



HOCHSCHULE HEILBRONN

Bachelor Thesis (SPO NUMMER)

Physik basierter Charaktercontroller mit Unity Machine Learning

Simon Grözinger*

16. August 2024

Eingereicht bei Prof. Dr. Tim Reichert
Zweitprüfer: Ulrich Straus

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

Inhaltsverzeichnis

Abkürzungsverzeichnis	III
Abbildungsverzeichnis	IV
Tabellenverzeichnis	VI
Listings	VII
1 Einleitung	1
2 Grundlagen	2
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
3 Analyse Walker Demo	12
3.1 Lernumgebung	12
3.2 Training	15
3.3 Auswertung	17
4 Umsetzung Charaktercontroller	20
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 Laufrichtungen kombinieren	29
4.3 Unterschiedliche Charakter	36
4.3.1 Anforderungen	36
4.3.2 Anpassungen	37
4.3.3 Einrichtung	39
4.3.4 Auswertung	41
4.4 Gangbild Anpassungen	43
4.4.1 Belohnung für Beinwechsel	43
4.4.2 Belohnung für Energieminimierung	44
4.4.3 Belohnung Armpendel	46
4.4.4 Imitationslernen	46
5 Fazit	47
Literaturverzeichnis	48

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	Training Stehen mit neuer Belohnungsfunktion	24
4.3	Vergleich von Lauftraining mit Demo Belohnungsfunktion gegen Neue Belohnungsfunktion	24
4.4	Vergleich der Demo Belohnungsfunktion unter verschiedenen Zielgeschwindigkeiten	25
4.5	Vergleich Demo gegen Belohnungsfunktion mit 0.1 Limit	26
4.6	Vergleich Training Stehen mit Neuer Belohnungsfunktion (orange) und Angepasster Demo Belohnungsfunktion (blau)	26
4.7	Unterschiedliche Blickrichtungen Training Graphen (grün = vorwärts, orange = rückwärts, rosa = rechts, blau = links)	28
4.8	Training Blickrichtungsziel mit Lehrplan	31
4.9	Training Blickrichtungsziel (blau = Winkelabweichung +90 Grad, grün = Winkelabweichung +180 Grad)	32
4.10	Blickwinkel Änderung durch Zielannäherung	33
4.11	Training Blickrichtungsziel mit Wechsel bei durchschnittlicher Blickbelohnung von 0.7	34
4.12	Spherecast in Blickrichtung	35
4.13	Training Blickrichtungsziel mit Wechsel bei Blickkontakt von 2 bzw. 3 Sekunden (orange = 3 sek, grün = 2 sek)	36
4.14	Mixamo Charakter Y Bot	39
4.15	Körperteilkomponente	40
4.16	Walker Agentkomponente	40

4.17 Training Mixamo Charakter	42
4.18 Mixamo Versuch 10 Gangbild	43
4.19 Beinwechsel Belohnung	44
4.20 Mixamo Versuch 11 Gangbild	44
4.21 Energiespar Belohnung	45
4.22 Mixamo Versuch 12 Gangbild	46

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	27
4.4	Laufrichtung Modell wechseln	29
4.5	Lehrplan für das Blickziel	30
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.[\[3\]](#)

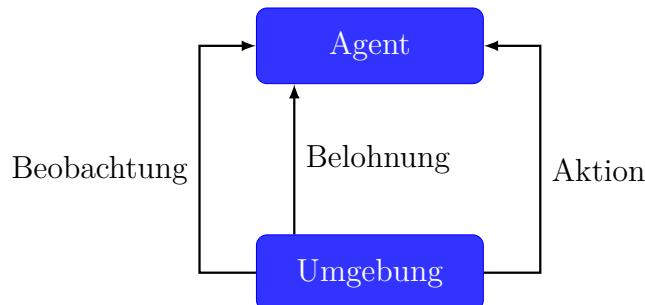


Abbildung 2.1: Verstärkendes Lernen Ablauf

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-Netzwerkommunikation.[\[1\]](#)

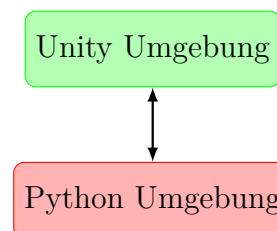


Abbildung 2.2: Unity ML-Agents Aufbau

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.

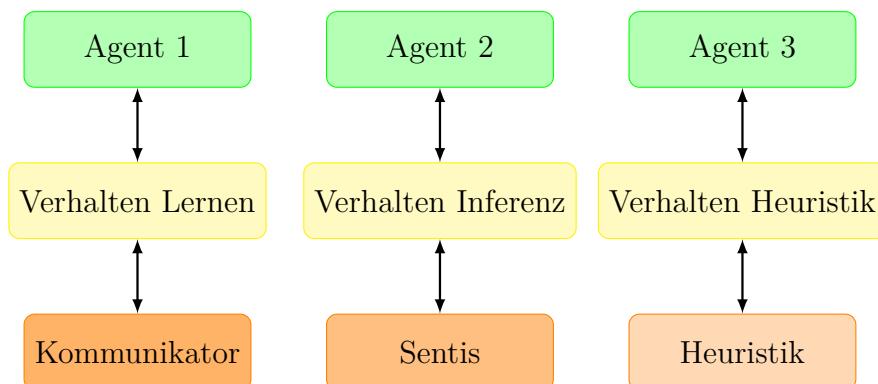


Abbildung 2.3: Unity ML-Agents Aufbau Unity Umgebung

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.

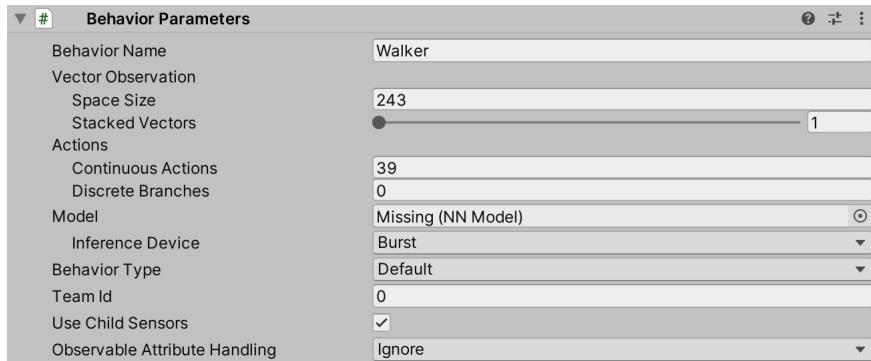


Abbildung 2.4: Unity ML-Agents Verhalten Parameter Komponente

- 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.

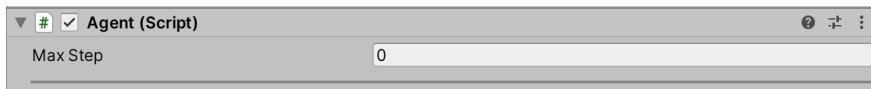


Abbildung 2.5: Unity ML-Agents Agenten Komponente

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.

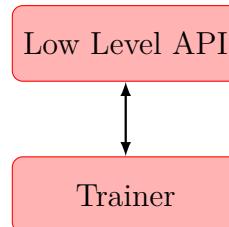


Abbildung 2.7: Unity ML-Agents Aufbau Python Umgebung

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.

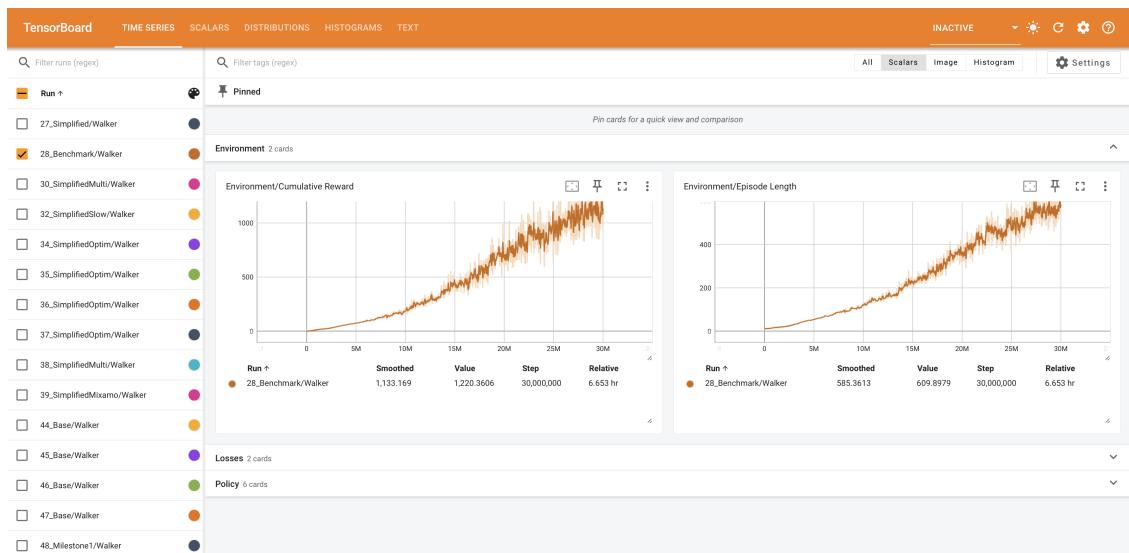


Abbildung 2.8: Tensorboard Ansicht

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.

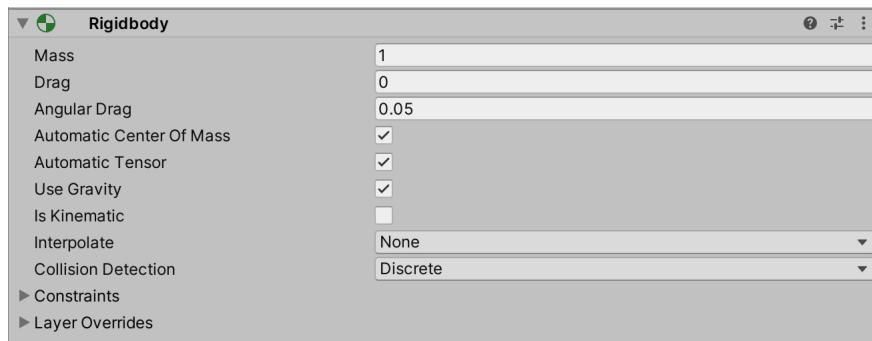


Abbildung 2.9: Unity ML-Agents Physik Festkörper

- 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.

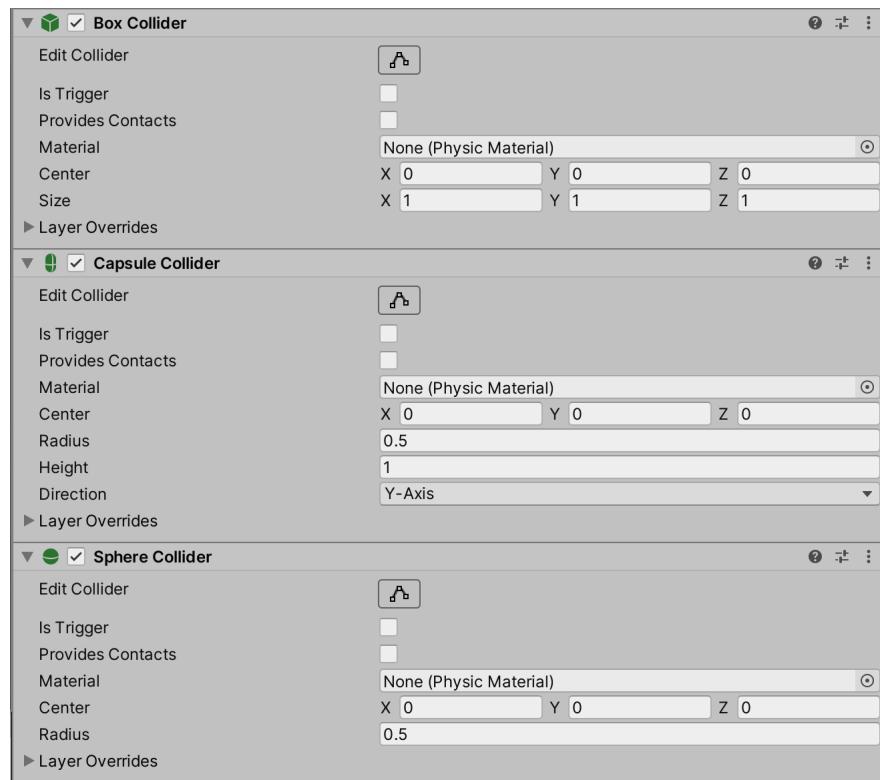


Abbildung 2.10: Unity ML-Agents Physik Kollisionskomponenten

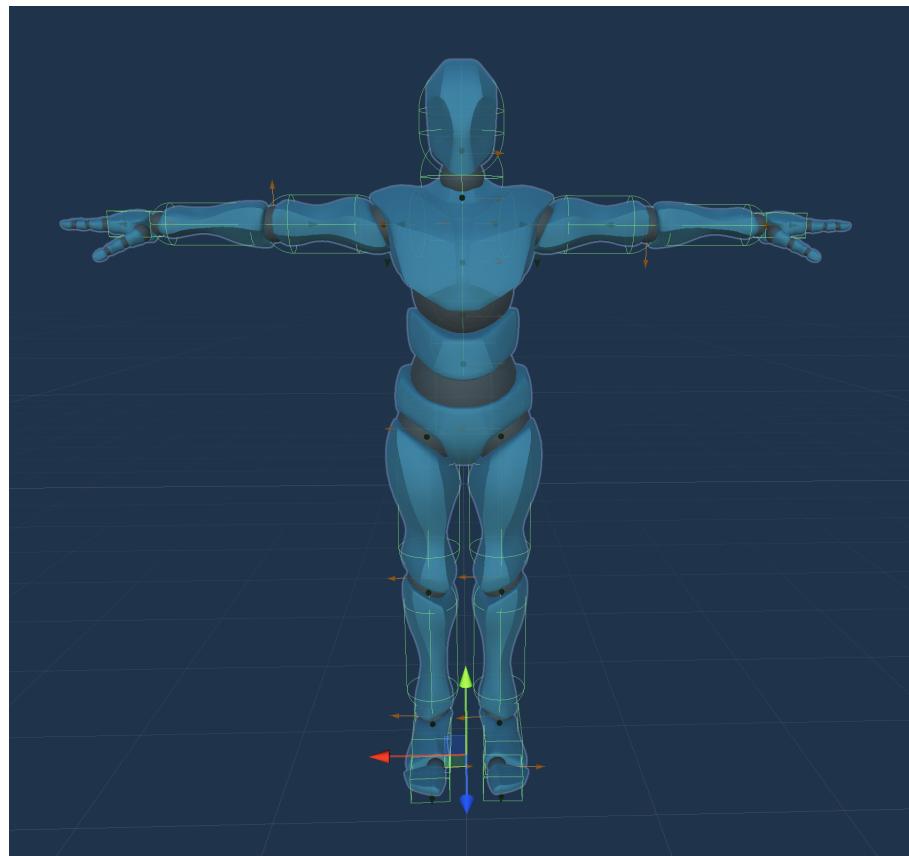


Abbildung 2.11: Unity ML-Agents Physik Charakter vereinfacht mit Kollisionskomponenten

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.

