



Bachelor Thesis

Physik basierter Charaktercontroller mit Unity Machine Learning

Simon Grözinger*

21. August 2024

Erstprüfer: Prof. Dr. Tim Reichert

Zweitprüfer: Ulrich Straus

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

Eidesstattliche Erklärung

Hiermit erkläre ich eidesstattlich, dass die vorliegende Arbeit von mir selbstständig und ohne unerlaubte Hilfe angefertigt wurde, insbesondere, dass ich alle Stellen, die wörtlich oder annähernd wörtlich oder dem Gedanken nach aus Veröffentlichungen und unveröffentlichten Unterlagen und Gesprächen entnommen worden sind, als solche an den entsprechenden Stellen innerhalb der Arbeit durch Zitate kenntlich gemacht habe, wobei in den Zitaten jeweils der Umfang der entnommenen Originalzitate kenntlich gemacht wurde. Ich bin mir bewusst, dass eine falsche Versicherung rechtliche Folgen haben wird.

Ort, Datum

Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis	V
Tabellenverzeichnis	VII
Listings	VIII
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	18
4. Umsetzung Charaktercontroller	21
4.1. Nutzersteuerung	21
4.1.1. Anforderungen	21
4.1.2. First- und Thirdperson-Steuerung	21
4.1.3. Top-Down-Steuerung	22
4.2. Zusätzliche Bewegungsabläufe	23
4.2.1. Anforderungen	23
4.2.2. Separate Bewegungsabläufe	23
4.2.3. Laufrichtungen kombinieren	30
4.3. Charakterkompatibilität und Konfiguration	38
4.3.1. Anforderungen	39
4.3.2. Anpassungen	39
4.3.3. Einrichtung	41
4.3.4. Auswertung	44
4.4. Gangbild Anpassungen	46
4.4.1. Beinwechsel	46
4.4.2. Energieminimierung	48
4.4.3. Armpendel	49
4.4.4. Imitationslernen	51
5. Zusammenfassung und Ausblick	53
Literaturverzeichnis	54
A. Training Dokumentation	55

B. Codeausschnitte	58
B.1. Angepasstes Körperteil Skript	58
B.2. Angepasstes Agenten Skript	61
B.3. Angepasstes Zielsteuerung Skript	68
B.4. Angepasstes Orientierungsobjekt Skript	69
B.5. Implementierung Spherecast	70
B.6. Implementierungen der Belohnungen für eine verbesserte Gehbewegung	70
B.7. Implementierung von Deep Mimic Körperhaltungsbelohnung	72
B.8. Initialisierung der Körperhaltung von Animation	72

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. Agent Konfiguration	14
3.4. Gelenk Motor Steuerung	14
3.5. Walker Demo Geschwindigkeitbelohnungsfunktion	17
3.6. Walker Demo Blickbelohnungsfunktion	17
3.7. Walker Demo Training Graphen	19
3.8. Walker Demo Analyse Gangbild	20
4.1. Neue Sigmoid Geschwindigkeitbelohnungsfunktion	24
4.2. Training stehen mit neuer Belohnungsfunktion	24
4.3. Vergleich von Lauftraining mit Demo Belohnungsfunktion gegen Neue 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. Vergleich Training stehen mit Neuer Belohnungsfunktion (orange) und angepasster Demo Belohnungsfunktion (blau)	27
4.7. Unterschiedliche Blickrichtungen Training Graphen (grün = vorwärts, orange = rückwärts, rosa = rechts, blau = links)	29
4.8. Training Blickrichtungsziel mit Lehrplan	32
4.9. Training Blickrichtungsziel (blau = Winkelabweichung ± 90 Grad, grün = Winkelabweichung ± 180 Grad)	34
4.10. Blickwinkel Änderung durch Zielannäherung	35
4.11. Training Blickrichtungsziel mit Wechsel bei durchschnittlicher Blickbelohnung von 0.7	36
4.12. Spherecast in Blickrichtung	37
4.13. Training Blickrichtungsziel mit Wechsel bei Blickkontakt von 2 bzw. 3 Sekunden (orange = 3 sek, grün = 2 sek)	38
4.14. Alte Architektur	39
4.15. Neue Architektur	40
4.16. Mixamo Charakter Y Bot	41

4.17. Körperteilkomponente	43
4.18. Walker Agentkomponente	43
4.19. Mixamo Charakter Gangbild mit Walker Demo Gelenklimits und Gewichten	44
4.20. Mixamo Charakter Training mit Walker Demo Gelenklimits und Gewichten .	45
4.21. Beinwechsel Belohnung	47
4.22. Mixamo Charakter Gangbild mit Beinwechselbelohnung	47
4.23. Energiespar Belohnung	48
4.24. Mixamo Charakter Gangbild mit Energiesparbelohnung	49
4.25. Pendelsumme Distanz veranschaulichung	50
4.26. Armpendel Bestrafung	50
4.27. Mixamo Charakter Gangbild mit Körperhaltungbelohnung	52

Tabellenverzeichnis

3.1. Walker Agent Körperteile	13
3.2. Walker Agent Beobachtung	15
3.3. Walker Agent Körperteil Beobachtung	16
3.4. Walker Agent Aktion	16
4.1. Mixamo Charakter Körperteile	42
A.1. Training Documentation	57

Listings

2.1. Agent Funktionen	4
2.2. Trainer Konfigurationsdatei	6
4.1. Nutzersteuerung für First- und Thirdperson	21
4.2. Nutzersteuerung für Top-Down	22
4.3. Blickrichtung festlegen mit Richtungs Enum	27
4.4. Laufrichtung Modell wechseln	30
4.5. Lehrplan für das Blickziel	31
4.6. Agent Aktion in Bewegung umwandeln	40
B.1. Körperteil Skript	58
B.2. Agenten Skript	61
B.3. Zielsteuerung Skript	68
B.4. Orientationsobjekt Skript	69
B.5. Implementierung von Spherecast um Blickkontakt überprüfen zu können	70
B.6. Implementierung von zusätzliche Belohnungen für Gehbewegungsanpassungen	70
B.7. Implementierung von Deep Mimic Körperhaltungsbelohnung	72
B.8. Implementierung Initialisierung des Charakters mit Körperhaltung aus Animation	72

1. Einleitung

Seit einigen Jahren steigt die Präsenz von AI und somit maschinellem Lernen in unserem Alltag. Systeme wie ChatGPT sind nicht mehr weg zu denken. Ein interessantes Feld ist dabei auch die Steuerung von Robotern, speziell Humanoiden Robotern. Die Roboter werden vorab meist in einer virtuellen Trainingsumgebung trainiert, um negative Auswirkungen wie Maschinenschäden zu verhindern und gleichzeitig die Notwendigkeit für die teure Anschaffung mehrerer Roboter zu vermeiden. Der Schritt zur Verwendung ähnlicher Systeme in der Videospielbranche ist daher nicht weit entfernt.

Nvidia und Ubisoft zeigen schon seit 2020 Prototypen zur Verwendung von maschinellem Lernen im Prozess der Charaktersteuerung bzw. Charakteranimation.^{[5][1]} Im Fall von Ubisoft wird klar gezeigt, welche Komplexität das Animationssystem ihrer top Titel aufweist. Maschinelles Lernen ermöglicht die Reduktion dieser Systeme in antrainierten Modellen. Speziell die Verwendung von physikalisch simulierten Charakteren ermöglicht einen Lernprozess vergleichbar mit dem eines Menschen.

Ziel der Arbeit ist es, einen Charakter physikalisch in der Unity Spieleumgebung zu simulieren. Der Charakter soll mit maschinellen Lernverfahren darauf trainiert werden, das Gleichgewicht zu halten und sich zu einem Ziel zu bewegen. Als Basis für die Trainingsumgebung soll die im Unity ML-Agents Framework entwickelte Walker Demo zum Einsatz kommen. Die Demo soll erweitert werden, sodass der Walker über Tastatureingaben gesteuert und somit in Videospielen als Ersatz für traditionelle Animationssysteme verwendet werden kann. Um die Vielfalt von Charakteranimationen abzudecken wird analysiert, wie weitere Bewegungsabläufe in das bestehende System eingefügt werden können. Außerdem soll auch die Kompatibilität mit unterschiedlichen Charaktermodellen geprüft werden.

Zu Beginn wird im Kapitel Grundlagen das Teilgebiet Verstärkendes Lernen aus dem Fachgebiet maschinelles Lernen erklärt. Darauf aufbauend wird anschließend der Aufbau, die Komponenten und Implementierungsschnittstellen der Unity ML-Agents Library aufgezeigt. Abgeschlossen werden die Grundlagen mit einer Übersicht der für die Simulation verwendeten Physikkomponenten in Unity. Auf die Grundlagen folgt eine Analyse der Walker Demo. Die Walker Demo ist eine Trainingsumgebung, welche innerhalb des Unity ML-Agents Projekts zur Demonstration der Fähigkeiten der Library entwickelt wurde. Im weiteren Verlauf der Arbeit dient diese Demonstration als Basis für den entwickelten Charaktercontroller. In Kapitel 4 wird näher auf die Entwicklung und somit den ausgeführten Versuchen sowie die darauf folgenden Ergebnisse eingegangen. Kapitel 4 teilt sich dabei in 4 Teile. In Teil 1 wird die Nutzersteuerung thematisiert. Teil 2 ermittelt, welche Anpassungen am Trainingsablauf gemacht werden müssen, um gängige Bewegungsabläufe eines Videospiel Charakters zu ermöglichen. In Teil 3 - Charakterkompatibilität und Konfiguration - werden die Komponenten der Demo angepasst, um die Konfiguration zu vereinfachen sowie das Steuern von unterschiedlichen Charaktermodellen zu gewährleisten. Teil 4 analysiert Anpassungen, um das erlernte Gangbild natürlicher zu gestalten. Abschließend werden in Zusammenfassung und Ausblick die in der Arbeit umgesetzten Prototypen auf ihre Anwendbarkeit in Videospielen analysiert und ein Ausblick für weitere Forschungen sowie bereits existierende Forschungen in anderen Softwareumgebungen aus diesem Bereich aufgezeigt.

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.^[7]

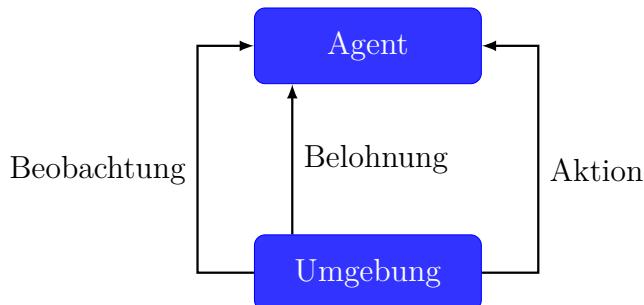


Abbildung 2.1.: Verstärkendes Lernen Ablauf

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.[3]

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.[3]

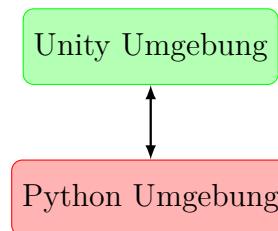


Abbildung 2.2.: Unity ML-Agents Aufbau

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

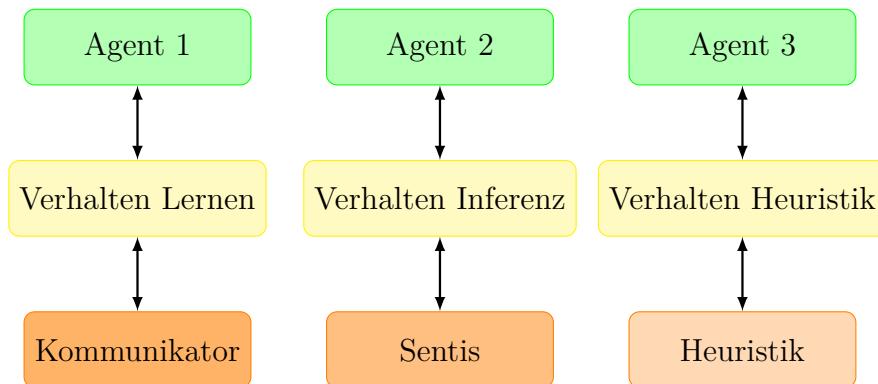


Abbildung 2.3.: Unity ML-Agents Aufbau Unity Umgebung

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

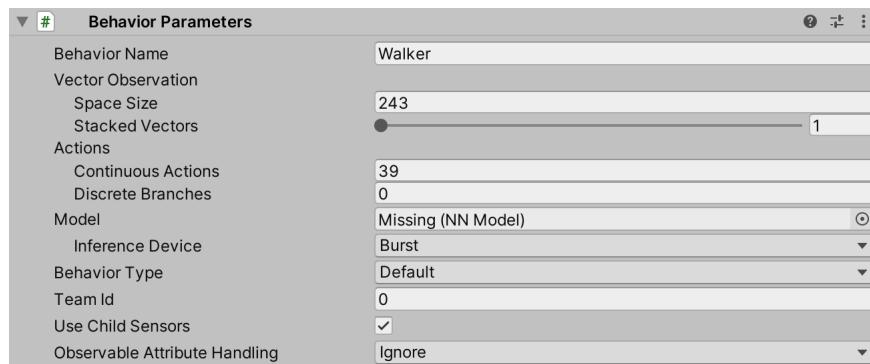


Abbildung 2.4.: Unity ML-Agents Verhalten Parameter Komponente

Die Parameter aus Abbildung 2.4 werden in folgender Auflistung erklärt.

- 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.

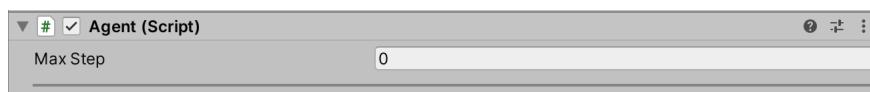


Abbildung 2.5.: Unity ML-Agents Agenten Komponente

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.

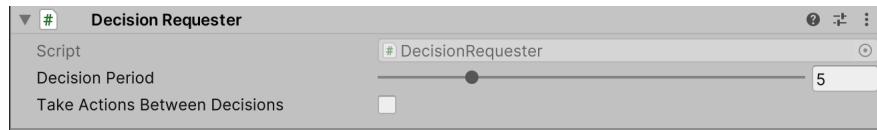


Abbildung 2.6.: Unity ML-Agents Entscheidung Anfragen Komponente

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. Abbildung 2.7 zeigt die Struktur der Python-Umgebung. 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.

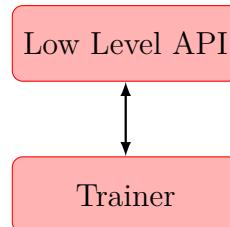


Abbildung 2.7.: Unity ML-Agents Aufbau Python Umgebung

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, eine Technik für das frühe Beenden einer Trainingsepisode zur Vermeidung von Training in unvorteilhaften Zuständen, ist die erreichte Episodenlänge ebenfalls sehr aussagekräftig, da sie anzeigt, wie lange der Agent in der Umgebung bestehen kann. Unter der Rubrik "Policy" finden sich auch die Graphen, welche den Verlauf der Hyperparameter darstellen. Die linke Seitenleiste listet alle im aktuellen Verzeichnis gespeicherten Trainingseinheiten auf. Darüber können Trainingsets ausgewählt und anschließend in den Graphen durch unterschiedlich farbige Linien verglichen werden. Diese Visualisierungen ermöglichen eine detaillierte Analyse und den Vergleich verschiedener Trainingsläufe, was zur Optimierung des Trainingsprozesses beiträgt.

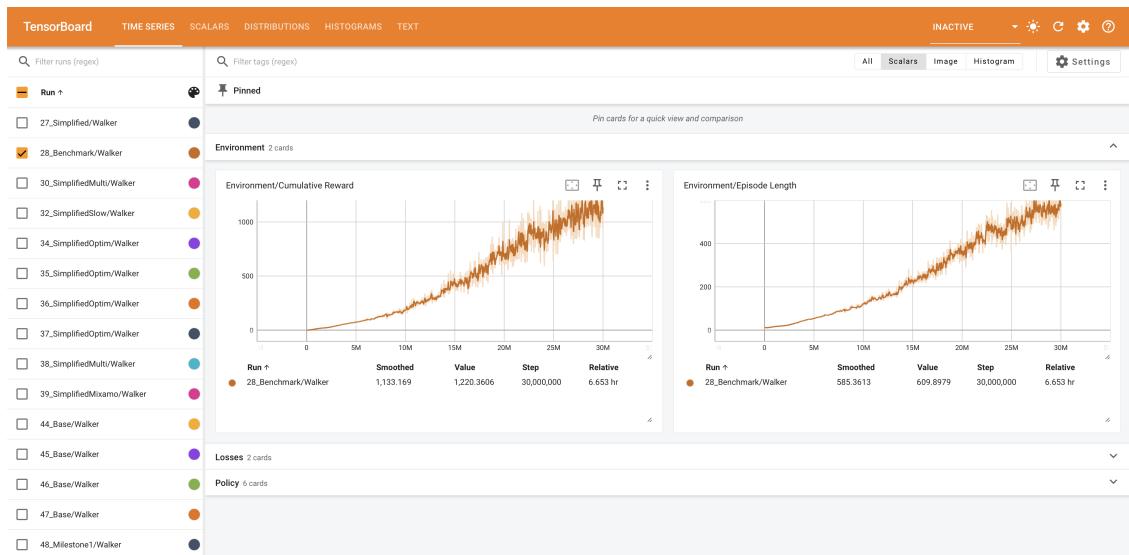


Abbildung 2.8.: Tensorboard Ansicht

Nicht immer geben die vorgefertigten Graphen alle relevanten Informationen wieder. Um die erfassten Daten und damit die Graphen zu erweitern, bietet Unity ML-Agents über die Statistikrekorder die "Statistik Hinzufügen" Funktion. Neu hinzugefügte Werte werden über die Episode und alle Umgebungen aggregiert. Als Aggregationsmethode kann zwischen 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.

