Listing 4.1: Nutzersteuerung für First- und Thirdperson

#### 4.1.3 Top-Down-Steuerung

Das Grundgerüst für die Top-Down-Steuerung ist das gleiche wie bei der First- oder Third-Person-Steuerung. Es unterscheidet sich jedoch darin, dass nicht die Mausbewegung, sondern die Mausposition eingelesen wird. Die Mausposition wird zunächst von einer Pixelkoordinate in eine relative Koordinate im Bereich von 0 bis 1 normiert. Diese Normierung gewährleistet eine konsistente Steuerung unabhängig von der Bildschirmgröße. Anschließend wird die relative Koordinate in einen Bereich von -1 bis 1 konvertiert, um die Position in Relation zum Bildschirmsmittelpunkt darzustellen, anstatt zur unteren linken Ecke. Der Vektor, bestehend aus den relativen Koordinaten, gibt die Blickrichtung an. Um die Zielposition zu berechnen, wird die Tastatureingabe mit den Richtungsvektoren der Weltachsen multipliziert. Abschließend wird das Ziel an die berechnete Position gesetzt, wodurch der Läufer in die angegebene Richtung läuft.

```

1 public virtual void FixedUpdate()
2 {
3     //Einlesen Tastatur Input
4     float inputHor = Input.GetAxis("Horizontal");
5     float inputVert = Input.GetAxis("Vertical");
6
7     Vector3 position;
8
9     //Einlesen Maus Position
10    Vector3 mousePos = Input.mousePosition;
11
12    //Maus Position normalisieren relativ zu Bildschirmauflösung
13    float normalizedMouseX = 2 * (mousePos.x / Screen.width) -
14        1;
15    float normalizedMouseY = 2 * (mousePos.y / Screen.height) -
16        1;
15    mousePos = new Vector3(normalizedMouseX, 0,
17                           normalizedMouseY);
16
17     //Berechnung der Rotation
```

```
18     rotation = Quaternion.LookRotation(mousePos, Vector3.up);  
19  
20     //Position berechnen  
21     position = root.position + Vector3.forward * inputVert +  
22         Vector3.right * inputHor;  
23  
24     //Setzen der Zielposition  
25     target.position = position;  
26 }
```

Listing 4.2: Nutzersteuerung für Top-Down

## 4.2 Zusätzliche Bewegungsabläufe

### 4.2.1 Anforderungen

Dieses Kapitel beschäftigt sich mit den Einschränkungen der Walker-Demonstration im Bezug auf unterschiedliche Bewegungsrichtungen. Das trainierte Modell der Walker Demo beherrscht nur die Fortbewegung in Blickrichtung. Der Läufer ist auch nicht darauf trainiert stehen zu bleiben. Das resultiert darin, dass der Läufer fällt sobald der Nutzer keinen Tastaturinput gibt. Es werden Anpassungen am Trainingsablauf getestet um den Läufer darauf zu trainieren unabhängig von der Bewegungsrichtung eine zusätzliche Blickrichtung zu halten, sowie auf der stelle zu stehen. Zu beginn werden unterschiedliche Bewegungsabläufe in einzelnen Modellen trainiert um zu prüfen ob die Limits des Läufers das erlernen dieser erlauben. Anschließend werden unterschiedliche Ansätze getestet um die Bewegungsabläufe in einem System zu kombinieren.

### 4.2.2 Separate Bewegungsabläufe

Das erste Ziel ist es dem Läufer das stehen bleiben beizubringen. Hierfür soll der Läufer sich möglichst wenig von der aktuellen Position bewegen. Die Hauptaufgabe ist es hierbei das Gleichgewicht zu halten.

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.

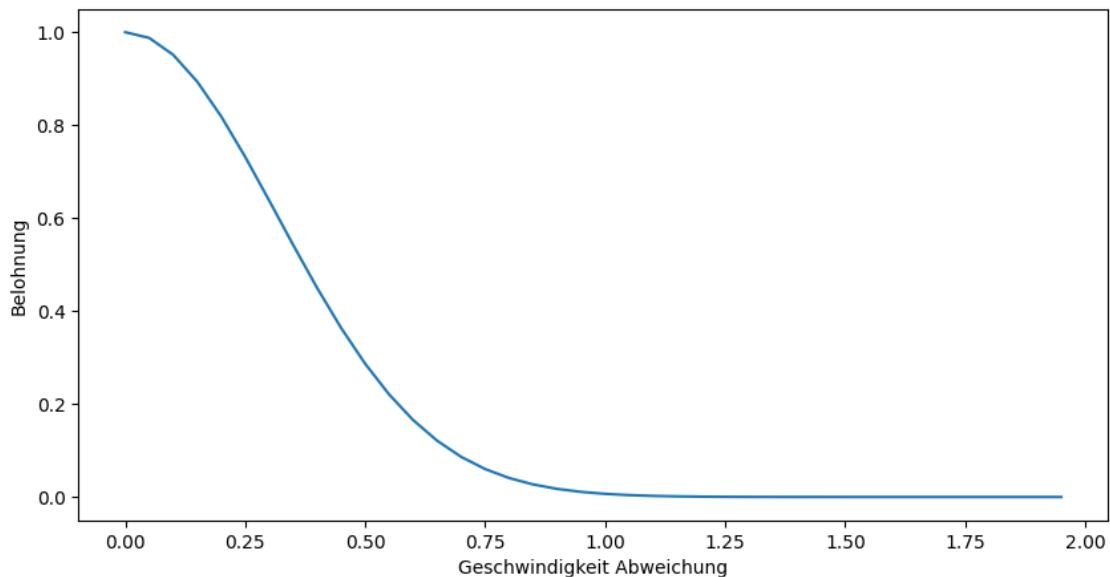


Abbildung 4.1: Neue Sigmoid Geschwindigkeit Belohnungsfunktion

Als neue Belohnungsfunktion wird eine ähnliche Sigmoid Funktion genutzt. Die Belohnungsfunktion nimmt für eine perfekte Übereinstimmung zwischen aktueller Geschwindigkeit und Zielgeschwindigkeit eine Belohnung von 1 an. Mit steigender Abweichung nimmt auch die Belohnung immer weiter ab. Ist eine Abweichung von größer 1 erreicht ist die Belohnung 0. Die Belohnungsfunktion aus Abbildung 4.1 wird daher ab hier Neue Belohnungsfunktion genannt.

Der Walker konnte mit der DeepMimic Belohnungsfunktion lernen auf der Stelle zu stehen. Abbildung 4.2e, 4.2a 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 4.2c, 4.2d.