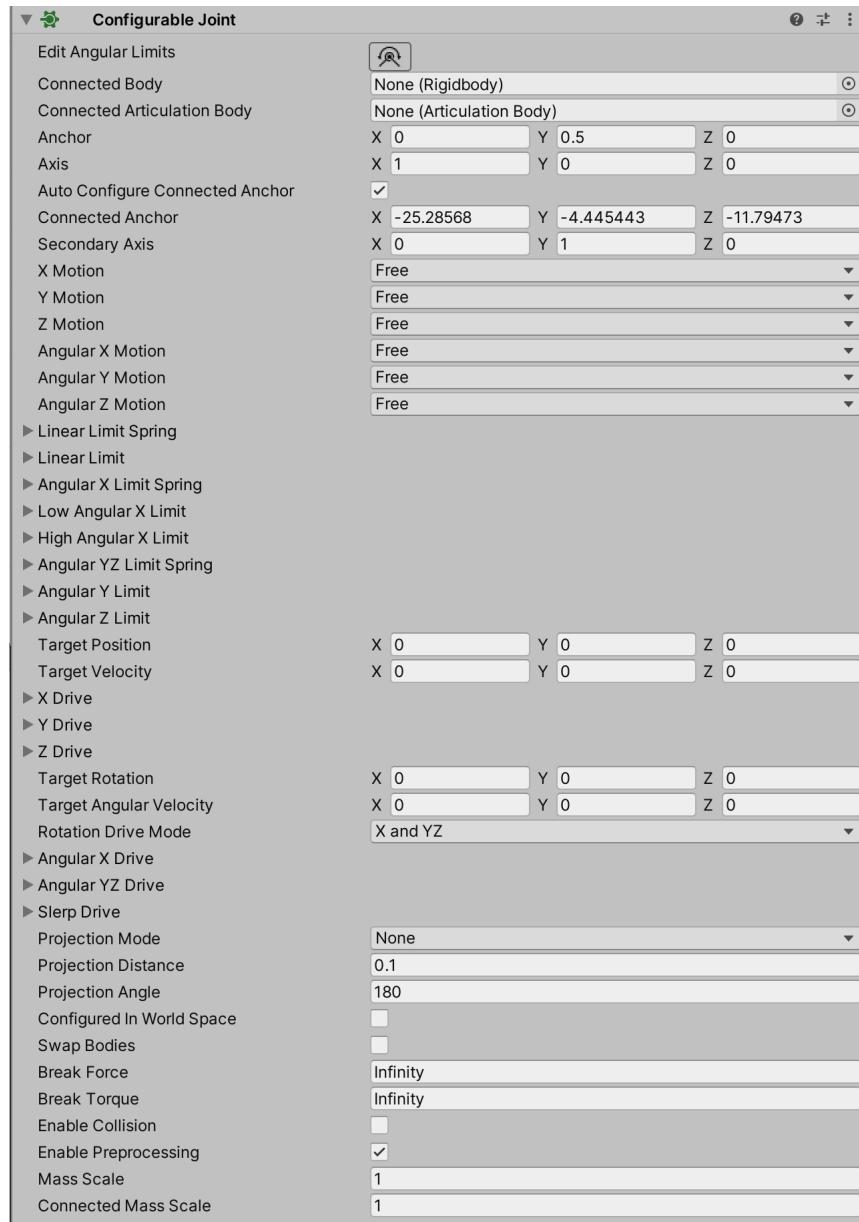


Abbildung 2.12: Unity ML-Agents Physik Gelenk

- **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.

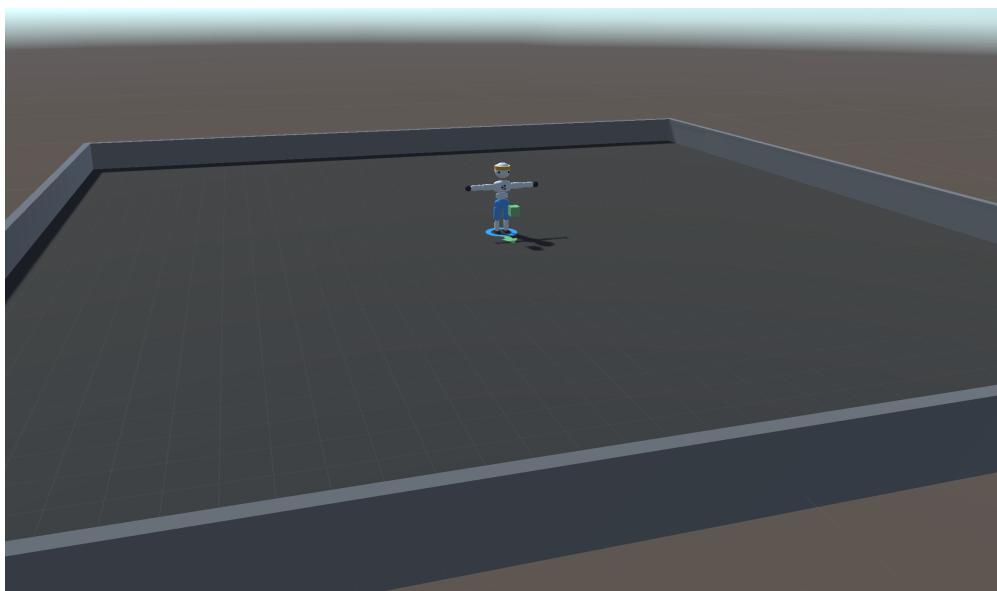


Abbildung 3.1: Walker-Demo Umgebung

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.

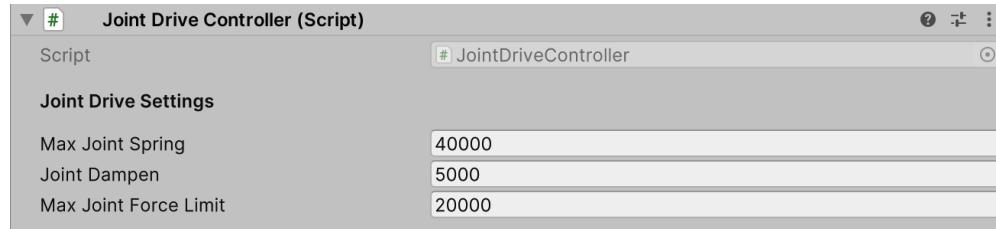


Abbildung 3.3: Gelenk Motor Steuerung

- **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).

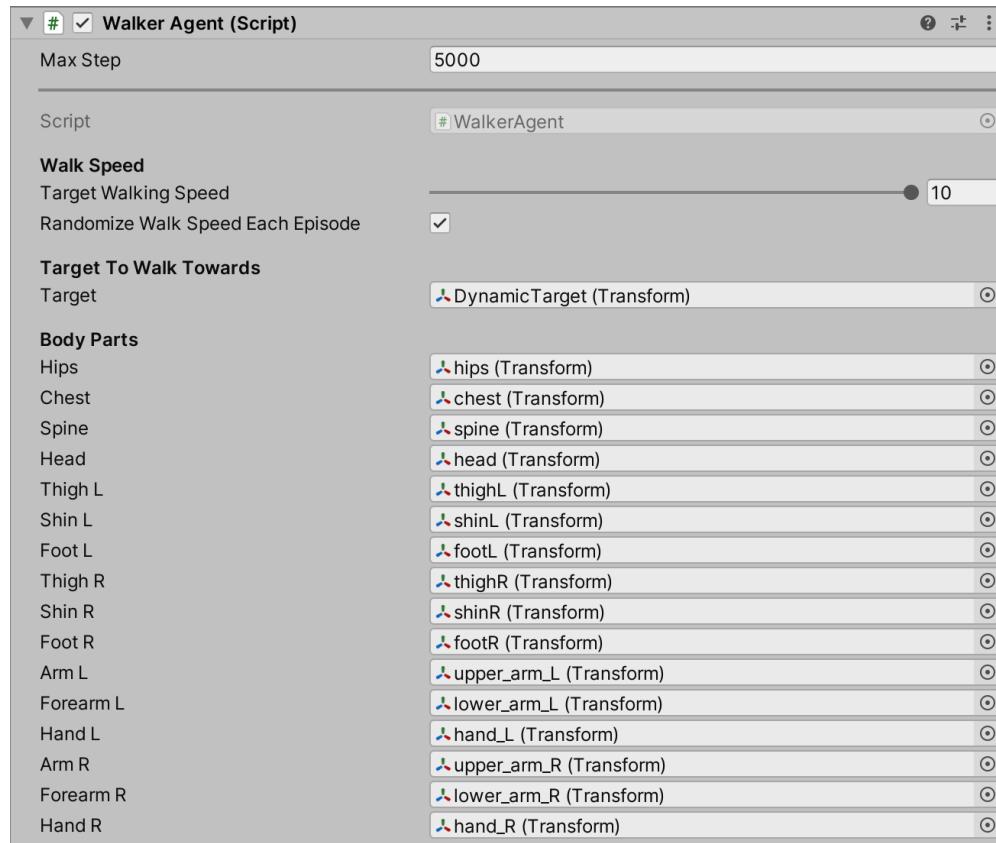


Abbildung 3.4: Agent Konfiguration

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.

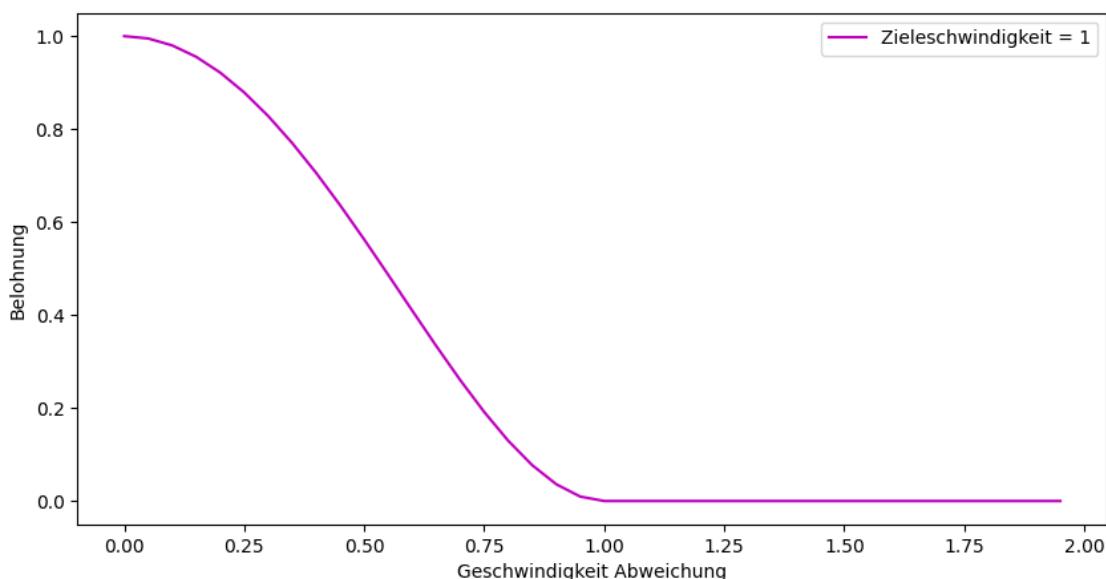


Abbildung 3.5: Walker Demo Match Velocity Belohnungsfunktion

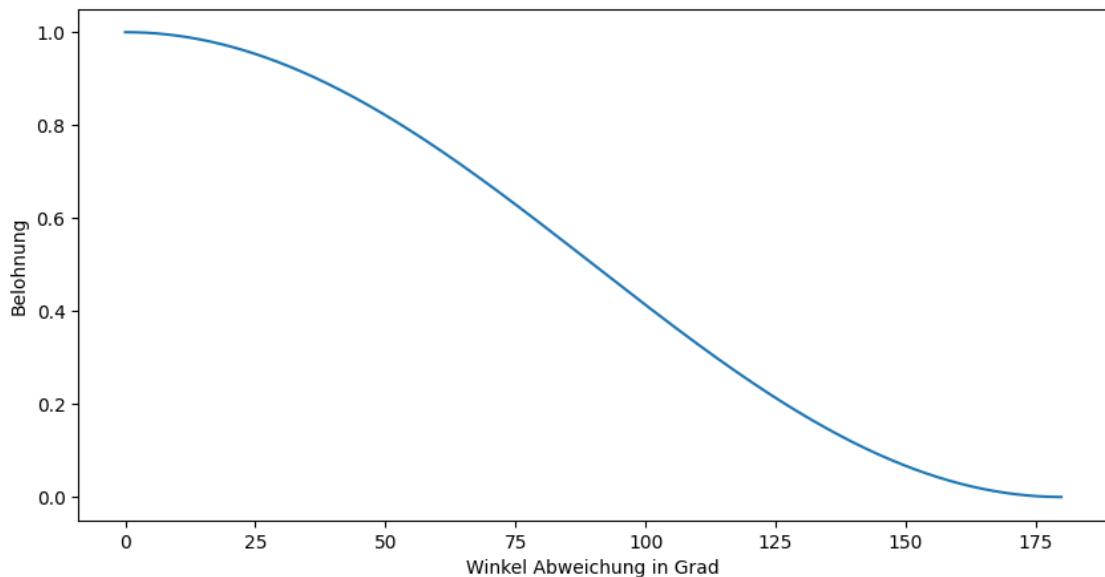


Abbildung 3.6: Walker Demo Look At Target Belohnungsfunktion

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 lernt 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%.