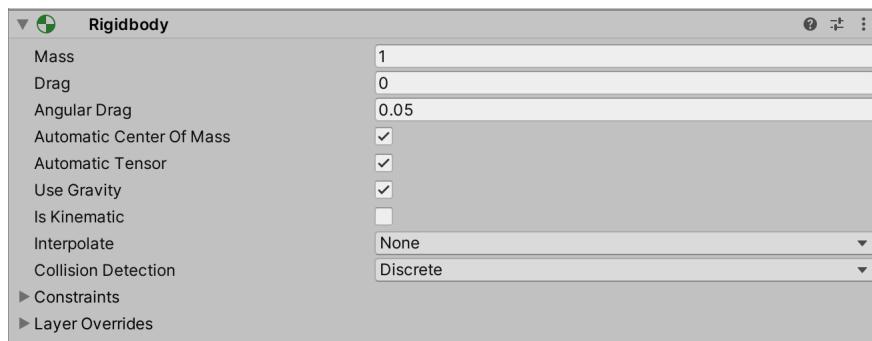


Abbildung 2.9.: Unity ML-Agents Physik Festkörper

Die Parameter der Komponente aus Abbildung 2.9 werden in folgender Auflistung erläutert.

- 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.

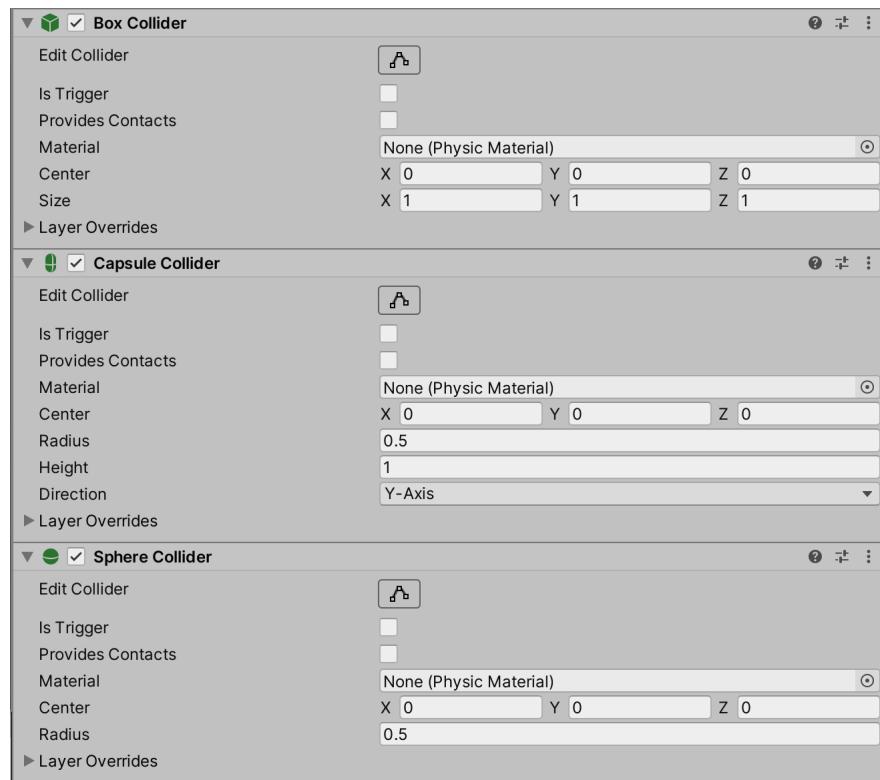


Abbildung 2.10.: Unity ML-Agents Physik Kollisionskomponenten

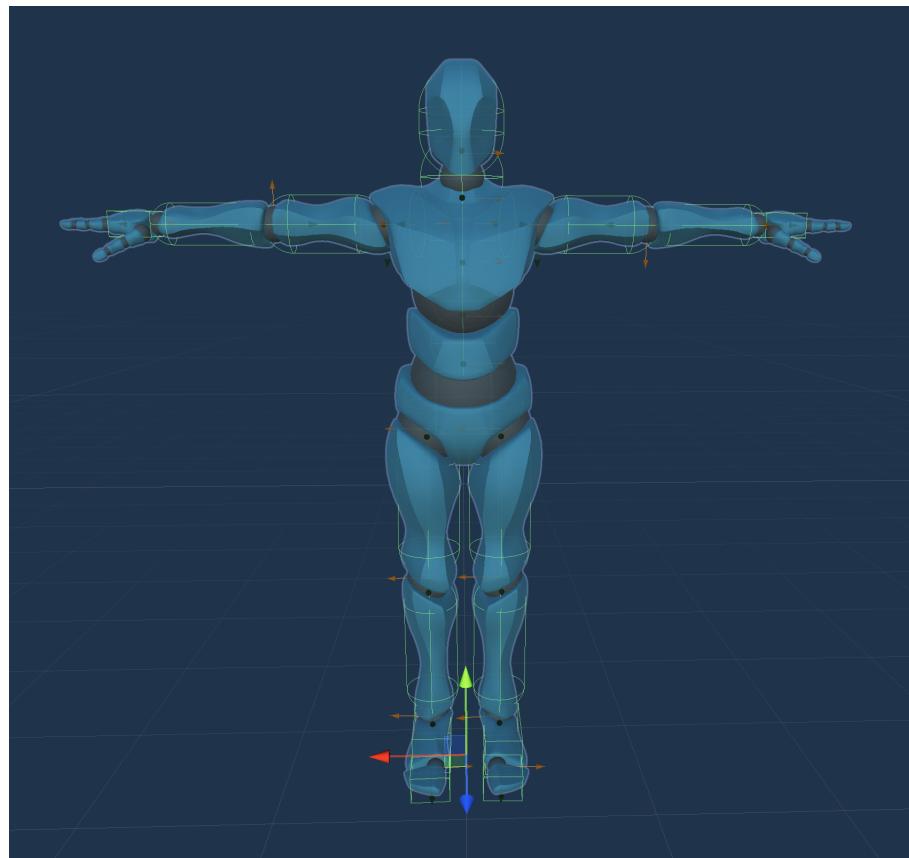


Abbildung 2.11.: Unity ML-Agents Physik Charakter vereinfacht mit Kollisionskomponenten

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 zeigt die Gelenkkomponente im Unity Inspector, in der Auflistung darunter werden ihre Parameter näher erklärt.

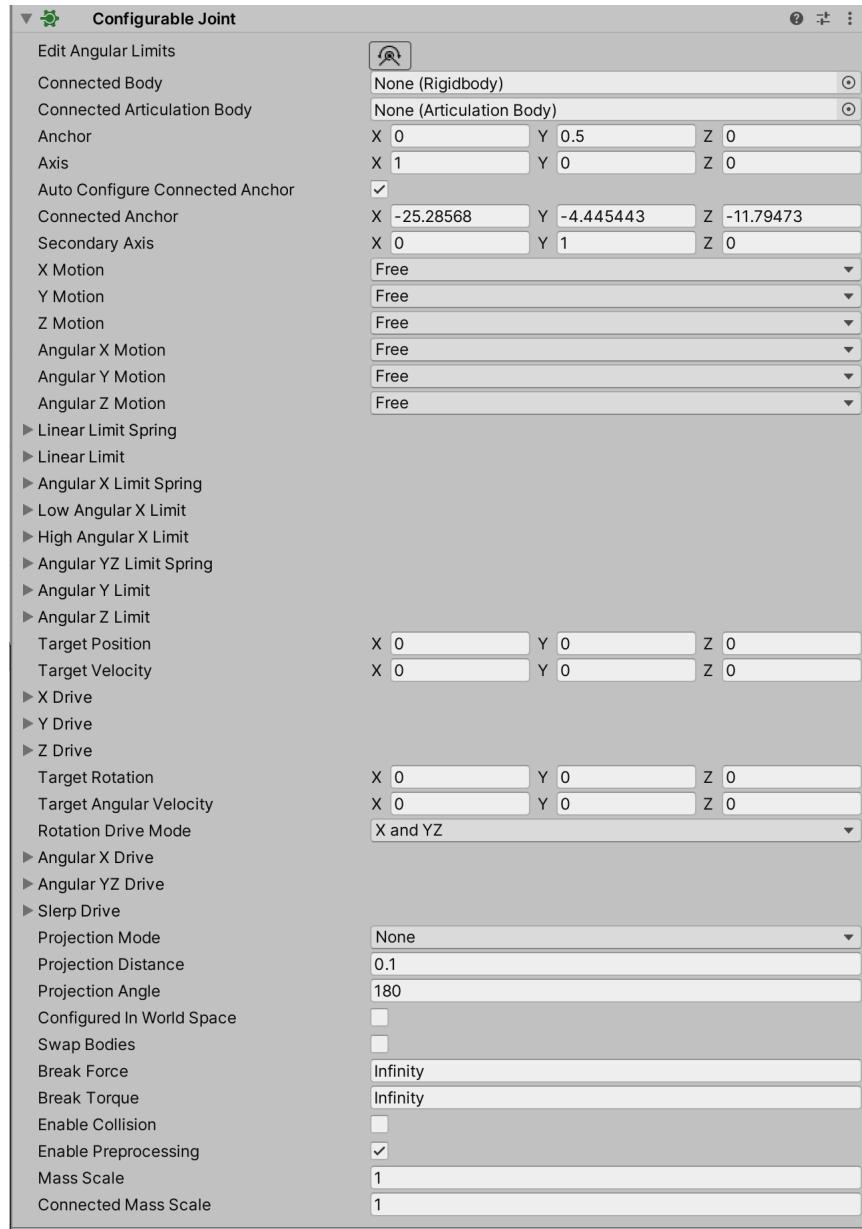


Abbildung 2.12.: Unity ML-Agents Physik Gelenk

- **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.

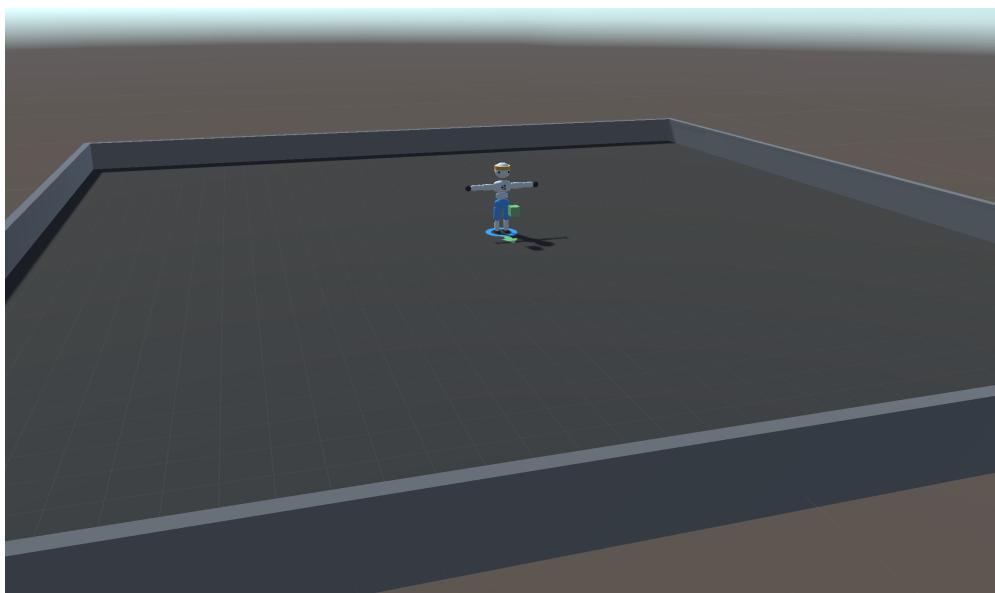


Abbildung 3.1.: Walker-Demo Umgebung

Der Läufer (zu sehen in Abbildung 3.2) 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 3.3 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 3.4 bestimmen die Stärke, mit welcher die Gelenke in die Zielstellung bewegt werden. Die Joint Drive Settings werden unterhalb von Abbildung 3.4 einzeln erklärt. 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.

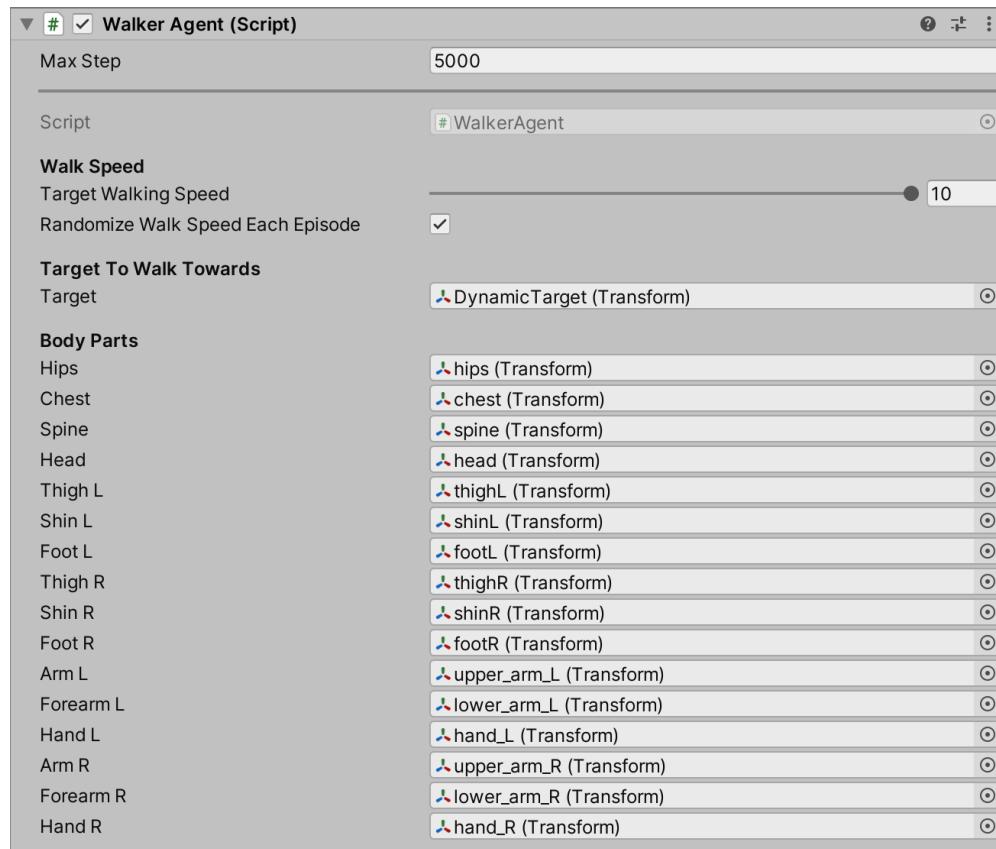


Abbildung 3.3.: Agent Konfiguration

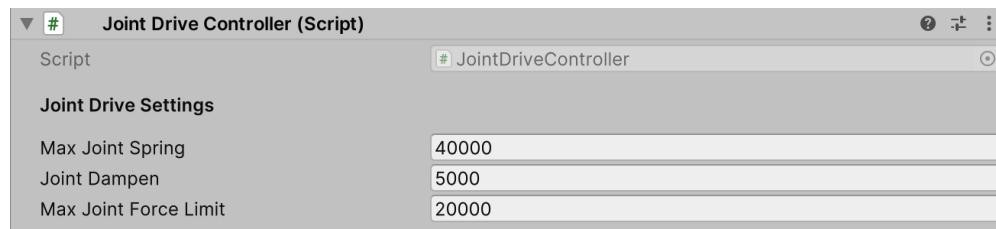


Abbildung 3.4.: Gelenk Motor Steuerung

- **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).

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. Um die Positionen und Rotationen der Körperteile mit möglichst wenig Ablenkung einzufangen, wird ein Stabilisierungsobjekt verwendet. Das Stabilisierungsobjekt wird in jedem Physikupdate auf die neue Hüftposition gesetzt. Die Richtung wird zusätzlich von der Hüfte in Richtung Ziel festgelegt. Die Verwendung dieses Stabilisierungsobjektes hat zur Folge, dass alle Bewegungen relativ zu einer gemeinsamen Basis abgebildet werden. Koordinatenunterschiede spielen dadurch keine Rolle. Der Läufer kann ein einziges Verhalten in Zielrichtung erlernen.

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 (abgebildet in 3.5) 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 (abgebildet in 3.6) bewertet. 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.