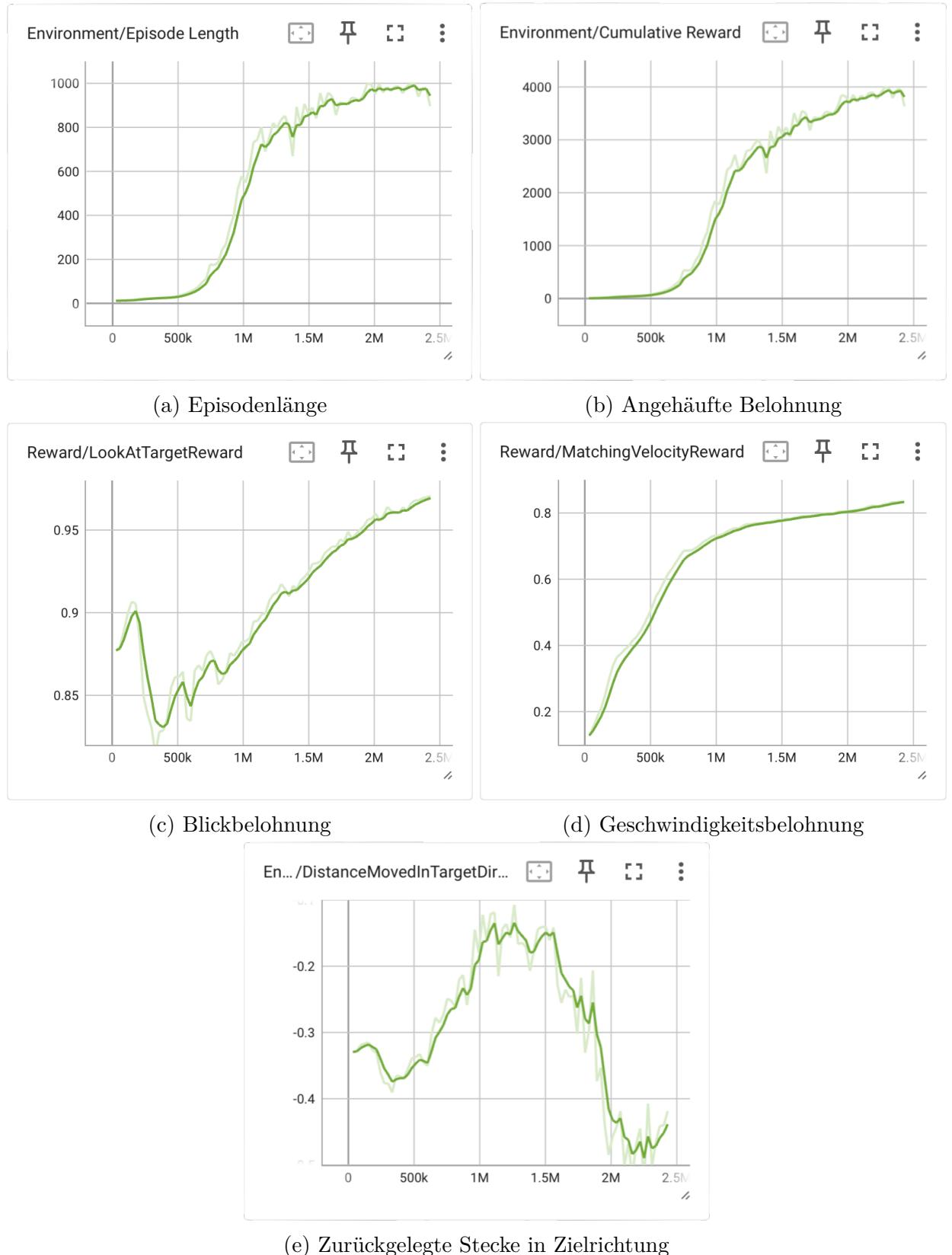


Abbildung 4.2: Versuch 4 Training Graphen

Mit zufälliger Zielgeschwindigkeit zu einem Ziel zu laufen wie im Ursprünglichen Verhalten konnte damit jedoch nicht zufriedenstellend erlernt werden. Die Abbildung 4.3 zeigt mit der orangenen Linie die Leistung der Neuen Belohnungsfunktion und mit der rosa 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 4.4). Diese Anpassung verbessert die Generalisierung zwischen den wechselnden Geschwindigkeiten um ein vielfaches.

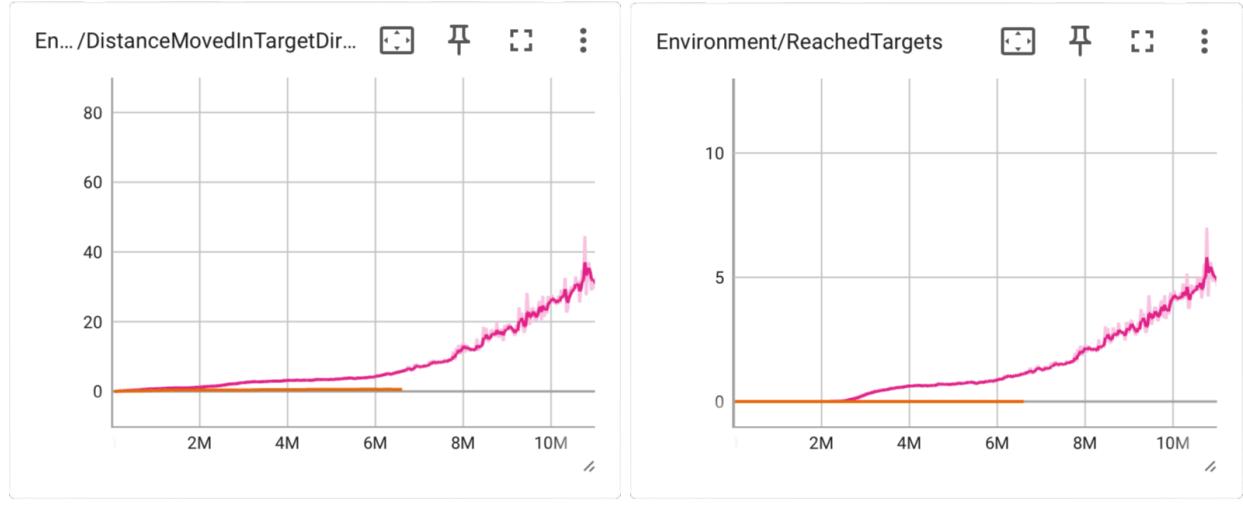


Abbildung 4.3: Vergleich von Lauftraining mit Demo Belohnungsfunktion gegen DeepMimic Belohnungsfunktion

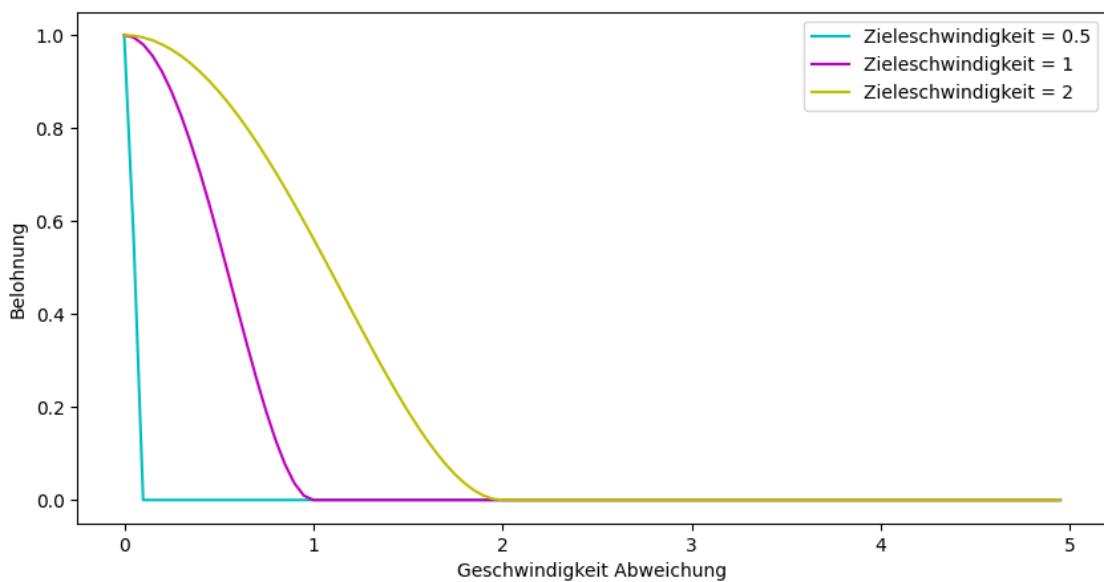


Abbildung 4.4: Vergleich der Demo Belohnungsfunktion unter verschiedenen Zielgeschwindigkeiten

Mit dieser Erkenntnis aus wird eine neue Anpassung untersucht. In der folgenden Anpassung bleibt die Belohnungsfunktion weitestgehend Unverändert. Lediglich das obere Limit ab welchem die Funktion eine Belohnung von 0 annimmt, wird auf ein minimum von 0.1

beschränkt. Somit kann sichergestellt werden, dass im Bereich der normalen Fortbewegung keine Veränderung auftritt. Mit der Demo Belohnungsfunktion konnten nur Annäherungen an eine Zielgeschwindigkeit von 0 genutzt werden. Bei immer weiterer Annäherung an 0 wird das Spektrum an akzeptablen Geschwindigkeiten bevor die Belohnung 0 ist nahezu unerreichbar (siehe Abbildung 4.5). Mit dem Limit von 0.1 ist der Bereich an Abweichungen für die die Belohnungsfunktion einen Wert größer 0 annimmt groß genug. Der Läufer kann durch ausprobieren Belohnungen über 0 erreichen, wodurch eine Richtung für die Optimierung ermittelt werden kann. Somit kann der Läufer die Belohnung optimieren.