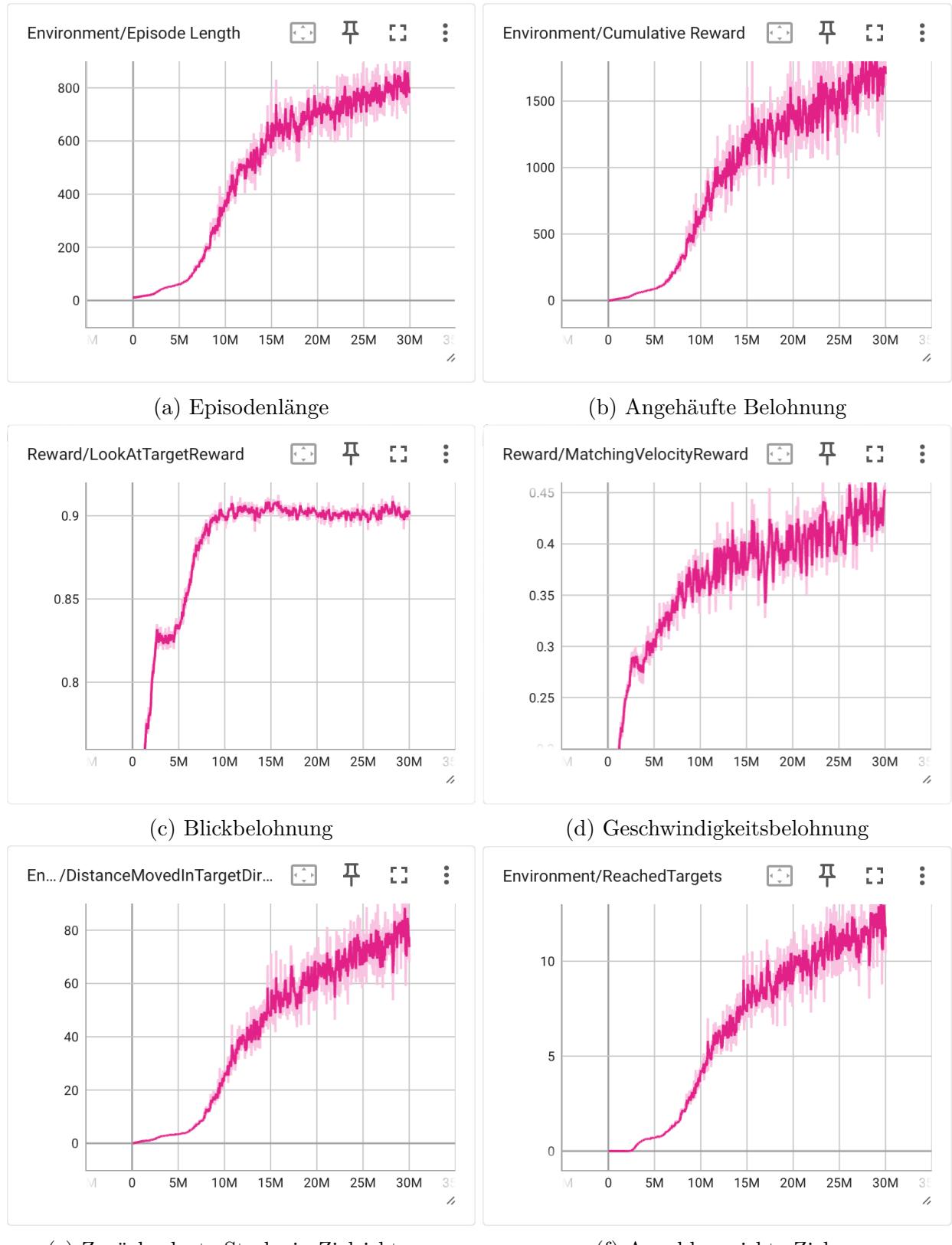


Abbildung 3.7: Walker Demo Training Graphen

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. Der Läufer wechselt periodisch das Standbein, setzt den einen Fuß vor den anderen und drückt sich über das Standbein voran. Durch die vereinfachte Darstellung der Füße kann der Läufer jedoch nicht richtig abrollen. Die Arme hält der Läufer nahezu horizontal und schwingt diese sehr stark um das Gleichgewicht halten zu können. Das Gangbild ist daher menschenähnlich, aber nicht detailliert genug um in realistischeren Anwendungsbereichen als natürliche Gehbewegung dargestellt zu werden.

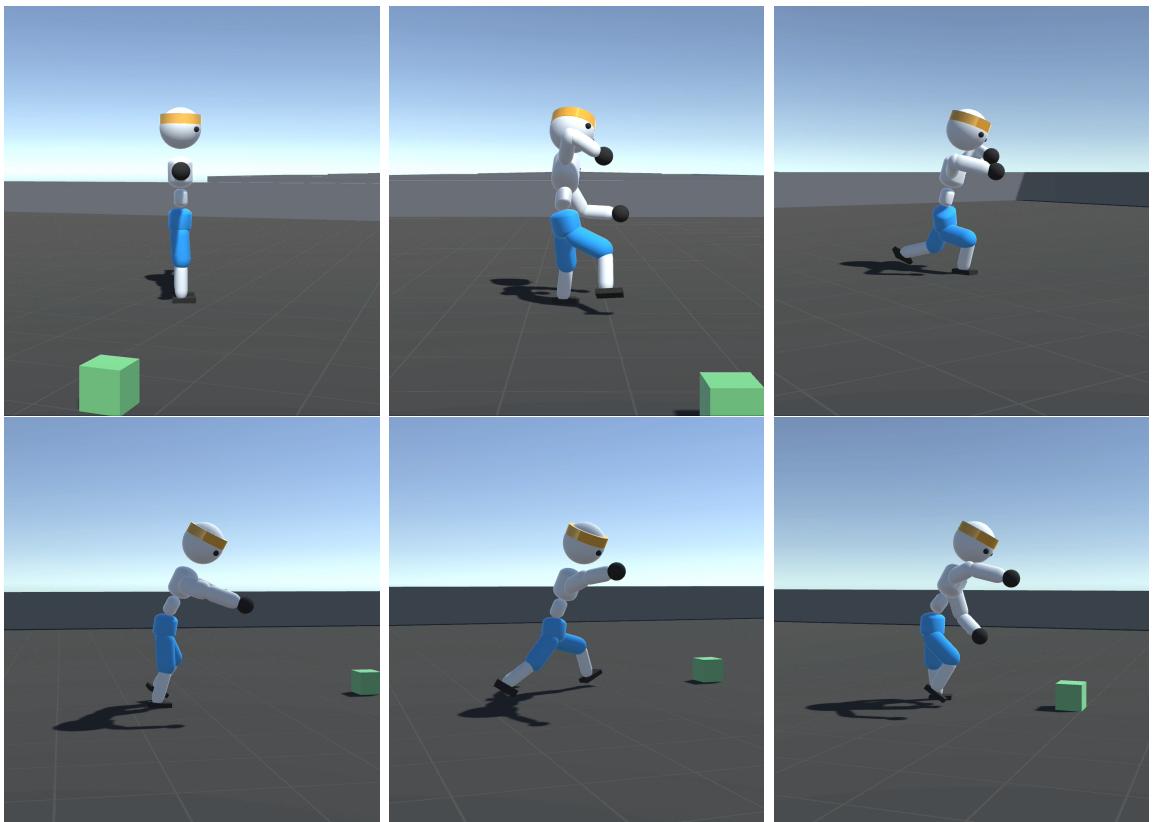


Abbildung 3.8: Walker Demo Analyse Gangbild

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 Bildschirmmittelpunkt 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 Bewegungsabläufe. Das trainierte Modell der Walker-Demo beherrscht derzeit nur die Fortbewegung in Blickrichtung und ist nicht in der Lage, auf der Stelle stehen zu bleiben. Dies führt dazu, dass der Läufer stürzt, sobald der Nutzer keinen Tastaturinput mehr gibt. Diese Einschränkungen lassen sich auf die Art und Weise zurückführen, wie das Modell trainiert wurde, da es ausschließlich auf die Vorwärtsbewegung optimiert wird. Um diese Probleme zu beheben, werden Anpassungen am Trainingsablauf vorgenommen. Insbesondere wird untersucht, wie das Modell so erweitert werden kann, dass es unabhängig von der Bewegungsrichtung eine stabile Blickrichtung beibehält und die Fähigkeit erlangt, auf der Stelle zu stehen.

4.2.2 Separate Bewegungsabläufe

Das erste Ziel ist es, dem Läufer das Stehenbleiben beizubringen. Hierfür soll der Läufer sich möglichst wenig von der aktuellen Position wegbewegen, wobei die Hauptaufgabe darin besteht, das Gleichgewicht zu halten.

Um das Stehenbleiben zu erreichen, wird die Zielgeschwindigkeit auf 0 gesetzt, während das Ziel an der Startposition bleibt. Eine Herausforderung hierbei ist die ursprüngliche Demo-Belohnungsfunktion, die durch die Zielgeschwindigkeit dividiert. Da dies bei einer Zielgeschwindigkeit von 0 zu mathematischen Fehlern führt, war eine Anpassung der Belohnungsfunktion notwendig. Statt der ursprünglichen Funktion wurde eine alternative Belohnungsfunktion implementiert, um das Problem zu beheben. Es wird erwartet, dass die Neue Belohnungsfunktion die Lernfähigkeit des Modells verbessert, auf der Stelle zu stehen, ohne zu fallen. Es muss jedoch auch untersucht werden wie die Neue Belohnungsfunktion die Lernfähigkeit des ursprünglichen Verhaltens beeinflusst.

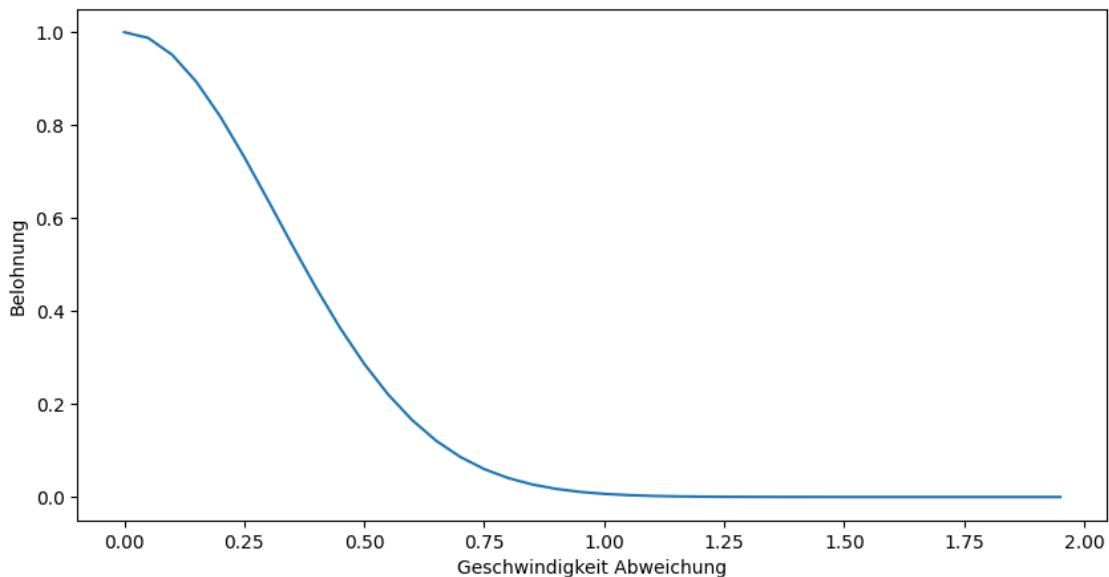


Abbildung 4.1: Neue Sigmoid Geschwindigkeit Belohnungsfunktion

Als Neue Belohnungsfunktion wird eine Sigmoid-Funktion genutzt, die eine glatte Abstufung der Belohnungen ermöglicht. Die Belohnung erreicht den Wert 1, wenn die aktuelle Geschwindigkeit perfekt mit der Zielgeschwindigkeit übereinstimmt. Mit zunehmender Abweichung von der Zielgeschwindigkeit sinkt die Belohnung stetig, und sobald die Abweichung den Wert 1 überschreitet, fällt die Belohnung auf 0. Die Belohnungsfunktion und ihre Parameter wurden so gewählt, dass die Neue Belohnungsfunktion der Demo Belohnungsfunktion ähnelt. Die Neue Belohnungsfunktion abgebildet in Abbildung 4.1 wird daher ab hier Neue Belohnungsfunktion genannt.

Der Walker konnte mit der Neuen Belohnungsfunktion erfolgreich lernen, auf der Stelle zu stehen. Dies wird in den Abbildungen 4.2a und 4.2b verdeutlicht. Die Abbildung 4.2b zeigt, wie die zurückgelegte Distanz um 0 herum schwankt, was darauf hinweist, dass der Läufer in der Lage ist, seine Position zu halten. Gleichzeitig erreicht die Episodenlänge in Abbildung 4.2a die maximale Länge von 1000, was bedeutet, dass der Läufer über die gesamte Episode hinweg stabil blieb.