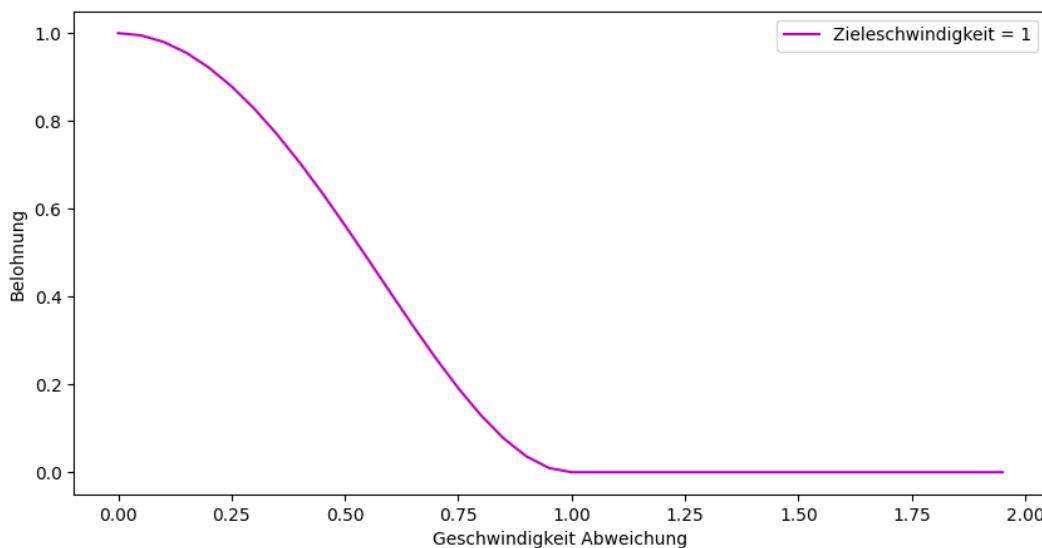


Abbildung 3.5.: Walker Demo Geschwindigkeitbelohnungsfunktion

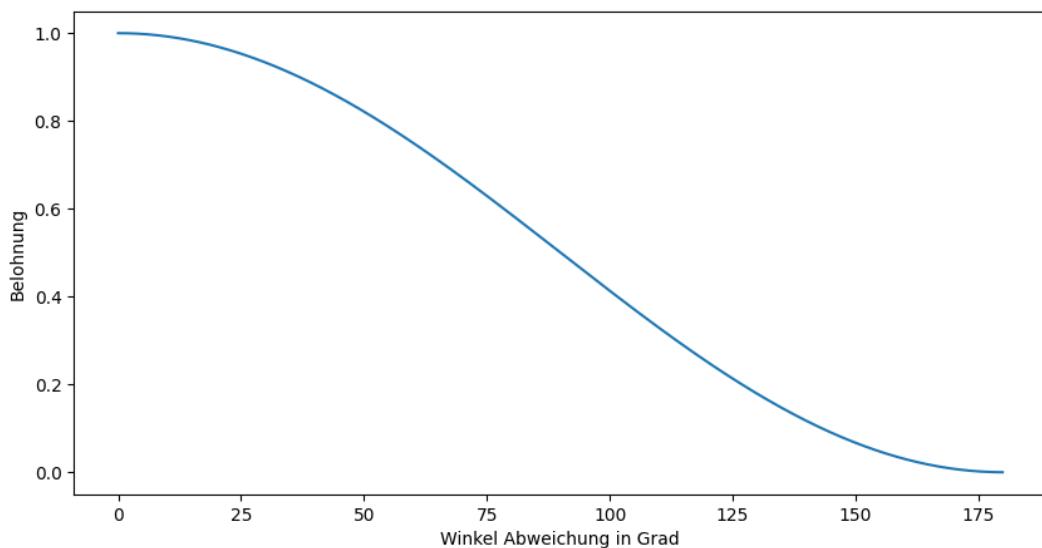


Abbildung 3.6.: Walker Demo Blickbelohnungsfunktion

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

Unity stellt für die Demonstrationen Konfigurationsdateien für alle unterstützten Lernalgorithmen bereit. Unity ML-Agents implementiert sowohl PPO als auch SAC. Beide Algorithmen werden erfolgreich für die Verwendung bei kontinuierlichen Kontrollproblemen, wie dem in dieser Arbeit thematisierten Kontrollproblem eines zweibeinigen Charakters, eingesetzt. Ein großes Aushängeschild des SAC-Algorithmus ist seine Sample-Effizienz, was bedeutet, dass mit weniger Simulationsschritten ein besseres Ergebnis erzielt wird. Bei ersten Versuchen mit der Walker-Demo ist dies auch zu beobachten. Jedoch zeigt sich, dass trotz der unterschiedlichen Anzahl an Simulationsschritten die verbrachte Zeit, um ein Ergebnis zu erreichen, vergleichbar ist. Dies deutet darauf hin, dass der SAC-Algorithmus zwar effizienter mit den verfügbaren Daten umgeht, aber nicht zwangsläufig schneller zu einem brauchbaren Ergebnis führt. Ein Nachteil dieser Effizienz ist, dass durch die geringere Anzahl an Simulationsschritten auch weniger Daten erfasst werden, was zu weniger aufschlussreichen Auswertungen führt. Zudem zeigte sich in den ersten Tests, dass der PPO-Algorithmus zu einem natürlicheren Verhalten des Charakters führt. Aus diesen Gründen wird in folgenden Kapiteln der PPO-Algorithmus zur Entwicklung verwendet.

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%.

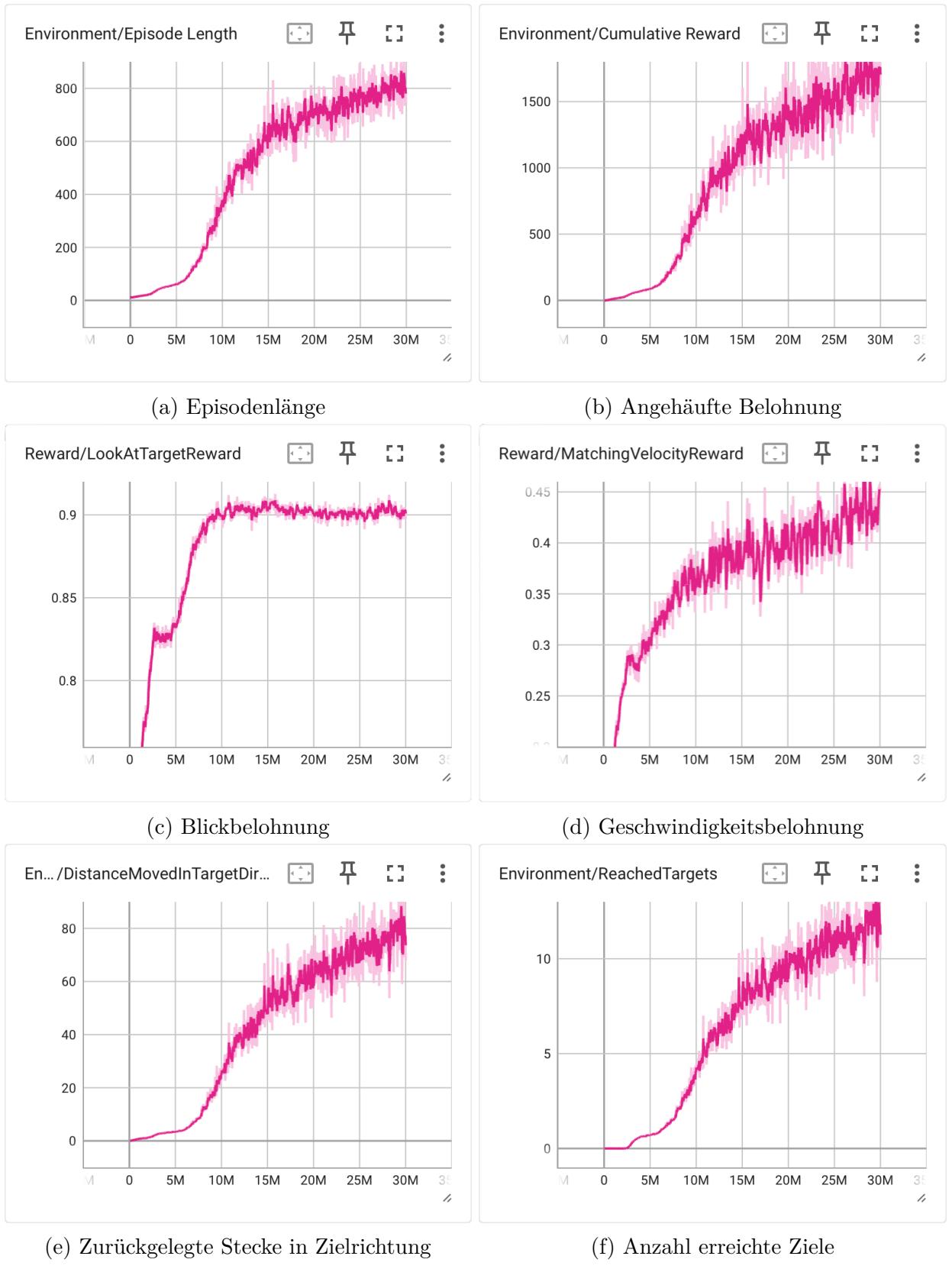


Abbildung 3.7.: Walker Demo Training Graphen

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. 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.

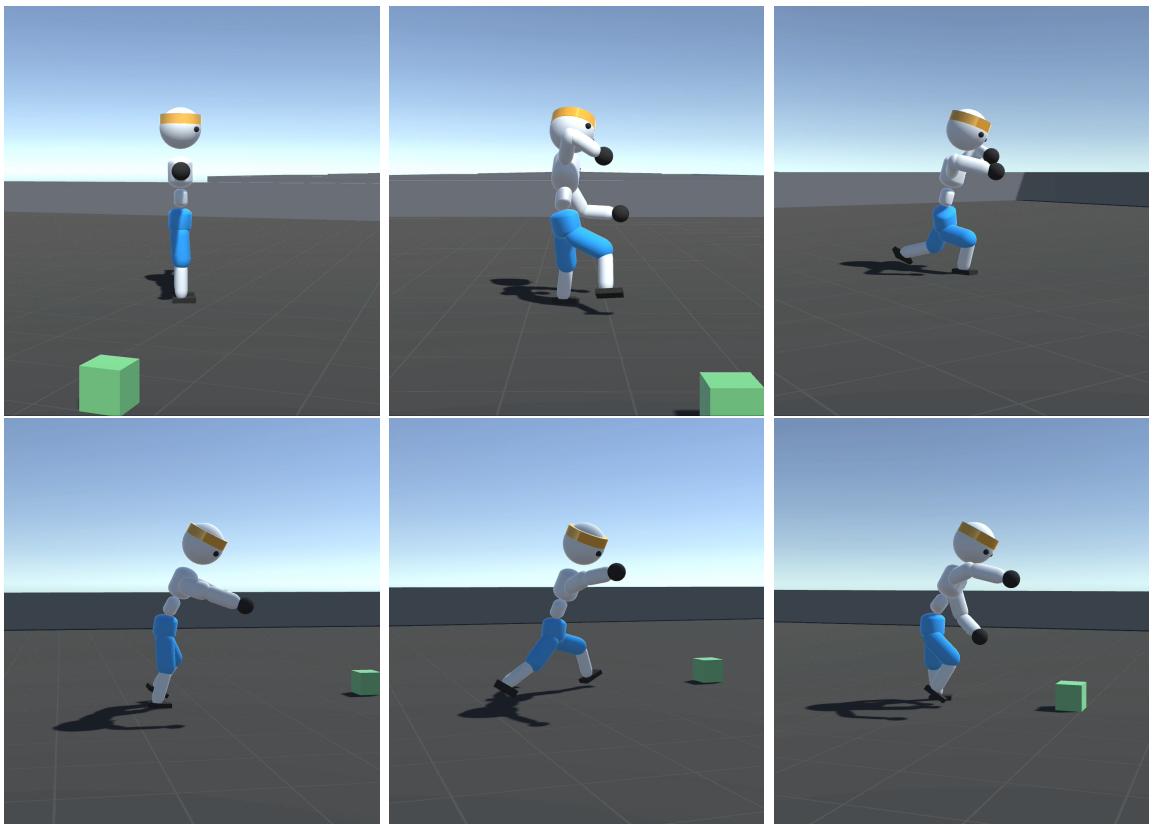


Abbildung 3.8.: Walker Demo Analyse Gangbild

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;
23
24     //Setzen der Zielposition
25     target.position = position;
26 }
```

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,
16                             normalizedMouseY);
17
18     //Berechnung der Rotation
19     rotation = Quaternion.LookRotation(mousePos, Vector3.up);
20
21     //Position berechnen
22     position = root.position + Vector3.forward * inputVert +
23                 Vector3.right * inputHor;
24
25     //Setzen der Zielposition
26     target.position = position;
27 }
```

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 weg bewegen, 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.

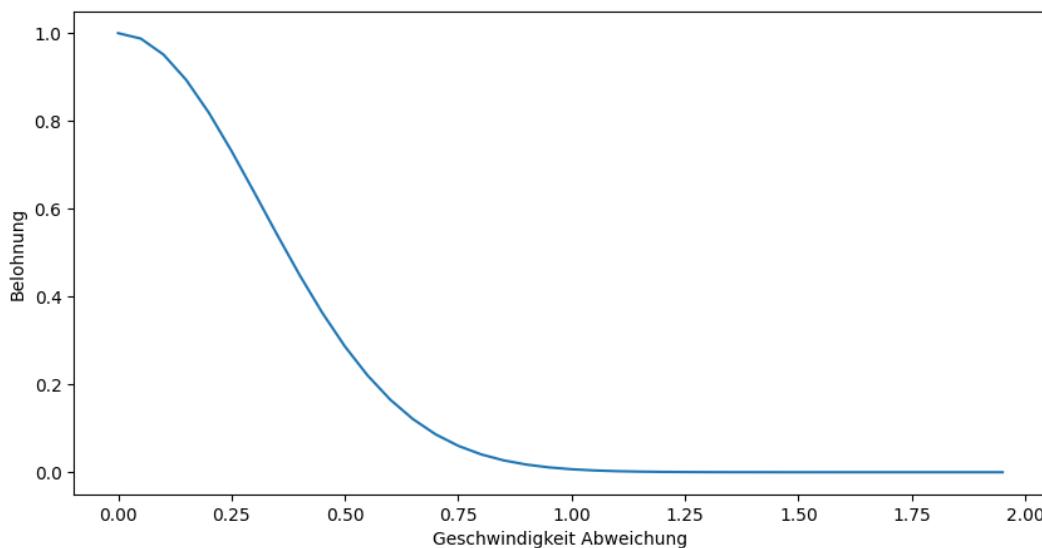


Abbildung 4.1.: Neue Sigmoid Geschwindigkeitbelohnungsfunktion

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.

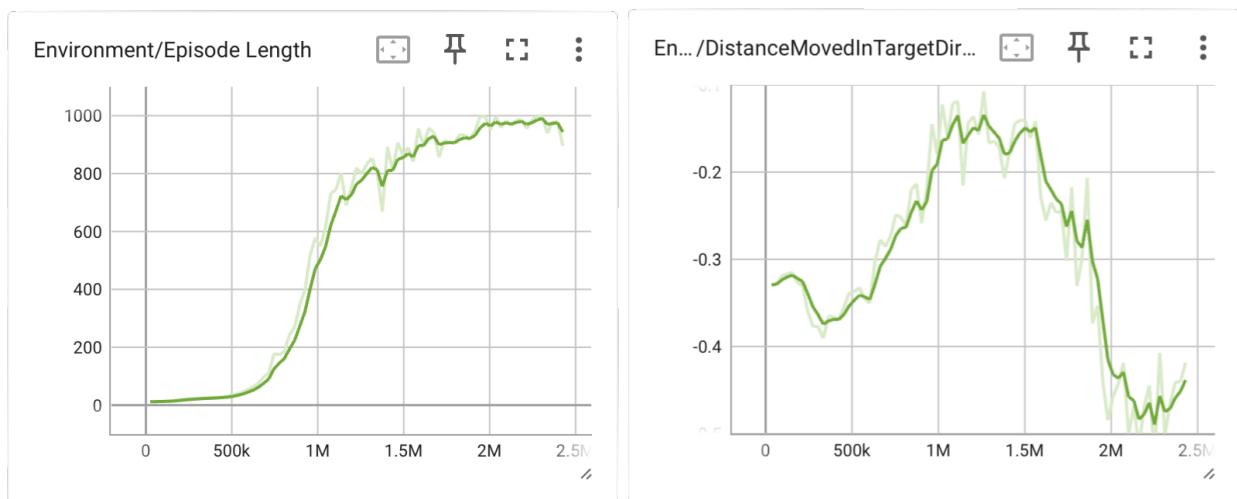


Abbildung 4.2.: Training stehen mit neuer Belohnungsfunktion

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.

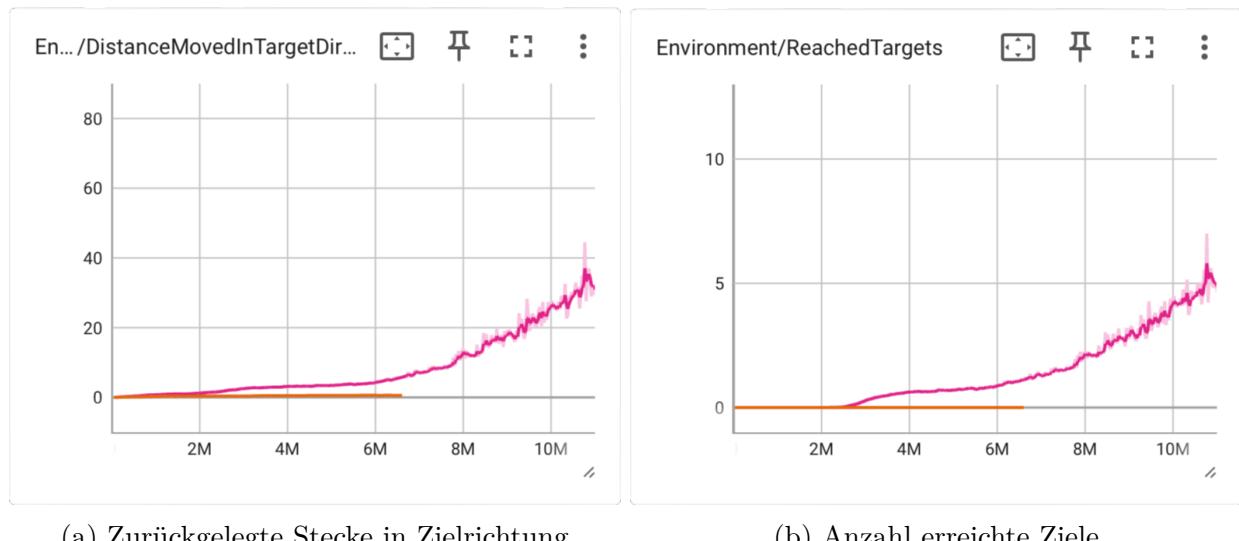


Abbildung 4.3.: Vergleich von Lauftraining mit Demo Belohnungsfunktion gegen Neue Belohnungsfunktion

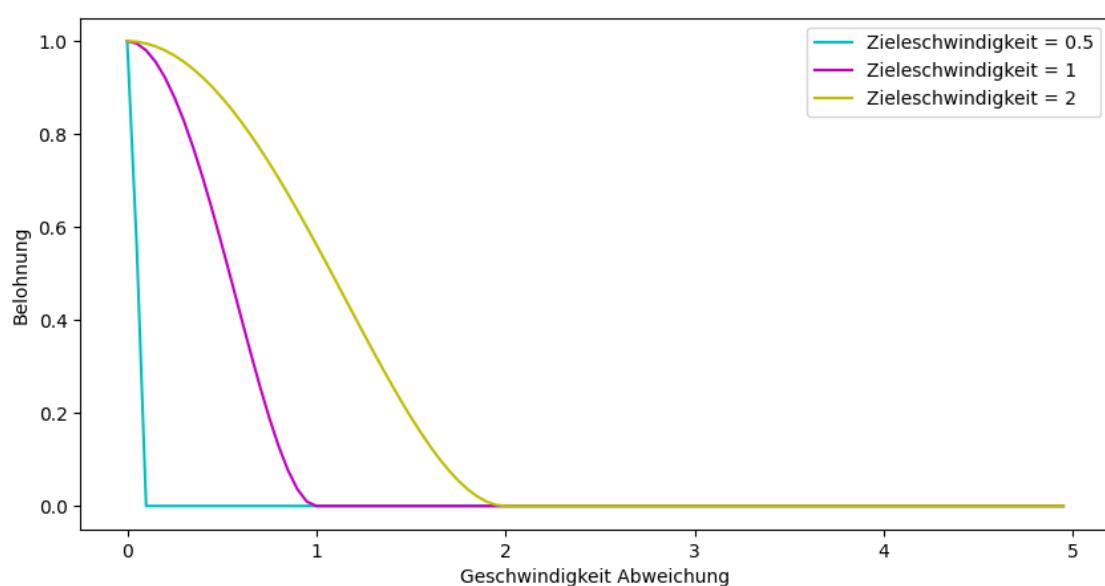


Abbildung 4.4.: Vergleich der Demo Belohnungsfunktion unter verschiedenen Zielgeschwindigkeiten

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.