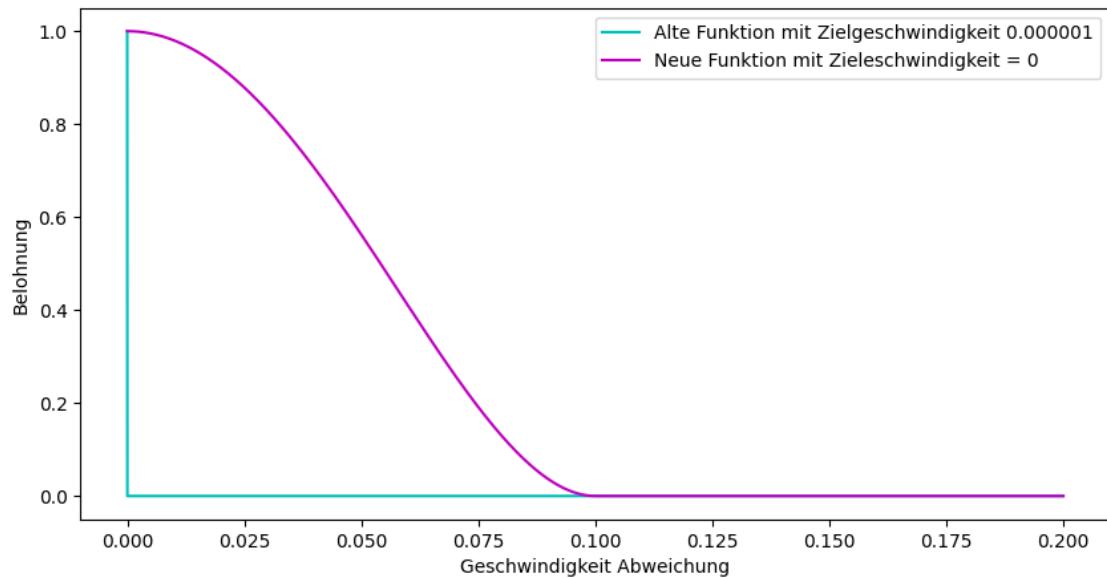


Abbildung 4.5: Vergleich Demo gegen Belohnungsfunktion mit 0.1 Limit

Das auf einer Stelle stehen hat der Läufer damit in einem separaten Training auch erlernt. Abbildung 4.6a zeigt das der Läufer die maximale Episoden Länge von 1000 erreicht hat ohne zu fallen. Die bewegte Distanz hat sich auch 0 angenähert (siehe 4.6e). Die Belohnungen wurden auch weitestgehend optimiert siehe Abbildung 4.6c, 4.6d.

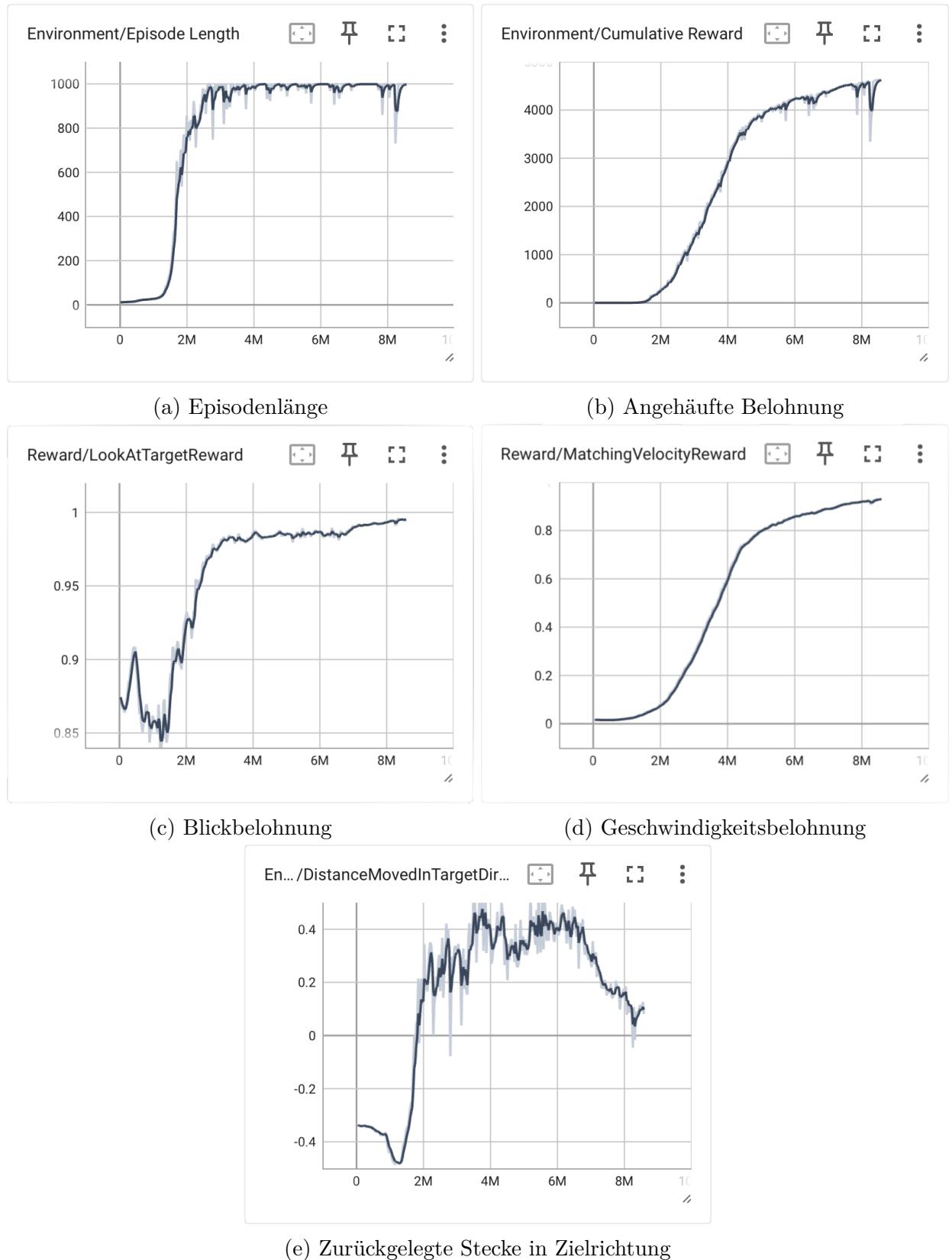


Abbildung 4.6: Versuch 5 Training Graphen

Nach dem erfolgreichen stehen auf einer festen Position, sind die nächsten Bewegungsziele die Bewegung zum Ziel mit unterschiedlichen Blickrichtungen. Die Extremfälle sind hier das Laufen rückwärts und seitwärts. Im folgenden wird daher untersucht ob das Laufen in