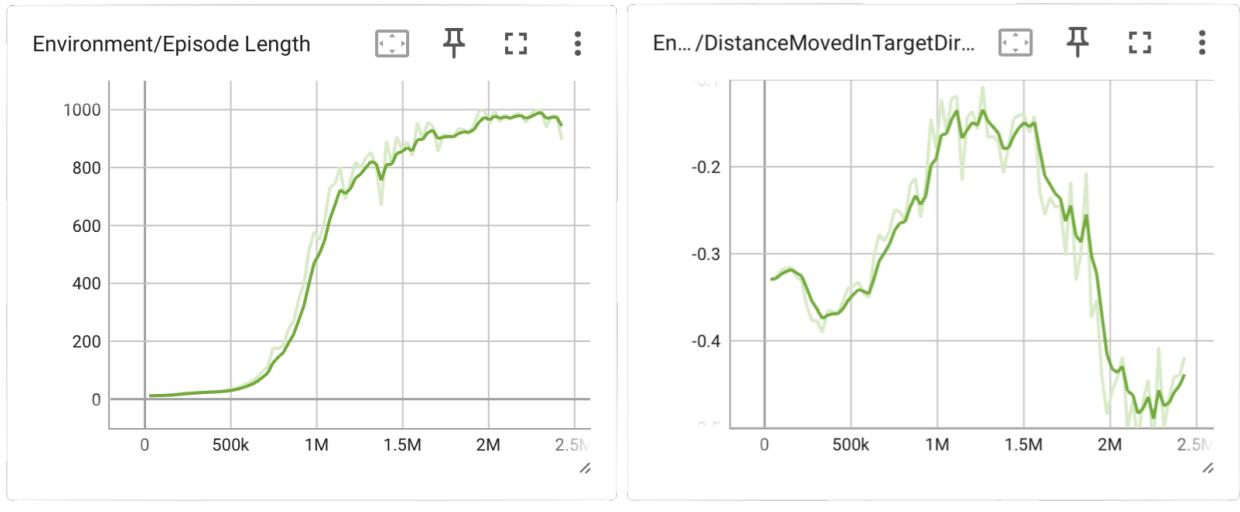


Abbildung 4.2: Training Stehen mit neuer Belohnungsfunktion

Mit zufälliger Zielgeschwindigkeit zu einem Ziel zu laufen, wie es im ursprünglichen Verhalten vorgesehen war, konnte jedoch mit der Neuen Belohnungsfunktion nicht zufriedenstellend erlernt werden. In Abbildung 4.3 ist die Leistung der Neuen Belohnungsfunktion durch die orangene Linie und die Leistung der Demo-Belohnungsfunktion durch die rosa Linie dargestellt. Die Abbildung 4.4 zeigt, dass die ursprüngliche Belohnungsfunktion die Fehlertoleranz in Abhängigkeit von der Zielgeschwindigkeit dynamisch anpasst, was dem Modell ermöglicht, besser zwischen unterschiedlichen Geschwindigkeiten zu generalisieren. Die Ergebnisse zeigen, dass während die Neue Belohnungsfunktion gut für das Erlernen der Stabilisierung des Läufers auf der Stelle ist, die ursprüngliche Belohnungsfunktion sich besser für das Erlernen einer Gangbewegung in Zielrichtung eignet.

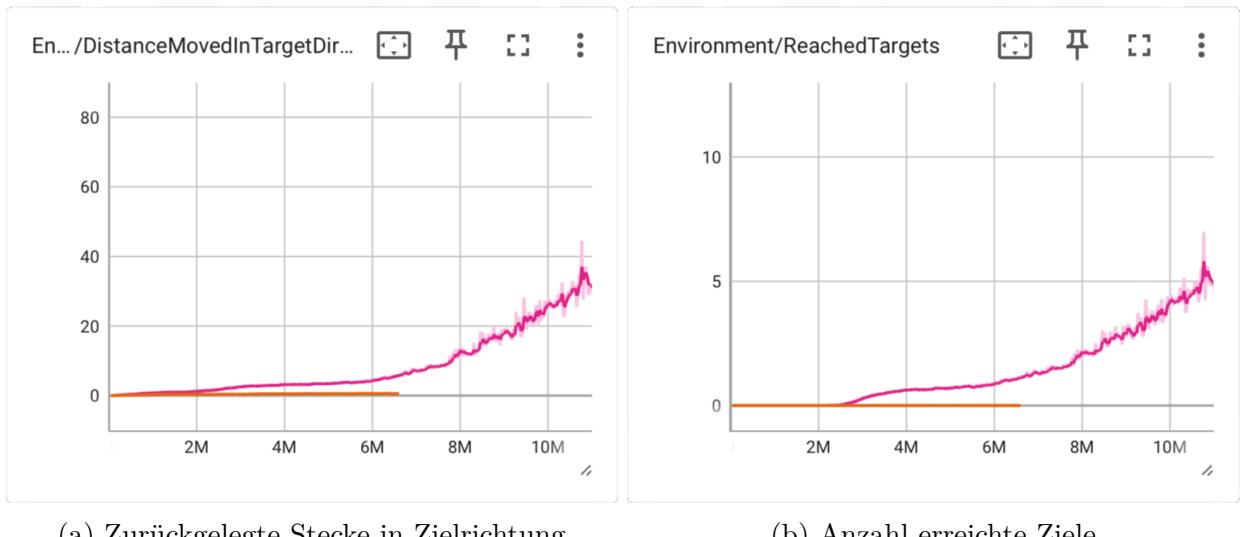


Abbildung 4.3: Vergleich von Lauftraining mit Demo Belohnungsfunktion gegen Neue Belohnungsfunktion

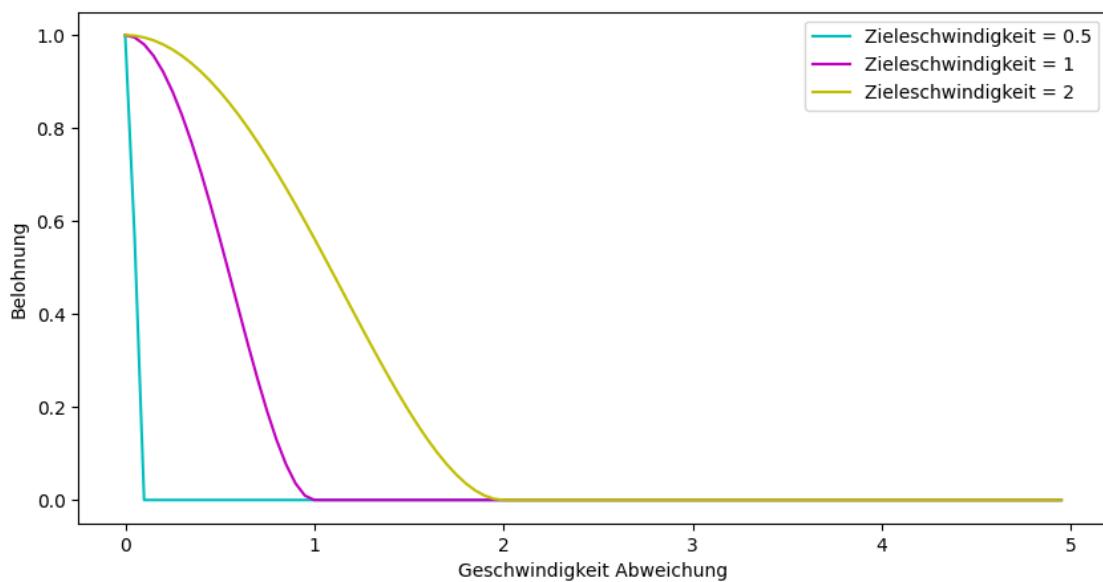


Abbildung 4.4: Vergleich der Demo Belohnungsfunktion unter verschiedenen Zielgeschwindigkeiten

Mit dieser Erkenntnis 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. Diese Anpassung stellt sicher, dass im Bereich der normalen Fortbewegung keine unerwünschten Veränderungen auftreten. Das Problem mit der ursprünglichen Demo-Belohnungsfunktion bestand darin, dass bei Annäherung an eine Zielgeschwindigkeit von 0 das Spektrum an akzeptablen Geschwindigkeiten, bevor die Belohnung auf 0 sinkt, extrem eng wurde (siehe Abbildung 4.5). Dies machte es nahezu unmöglich, sinnvolle Lernfortschritte in diesem Bereich zu erzielen. Durch die Einführung eines Limits von 0,1 wird der Bereich der Geschwindigkeitsabweichungen, für die die Belohnungsfunktion einen Wert größer 0 annimmt, ausreichend vergrößert. Dies ermöglicht dem Läufer, durch Ausprobieren Belohnungen über 0 zu erreichen, was wiederum eine Richtung für die Optimierung des Verhaltens bietet. Somit kann der Läufer die Belohnung optimieren.

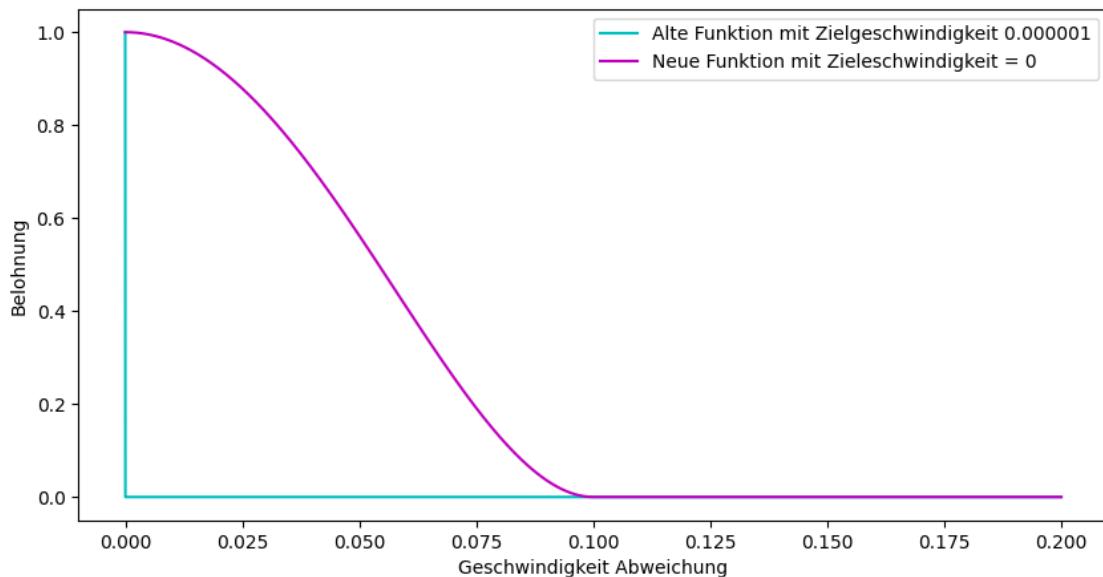


Abbildung 4.5: Vergleich Demo gegen Belohnungsfunktion mit 0.1 Limit

Mit der angepassten Demo-Belohnungsfunktion konnte nach etwas mehr Trainingsschritten ebenfalls die maximale Episodenlänge von 1000 erreicht werden (siehe Abbildung 4.6a). Dies zeigt, dass der Läufer in der Lage war, über eine längere Trainingsdauer hinweg die erforderliche Stabilität zu erlernen. Die in Abbildung 4.6b dargestellte zurückgelegte Distanz nähert sich gegen Ende fast 0, was darauf hindeutet, dass der Läufer seine Position sehr gut halten kann. Dieses Ergebnis zeigt, dass auch mit der angepassten Demo-Belohnungsfunktion ein vergleichbares Niveau an Stabilität erreicht werden kann, ohne den Lernvorgang für das ursprüngliche Verhalten zu beeinflussen.

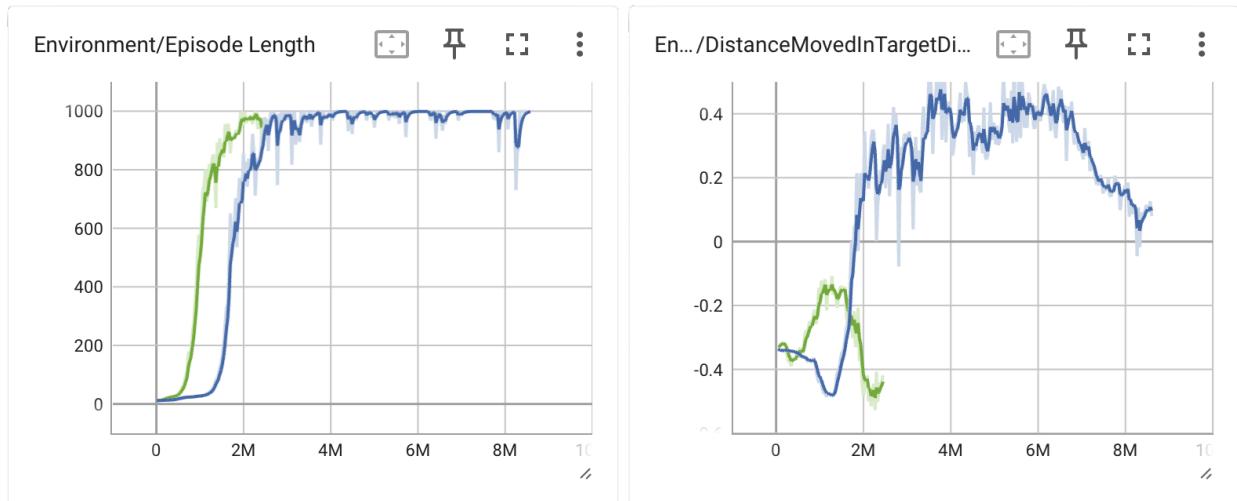


Abbildung 4.6: Vergleich Training Stehen mit Neuer Belohnungsfunktion (orange) und An gepasster Demo Belohnungsfunktion (blau)