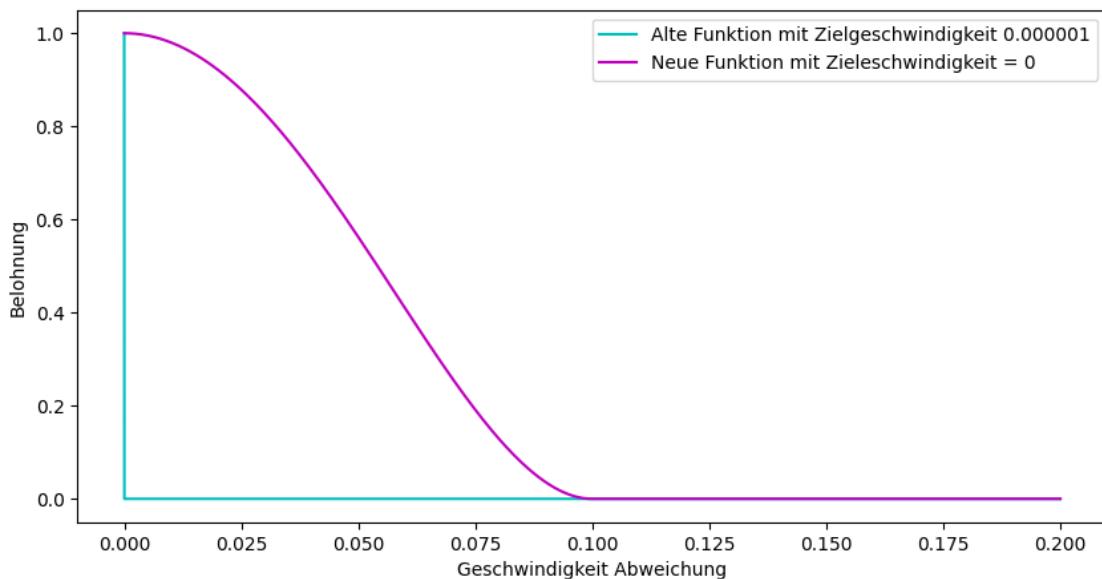


Abbildung 4.5.: Vergleich Demo gegen Belohnungsfunktion mit 0.1 Limit

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.

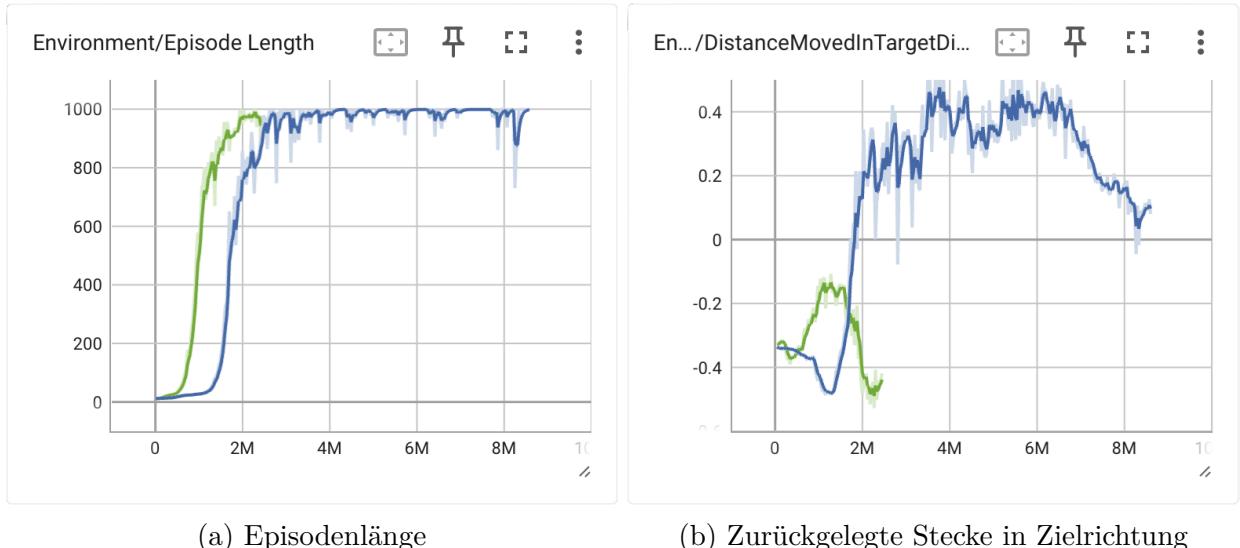


Abbildung 4.6.: Vergleich Training stehen mit Neuer Belohnungsfunktion (orange) und angepasster Demo Belohnungsfunktion (blau)

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 Seitwärtslauen. 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 Vorwärtsbewegung 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 festlegen mit Richtungs Enum

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. Bei der Bewegungsrichtung seitlich - links ist jedoch eine starke Abweichung zu verzeichnen. Diese Abweichung zeigt sich in einer geringeren Übereinstimmung der Geschwindigkeit sowie weniger erreichten Zielen und einer kürzeren zurückgelegten Distanz im Vergleich zur seitlichen Bewegung nach rechts. Da die seitlichen Bewegungsrichtungen symmetrisch zueinander identische Ergebnisse erzielen sollten, wird davon ausgegangen, dass diese Abweichung auf die zufällige Natur des maschinellen Lernprozesses und der Trainingsumgebung zurückzuführen ist. Die zufällige Platzierung des Laufziels kann beispielsweise durch das platzieren von weiter entfernten Zielen die Schwierigkeit zufällig beeinflussen. Weiterhin kann auch die zufällig gewählte Geschwindigkeit und Startrotation die Schwierigkeit verändern. Generell sollten sich die Abweichungen über die vielen Trainingsepisoden relativieren, es ist jedoch nicht auszuschließen, dass das Training dadurch unterschiedlich verläuft.

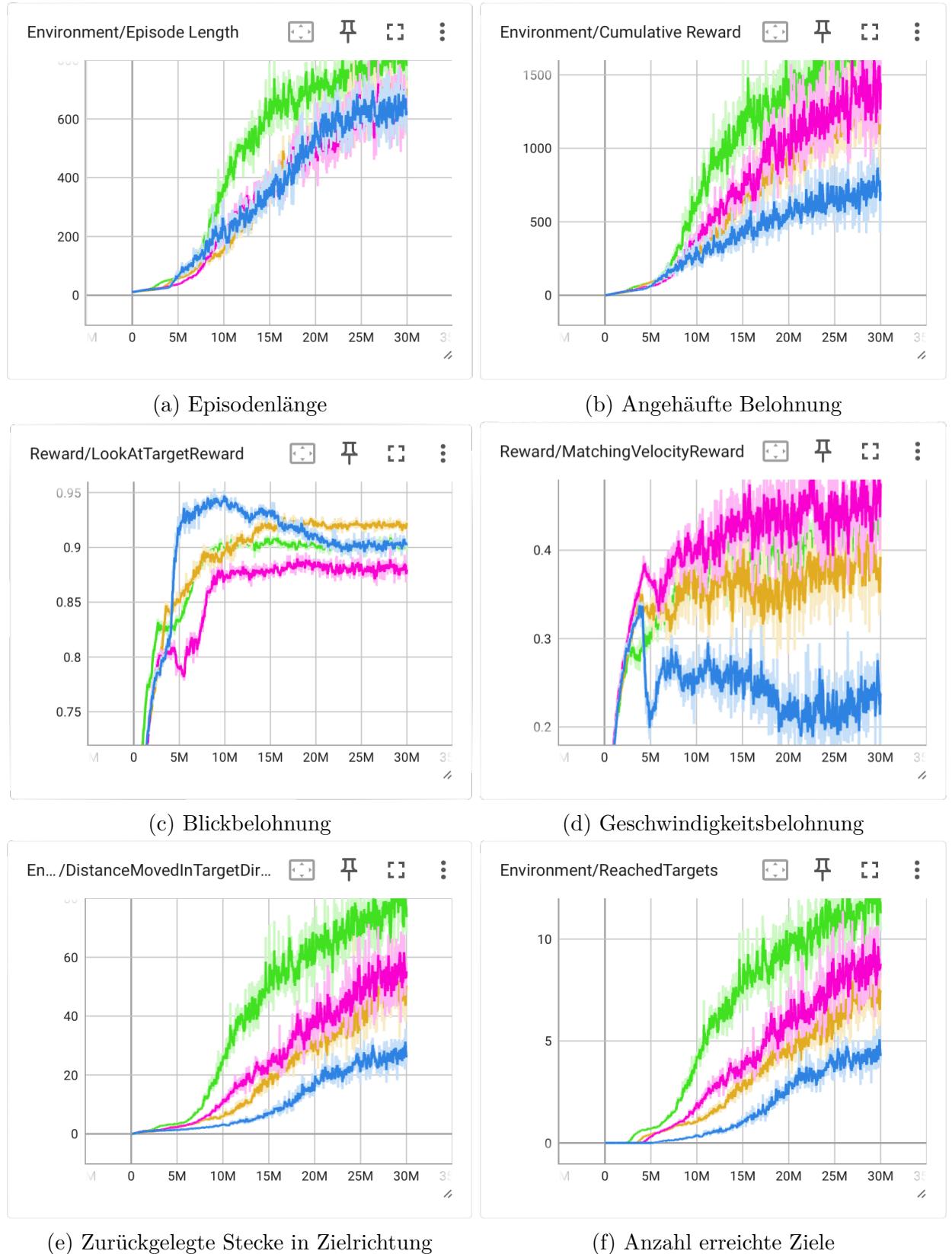


Abbildung 4.7.: Unterschiedliche Blickrichtungen Training Graphen (grün = vorwärts, orange = rückwärts, rosa = rechts, blau = links)

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, 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, dass die initiale Lernepisode zu lang war, wodurch das Verhalten bereits zu stark auf die spezifischen Gegebenheiten der ersten Lernphase angepasst war.

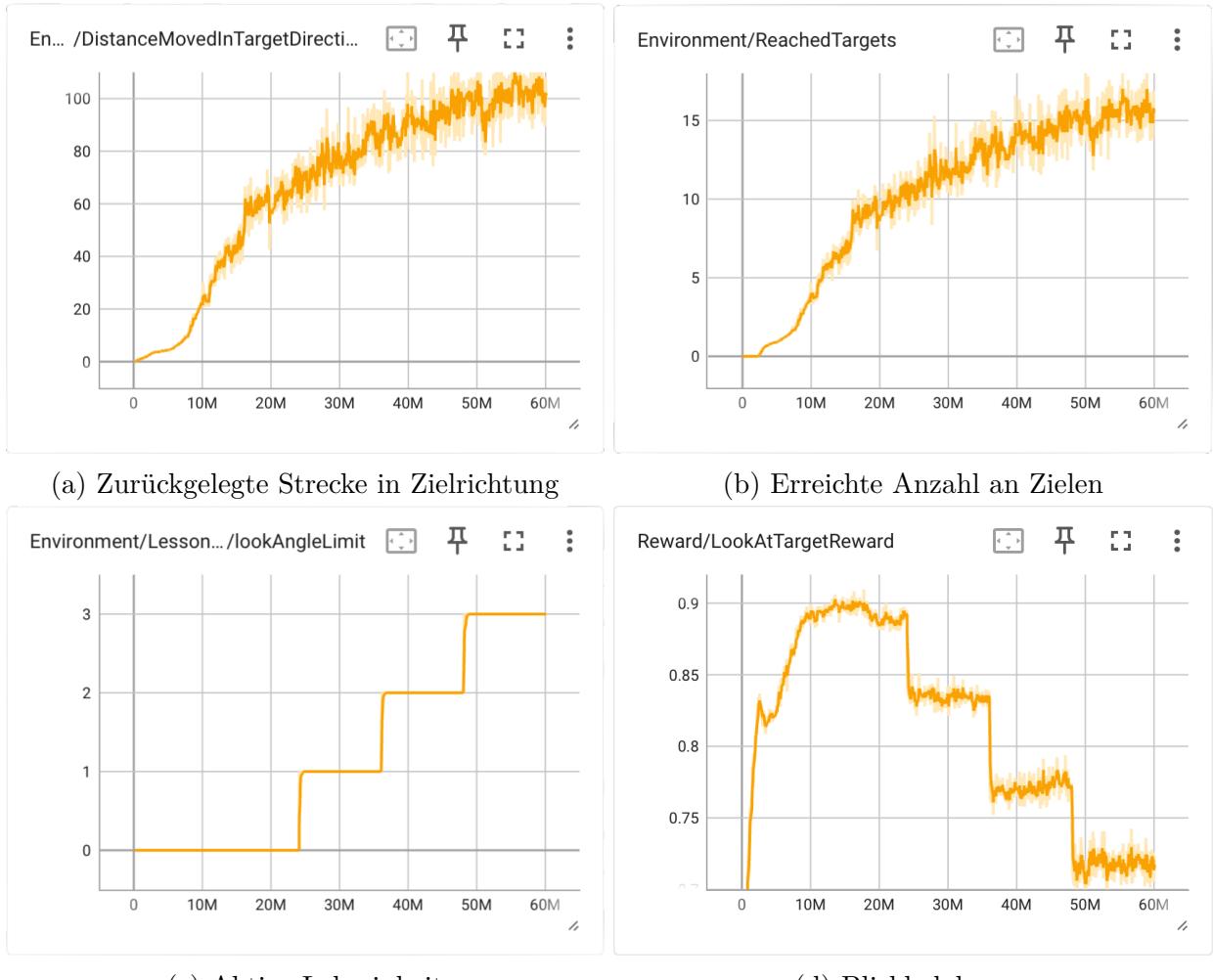


Abbildung 4.8.: Training Blickrichtungsziel mit Lehrplan

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 Ziele zu umgehen. Um dieses Problem zu adressieren, kann die Belohnungsfunktion angepasst oder zusätzliche Belohnungsfunktionen hinzugefügt werden.

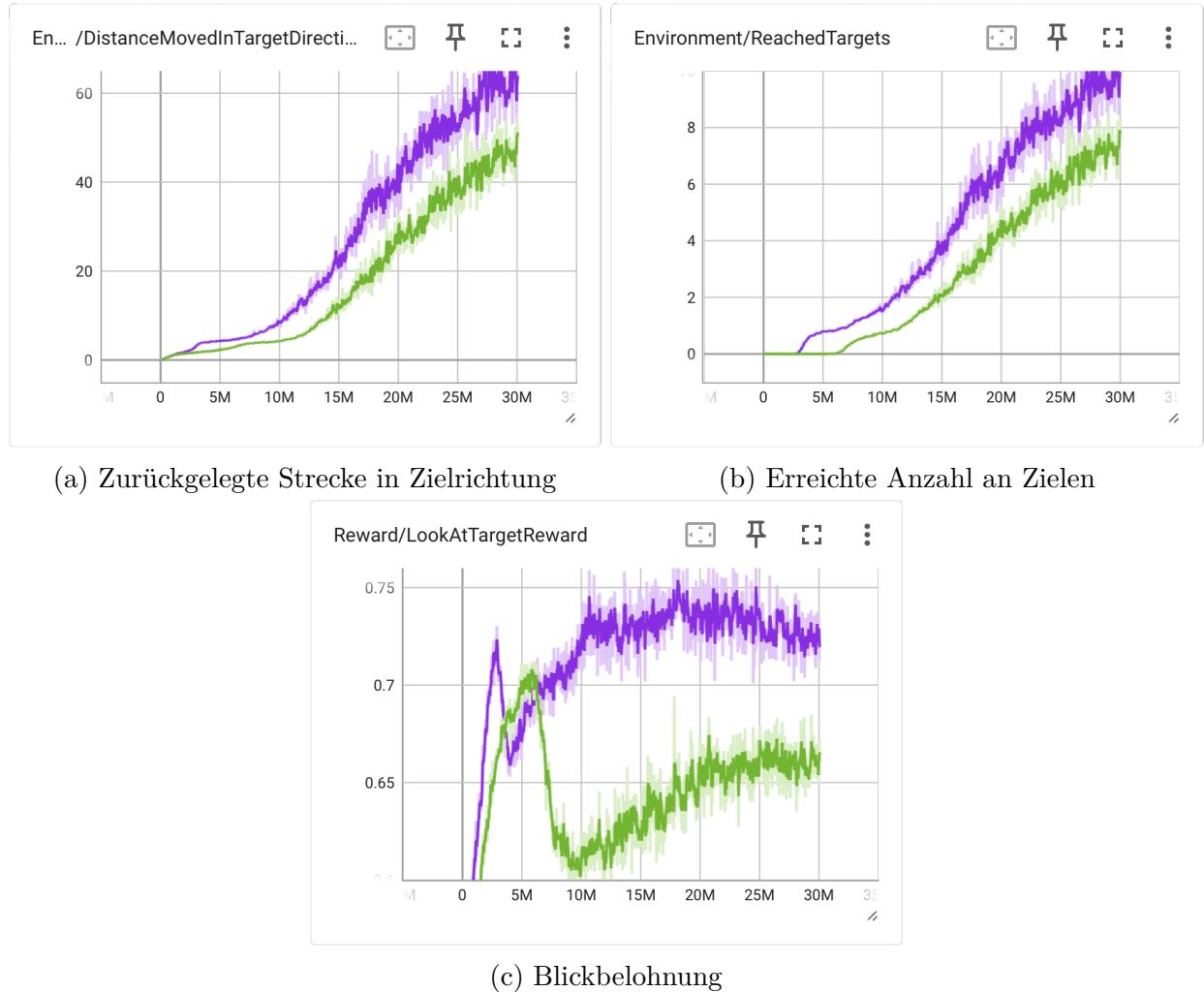


Abbildung 4.9.: Training Blickrichtungsziel (blau = Winkelabweichung ± 90 Grad, grün = Winkelabweichung ± 180 Grad)

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 Kopfniegsbestrafung 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.