unterschiedliche Richtungen relativ zur Blickrichtung für den Läufer erlernbar ist. Um das zu realisieren wurde die Blickrichtung Belohnungsfunktion relativ zur Zielrichtung angepasst. Bei Vorwärtsbewegung ist die Blickrichtung gleich der Zielrichtung. Bei Seitlicherbewegung ist die Blickrichtung im rechten Winkel zur Zielrichtung. Bei der Rückwärtsbewegung ist die Blickrichtung entgegen der Zielrichtung. In 4.3 ist die Implementierung für das bestimmen der Blickrichtung 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     var lookAtTargetReward = (Vector3.Dot(lookDirection,
28                                         headForward) + 1) * 0.5F;
29 }
```

Listing 4.3: Blickrichtung Enum und Belohnung

Das gehen in Zielrichtung wurde durch die Änderungen nicht beeinflusst. Separate Trainings zu den drei anderen Laufrichtungen waren erfolgreich. Abbildung 4.7e, 4.7f zeigen die zurückgelegte Distanz und die Anzahl an erreichten Zielen in einer Trainingsepisode. Die Ergebnisse der 3 Laufrichtungen sind alle vergleichbar mit den Ergebnissen der Demo. Die Abweichung der Laufrichtung Links ist vermutlich der zufälligen Natur des Trainings anzurechnen.

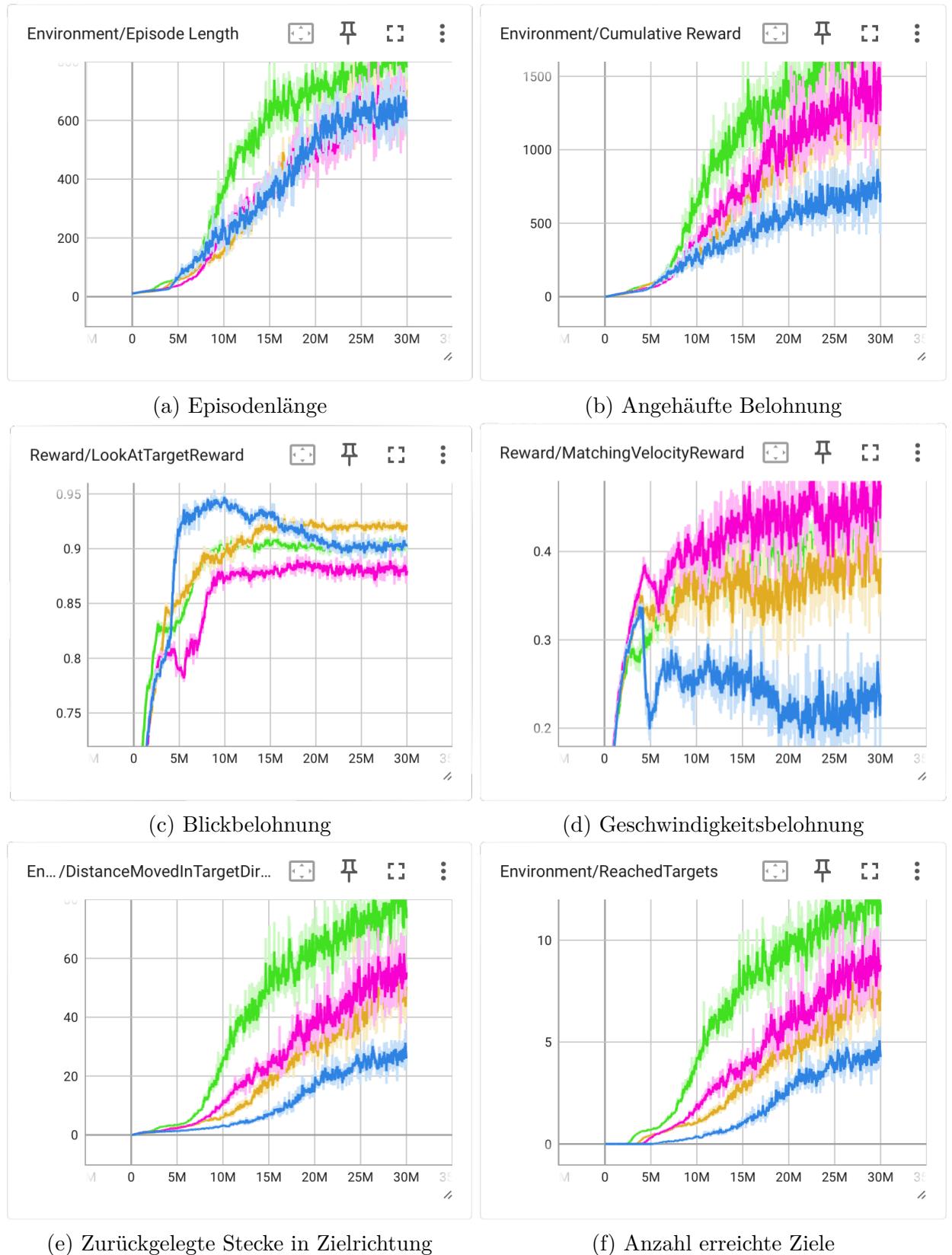


Abbildung 4.7: Unterschiedliche Blickrichtungen Training Graphen (grün = vorwärts, orange = rückwärts, rosa = rechts, blau = links)

### 4.2.3 360 Grad Blickziel

Der erste Ansatz Look at target mit Winkelabweichung von Zielrichtung platziert. Winke labweichung von 30 grad zu gering, dadurch ist Belohnungsverlust nicht signifikant genug um ein neues Verhalten zu lernen. (ignoriert Look Target und nimmt bei größerer Winke labweichung Belohnungsverlust in kauf) 102 / 103 Winkelabweichung von 90 grad, Läufer lernt Blickrichtung leicht anzupassen. (läuft leicht schräg?) 104 Winkelabweichung von 180 grad, Läufer lernt durch nach unten schauen für schwere Blickziele kann die Belohnung von negativen Werten auf 0 begrenzt werden. 105 Hinzufügen von extra Bestrafung für Blick richtung Abweichung vom Horizont. (Läufer läuft nur rückwärts, vermutlich weil durch die nicht bewegten Ziele und das vorwärts bewegen die Ziele mit einer hohen Wahrscheinlichkeit im laufe der Episode sich hinter dem Läufer befinden.) 112 Look Target in jedem Update neu gesetzt um Blickrichtungswinkel beizubehalten und Ziel wird nur neu gesetzt wenn Blickbelohnung durchschnittlich größer 0.7 innerhalb der Episode. (Läuft nur noch seitwärts, dadurch werden alle Winkelabweichungen im durchschnitt am besten erreicht) 113 Belohnungsabfall gesteigert um bei größerer Abweichung kaum noch Belohnung erreicht wird. (Schaut recht zuverlässig zu Ziel aber lernt nicht zu laufen) 114 Ziel wird neu gesetzt wenn Läufer 3 sek auf das Ziel schaut (mit sperecast). (Schaut zuverlässig zu Ziel aber lernt sehr langsam das Laufen) 117 Verringern der Look at Zeit auf 2 sek. (Lernt gut zu laufen aber Blickbelohnung fällt extrem ab, schaut nicht auf Ziel) 118 / 119

### 4.2.4 4 Laufrichtungen

#### 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 4.4: Laufrichtung Modell wechseln

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

## 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 4.5: Laufrichtung zufällig zum Start und beim erreichen von Ziel

Codeausschnitt 4.5 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 aufgerufen. 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. 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 4.8).

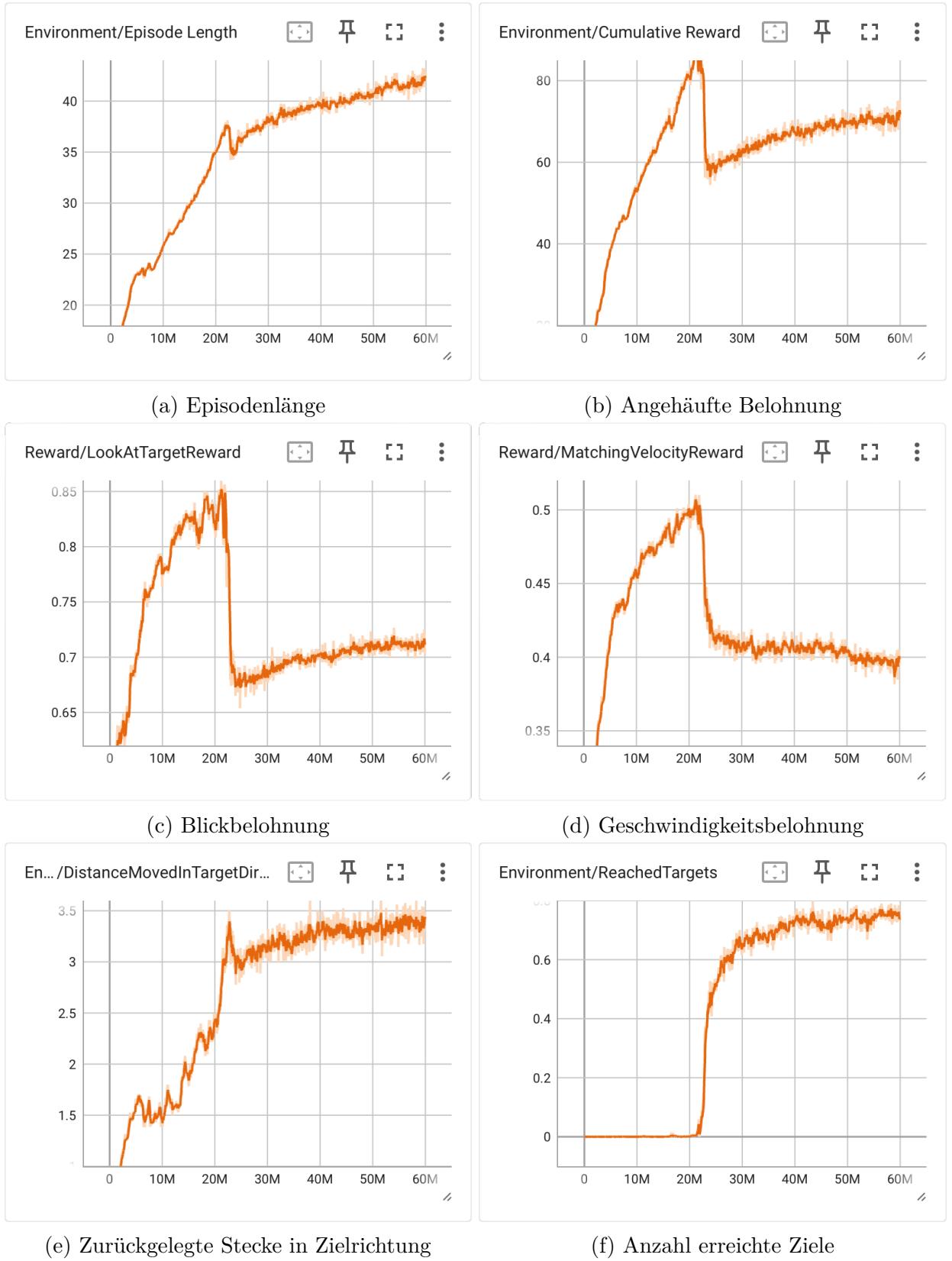