Nach dem erfolgreichen Erlernen des Stehens auf einer festen Position sind die nächsten Bewegungsziele die Fortbewegung zum Ziel mit unterschiedlichen Blickrichtungen. Die Extremfälle umfassen dabei das Rückwärts- und Seitwärtslaufen. In diesem Abschnitt wird

untersucht, ob der Läufer in der Lage ist, Bewegungen in verschiedene Richtungen relativ zu seiner Blickrichtung zu erlernen. Um dies zu ermöglichen, wurde die Belohnungsfunktion angepasst, sodass sie die Blickrichtung relativ zur Zielrichtung berücksichtigt. Bei der Vorrwärtsbewegung entspricht die Blickrichtung der Zielrichtung. Bei der Seitwärtsbewegung steht die Blickrichtung im rechten Winkel zur Zielrichtung, und bei der Rückwärtsbewegung ist die Blickrichtung entgegengesetzt zur Zielrichtung. Die Implementierung zur Bestimmung der Blickrichtung ist in 4.3 dargestellt.

```

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 negativ beeinflusst. Separate Trainings für die drei anderen Laufrichtungen – seitlich nach links, seitlich nach rechts und rückwärts – waren ebenfalls erfolgreich. Abbildungen 4.7e und 4.7f zeigen die zurückgelegte Distanz und die Anzahl der erreichten Ziele in einer Trainingsepisode. Die Ergebnisse für die drei Laufrichtungen sind alle vergleichbar mit den Ergebnissen der ursprünglichen Demo. Die Tatsache, dass die Ergebnisse “vergleichbar” sind, bedeutet, dass der Läufer in der Lage war, eine ähnliche Anzahl von Zielen zu erreichen und ähnliche Distanzen zurückzulegen wie in der ursprünglichen Demo, unabhängig von der Laufrichtung. Insgesamt zeigen die Ergebnisse, dass der Läufer mit den Einschränkungen der Gelenke und der implementierten Belohnungen dazu in der Lage ist, die Bewegung zum Ziel mit allen vier Blickrichtungen zu meistern. erwähnung ausnahme bewegung links

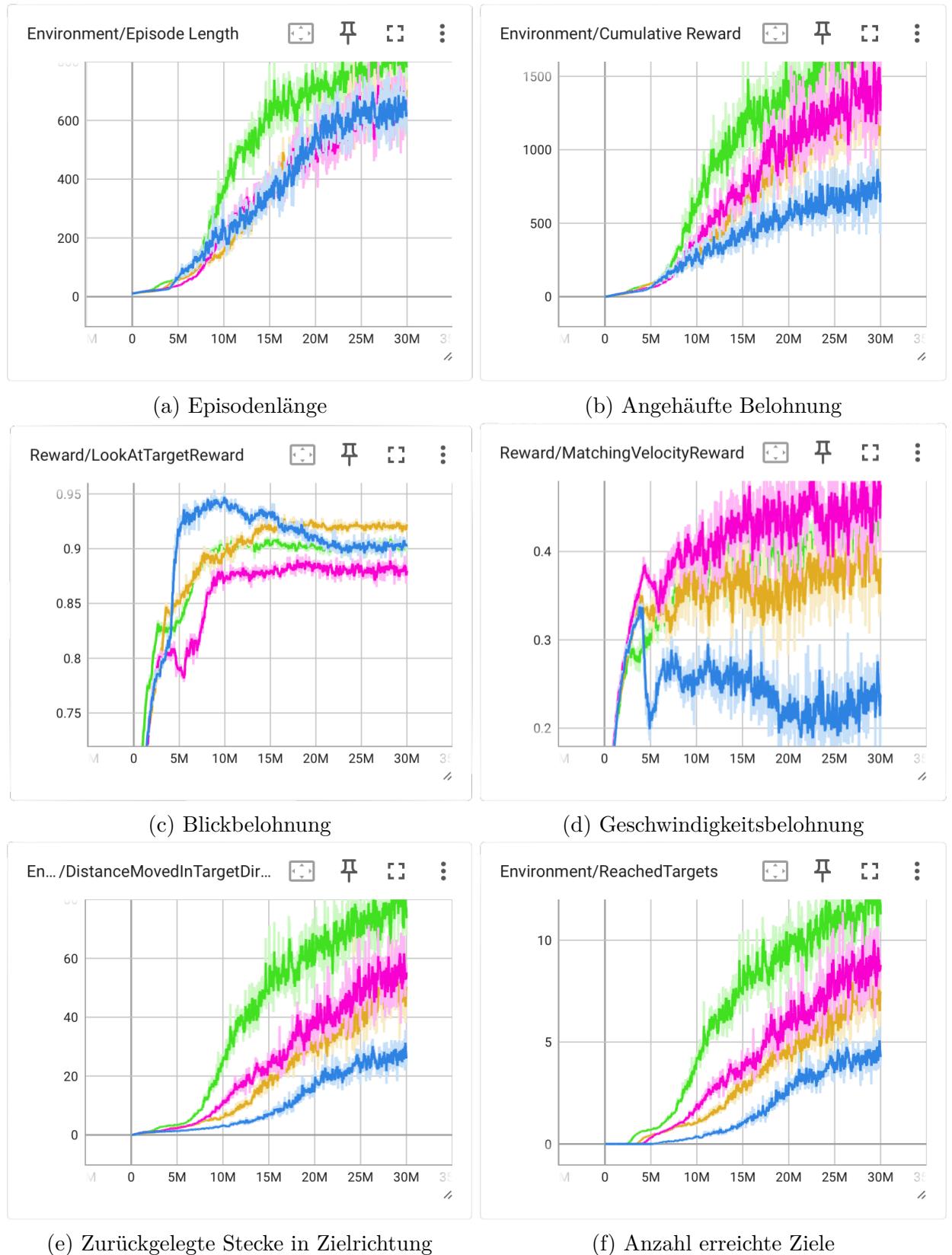


Abbildung 4.7: Unterschiedliche Blickrichtungen Training Graphen (grün = vorwärts, orange = rückwärts, rosa = rechts, blau = links)

4.2.3 Laufrichtungen kombinieren

Die Charaktersteuerung erfordert je nach Tastatureingabe eine unterschiedliche Bewegungsrichtung. Um dies zu erreichen, wird die Funktion zum Wechseln des verwendeten Modells im Unity ML-Agents Agent genutzt. Mit dieser Funktion wird in der folgenden Implementierung zwischen den separaten Bewegungsmodellen gewechselt, um alle Bewegungsrichtungen mit einer Steuerung abzudecken. Es wird erwartet, dass die Bewegung in die einzelnen Richtungen funktioniert, dass der Läufer jedoch beim Wechsel zwischen den Modellen das Gleichgewicht nicht halten kann. Das Balance Problem bei Übergängen zwischen den Modellen ist darauf zurückzuführen, dass die Bewegungsmodelle separat voneinander trainiert wurden, und der Agent somit im Training den Zustandswechsel bzw. die Zustände der anderen Bewegungsabläufe nie gelernt hat. Der Läufer kann daher durch die plötzliche Veränderung in den Bewegungsabläufen das Gleichgewicht verlieren.

```

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 jedoch ohne Ausnahme. Um dieses Problem zu lösen, ergeben sich zwei grundlegende Verfahren. Zum einen kann versucht werden, die

Übergänge zwischen den Modellen zu optimieren. Ein zweiter Ansatz ist es, die unterschiedlichen Bewegungsabläufe in einem einzigen Modell anzutrainieren. Im Folgenden wird eine Lösung basierend auf dem zweiten Ansatz entwickelt und ausgewertet.

Der erste Versuch, alle Bewegungsrichtungen in einem Modell anzulernen, wurde von den Methoden der Walker Demo inspiriert. Ähnlich wie beim Laufziel wurde ein zusätzliches Zielobjekt hinzugefügt, das zufällig platziert wurde. Um die Komplexität des Lernprozesses nicht von Anfang an zu hoch anzusetzen, wurde das Blickziel – also die Winkelabweichung zwischen Zielrichtung und Blickrichtung – schrittweise über einen Lehrplan angepasst. Zu Beginn wurde das Blickziel mit einer Winkelabweichung im Bereich von -5 bis 5 Grad platziert, um sicherzustellen, dass der Läufer zunächst nur geringe Anpassungen in der Blickrichtung vornehmen muss. Im Laufe des Trainings wurde dieser Bereich allmählich auf -90 bis 90 Grad erweitert, um die Lernanforderungen schrittweise zu steigern (siehe Codeausschnitt 4.5). Das Blickziel wurde jedes Mal neu gesetzt, sobald ein Ziel erreicht wurde.

Diese Methode wurde gewählt, um mit geringer Anfangsschwierigkeit das Erlernen des grundlegenden Verhaltens zu erleichtern. Bei zu großer Anfangsschwierigkeit kann es dazu kommen, dass der Läufer ein falsches Verhalten lernt oder gar keine zielführende Lösung findet. Ein potenzielles Risiko bei diesem Ansatz besteht jedoch darin, dass in der Startphase ein Verhalten gefestigt wird, das in späteren Etappen nicht in der Lage ist, auf größere Abweichungen des Blickwinkels zu adaptieren. Der Erfolg dieses Ansatzes wird davon abhängen, ob der Läufer in der Lage ist, das während der frühen Lernphasen erlernte Verhalten flexibel anzupassen, wenn die Anforderungen in späteren Phasen erhöht werden.

```

1  lookAngleLimit:
2      curriculum:
3          - name: min max 5 degree look target deviation
4              completion_criteria:
5                  measure: progress
6                  behavior: Walker
7                  threshold: 0.4
8                  signal_smoothing: true
9                  value: 5.0
10             - name: min max 30 degree look target deviation
11                 completion_criteria:
12                     measure: progress
13                     behavior: Walker
14                     threshold: 0.6
15                     signal_smoothing: true
16                     require_reset: true
17                     value: 30.0
18             - name: min max 60 degree look target deviation
19                 completion_criteria:
20                     measure: progress
21                     behavior: Walker
22                     threshold: 0.8
23                     signal_smoothing: true
24                     require_reset: true
25                     value: 60.0
26             - name: min max 90 degree look target deviation
27                 completion_criteria:
28                     measure: progress
29                     behavior: Walker

```

```

30     threshold: 1.0
31     signal_smoothing: true
32     require_reset: true
33     value: 90.0

```

Listing 4.5: Lehrplan für das Blickziel

Der Läufer lernt einen stabilen Gang (siehe Abbildungen 4.8a und 4.8b). Die Winkelabweichung von 5 Grad ist jedoch mit der aktuellen Belohnungsfunktion nahezu zu vernachlässigen. Bereits vor den folgenden Lehreinheiten hat sich ein Verhalten so stark gefestigt, dass die Lehreinheiten keine Wirkung mehr zeigen (siehe Abbildungen 4.8c und 4.8d). Diese Ergebnisse zeigen, dass die Auswirkung der 5 Grad Winkelabweichung zu gering war, um das Verhalten auf Abweichungen der Zielblickrichtung vorzubereiten. Ein weiteres Problem war es, dass die initiale Lernepisode zu lang war, wodurch das Verhalten bereits zu stark auf die spezifischen Gegebenheiten der ersten Lernphase angepasst war.

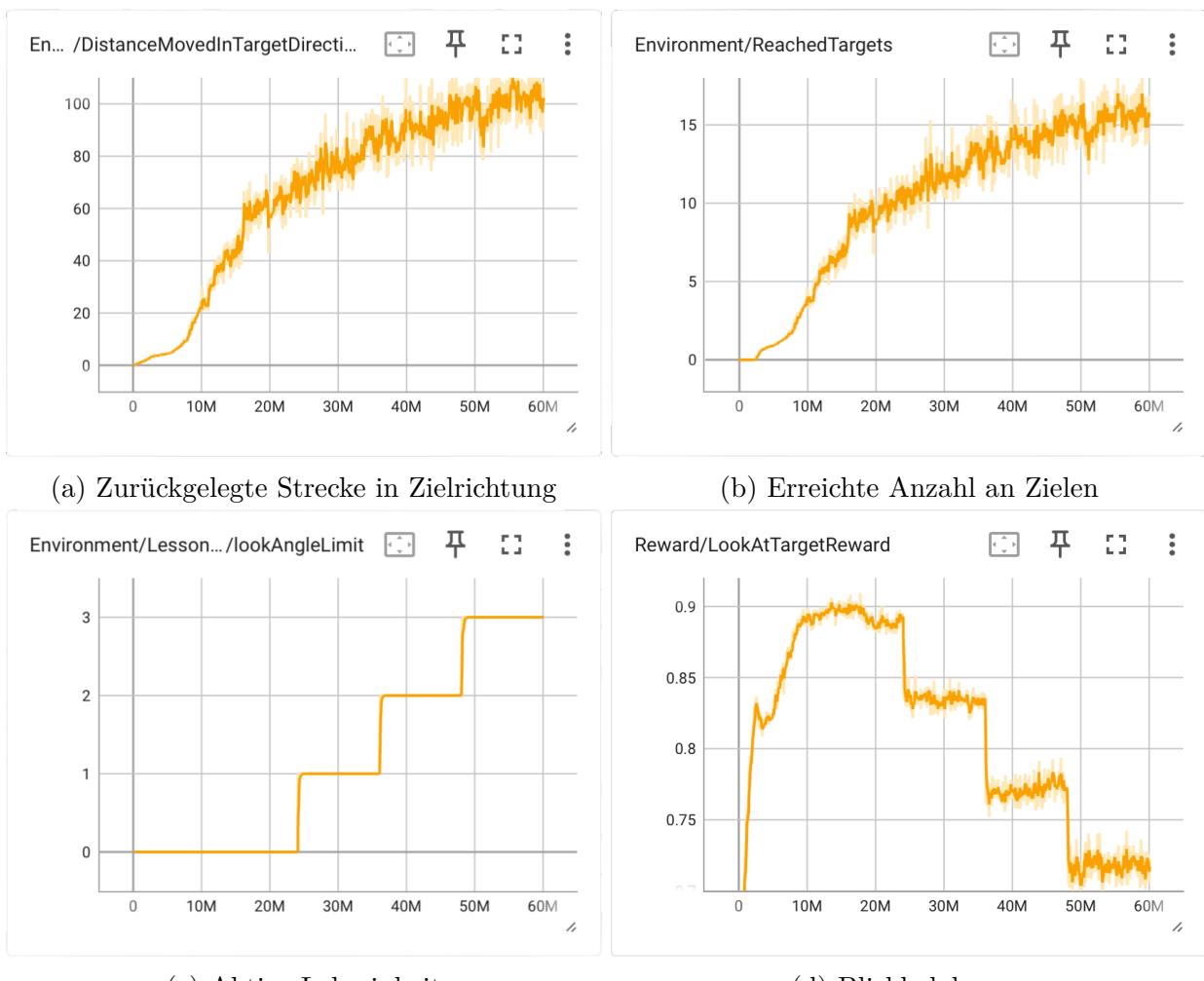


Abbildung 4.8: Training Blickrichtungsziel mit Lehrplan

Um von Beginn an ein generell gültiges Verhalten anzutrainieren und somit das Problem der Spezialisierung des Modells auf eine bestimmte Blickrichtung zu umgehen, wurden Trainings ohne Lehrplan durchgeführt. Dabei wurde ein Training mit einer Winkelabweichung von +90 Grad und eines mit +180 Grad durchgeführt. In beiden Trainings lernte

der Läufer jedoch kaum, seine Blickrichtung an die neue Zielrichtung anzupassen. Insbesondere beim Training mit Winkelabweichungen von bis zu ± 180 Grad entdeckte der Läufer eine Möglichkeit, negative Belohnungen zu umgehen, indem er seine Blickrichtung vertikal nach unten ausrichtete. Dies ist möglich, weil die Blickrichtung vertikal nach unten entlang der Y-Achse verläuft. Bei der Berechnung der Blickbelohnung wird die Y-Komponente der Blickrichtung auf 0 gesetzt, da die Zielrichtung ohne Höhenkomponente festgelegt wird, um Höhenunterschiede zwischen Hüfte und Ziel zu ignorieren. Durch dieses Schlupfloch kann der Läufer negative Belohnungen verhindern, ohne seine Gangart nennenswert anzupassen. Diese Beobachtung zeigt eine Schwäche in der aktuellen Belohnungsfunktion, gleichermaßen demonstriert es aber auch gut, wie der Agent lernt. Der Agent hat keinerlei Verständnis über die gelernten Bewegungsabläufe; er versucht lediglich, die Belohnung zu maximieren und nutzt dabei jedes verfügbare Mittel. Im Training findet der Agent so häufig Wege, die bei der Entwicklung vermeintlich sorgfältig gesetzten Zielen zu umgehen. Um dieses Problem zu adressieren, kann die Belohnungsfunktion angepasst werden, oder zusätzliche Belohnungsfunktionen hinzugefügt werden.