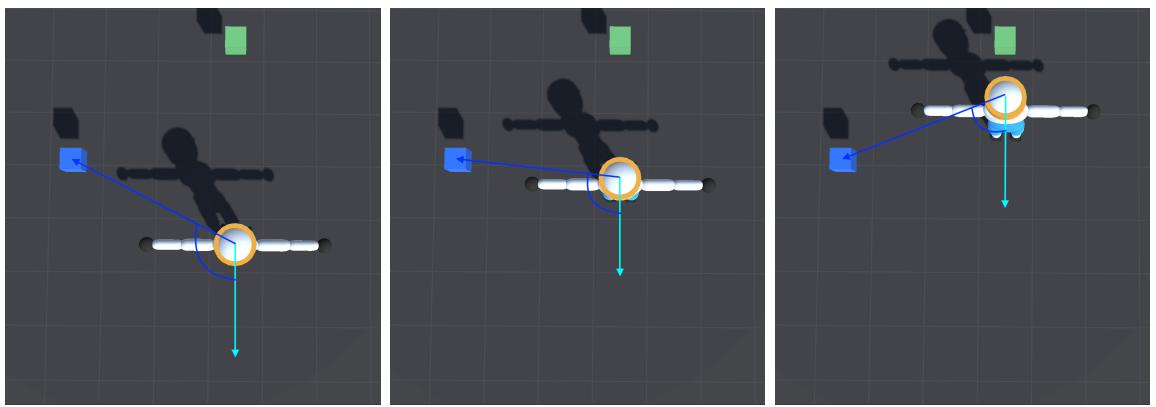


Abbildung 4.10.: Blickwinkel Änderung durch Zielannäherung

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 Abbildung 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 Zielvorgaben 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 an die neuen Ziele anpassen kann.



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 ist eine Technik, die verwendet wird, um eine kugelförmige Kollisionsabfrage durchzuführen. Zusätzlich zur Funktionsweise eines Raycast ermöglicht der Spherecast, über den Kugeldurchmesser die Blickfeldgröße 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.

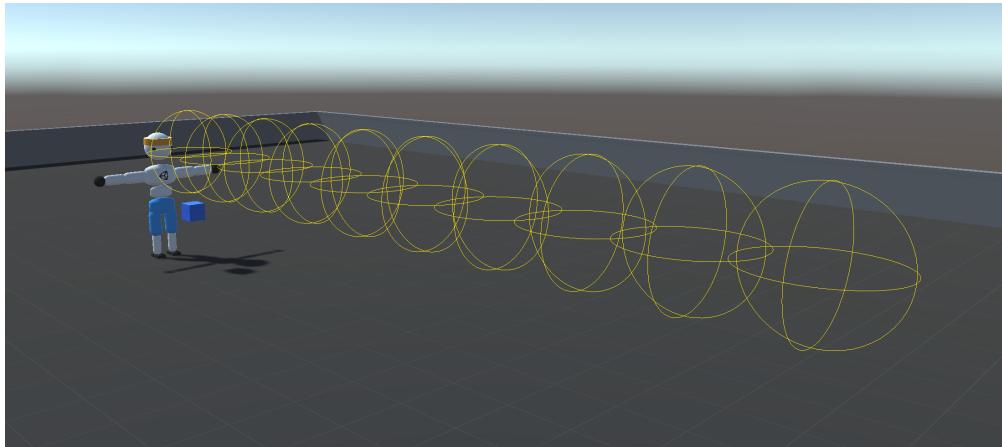


Abbildung 4.12.: Spherecast in Blickrichtung

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 ist nach 120 Millionen Trainingsschritten 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. Ein weiteres Problem, welches bei der Auswertung entdeckt wurde, ist, dass die Belohnung auch ohne erreichen der Blickziele optimiert werden kann. Es ist für den Agenten möglich, die Schwierigkeit des wechselnden Blickziels zu umgehen, indem er den Blickkontakt zum Ziel nie für 2 volle Sekunden hält. Dieses Problem muss in kommenden Versuchen berücksichtigt werden.ende ??

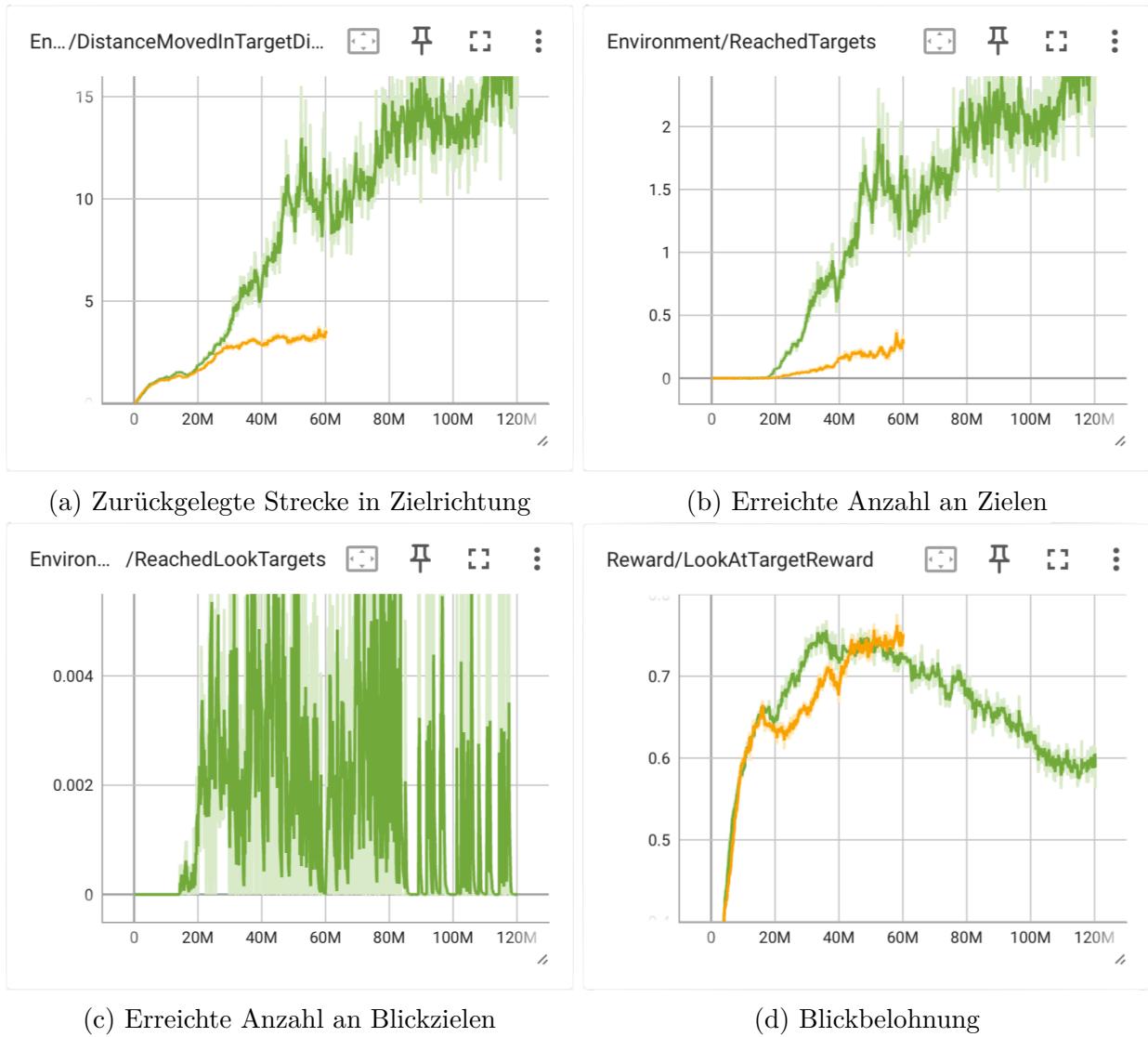


Abbildung 4.13.: Training Blickrichtungsziel mit Wechsel bei Blickkontakt von 2 bzw. 3 Sekunden (orange = 3 sek, grün = 2 sek)

4.3. Charakterkompatibilität und Konfiguration

Spiele verwenden die unterschiedlichsten Charaktere, nicht nur im Bezug auf das Aussehen, sondern auch hinsichtlich der Komplexität der Körperteile und der Anzahl der Knochen. Um den Charaktercontroller vielseitig einsetzbar zu gestalten, werden in diesem Kapitel die Walker-Demo-Komponenten angepasst, um den Einrichtungsprozess zu vereinfachen und unterschiedliche Charakterkörperstrukturen zu ermöglichen. Zunächst 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-Charaktermodells dargestellt. Zum Schluss wird der Mixamo-Charakter trainiert und das Ergebnis ausgewertet. Die Wahl des Mixamo-Modells als Beispiel soll zeigen, wie die Einrichtung mit den Vereinfachungen funktioniert und wie das Training bei der Verwendung von komplexeren Charaktermodellen beeinflusst wird.

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. Diese Änderungen zielen darauf ab, die Flexibilität und Benutzerfreundlichkeit des Walker-Demo-Agenten zu verbessern. Manuelle Konfiguration soll auf das nötigste reduziert werden, gleichzeitig soll aber auch eine Skalierbare Lösung für unterschiedliche Charakterkörperstrukturen implementiert werden.

4.3.2. Anpassungen

Der Aufbau der Walker-Demo-Skripte, zu sehen in Abbildung 4.14, verteilt die Aufgaben auf viele unterschiedliche Komponenten. Diese Aufteilung erschwert die Einrichtung und das Erweitern der Funktionalität erheblich. Die Aufgabenverteilung ist zudem nicht sinnvoll durchdacht. Ein Beispiel dafür ist die Zielsteuerung, die eigenständig funktioniert, während das Stabilisierungsobjekt vom Agenten aktualisiert werden muss.

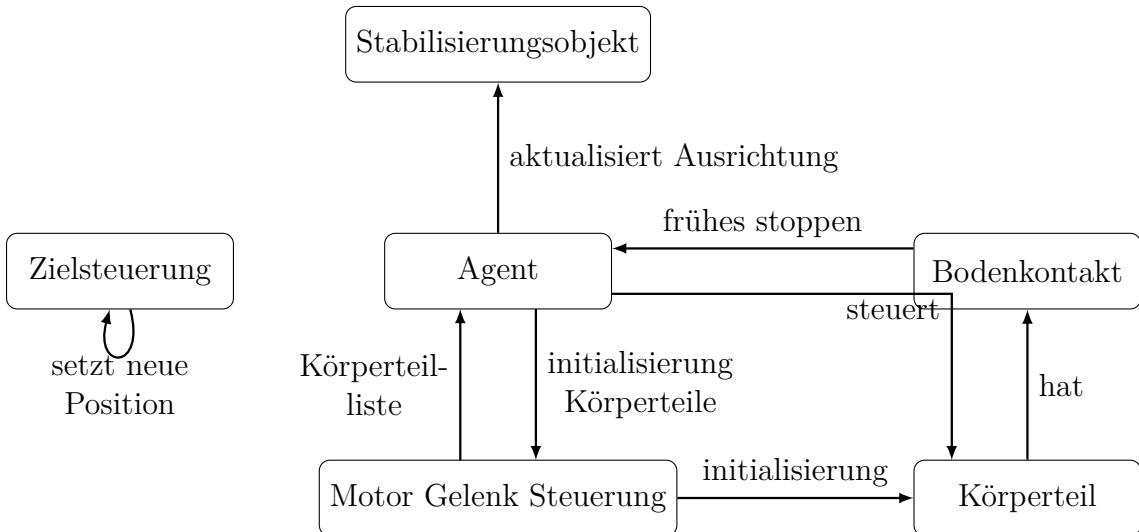


Abbildung 4.14.: Alte Architektur

Im neuen Ansatz (siehe Abbildung 4.15) wird die Steuerung und die Datenhaltung der Körperteile in eine Unitykomponente integriert. Zudem wird der Ablauf des Lernprozesses vollständig im Agenten implementiert. Externe Objekte wie die Zielsteuerung oder das Stabilisierungsobjekt werden, wo nötig, vom Agenten erstellt. Das Stabilisierungsobjekt benötigt, sobald es einmal initialisiert ist, keinen weiteren Input für die Aktualisierung der Ausrichtung. Aus diesem Grund wird die Aktualisierung fortan vom Stabilisierungsobjekt selbst ausgeführt. Um mehr Kontrolle über die Zielsetzung zu erlangen, wird die Zielsetzung

über ein Event vom Agenten ausgelöst. Die Funktionen der Bodenkontaktkomponente werden zuletzt ebenfalls in die Körperteilkomponente verschoben, um doppelte Konfigurationen von Referenzen zu reduzieren. Als Ergebnis dieser Umstrukturierung kann der Agent als Gehirn des Läufers interpretiert werden, während die Körperteile die Muskeln und Nerven darstellen. Diese neue Architektur bietet den Vorteil, dass die Körperteilkomponenten der Unterobjekte vom Agenten beim Programmstart als Liste abgefragt werden können, wodurch theoretisch jede beliebige Körperstruktur gesteuert werden kann.

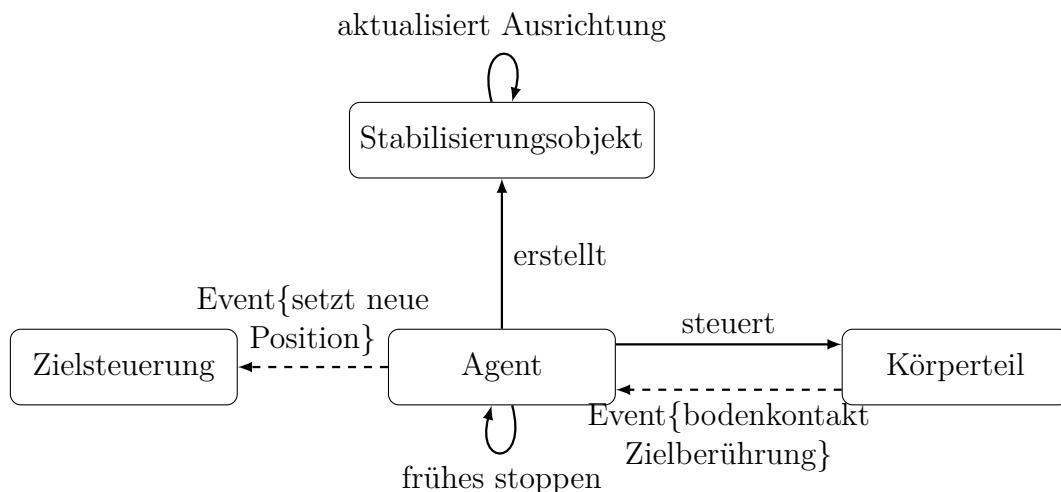


Abbildung 4.15.: Neue Architektur

Um die Steuerung von beliebigen Körperstrukturen zu ermöglichen, müssen zudem die Erstellung der Beobachtung sowie das Ausführen der Aktion je nach Körperstruktur dynamisch angepasst werden. Bei der Initialisierung der Körperteile wird geprüft, ob das Körperteil über eine Gelenkkomponente verfügt. Ist das der Fall, werden die Freiheitsgrade dieser Gelenke bestimmt. Die Freiheitsgrade geben anschließend an, welche Felder für das Körperteil in der Beobachtung hinzugefügt und in der Aktion ausgelesen werden müssen. Die Beobachtung wurde bereits im ursprünglichen Walker-Agent-Skript dynamisch für jedes Körperteil in einer Liste erstellt. Die Aktion wurde jedoch bisher statisch für jedes Körperteil ausgelesen. Im neuen Ansatz, wie in Codeausschnitt 4.6 dargestellt, wird die Aktion nun dynamisch ausgewertet, basierend auf den zuvor ermittelten Freiheitsgraden.

```

1 public override void OnActionReceived(ActionBuffers actionBuffers)
2 {
3     var continuousActions = actionBuffers.ContinuousActions;
4     int i = -1;
5
6     foreach (Bodypart bp in bodyparts)
7     {
8         //Körperteil mit steifem oder keinem Gelenk wird
9         //ignoriert
10        if (bp.dof.sqrMagnitude <= 0) continue;
11        float targetRotX = bp.dof.x == 1 ?
12            continuousActions[++i] : 0; // wenn Gelenk um
13            X-Achse beweglich ist wird Wert aus Aktion ausgelesen

```

```

11     float targetRotY = bp.dof.y == 1 ?
12         continuousActions[++i] : 0; // - Y-Achse -
13     float targetRotZ = bp.dof.z == 1 ?
14         continuousActions[++i] : 0; // - Z-Achse -
15     float jointStrength = continuousActions[++i];
16     bp.SetJointTargetRotation(targetRotX, targetRotY,
17                               targetRotZ); //Zielrotation setzen
18     bp.SetJointStrength(jointStrength); //Gelenkstärke
19     setzen
20 }
21 }
```

Listing 4.6: Agent Aktion in Bewegung umwandeln

4.3.3. Einrichtung

Der ausgewählte Charakter ist der Y Bot-Charakter, welcher in Abbildung 4.16 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. Die Entscheidung, das Skelett zu vereinfachen, ist ein gängiger Ansatz, um die Trainingszeit und -komplexität zu verringern. Durch die Zusammenfassung der Finger mit dem Handknochen wird die Anzahl der zu steuernden Gelenke reduziert, was wiederum die Anzahl der Optionen verringert, die der Agent im Verlauf des Trainings ausprobieren muss.