Abbildung 4.8: Versuch 8 Training Graphen

## **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.

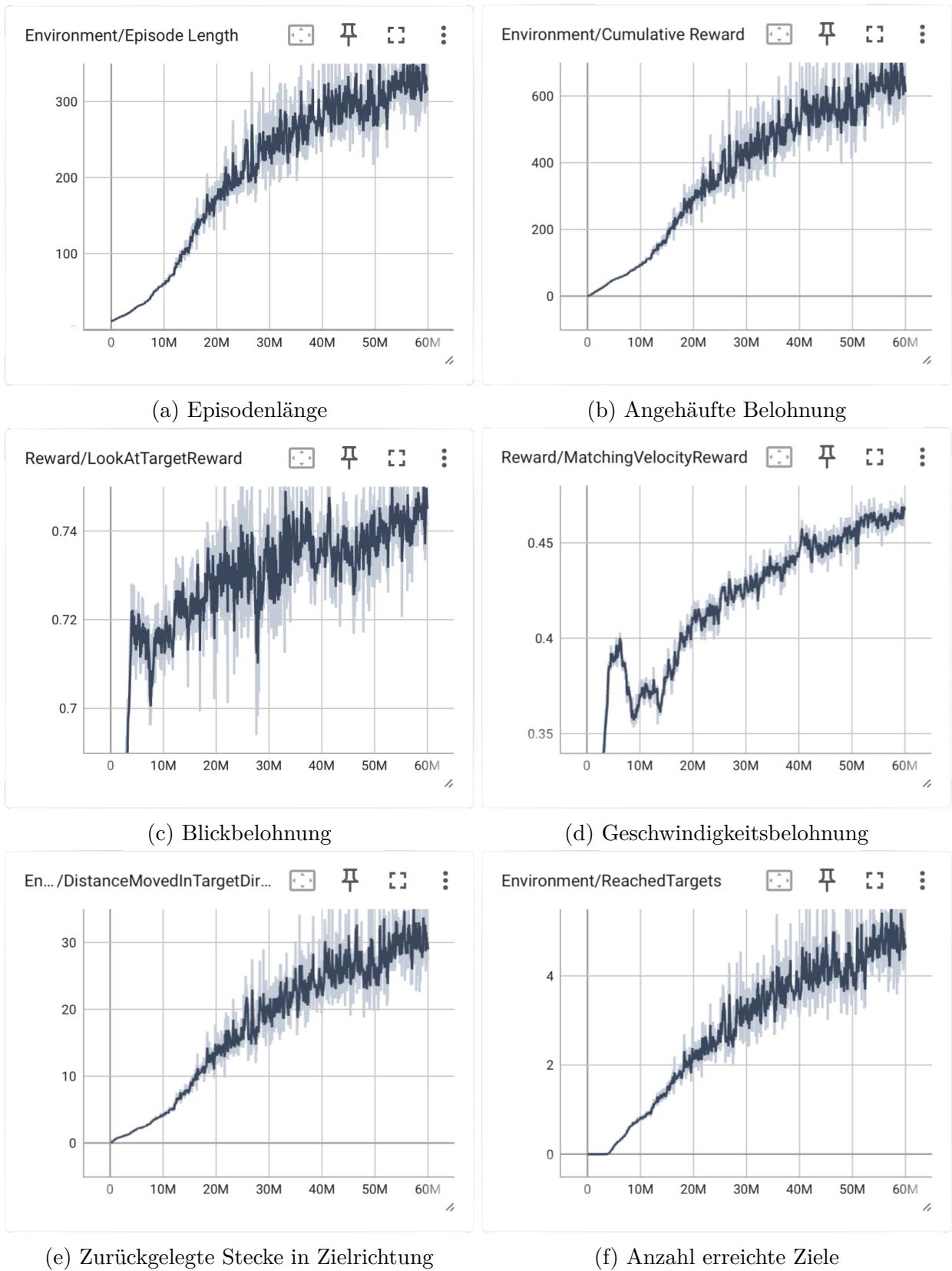


Abbildung 4.9: Versuch 9 Training Graphen

Das gleichbleiben der Laufbewegung über die komplette Trainingsepisode hat zur Folge dass das Training stabiler verläuft siehe Abbildung 4.9. 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 Unterschiedliche Charakter

Spiele verwenden die unterschiedlichsten Charaktere, nicht nur das Aussehen sondern auch die Komplexität der Körperteile und die Anzahl der Knochen variiert. Um den Charaktercontroller vielseitig einsetzbar zu gestalten wird in diesem Kapitel die Walker Demo Komponenten angepasst um den Einrichtungsprozess zu vereinfachen und unterschiedliche Charakter Körperstrukturen zu erlauben. Es wird analysiert wie das Agentenskript angepasst werden muss um Charaktere mit unterschiedlichen Körperstrukturen zu steuern und zu trainieren. Anschließend wird der Einrichtungsprozess am Beispiel eines Mixamo 3D Charakter Modells dargestellt. Zum Schluss wird der Mixamo Charakter trainiert und das Ergebnis ausgewertet.

### 4.3.1 Anforderungen

Der Agent der Walker Demo setzt für jedes Körperteil eine separate Referenz, die über den Inspector in Unity konfiguriert werden muss. Um die damit verbundene Einschränkung einer festgelegten Konfiguration von Körperteilen zu beheben, soll eine flexible Liste von Körperteilen konfiguriert werden. Bei der Zustandserfassung der Körperteile soll anschließend der Zustand aller Körperteile in der Liste erfasst werden. Die Aktion soll gleichermaßen Zielwinkel und Gelenkstärke für alle Körperteile der Liste beinhalten. Um unnötige Komplexität bei der Beobachtung und Aktion zu vermeiden, sollen die Zielwinkel nur für bewegbare Gelenkkachsen bestimmt werden. Die Gelenkstärke soll für komplett versteifte Gelenke ebenfalls ausgelassen werden. Um die Konfiguration weiter zu erleichtern, sollen die Körperteile zudem automatisch dem Walker-Agenten hinzugefügt werden. Schließlich soll auch das Stabilisierungsobjekt automatisch generiert werden.

### 4.3.2 Anpassungen

Um ein Objekt als Körperteil zu definieren, wurde die Bodypart-Klasse in ein MonoBehaviour-Unity-Skript umgewandelt. Zusätzlich wurden die Funktionen zur Steuerung der Körperteile von der Gelenk-Motor-Steuerung in das Körperteil-Skript verlagert. Dadurch ist es möglich, die Körperteile direkt über das Körperteil-Skript zu steuern, ohne den Umweg über die Gelenk-Motor-Steuerung. Die Körperteilkomponente initialisiert sich beim Programmstart und ist anschließend sowohl für die Steuerung als auch für die Aktualisierung der Zustandsparameter des Körperteils zuständig. Jedes Körperteil benötigt eine Festkörperkomponente. Zusätzlich wird überprüft, ob das Objekt über eine Gelenkkomponente verfügt. Existiert eine solche Komponente, wird diese eingerichtet und die Freiheitsgrade bestimmt. Diese Freiheitsgrade geben an, welche Felder für das Körperteil in der Beobachtung und Aktion hinzugefügt werden müssen.

Der Walker-Agent sucht beim Programmstart alle Körperteile und speichert sie in einer Liste ab. Beim Erstellen der Beobachtung und beim Umwandeln der Aktion in eine Bewegung wird über die Körperteilliste iteriert, und für jedes Körperteil wird die entsprechende Beobachtung erstellt oder die Aktion ausgeführt. Das Stabilisierungsobjekt wird auch automatisch beim initialisieren des Agenten erstellt (siehe 4.6).