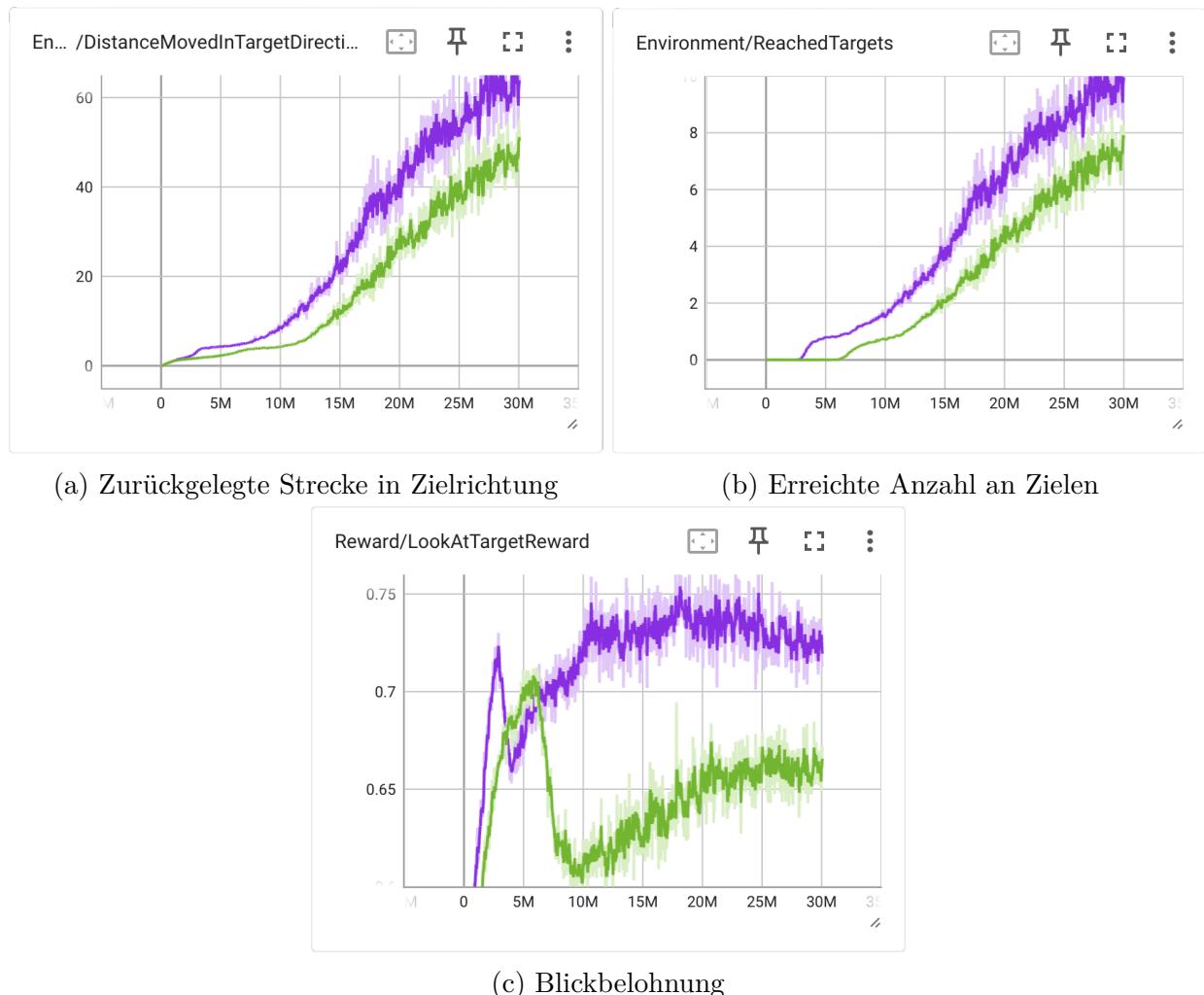


Abbildung 4.9: Training Blickrichtungsziel (blau = Winkelabweichung $+90$ Grad, grün = Winkelabweichung $+180$ Grad)

Um das Schlupfloch in der Blickbelohnung zu schließen, wurde eine neue Bestrafung eingeführt: die Kopfniegsbestrafung. Diese Bestrafung sorgt dafür, dass der Läufer für das

Neigen des Kopfes bestraft wird, wodurch er gezwungen wird, den Kopf aufrecht zu halten. Im Training mit der neuen Kopfnieigungsbestrafung lernt der Läufer jedoch langsamer und findet stattdessen einen neuen Ausweg, um die Belohnungen generell zu optimieren, ohne unterschiedliche Gangarten zu erlernen. Durch die Eigenschaft des Trainings, dass der Winkel zwischen Läufer, Ziel und Blickziel nur bei der Platzierung des Blickziels eingehalten wird, vergrößern sich die Winkel, wenn sich der Läufer dem Ziel nähert, ausnahmslos. Dies führt dazu, dass die beste Lösung für den Läufer darin besteht, sich rückwärts zu bewegen, da die Wahrscheinlichkeit, dass die Winkelabweichung kleiner ist, wesentlich höher ist. Die Abbildung 4.10 zeigt, wie sich der Blickwinkel ändert, wenn der Läufer sich dem Ziel nähert, und verdeutlicht, warum das rückwärtsgerichtete Gehen eine vorteilhafte Strategie für den Läufer darstellt.

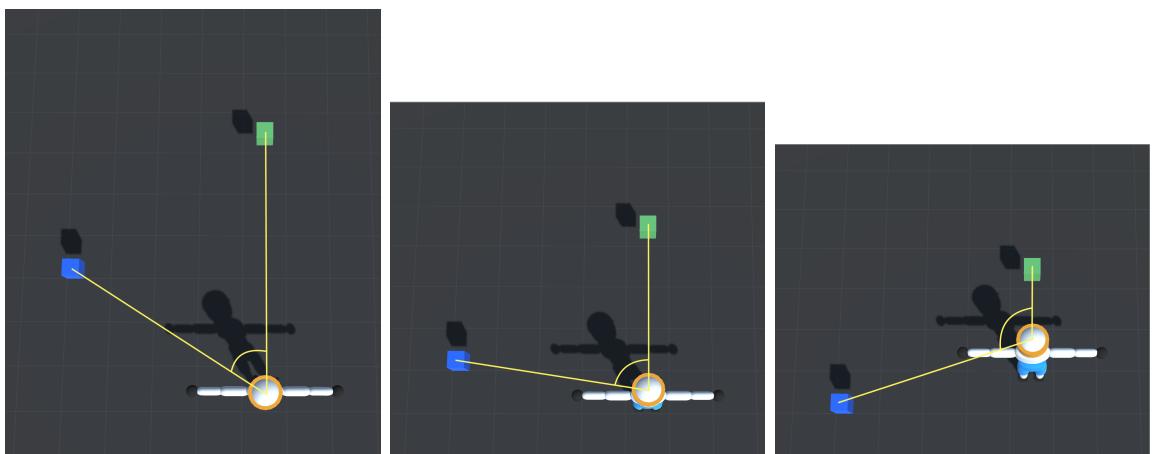


Abbildung 4.10: Blickwinkel Änderung durch Zielannäherung

Das Problem wurde behoben, indem das Blickziel in nachfolgenden Trainings kontinuierlich mit jedem Update neu platziert wurde, um somit den Blickwinkel gleich zu halten. In den bisherigen Trainingseinheiten hat der Läufer zudem eine Gangart optimiert, um alle Blickziele mittelmäßig zu erreichen. Um den Läufer zu motivieren, alle Blickziele stärker einzuhalten, wurde die Implementierung geändert, sodass der Blickwinkel beim Erreichen des Laufziels nur noch gewechselt wird, wenn die durchschnittliche Blickbelohnung über dem Schwellenwert von 0.7 liegt. Mit dieser neuen Bedingung beim Erreichen des Blickziels erreicht der Läufer die ersten Blickziele erfolgreich, stagniert jedoch anschließend an den neu gesetzten Blickzielen (siehe 4.11). Die kontinuierliche Neuplatzierung des Blickziels bei jedem Update hat dazu geführt, dass der Läufer nicht länger von einer konstanten Blickrichtung profitiert und gezwungen ist, seine Gangart besser an die Ziervorgaben anzupassen. Der Läufer braucht zu Beginn eine ganze Weile um zu lernen sich bis zum Ziel zu bewegen. Dadurch dass das Blickziel erst nach dem erreichen des Ziels mit einer durchschnittlichen Blickbelohnung von über 0,7 neu gesetzt wird, verbringt der Läufer zu viel Zeit mit den gleichen Zielen. Als Folge kommt es wieder dazu, dass der Agent ein Verhalten optimiert welches er anschließend nicht schafft an die neuen Ziele anzupassen.



Abbildung 4.11: Training Blickrichtungsziel mit Wechsel bei durchschnittlicher Blickbelohnung von 0.7

Um von Anfang an und separat vom Erlernen des Laufens die Blickbelohnung zu erlernen, wird die Bedingung für das Erreichen eines Blickziels erneut angepasst. Im folgenden Versuch wird der Blickzielwinkel gewechselt, sobald der Läufer 3 Sekunden auf das Ziel blickt. Implementiert wurde das Ganze mit einem Spherecast, um gleichzeitig die Genauigkeit, mit der der Läufer auf das Ziel schauen muss, anpassen zu können (siehe Abbildung 4.12). Spherecast, eine Technik, die verwendet wird, um eine kugelförmige Abfrage um die Blickrichtung des Läufers durchzuführen, ermöglicht es, über den Kugeldurchmesser die Blickfeld Größe anzupassen. je nach Kugelgröße und Dauer des Blickkontaktes kann die Schwierigkeit variiert werden. Grundlegend lässt sich darüber aber die Abhängigkeit zum Lernverlauf der Fortbewegung lösen. Gleichermassen kann zusätzlich auch die geforderte Präzision angepasst werden.

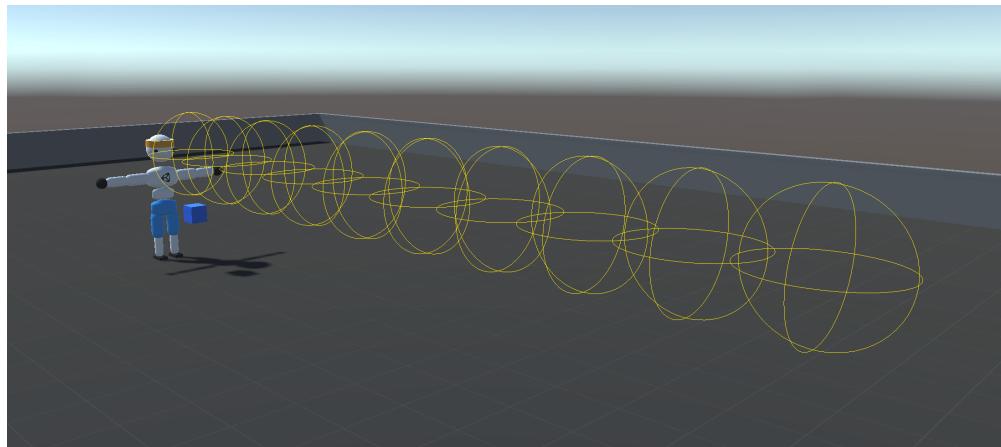


Abbildung 4.12: Spherecast in Blickrichtung

Das Training wurde einmal mit 3 Sekunden und einmal mit 2 Sekunden Blickkontaktzeit zum Erreichen des Blickziels durchgeführt, wobei das Training mit 2 Sekunden Blickkontaktzeit besser abschneidet. Trotz dieser Anpassungen ist der Schwierigkeitsgrad des Trainings noch sehr hoch. Selbst das bessere der beiden Trainings nach 120 Millionen Trainingsschritten ist noch weit davon entfernt, stabil mehrere Ziele am Stück zu erreichen. Es ist zu vermuten, dass weitere Anpassungen erforderlich sind um die gewünschte Stabilität zu erreichen.