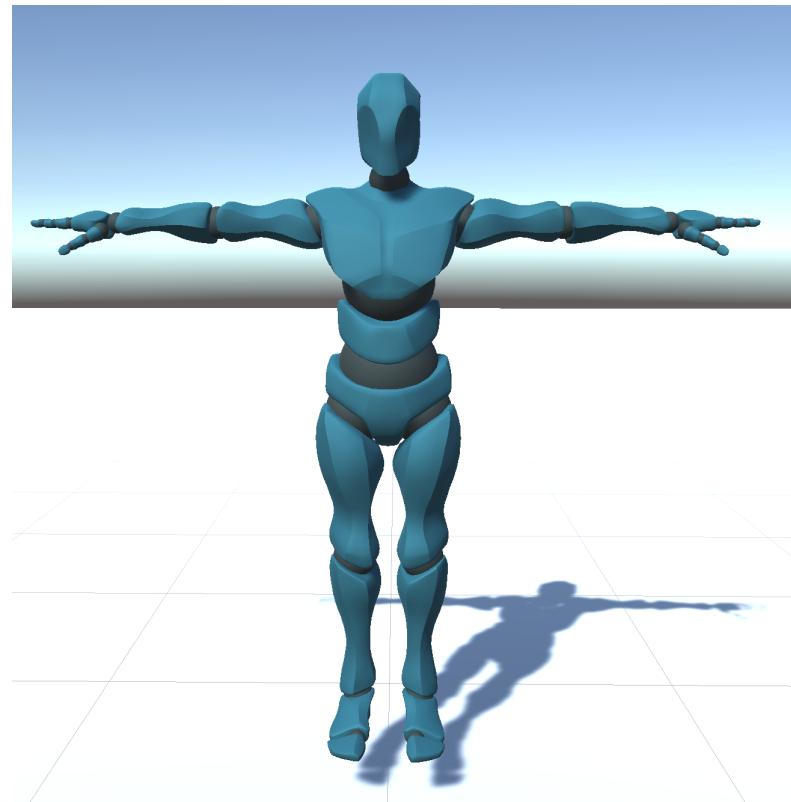


Abbildung 4.16.: Mixamo Charakter Y Bot

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. Slerp (Spherical Linear Interpolation) ist besonders nützlich in physikbasierten Simulationen, da es eine kontinuierliche und glatte Rotation zwischen zwei Orientierungen gewährleistet, was zu natürlicheren Bewegungen führt. 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 zwei Wirbeln anstatt einem 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. Diese Vereinfachungen tragen ebenfalls dazu bei die Komplexität möglichst nah der Walker Demo anzugeleichen. In Szenarien mit komplexeren Skills und Bewegungsabläufen kann die Vereinfachung zu Einschränkungen führen. Für das Erlernen einer Gehbewegung sind die Vereinfachungen des Oberkörpers jedoch ein kleiner Abstrich für die damit gewonnene Trainingsdauer.

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

Die Körperteilkomponente übernimmt die Konfigurationsparameter der Gelenkmotorsteuerung (siehe Abbildung 4.17). Parameter wie Stärke und maximale Rotationsgeschwindigkeit können angepasst werden. Die Standardwerte der Walker Demo bieten hier eine solide Grundlage für die meisten Anwendungsfälle. Bei Bedarf kann auch das Feld “Trigger Touching Ground“ aktiviert werden, um ein Event auszulösen, sobald das Körperteil den Boden berührt. Dieses Feature wird verwendet um, sobald ein Körperteil den Boden berührt, im Agenten ein frühes Stop auszulösen.

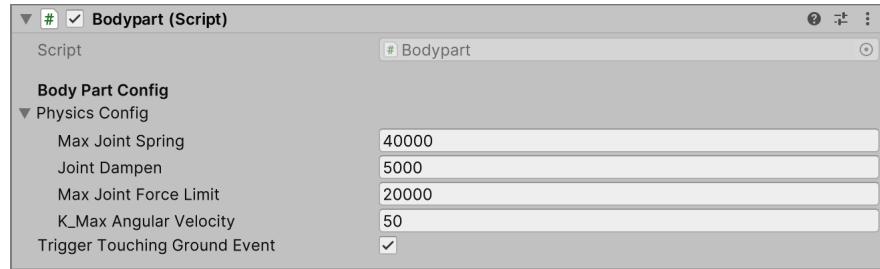


Abbildung 4.17.: Körperteilkomponente

Ist der Körper fertig konfiguriert, wird zuletzt das Walker Agent Skript (zu sehen in 4.18) zusammen mit den Behaviour-Parametern und dem Decision Requester hinzugefügt. Um die angepasste Walker Agent Komponente nutzen zu können, müssen noch die Referenzen für das Zielobjekt sowie das Hauptkörperteil und den Kopf gesetzt werden. Diese Referenzen werden für die Berechnung von relativen Positionen sowie die Belohnung genutzt. Soll das Ziel bei Berührung neu gesetzt werden, muss auch das Event “On Touched Target“ mit der Platzierungsfunktion der Zielsteuerung verknüpft werden. Zusätzlich können Parameter für das Training, wie Zielgeschwindigkeit und die Zufallsgenerierung der Startrotation, angepasst werden.

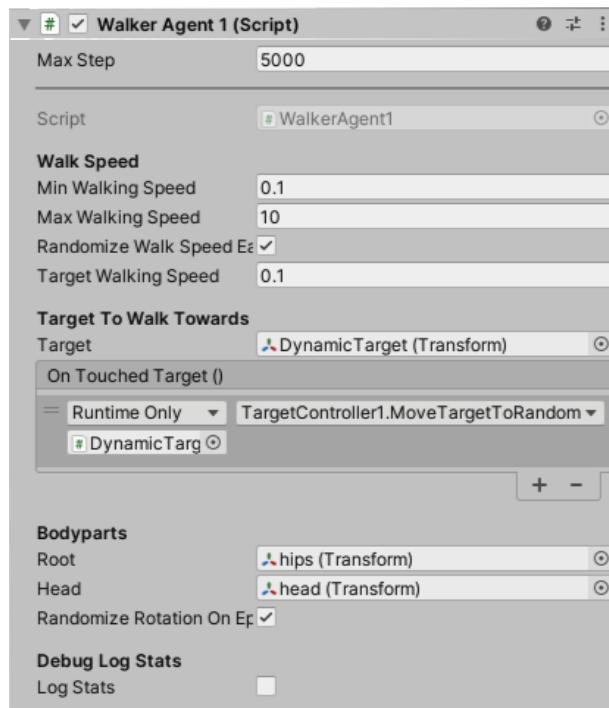


Abbildung 4.18.: Walker Agentkomponente

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 4.20). In der Abbildung 4.19 wird das erlernte Gangbild gezeigt. Der Läufer lernt in diesem Fall nicht das Laufen, sondern galoppiert zum Ziel.

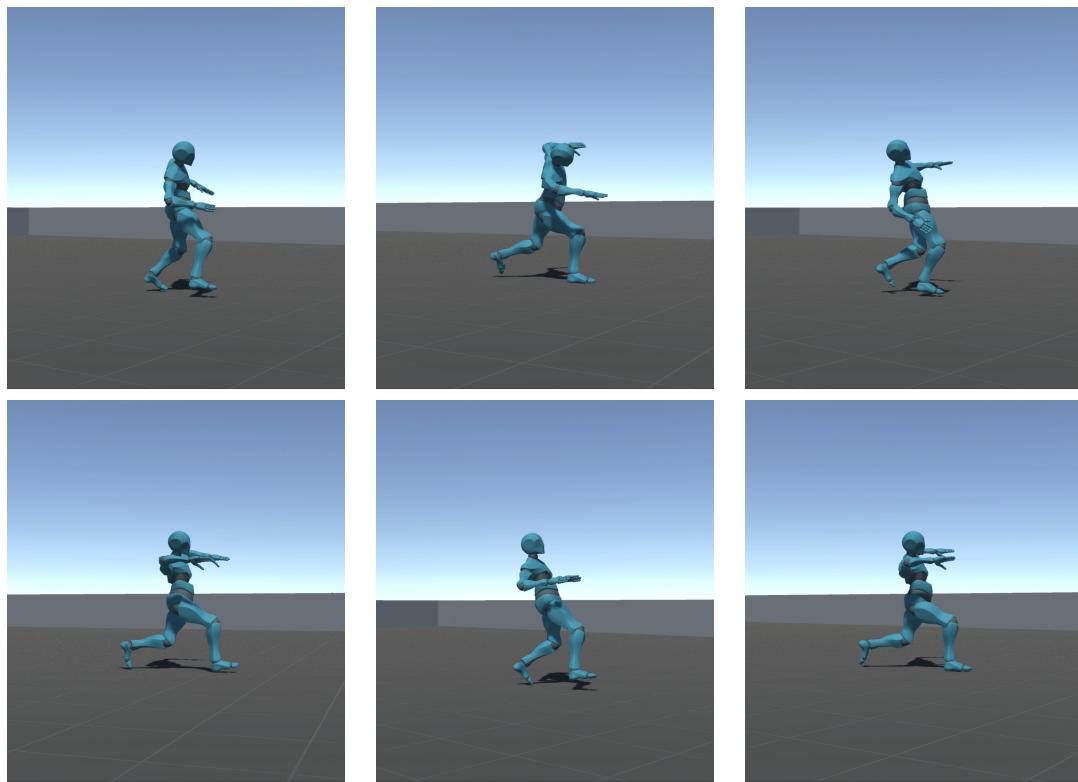


Abbildung 4.19.: Mixamo Charakter Gangbild mit Walker Demo Gelenklimits und Gewichten

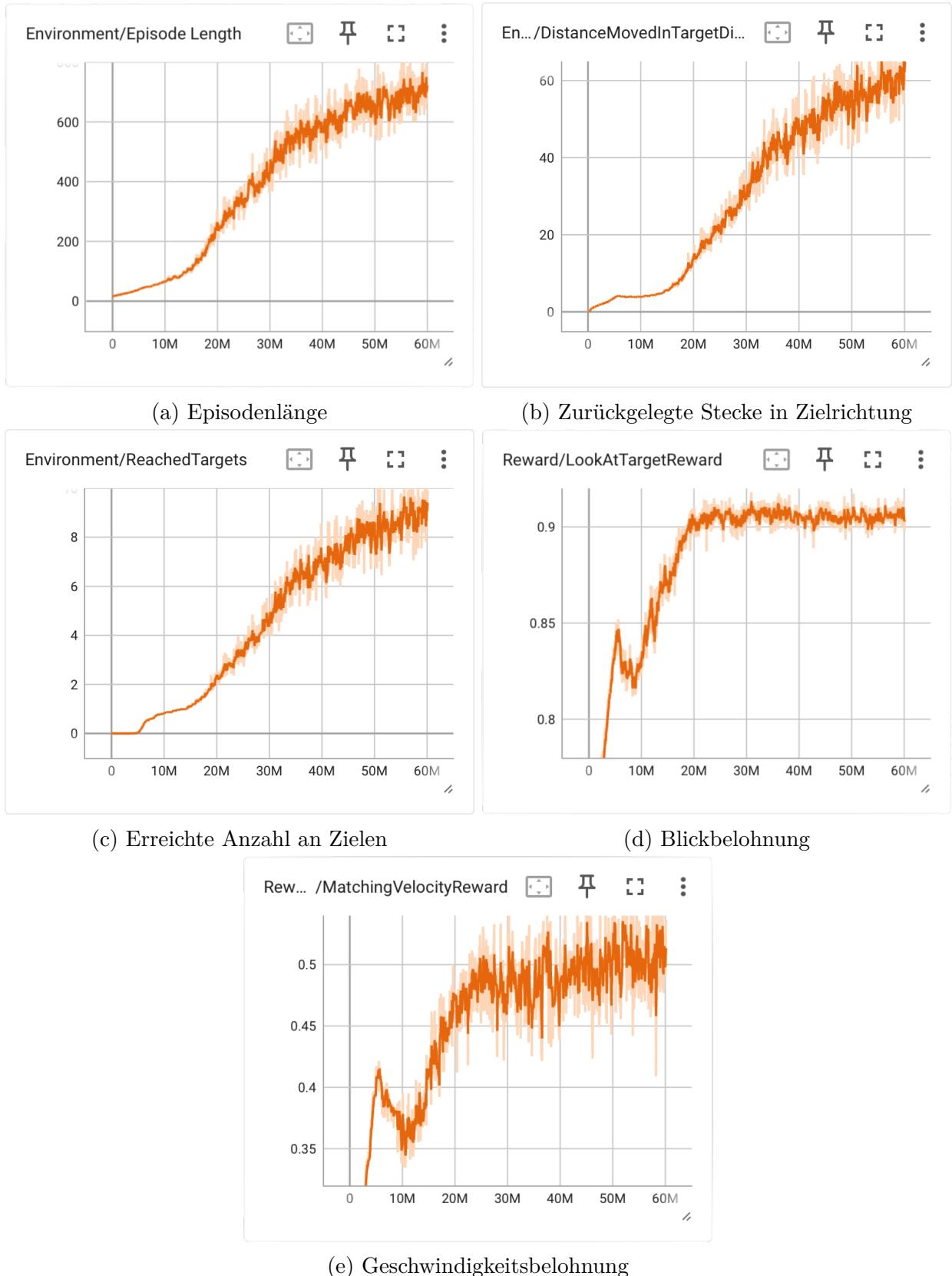


Abbildung 4.20.: Mixamo Charakter Training mit Walker Demo Gelenklimits und Gewichten

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. Diese Gangphasen umfassen verschiedene Stadien der Fortbewegung, wie das Abstoßen, das Schwingen des Beins und den Fersenauftritt, die zusammen ein fließendes und ausgewogenes Gehen ermöglichen. 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 ausreichend. Die gesteigerte Komplexität des 3D-Modells beim Mixamo Charakter führt bei der Wahrnehmung zu einem gesteigerten Realismus. Der Mixamo Charakter lernt zudem eher ein Galoppieren als das Laufen, welches der Walker Demo Läufer nach dem Training aufweist. Durch den gesteigerten Realismus und die Verschlechterung des Gangbilds wird im folgenden Kapitel untersucht, wie zusätzliche Belohnungen das erlernte Gangbild beeinflussen können. Dabei werden gezielt Belohnungen eingeführt, die darauf abzielen, das Gesamterscheinungsbild der Fortbewegung zu verbessern.

4.4.1. 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, der von 0 startet, sobald das vorderste Bein in Zielrichtung gewechselt wurde. Ausgehend von der verstrichenen Zeit seit dem letzten Wechsel erhält der Läufer dann eine Strafe. Wie in Abbildung 4.21 zu sehen ist, bleibt die Bestrafung bis 1,2 Sekunden bei 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. Initial wurde ein Schwellenwert für die Bestrafung von 2 Sekunden von einem ähnlichen Projekt aus dem Youtube Video “AI Learns to Walk (deep reinforcement learning)“ übernommen.^[8] Nach dem Training war jedoch deutlich zu sehen, dass eine Beinwechselperiode von 2 Sekunden zu lang ist. Durch eigene Experimente des Verfassers wurde eine durchschnittliche Beinwechselzeit von 1,2 Sekunden ermittelt. Für die Bestimmung dieses Wertes wurden Videoaufnahmen des Gehens erstellt, anhand derer anschließend die Dauer eines Beinwechsels festgelegt werden konnte.

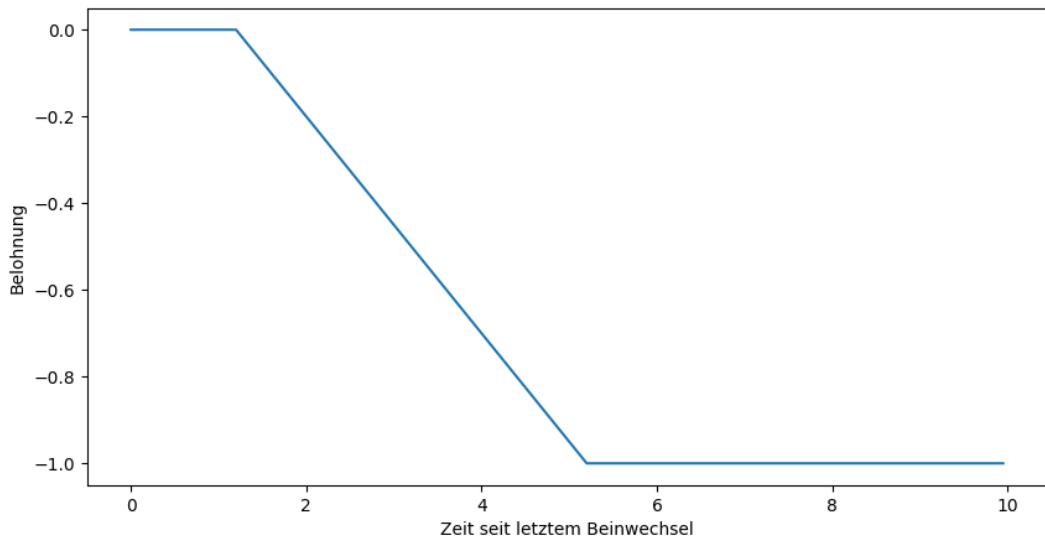


Abbildung 4.21.: Beinwechsel Belohnung

Das Einführen dieser Bestrafung hat beim Training erfolgreich dazu geführt, dass der Läufer in regelmäßigen Abständen das Standbein wechselt (siehe Abbildung 4.22). Diese Anpassung führte zu einer spürbaren Verbesserung der Bewegungsdynamik des Läufers, insbesondere in Bezug auf die Regelmäßigkeit der Schritte. Es wird das Verharren in einer Position verhindert, was das galoppieren bestraft, und so zu einer flüssigeren und natürlicheren Fortbewegung führt.