```
1 public override void Initialize()
2 {
3     //Stabilisierungs- / Orientierungsobjekt erstellen
4     GameObject orientationObject = new
5         GameObject("OrientationObject");
6     orientationObject.transform.parent = transform;
7     walkOrientationCube =
8         orientationObject.AddComponent<OrientationCubeController1>();
9     walkOrientationCube.root = root;
10    walkOrientationCube.target = target;
11
12    //Körperteile initialisieren und zur Liste hinzufügen
13    foreach (Bodypart bps in
14        root.GetComponentsInChildren<Bodypart>())
15    {
16        bps.Initialize();
17        bps.onTouchingGround.AddListener(OnTouchingGround);
18        bp.onTouchedTarget.AddListener(OnTouchedTarget);
19        bodyparts.Add(bps);
20    }
21
22    //Körperteile zurücksetzen
23    public override void OnEpisodeBegin()
24    {
25        foreach (Bodypart bps in bodyparts)
26        {
27            bps.Reset();
28        }
29
30    //Beobachtung für Körperteil hinzufügen
31    public void CollectObservationBodyPart(Bodypart bps, VectorSensor
32        sensor)
33    {
34        sensor.AddObservation(bps.touchingGround);
35
36        sensor.AddObservation(m_OrientationCube.transform
37            .InverseTransformDirection(bps.rb.velocity));
38        sensor.AddObservation(m_OrientationCube.transform
39            .InverseTransformDirection(bps.rb.angularVelocity));
40
41        sensor.AddObservation(m_OrientationCube.transform
42            .InverseTransformDirection(bps.rb.position - root.position));
43
44        if (bps.dof.sqrMagnitude <= 0) return;
45
46        sensor.AddObservation(bps.rb.transform.localRotation);
47        sensor.AddObservation(bps.currentStrength /
48            bps.physicsConfig.maxJointForceLimit);
```