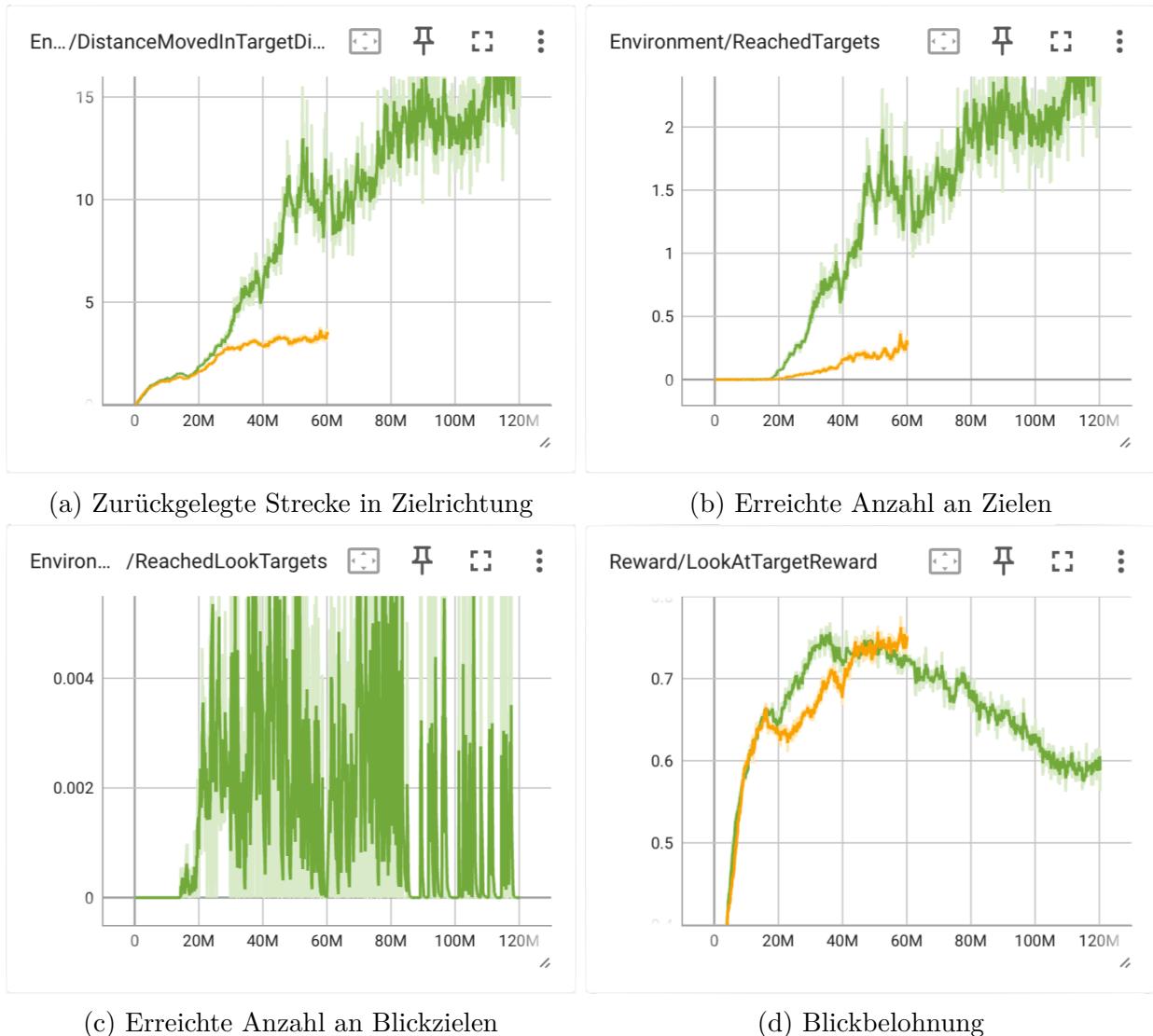


Abbildung 4.13: Training Blickrichtungsziel mit Wechsel bei Blickkontakt von 2 bzw. 3 Sekunden (orange = 3 sek, grün = 2 sek)

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 Gelenkachsen 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. Als MonoBehaviour kann das Skript im Unity Editor jedem beliebigen Objekt hinzugefügt werden. 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
10    //Körperteile initialisieren und zur Liste hinzufügen
11    foreach (Bodypart bps in
12        root.GetComponentsInChildren<Bodypart>())
13    {
14        bps.Initialize();
15        bps.onTouchingGround.AddListener(OnTouchingGround);
16        bp.onTouchedTarget.AddListener(OnTouchedTarget);
17        bodyparts.Add(bps);
18    }

```