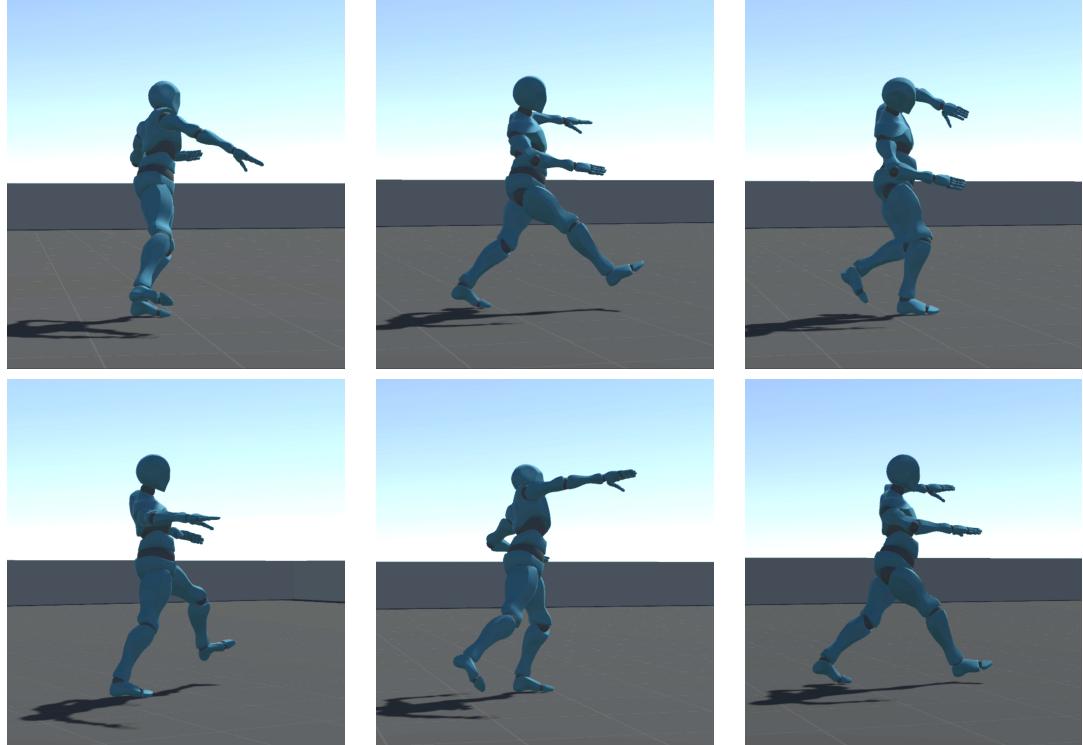


Abbildung 4.22.: Mixamo Charakter Gangbild mit Beinwechselbelohnung

4.4.2. Energieminimierung

Ein großer Einfluss auf die Entwicklung des menschlichen Gangbilds ist die Energieminimierung. Der Läufer hat keine Wahrnehmung von Aufwand, daher sind die erlernten Bewegungsabläufe oft alles andere als effizient. Um das Gangbild weiter zu verbessern, wird eine Belohnung eingeführt, die den Agenten dafür 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 Energieminimierung eine Funktion verwendet, bei der die Bestrafung zwischen 150 und 3150 Watt linear ansteigt (siehe Abbildung 4.23). Die Wahl der Energie-Schwellenwerte orientiert sich an den im Artikel “Assessing the metabolic cost of walking: The influence of baseline subtractions“ angegebenen durchschnittlichen Energieaufwänden.[9] Für eine Geschwindigkeit von 0,2 m/s bis 1,9 m/s wird ein Energieaufwand von 2,14 bis 6,90 Watt pro kg festgelegt. Mit einem Gewicht des Läufers von 109 kg ergibt sich ein Bereich von 233 bis 752 Watt bei einer vergleichbaren Effizienz. Dieser Bereich ist weit genug gefasst, sodass der Läufer Belohnungen über 0 erreichen kann. Ergebnisse mit steigender Effizienz führen zu nahezu vernachlässigbaren Bestrafungen. Die Funktion ergibt für 233 Watt eine Bestrafung von 0,027 und für 752 Watt eine Bestrafung von 0,2. Aufgrund der variablen Sensitivität der Geschwindigkeitsbelohnungsfunktion erreicht der Läufer selten eine solch hohe Geschwindigkeit, sodass die Bestrafung durch das Optimieren der Gangbewegung nahezu auf 0 reduziert werden kann.

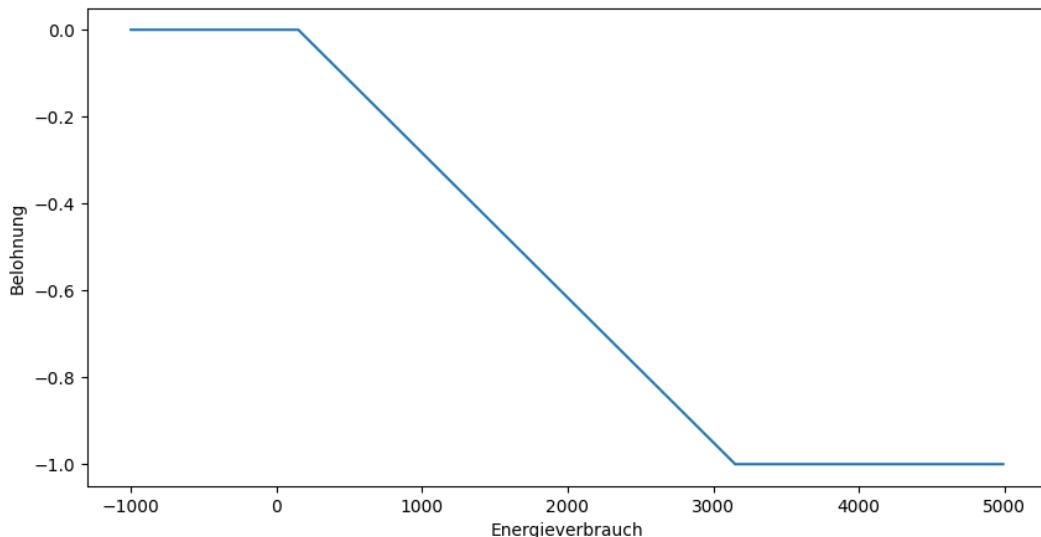


Abbildung 4.23.: Energiespar Belohnung

Die Energieminimierung bringt den Läufer dazu, seine Bewegungen zu optimieren und somit zu minimieren (siehe Abbildung 4.24). Das Gangbild wirkt dadurch natürlicher, insbesondere weil die Bein- und Körperbewegungen wesentlich verbessert wurden. Zuvor hat der Läufer beim Laufen den Oberkörper über das Standbein gekippt, nach der Einführung der Energiesparbelohnung wurde die Oberkörperbewegung minimiert. Allerdings hat die Energieminimierung auch dazu geführt, dass die Arme so gut wie gar nicht mehr bewegt werden. Das Fehlen der Armbewegung tritt im Gesamtbild der natürlichen Bewegung negativ auf.

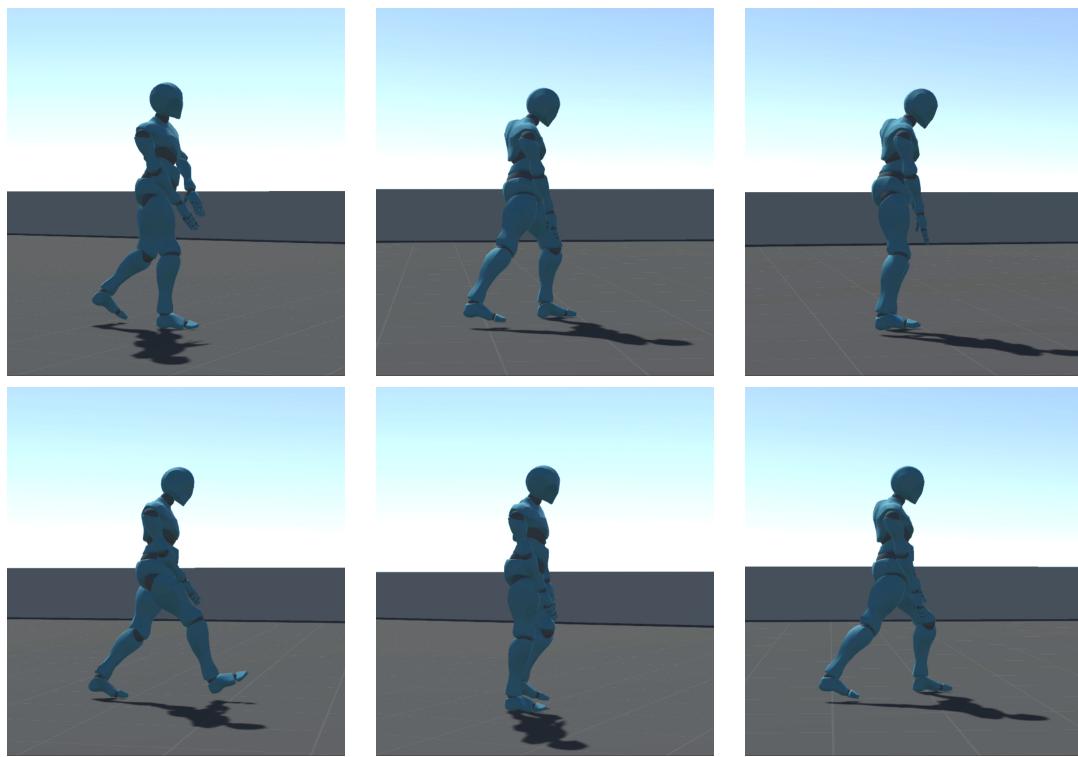
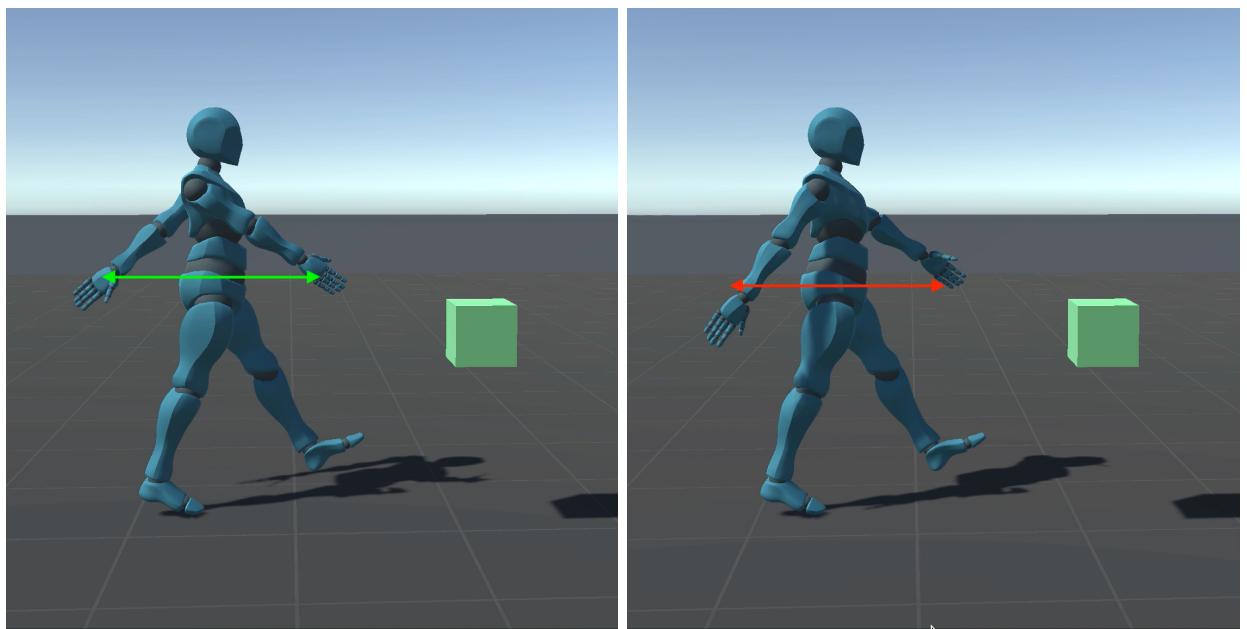


Abbildung 4.24.: Mixamo Charakter Gangbild mit Energiesparbelohnung

4.4.3. 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.[4]

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. Um die Arme während des Laufens gegensätzlich zu den Beinen zu schwingen, wird die Verwendung einer weiteren Bestrafung getestet. Für die Bestrafung wird die Distanz der Hände, sowie Hüfte zum Ziel berechnet. Abhängig von der führenden Seite (Bein vorne) werden die Werte so aufsummiert, dass bei korrekter Armorientierung ein positiver Wert und gleicherweise bei falscher Armorientierung ein negativer Wert herauskommt. Ist das linke Bein vorne, sieht die Berechnung wie folgt aus: $(\text{hüftdistanz} - \text{rechteHandDistanz}) + (\text{linkeHandDistanz} - \text{hüftDistanz})$ ist das rechte Bein vorne, werden die Hand Distanzen vertauscht und man erhält $(\text{hüftdistanz} - \text{linkeHandDistanz}) + (\text{rechteHandDistanz} - \text{hüftDistanz})$. Die errechnete Distanz wird in Abbildung 4.25 dargestellt.



(a) Richtige Armpendel Richtung (Distanz positiv)
 (b) Falsche Armpendel Richtung (Distanz negativ)

Abbildung 4.25.: Pendelsumme Distanz veranschaulichung

Anschließend wird die Distanz mit einer Clip ähnlichen Funktion (siehe Abbildung 4.26) auf einen Bereich von -1 bis 1 beschränkt, abschließend werden die Werte auf einen Bereich von -1 bis 0 skaliert.

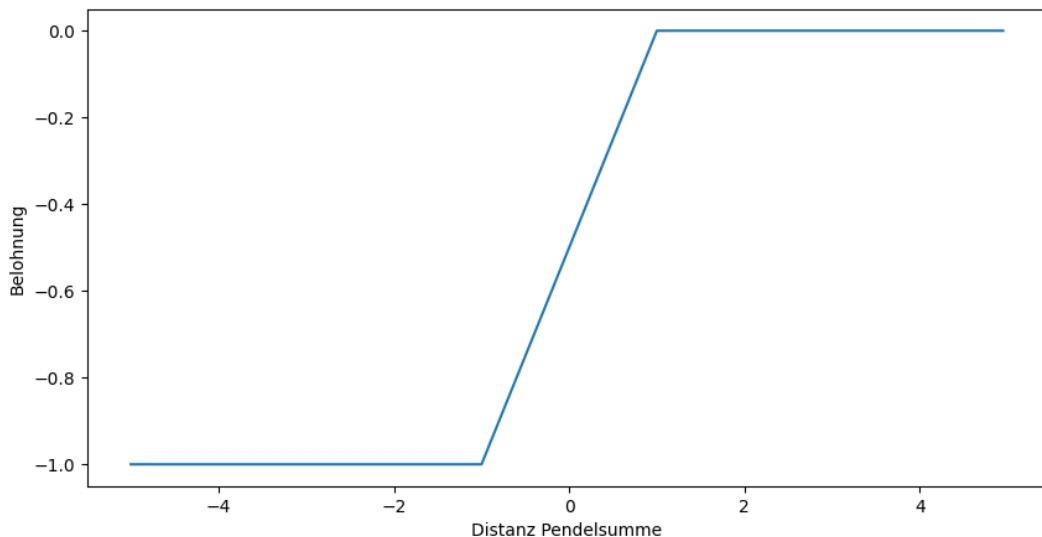


Abbildung 4.26.: Armpendel Bestrafung

Die neu eingeführte Armpendel Bestrafung hat nicht die erwarteten Ergebnisse erzielt. Zu Beginn des Trainings dominiert die Bestrafung so stark mit -0,48 Punkten pro Trainingsschritt. Da der Läufer sich entscheidet nur diese Bestrafung zu optimieren und gleichzeitig so schnell wie möglich das frühe Stop auszulösen, um zusätzliche Bestrafung zu verhindern. Um zu evaluieren ob die entwickelte Belohnungsfunktion das gewollte Verhalten verstärkt, muss hier ein Training mit weniger starken Bestrafungen durchgeführt werden.

4.4.4. Imitationslernen

Imitationslernen ist eine weit verbreitete Technik, um Agenten im verstärkenden Lernen ein bestehendes Verhalten anhand von Aufzeichnungen anzutrainieren. Ein Vorteil des Imitationslernens besteht darin, dass natürliche Verhaltensmuster aus der Referenz übernommen werden und somit weniger spezialisierte Belohnungen erforderlich sind. Als Referenz für die Steuerung menschlicher Charaktere werden Bewegungsaufnahmen von Menschen verwendet. Unity ML-Agents bietet für das Imitationslernen zwei Algorithmen an: Behavioral Cloning und Generative Adversarial Imitation Learning. Beide verwenden als Referenz eine aufgenommene Demonstration. Zum Aufnehmen einer Demonstration existiert die Demonstration Recorder Komponente, die auf ein Objekt mit Agent hinzugefügt wird. Anschließend kann das heuristische Verhalten des Agenten aufgezeichnet werden. Im Fall der Läufer-Trainingsumgebung ist es jedoch schwierig, ein solches Verhalten zu entwickeln, da zusätzlich zu den Gelenkrotationen, die aus Bewegungsaufnahmen entnommen werden können, auch die Gelenkstärke benötigt wird. Aufgrund dieser Anforderungen sind die von Unity ML-Agents verfügbaren Systeme für das Imitationslernen in diesem Fall nicht geeignet. Daher wurde das Imitationslernen basierend auf den in der Arbeit “DeepMimic: Example-Guided Deep Reinforcement Learning of Physics-Based Character Skills“ entwickelten Imitationsbelohnungsfunktionen getestet.^[6] Als Referenzbewegung werden Motion Capture Daten aus der “SFU Motion Capture Database“ genutzt.^[2]

Zu Beginn einer Trainingsepisode wurde die Referenzanimation zurückgesetzt und der Läufer auf die gleiche Startpose gesetzt. Der Agent erhält zusätzlich zu seiner normalen Beobachtung auch eine Phasenvariable, die die aktuelle Phase der Animation widerspiegelt. Es wurde untersucht, wie sich die Kombination der Körperhaltungsbelohnung (Posereward) aus dem Deep Mimic Artikel mit den Walker Demo Belohnungen auf das Verhalten auswirkt. Die Körperhaltungsbelohnung belohnt die Übereinstimmung der relativen Körperteilrotationen und ist somit positionsunabhängig. Da jedoch die Hüfte bzw. das Hauptkörperteil eines Charakters keine Referenz hat, zu der diese Rotation relativ bestimmt werden kann, wurde eine Ausnahmeregel für die Hüfte entwickelt, die nur die Rotation in zwei Richtungen vergleicht: die X- und Z-Rotation. Die Rotation um die Y-Achse wird ignoriert, um unterschiedliche Orientierungen des Körpers und damit unterschiedliche Zielrichtungen zuzulassen, ohne diese zu bestrafen. Der Läufer entwickelte ein rückwärts übergebeugtes Gehverhalten (siehe Abbildung 4.27), es ist unklar, ob der Läufer die Referenzbewegung teilweise imitiert hat. Der Auslöser für das merkwürdige Gangbild ist ebenfalls nicht eindeutig. Eine mögliche Ursache könnte ein Fehler im Vergleich der Hüftrotation sein.