```

47 }
48
49 //Körperteilbeobachtungen an Beobachtung anfügen
50 public override void CollectObservations(VectorSensor sensor)
51 {
52     foreach (Bodypart bps in bodyparts)
53     {
54         CollectObservationBodyPart(bps, sensor);
55     }
56 }
57
58 //Aktion in Bewegung umwandeln
59 public override void OnActionReceived(ActionBuffers actionBuffers)
60 {
61     var continuousActions = actionBuffers.ContinuousActions;
62     int i = -1;
63
64     foreach (Bodypart bp in bodyparts)
65     {
66         if (bp.dof.sqrMagnitude <= 0) continue;
67         float targetRotX = bp.dof.x == 1 ? continuousActions[++i] :
68             0;
69         float targetRotY = bp.dof.y == 1 ? continuousActions[++i] :
70             0;
71         float targetRotZ = bp.dof.z == 1 ? continuousActions[++i] :
72             0;
73         float jointStrength = continuousActions[++i];
74         bp.SetJointTargetRotation(targetRotX, targetRotY,
75             targetRotZ);
76         bp.SetJointStrength(jointStrength);
77     }
78 }

```

Listing 4.6: Ausschnitt Angepasstes Walker Agent Skript

### 4.3.3 Einrichtung

Der ausgewählte Charakter ist der Y Bot Charakter welcher in Abbildung 4.10 zu sehen ist. Der Y Bot besteht ausgenommen der Finger aus 22 Knochen. Um das Training zu beschleunigen werden für alle Versuche die Finger mit dem Handknochen als ein Körperteil zusammengefasst.

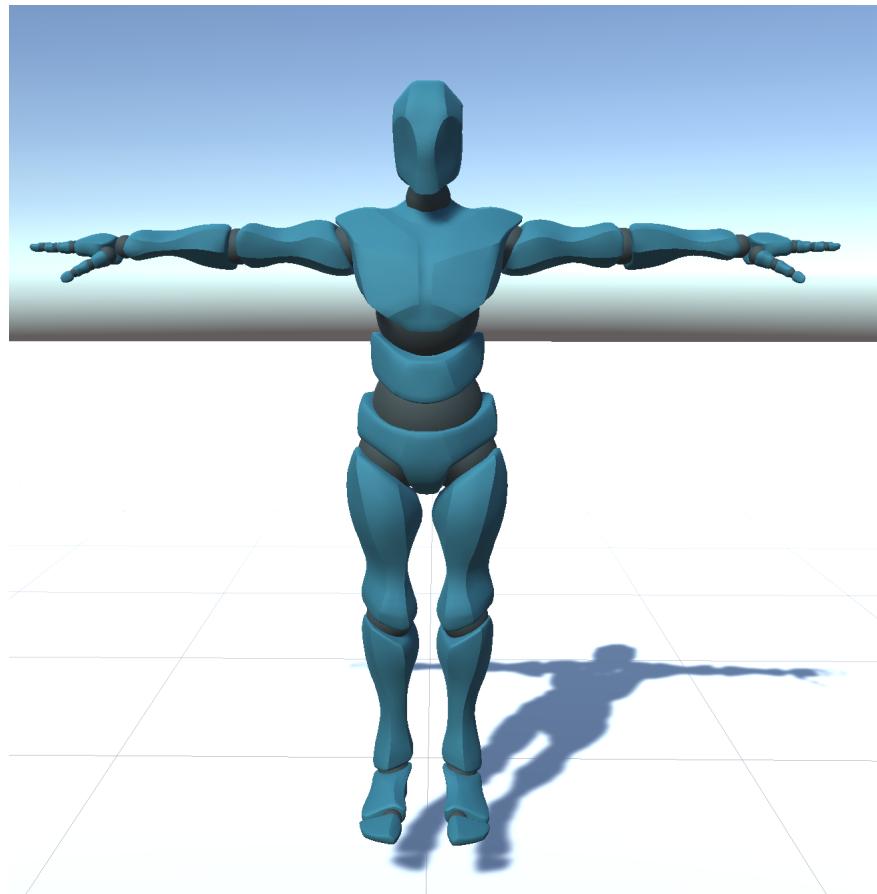


Abbildung 4.10: Mixamo Charakter Y Bot

Jedes Körperteil benötigt für das Steuern und Trainieren mit dem Walker Agenten Skript eine Kollisionskomponente, eine Festkörperkomponente und eine Körperteilkomponente. Zusätzlich müssen Gelenkkomponenten hinzugefügt werden, um die Körperteile miteinander zu verbinden. Dabei wird die Gelenkkomponente jeweils auf das untergeordnete Körperteil angewendet, während das übergeordnete Körperteil als verbundener Körper referenziert wird. Die Kollisionskomponente soll das Körperteil in vereinfachter Form und Größe darstellen, um die Berechnungen zu optimieren. Bei den Festkörpern müssen das Gewicht und der Schwerpunkt festgelegt werden, um eine realistische physikalische Simulation zu gewährleisten. In der Gelenkkomponente können Bewegungen durch das Festlegen von Winkellimits gesperrt oder limitiert werden. Für die Rotationsberechnung wird der Slerp-Modus verwendet, da dieser eine gleichmäßige Interpolation der Rotation ermöglicht. In der Körperteilkomponente können Parameter wie Stärke und maximale Rotationsgeschwindigkeit angepasst werden, wobei die Standardwerte aus der Walker Demo in den meisten Fällen ausreichend sind. Bei Bedarf kann auch das Feld "Trigger Touching Ground" aktiviert werden, um ein Event auszulösen, sobald das Körperteil den Boden berührt.

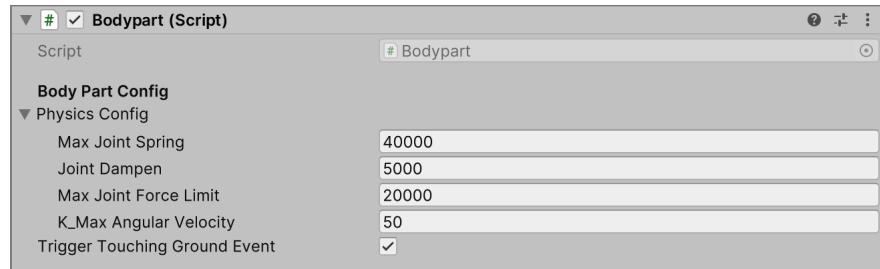


Abbildung 4.11: Körperteilkomponente

Ist der Körper fertig konfiguriert wird zuletzt das Walker Agent Skript und der Decision Requester hinzugefügt.

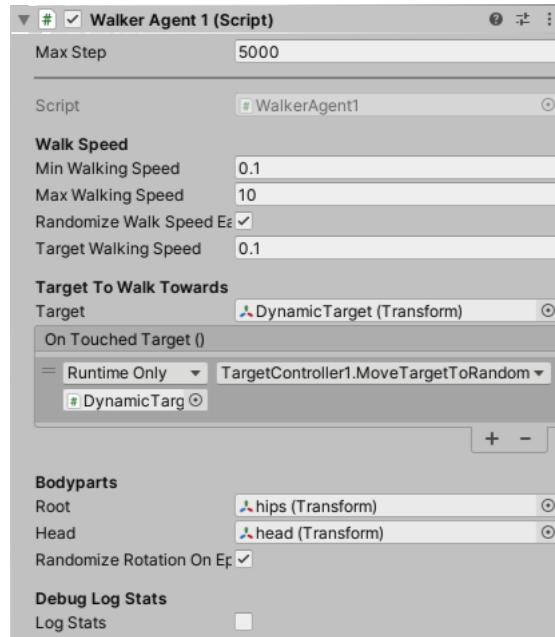


Abbildung 4.12: Walker Agentkomponente