```
18 }
19
20 //Körperteile zurücksetzen
21 public override void OnEpisodeBegin()
22 {
23     foreach (Bodypart bps in bodyparts)
24     {
25         bps.Reset();
26     }
27 }
28
29 //Beobachtung für Körperteil hinzufügen
30 public void CollectObservationBodyPart(Bodypart bps, VectorSensor
31 sensor)
32 {
33     sensor.AddObservation(bps.touchingGround);
34
35     sensor.AddObservation(m_OrientationCube.transform
36     .InverseTransformDirection(bps.rb.velocity));
37     sensor.AddObservation(m_OrientationCube.transform
38     .InverseTransformDirection(bps.rb.angularVelocity));
39
40     sensor.AddObservation(m_OrientationCube.transform
41     .InverseTransformDirection(bps.rb.position - root.position));
42
43     if (bps.dof.sqrMagnitude <= 0) return;
44
45     sensor.AddObservation(bps.rb.transform.localRotation);
46     sensor.AddObservation(bps.currentStrength /
47         bps.physicsConfig.maxJointForceLimit);
48 }
49
50 //Körperteilbeobachtungen an Beobachtung anfügen
51 public override void CollectObservations(VectorSensor sensor)
52 {
53     foreach (Bodypart bps in bodyparts)
54     {
55         CollectObservationBodyPart(bps, sensor);
56     }
57 }
58
59 //Aktion in Bewegung umwandeln
60 public override void OnActionReceived(ActionBuffers actionBuffers)
61 {
62     var continuousActions = actionBuffers.ContinuousActions;
63     int i = -1;
64
65     foreach (Bodypart bp in bodyparts)
66     {
67         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,
        targetRotZ);
    bp.SetJointStrength(jointStrength);
}
}

```

Listing 4.6: Ausschnitt Angepasstes Walker Agent Skript

4.3.3 Einrichtung

Der ausgewählte Charakter ist der Y Bot Charakter welcher in Abbildung 4.14 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.

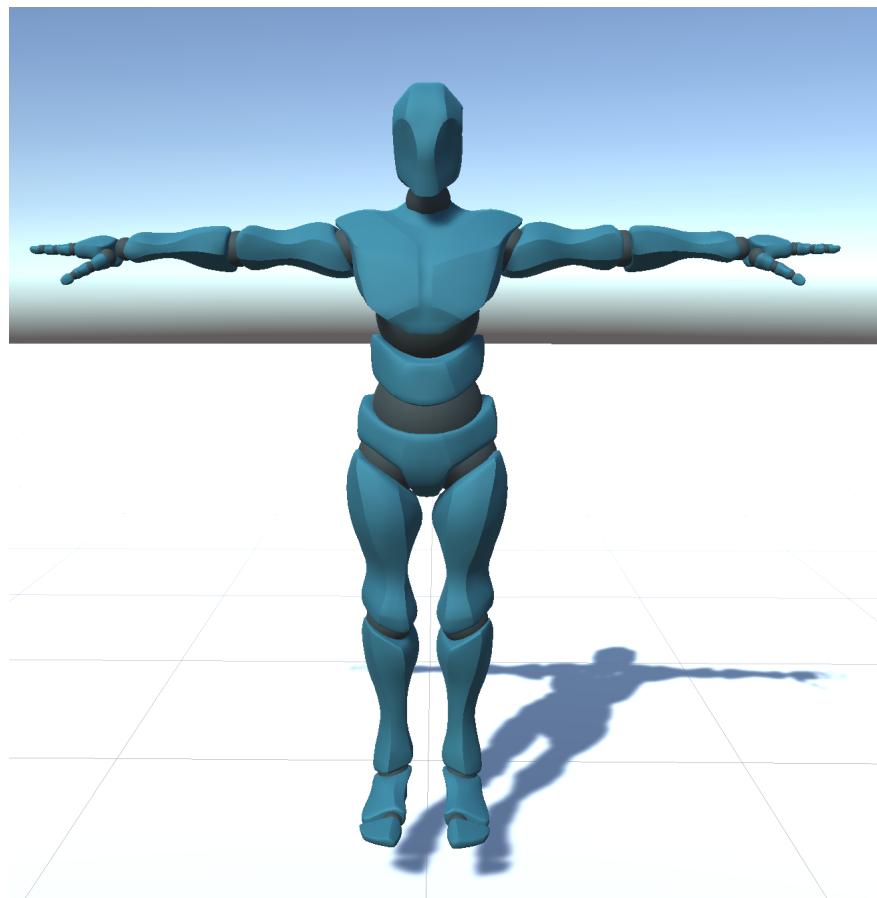


Abbildung 4.14: Mixamo Charakter Y Bot

Jedes Körperteil benötigt für das Steuern und Trainieren mit dem Walker Agenten Skript eine Kollisionskomponente, eine Festkörperkomponente und eine Körperteilkomponente. Zu-

sätzlich 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. Die Körperteilkomponente übernimmt die Konfigurationsparameter der Gelenk Motor Steuerung (siehe Abbildung 4.15). Parameter wie Stärke und maximale Rotationsgeschwindigkeit können 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.

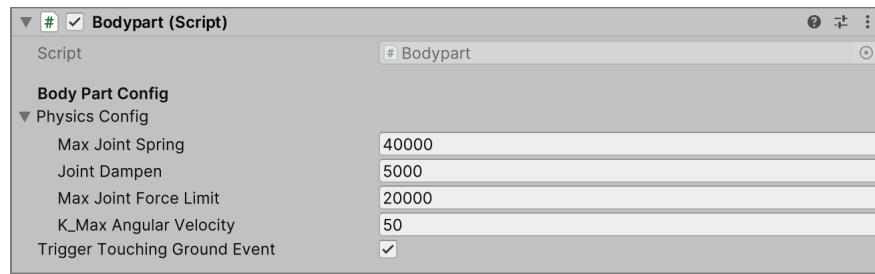


Abbildung 4.15: Körperteilkomponente

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

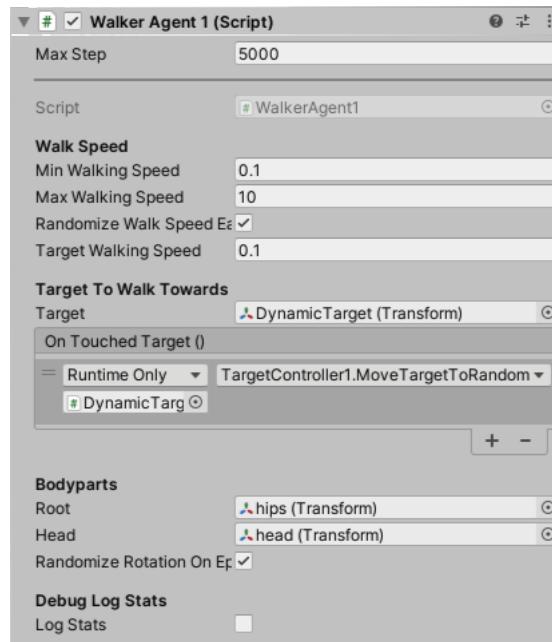


Abbildung 4.16: Walker Agentkomponente

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 etwas schlechteres 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 Abbildung 4.17). In der Abbildung 4.18 wird das erlernte Gangbild gezeigt. Der Läufer lernt in diesem Fall nicht das Laufen sondern galoppiert zum Ziel.

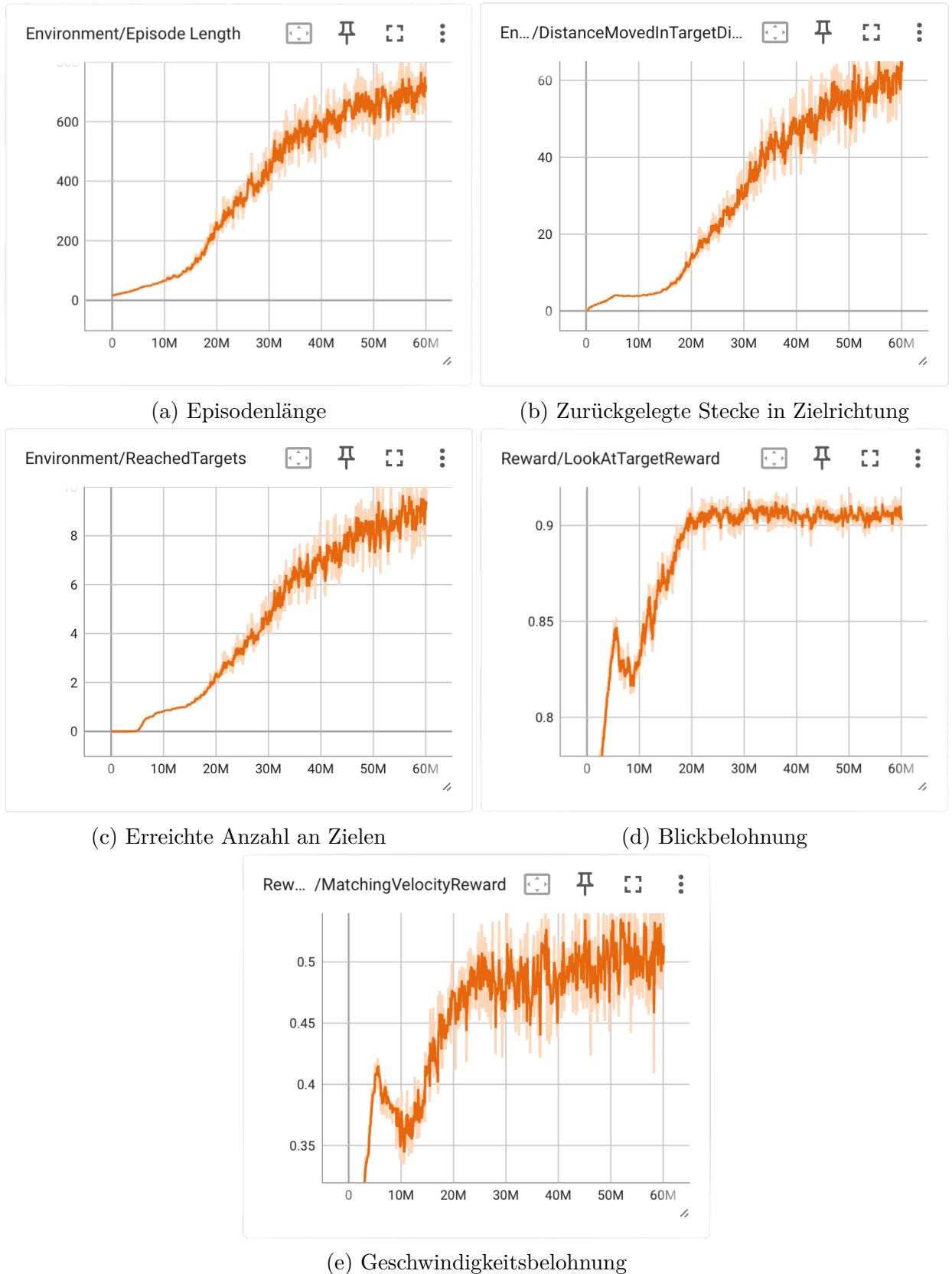


Abbildung 4.17: Training Mixamo Charakter

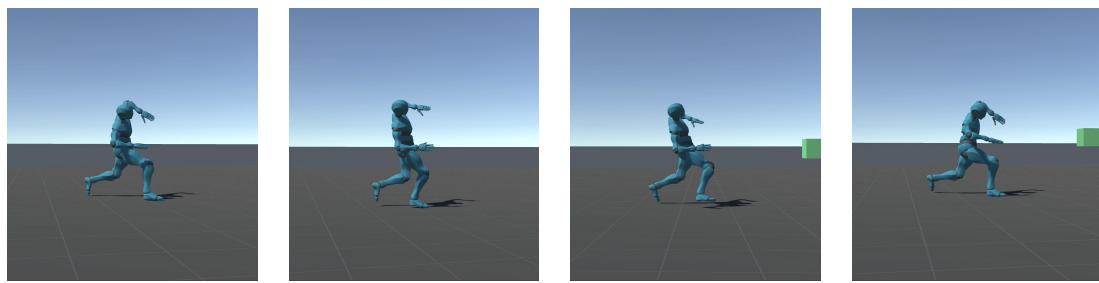


Abbildung 4.18: Mixamo Versuch 10 Gangbild

4.4 Gangbild Anpassungen

Das natürlich erlernte Gangbild eines gesunden Menschen ist sehr komplex. Im folgenden Kapitel wird das Gangbild des Mixamo Charakters basierend auf den Gangphasen der menschlichen Fortbewegung analysiert und angepasst. Wie bereits in der Analyse der Walker Demo erwähnt, ist das Gangbild des Walker Demo Läufers menschenähnlich und für die vereinfachte Darstellung des Charakters durchaus ausreichen. Die gesteigerte Komplexität des 3D Modells beim Mixamo Charakter führt bei der Wahrnehmung einem gesteigerten Realismus. Der Mixamo Charakter lernt zudem mehr ein galoppieren als das laufen welches der Walker Demo Läufer nach dem Training aufweist. Durch den gesteigerten Realismus und die Verschlechterung des Gangbilds wird in folgendem Kapitel getestet wie zusätzliche Belohnungen das erlernte Gangbild beeinflussen können.

4.4.1 Belohnung für Beinwechsel

Das Menschliche Gangbild besteht aus sich wiederholenden Phasen, die abwechselnd bei beiden Beinen durchlaufen werden. Um den Läufer dazu zu motivieren in einem regelmäßigen Intervall das vorangehende Bein zu wechseln wird ein Timer eingeführt welcher von 0 startet sobald das vorderste Bein in Zielrichtung gewechselt wurde. Ausgehend der verstrichenen Zeit seit dem letzten Wechsel bekommt der Läufer dann eine Strafe. Wie in Abbildung 4.19 zu sehen ist Bestrafung bis 1,2 Sekunden 0. Bleibt ein Bein länger als 1,2 Sekunden vorne steigt die Bestrafung linear an bis sie bei 5,2 Sekunden das Maximum von -1 erreicht.[4]

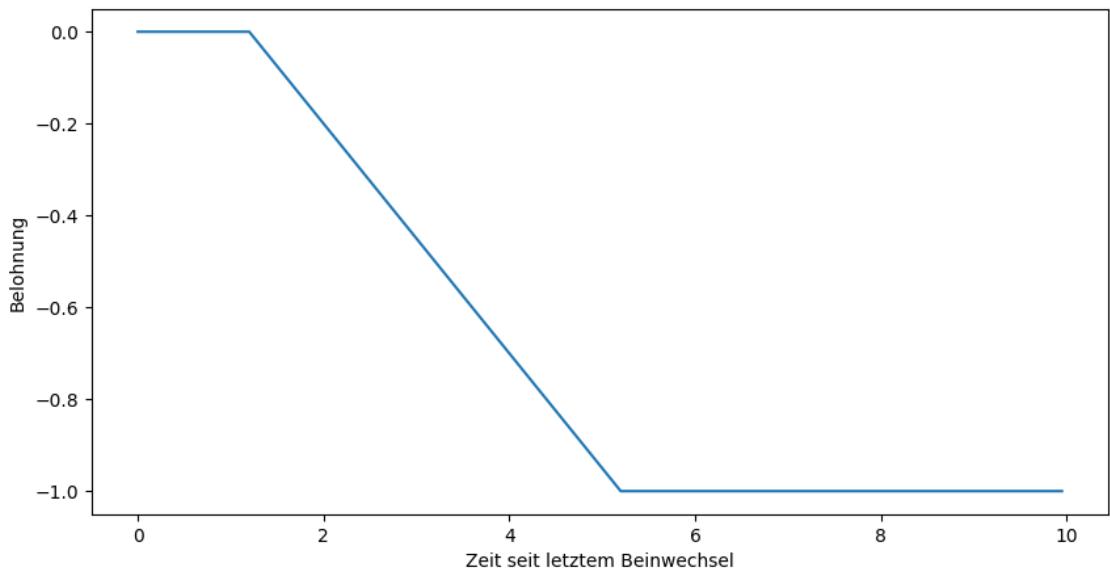


Abbildung 4.19: Beinwechsel Belohnung

Das einführen dieser Bestrafung hat beim Training erfolgreich dazu geführt das der Läufer in regelmäßigen Abständen das Standbein wechselt und somit ein natürlicheres Gangbild entwickelt (siehe Abbildung 4.20).

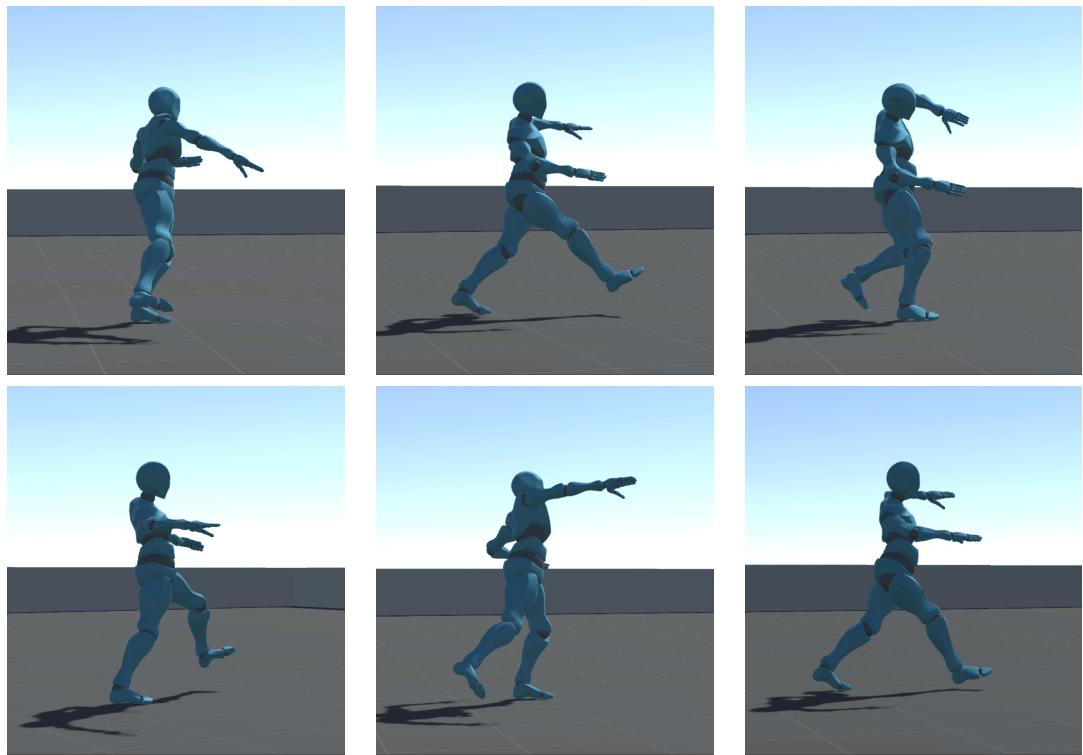


Abbildung 4.20: Mixamo Versuch 11 Gangbild

4.4.2 Belohnung für Energieminimierung

Ein großer Einfluss auf die Entwicklung des menschlichen Gangbilds ist die Energieminimierung. Der Läufer hat keine Wahrnehmung von Aufwand, daher sind die erlernten Be-

wegungsabläufe oft alles andere als effizient. 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. Ähnlich wie die Fußwechselbestrafung wird auch für die Energieminimerung eine Funktion verwendet welche die Bestrafung zwischen 150 und 3150 Watt linear ansteigt.

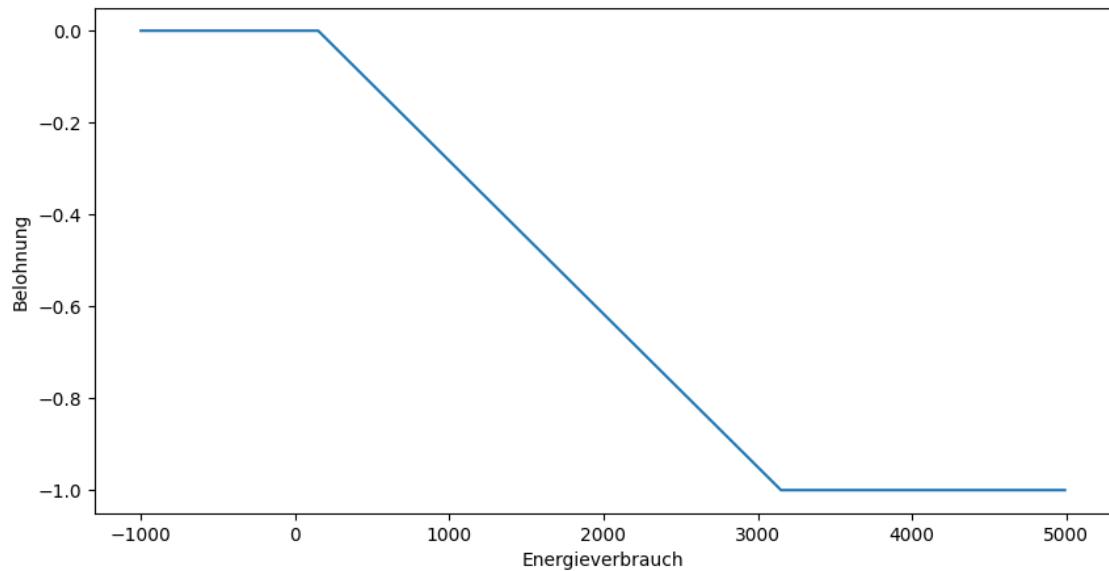


Abbildung 4.21: Energiespar Belohnung

Die Energieminimierung bringt den Läufer dazu seine Bewegungen zu optimieren und somit zu minimieren. Das Gangbild wirkt dadurch natürlicher. Vor allem die Bein und Körperbewegungen wurden wesentlich verbessert. Die Arme werden so gut wie gar nicht mehr bewegt um so viel Energie wie möglich zu sparen.

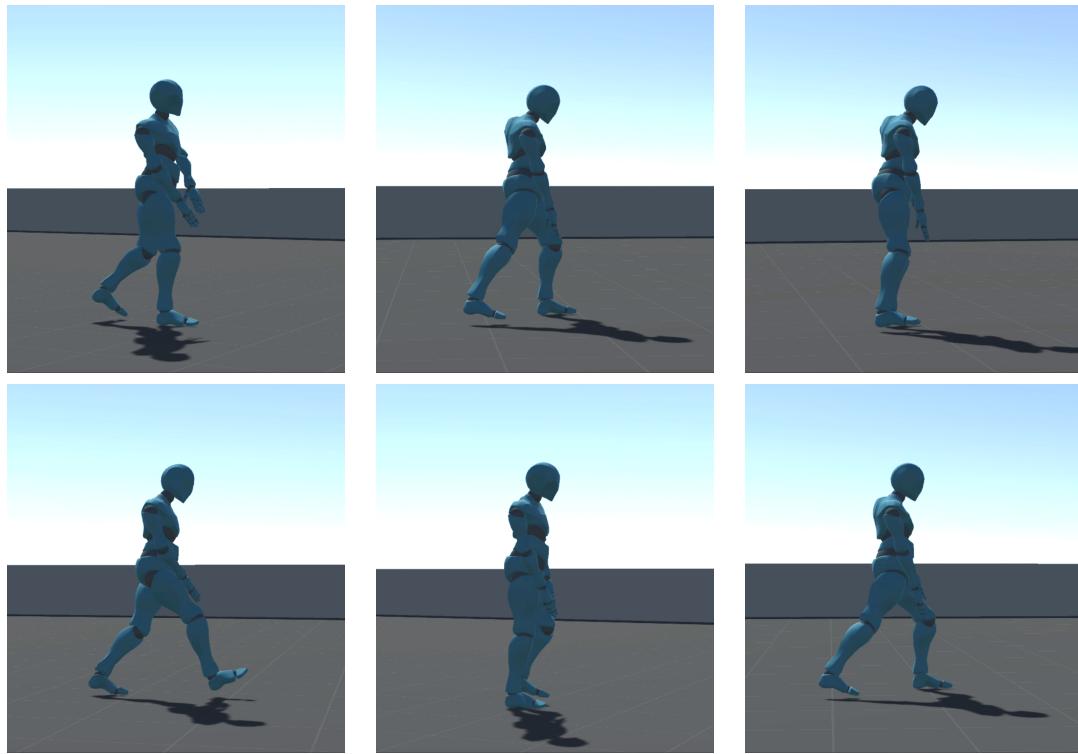


Abbildung 4.22: Mixamo Versuch 12 Gangbild

4.4.3 Belohnung Armpendel

Die Arme sind für eine natürliche Gehbewegung unabdinglich. Ein gesunder Mensch bewegt die Arme beim gehen gegensätzlich zu den Beinen. Einige Vermutungen warum der Mensch dieses Verhalten zeigt sind die Verbesserung der Stabilität, Energieminimierung oder die Abstammung von Vorfahren welche die Arme aktiv beim fortbewegen einsetzten.[\[2\]](#)

Der Grund ist für die Entwicklung eines natürlichen Charakterkontrollers jedoch zweit- rangig. Durch die vereinfachte physikalische Darstellung muss das für Menschen natürliche Verhalten teilweise über Umwege antrainiert werden.

4.4.4 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] Pieter Meyns, Sjoerd M Bruijn und Jacques Duysens. “The how and why of arm swing during human walking”. In: *Gait & posture* 38.4 (2013), S. 555–562.
- [3] Richard S Sutton und Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.
- [4] AI Warehouse. *AI Learns to Walk (deep reinforcement learning)*. 2023. URL: https://www.youtube.com/watch?v=L_4BPjLBF4E.