Die Gewichte der Körperteile wurden von der Walker Demo übernommen. Gleichermaßen wurden die Winkellimits für die Gelenke übernommen. Die zusätzlichen Körperteile wurden vereinfacht. Der Oberkörper besteht im Mixamo Modell aus den Schulterknochen sowie dem obersten Wirbel der Wirbelsäule. Die Wirbelsäule besteht im Mixamo Modell aus 2 Wirbeln anstatt dem einen Wirbel des Läufers aus der Demo. Zuletzt sind die Füße noch in Fuß und Vorderfuß aufgeteilt. Bei diesen Änderungen der Körperstruktur wurden die Gewichte und Winkellimits des vereinfachten Körpers auf die komplexeren Körperstrukturen aufgeteilt.

Körperteil	Verbundenes Körperteil	Gewicht	Winkellimits	Form
Hüfte	-	15kg	-	Kapsel
Wirbel 1	Hüfte	6kg	x(-20,20) y(-20,20) z(-15,15)	Kugel
Wirbel 2	Wirbel 1	4kg	-	Kugel
Wirbel 3	Wirbel 2	3kg	x(-20,20) y(-20,20) z(-15,15)	Kugel
Schulter LR	Wirbel 3	je 2kg	-	Kugel
Nacken	Wirbel 3	1kg	-	Kugel
Kopf	Nacken	6kg	x(-30,10) y(-20,20)	Kapsel
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	-	Quader
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 4kg	x(-20,20) y(-20,20) z(-20,20)	Quader
Vorderfuß LR	Fuß	je 1kg	-	Quader

Tabelle 4.1: Mixamo Charakter Körperteile

#### 4.3.4 Auswertung

Das Training dauert etwa doppelt so lange um ein ähnliches Resultat zu erreichen. Der Agent lernt mit dem Mixamo Modell lange in der Umgebung zu bestehen und erreicht dabei auch ein gutes Maß an Belohnung pro Schritt (siehe Graphen). In der Abbildung 4.13 wird das erlernte Gangbild gezeigt. Der Läufer lernt in diesem Fall nicht das Laufen sondern galoppiert zum Ziel.

106 Tensorboard Graphen einfügen

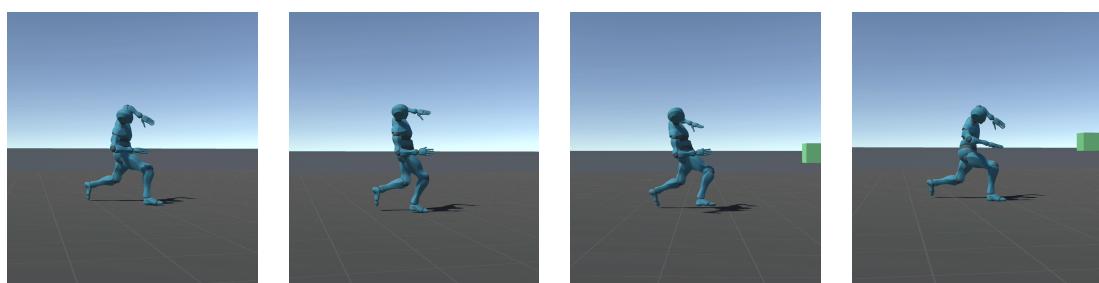


Abbildung 4.13: Mixamo Versuch 10 Gangbild

#### 4.4 Gangbild anpassungen

Das Gangbild des Läufers in der Walker Demo ist sofort als Laufen zu erkennen. Bei genauem hinschauen wird jedoch schnell klar das die Bewegung nicht **natürlich** ist. Das galoppieren des

Mixamo Charakters ist auf jedenfall nicht ausreichen um als Laufbewegung durchzugehen.

#### 4.4.1 Belohnung für Beinwechsel

Um sicher zu stellen das der Läufer während dem Training eine Laufbewegung lernt um das Ziel zu erreichen, wird im folgenden Versuch eine Bestrafung eingeführt welche den Läufer bestraft wenn ein Bein zu lange voraus geht.

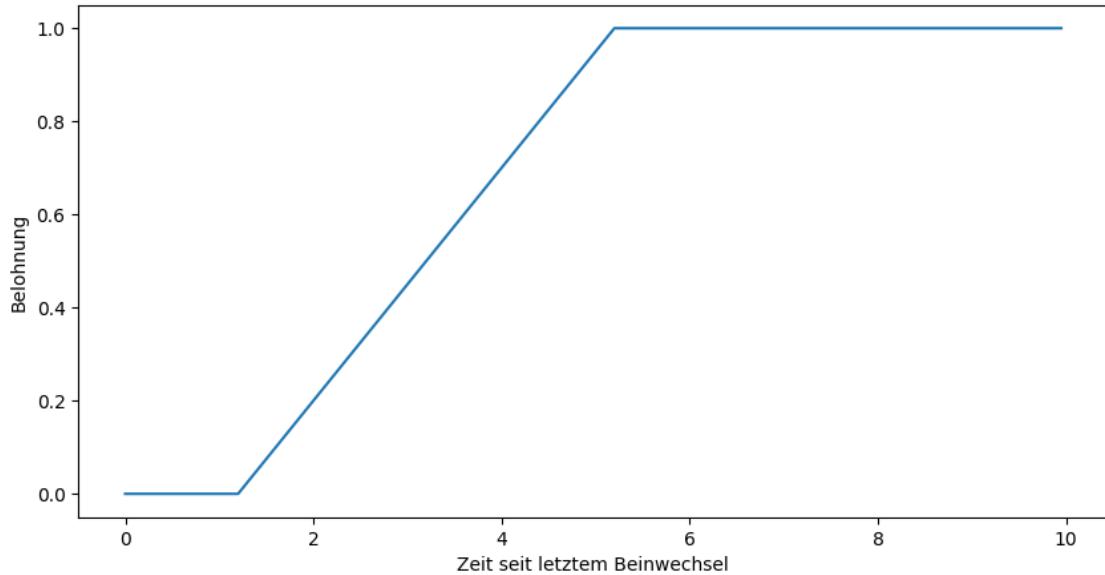


Abbildung 4.14: Beinwechsel Belohnung

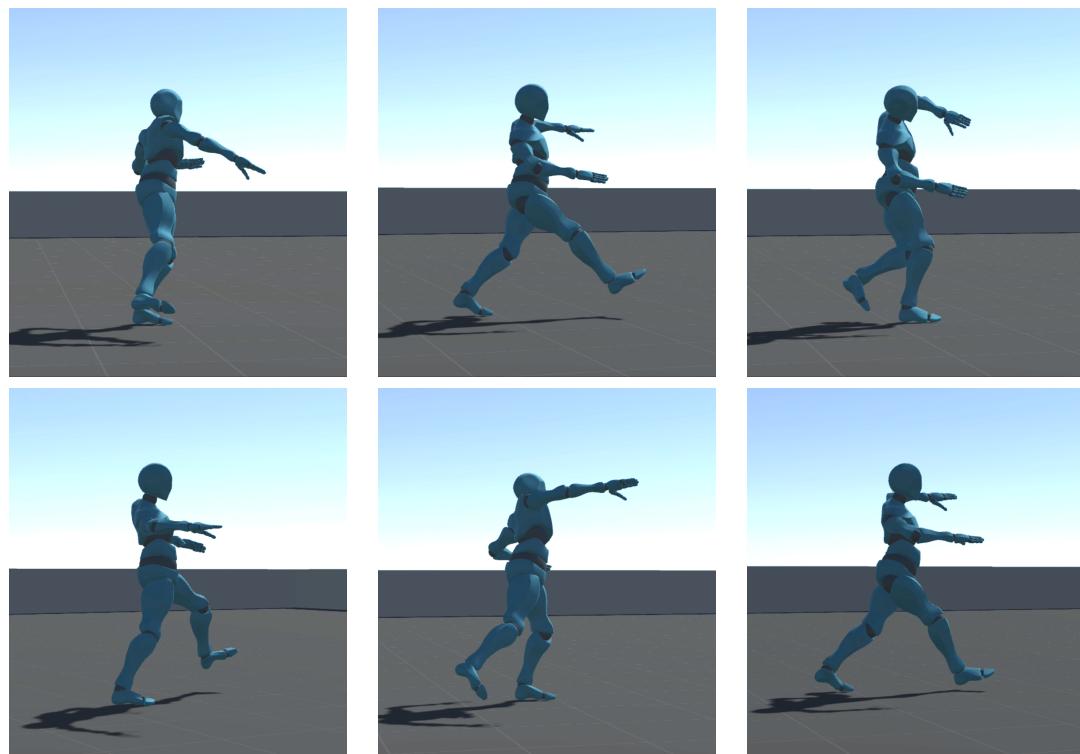


Abbildung 4.15: Mixamo Versuch 11 Gangbild

#### 4.4.2 Belohnung für Energieminimierung

Um das Gangbild weiter zu verbessern wird eine Belohnung eingeführt welche den Agenten belohnt wenn er so wenig wie möglich Kraft aufwendet um das Ziel zu erreichen. Genauer gesagt wird er dafür bestraft wenn die Gelenksteuerung einen zu hohen Energiekonsum aufweist.

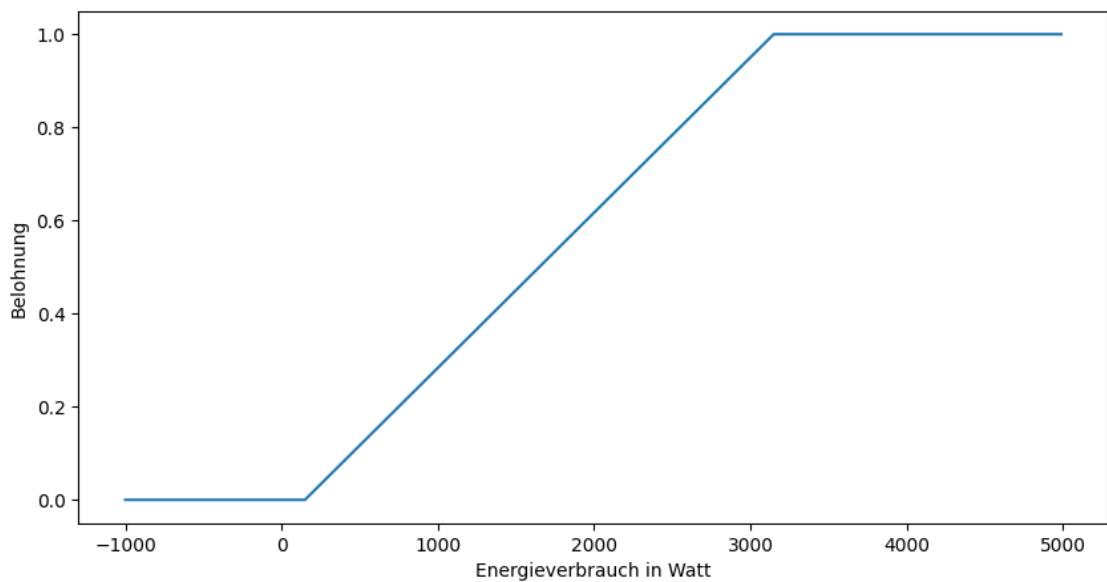


Abbildung 4.16: Energiespar Belohnung

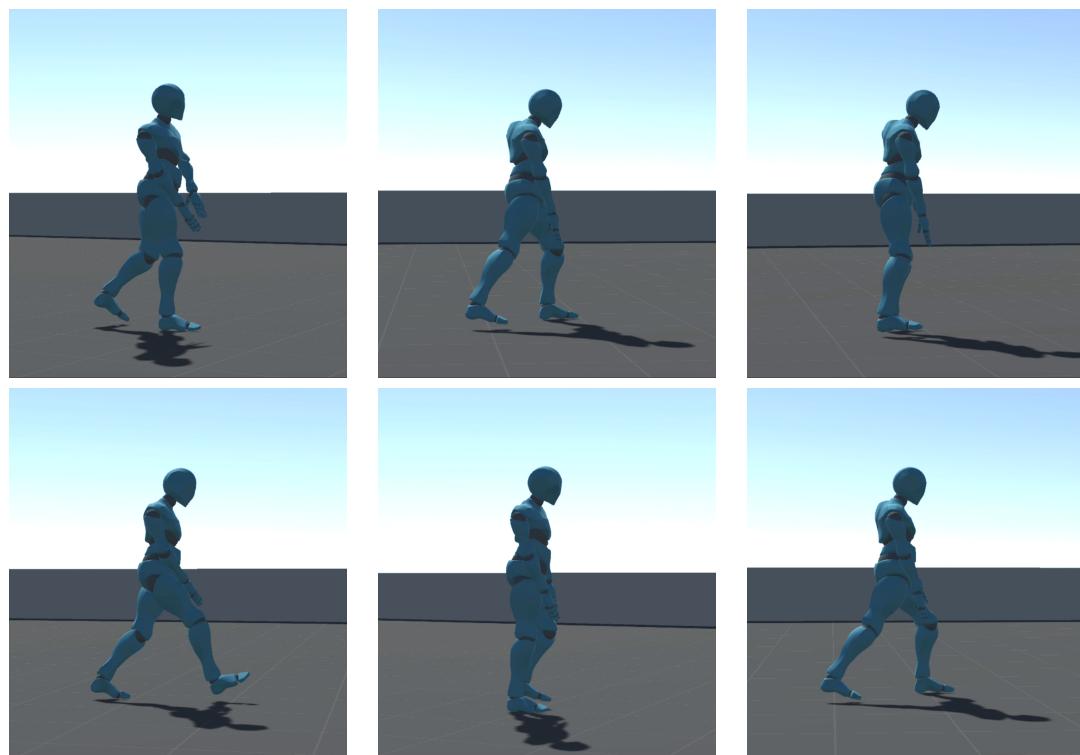


Abbildung 4.17: Mixamo Versuch 12 Gangbild

#### **4.4.3 Imitationslernen**

## 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.