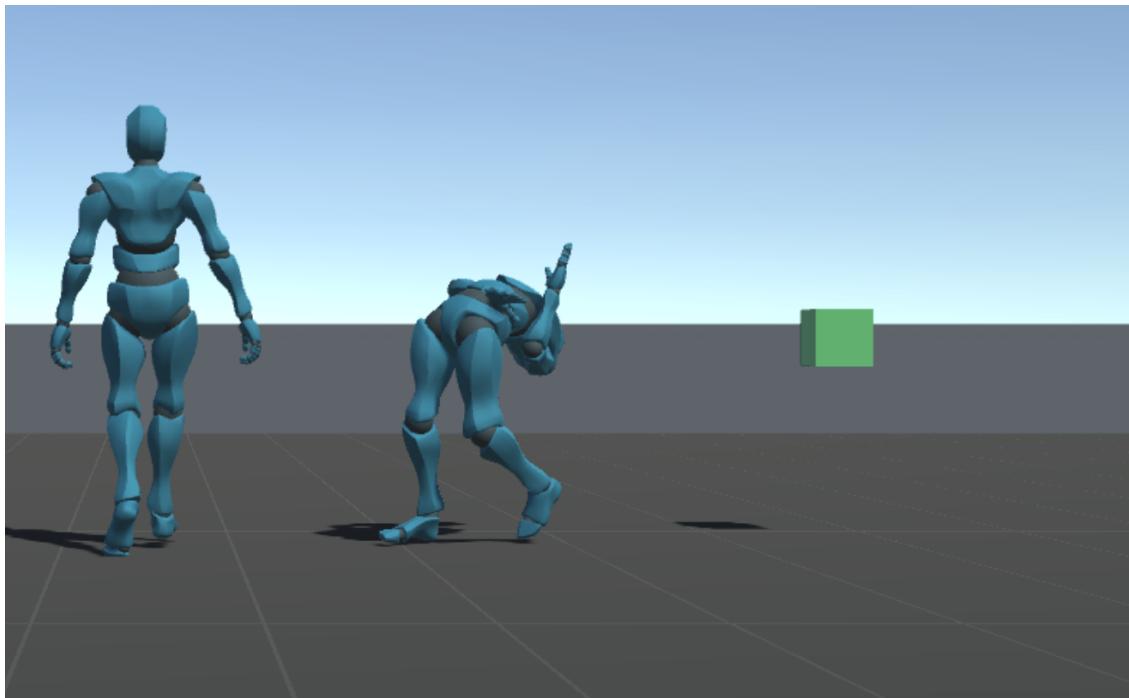


Abbildung 4.27.: Mixamo Charakter Gangbild mit Körperhaltungbelohnung

5. Zusammenfassung und Ausblick

Die Arbeit war teilweise erfolgreich darin einer Charaktersteuerung mit dem Unity ML-Agents Paket zu entwickeln und trainieren. Es konnten Anpassungen aufgezeigt werden, um das Modell an die spezifischen Anforderungen eines kontrollierbaren Spielcharakters anzupassen. Auch wenn diese im limitierten Zeitspektrum dieser Arbeit nicht kombiniert werden konnten, wurde klar ersichtlich, dass die Bewegungsabläufe für den Läufer erlernbar waren. Im weiteren konnte die Kompatibilität für unterschiedliche Charaktermodelle durch die Anpassungen erreicht werden. Das optische Erscheinungsbild der Gangbewegung konnte außerdem auch auf ein neues Level gebracht werden. Als größtes Problem zeigte sich jedoch das Antrainieren mehrerer Bewegungsabläufe (Skills) in einem einzigen Modell. Im Artikel “DeepMimic: Example-Guided Deep Reinforcement Learning of Physics-Based Character Skills” wird hierfür ein Skill Selector Modell trainiert, das als Input die Auswahl des aktuell geforderten Skills erhält und entscheidet, welcher Skill aktiviert werden soll. Der Skill Selector kann sogar mehrere Skills gleichzeitig aktivieren, wodurch fließende Übergänge zwischen den Bewegungen möglich sind.^[6] Dieses Verfahren erfordert jedoch tiefgreifende Anpassungen an den grundlegenden Konzepten der Lernalgorithmen und war daher in dieser Arbeit nicht realisierbar.

Aus diesen Ergebnissen lässt sich ableiten, dass mit dem Unity ML-Agents Paket durchaus charaktersteuernde Modelle trainiert werden können, die zusätzlich über Nutzerinput steuerbar sind. Somit erscheint der Ansatz, traditionelle Animationssysteme durch ein trainiertes Modell zu ersetzen, durchaus realistisch. Die weitere Entwicklung der thematisierten Anpassungen sowie umfassendere Tests zur Stabilität sind jedoch notwendig. Beispielsweise kann der Läufer aktuell keine äußeren Kraftimpulse kompensieren, was die Fähigkeit zur Erholung von Stürzen und das Anpassen an Unebenheiten im Untergrund zu wesentlichen Themen für zukünftige Arbeiten macht. Aus diesen Ergebnissen und Impulsen für zukünftige Implementierungen lässt sich ableiten, dass mit dem aktuellen Stand der Entwicklung höchstens sehr vereinfachte Spieletitel ein solches System in Erwägung ziehen sollten. Andere Arbeiten in diesem Bereich, wie die von Nvidia und Ubisoft entwickelten Systeme, zeigen jedoch, dass ähnliche Lösungen in anderen Entwicklungsumgebungen mit umfassenderen Ressourcen bereits erfolgreich umgesetzt wurden.^{[5][1]} Insbesondere Ubisoft nutzte denselben Lernalgorithmus für die Entwicklung ihrer Charaktersteuerung.

Zukünftig wird die Verwendung von maschinellem Lernen im Bereich der Charakteranimation sicher Realität werden. Abgesehen von kompetitiven Spielen, bei denen ein gleichbleibendes, vorhersehbares Verhalten erwünscht ist, könnte die Nutzung von physikalisch simulierten Charakterkontrollern durch maschinelles Lernen den Aufwand für das Erstellen natürlicher Bewegungsabläufe drastisch reduzieren. Zudem ermöglicht die physikalische Simulation eine Interaktion mit der Umgebung, die unter kinematischen Animationssystemen bisher nicht möglich war.

Literaturverzeichnis

- [1] Kevin Bergamin u. a. “DReCon: data-driven responsive control of physics-based characters”. In: *ACM Trans. Graph.* 38.6 (2019). ISSN: 0730-0301. DOI: [10.1145/3355089.3356536](https://doi.org/10.1145/3355089.3356536). URL: <https://doi.org/10.1145/3355089.3356536>.
- [2] Karunanidhi Durai Kumar Huang Geng Sai Charan Mahadevan Goh Jing Ying Kang-Kang Yin. *SFU Motion Capture Database*). URL: <https://mocap.cs.sfu.ca>.
- [3] 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>.
- [4] 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.
- [5] Xue Bin Peng u. a. “ASE: Large-scale Reusable Adversarial Skill Embeddings for Physically Simulated Characters”. In: *ACM Trans. Graph.* 41.4 (Juli 2022).
- [6] 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.
- [7] Richard S Sutton und Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.
- [8] AI Warehouse. *AI Learns to Walk (deep reinforcement learning)*. 2023. URL: https://www.youtube.com/watch?v=L_4BPjLBF4E.
- [9] Peter G. Weyand, Bethany R. Smith und Rosalind F. Sandell. “Assessing the metabolic cost of walking: The influence of baseline subtractions”. In: *2009 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*. 2009, S. 6878–6881. DOI: [10.1109/IEMBS.2009.5333126](https://doi.org/10.1109/IEMBS.2009.5333126).

A. Training Dokumentation

ID	Beschreibung	Ergebnis
1	Walker Demo als Vergleich für kommende Trainings	
2	Walker Demo mit angepasster halbiert maximal Geschwindigkeit	lernt schneller ein gutes Verhalten
3	Walker Demo mit angepassten Hyperparametern (batch und buffer size) um die Werte an die erhöhte Anzahl an parallelen Trainingsumgebungen anzupassen	lernt wesentlich langsamer
4	Walker Demo mit angepassten Hyperparametern num_epoch um mehrere Lernschritte auf die erfassten Daten auszuführen	lernt wesentlich langsamer
5	Walker Demo mit Laufrichtung als Goal Observation für Hyper Network training	für komplexe Probleme wie das kontinuierliche Kontrollproblem eines zweibeinigen Läufers viel zu langsam
6	Imitationslernen mit Mixamo Charakter und "kneeing soccer ball" Animation	lernt Teile der Bewegung aber hebt Bein nicht
7	Imitationslernen mit Mixamo Charakter und "backflip" Animation und zufälliger Initialisierung mit Pose von Animation	lernt nicht abzuspringen
8	Imitationslernen mit Mixamo Charakter und "laufen" Animation und zufälliger Initialisierung mit Pose von Animation	lernt nicht zu laufen
9	Walker Demo mit Mixamo Charakter, realistischen Gelenklimits und Körperteil Gewichten	lernt sich auf das Ziel zu zu bewegen aber fällt nach kurzer Zeit
10	Walker Demo mit Mixamo Charakter, realistischen Gelenklimits, Körperteil Gewichten und 1 extra Hidden Layer	lernt langsamer und auch mit mehreren Trainingsschritten kein besseres Ergebnis
11	Walker Demo mit Mixamo Charakter, realistischen Gelenklimits, Körperteil Gewichten und doppelte Anzahl an Knoten in jedem Layer	lernt langsamer und auch mit mehreren Trainingsschritten kein besseres Ergebnis
12	Walker Demo mit umstrukturiertem Code	lernt langsamer
13	Walker Demo mit Mixamo Charakter, gleiche Gelenklimits und Gewichte wie Demo	lernt trotzdem viel langsamer
14	Walker Demo mit umstrukturiertem Code aber Ziel wird wieder nur beim erreichen neu gesetzt	lernt in gleicher Zeit wie Demo vergleichbares Verhalten

15	Walker Demo mit Gelenkschon Belohnung (Gelenke gleichmäßig belasten)	Belohnung mit 3 multiplizierten Komponenten zu komplex
16	Walker Demo mit Gelenkschon Belohnung (Gelenke gleichmäßig belasten) und Belohnungen mit Gewichtung addiert	Läufer nutzt Blickbelohnung aus
17	Walker Demo auf der Stelle stehen mit Geschwindigkeit 0,0001	lernt nicht
18	Walker Demo auf der Stelle stehen mit neuer Geschwindigkeitsbelohnung	lernt sehr schnell gutes Verhalten
19	Walker Demo laufen mit neuer Geschwindigkeitsbelohnung	lernt sehr langsam
20	Walker Demo laufen mit Limit für Geschwindigkeitbelohnung Sensitivität	lernt schnell gutes Verhalten
21	Walker Demo rückwärts laufen	lernt gutes Verhalten
22	Walker Demo seitwärts (rechts) laufen	lernt gutes Verhalten
23	Walker Demo seitwärts (links) laufen	lernt etwas schlechteres Verhalten
24	Walker Demo mit Blickziel und Winkelabweichung Lernplan	ignoriert wechselndes Blickziel
25	Walker Demo mit Blickziel und Winkelabweichung fest auf 90 Grad	passt Blickrichtung leicht an
26	Walker Demo mit Blickziel und Winkelabweichung fest auf 180 Grad	schaut zu Boden
27	Walker Demo mit Mixamo Charakter Fehler mit Fuß-kollider behoben	lernt sehr gut aber galoppiert anstatt zu laufen
28	Walker Demo mit Mixamo Charakter mit Belohnung für Fußwechsel alles 2 Sekunden	galoppiert immer noch
29	Walker Demo mit Mixamo Charakter mit Belohnung für Fußwechsel alles 1,2 Sekunden	lernt zu laufen und erreicht stabil mehrere Ziele
30	Walker Demo mit Mixamo Charakter mit Belohnung Energieminimierung	Gangbild verbessert aber Arme ohne Bewegung
31	Walker Demo mit Blickziel, Winkelabweichung fest auf 180 Grad und Bestrafung für das neigen des Kopfs	läuft nur rückwärts
32	Walker Demo mit Blickziel, Winkelabweichung jedes Update gesetzt	läuft nur seitwärts
33	Walker Demo, Blickziel jedes Update gesetzt, Winkelabweichung nur bei erreichen von Ziel und durchschnittlicher Belohnung größer 0,7 neu gesetzt	läuft nur seitwärts
34	Walker Demo, Blickziel jedes Update gesetzt, Winkelabweichung nur bei erreichen von Ziel und durchschnittlicher Belohnung größer 0,7 neu gesetzt, extremerer Anstieg der Blickbelohnung	schaut zuverlässig zum Blickziel aber fällt sehr schnell

35	Walker Demo, Blickziel jedes Update gesetzt, Winkelabweichung bei Blickkontakt größer 3 Sekunden	schaut zuverlässig zum Blickziel aber fällt sehr schnell
36	Walker Demo, Blickziel jedes Update gesetzt, Winkelabweichung bei Blickkontakt größer 2 Sekunden	schaut zuverlässig zum Blickziel aber fällt auch schnell
37	Walker Demo, Blickziel jedes Update gesetzt, Winkelabweichung bei Blickkontakt größer 2 Sekunden, Beta höher (0,01) für mehr ausprobieren während dem Training	Entropy zu hoch Verhalten wird immer zufälliger anstatt sich zu festigen
38	Walker Demo, Blickziel jedes Update gesetzt, Winkelabweichung bei Blickkontakt größer 2 Sekunden, Beta höher (0,0075) für mehr ausprobieren während dem Training	ähnliche Leistung wie vor der Änderung

Tabelle A.1.: Training Documentation

B. Codeausschnitte

B.1. Angepasstes Körperteil Skript

```
1  using System.Collections.Generic;
2  using UnityEngine;
3  using UnityEngine.Events;
4
5  [System.Serializable]
6  public class PhysicsConfig
7  {
8      public float maxJointSpring = 40000f;
9      public float jointDampen = 5000f;
10     public float maxJointForceLimit = 20000f;
11     public float k_MaxAngularVelocity = 50f;
12 }
13
14 /*
15 * Bodypart
16 * implementes functions to control the joints and holds
17 * information about bodypart
18 */
19 [RequireComponent(typeof(Rigidbody))]
20 public class Bodypart : MonoBehaviour
21 {
22     [Header("Body Part Config")]
23     public PhysicsConfig physicsConfig;
24     public bool triggerTouchingGroundEvent;
25
26     [HideInInspector] public UnityEvent onTouchedGround;
27     [HideInInspector] public bool touchingGround;
28
29     [HideInInspector] public UnityEvent onTouchedTarget;
30
31     //component references
32     [HideInInspector] public Rigidbody rb;
33     [HideInInspector] public ConfigurableJoint joint;
34
35     //starting pos and rot
36     [HideInInspector] public Vector3 startingPos;
37     [HideInInspector] public Quaternion startingRot;
38
39     //bodypart information
40     [HideInInspector] public Vector3 dof; //degrees of freedom
41     [HideInInspector] public float currentStrength;
42     [HideInInspector] public float power;
```

```
43     void Awake()
44     {
45         //get component references
46         rb = GetComponent<Rigidbody>();
47         if (TryGetComponent(out ConfigurableJoint foundJoint))
48         {
49             joint = foundJoint;
50         }
51
52         //save starting pos and rot
53         startingPos = transform.position;
54         startingRot = transform.rotation;
55
56         //setup rigidbody max angular velocity
57         rb.maxAngularVelocity = physicsConfig.k_MaxAngularVelocity;
58
59         //setup base degrees of freedom
60         dof = new Vector3(0f, 0f, 0f);
61
62         if (joint)
63         {
64             //set joint settings from physics config
65             JointDrive jd = new JointDrive
66             {
67                 positionSpring = physicsConfig.maxJointSpring,
68                 positionDamper = physicsConfig.jointDampen,
69                 maximumForce = physicsConfig.maxJointForceLimit
70             };
71             joint.slerpDrive = jd;
72
73             //calculate degrees of freedom
74             dof.x = joint.angularXMotion !=
75                 ConfigurableJointMotion.Locked ? 1 : 0;
76             dof.y = joint.angularYMotion !=
77                 ConfigurableJointMotion.Locked ? 1 : 0;
78             dof.z = joint.angularZMotion !=
79                 ConfigurableJointMotion.Locked ? 1 : 0;
80         }
81     }
82
83     public void Initialize()
84     {
85         //setup events
86         onTouchedGround = new UnityEvent();
87         onTouchedTarget = new UnityEvent();
88
89         /// <summary>
90         /// Reset body part to initial configuration.
91         /// </summary>
92         public void ResetTransform()
```

```
91  {
92      rb.transform.position = startingPos;
93      rb.transform.rotation = startingRot;
94      rb.velocity = Vector3.zero;
95      rb.angularVelocity = Vector3.zero;
96      touchingGround = false;
97  }
98
99  public void ResetTransform(Vector3 position, Quaternion
100    rotation)
101 {
102     rb.transform.position = position;
103     rb.transform.rotation = rotation;
104     rb.velocity = Vector3.zero;
105     rb.angularVelocity = Vector3.zero;
106     touchingGround = false;
107 }
108 /// <summary>
109 /// Apply torque according to defined goal 'x, y, z' angle and
110   force 'strength'.
111 /// </summary>
112 public void SetJointTargetRotation(float x, float y, float z)
113 {
114     x = (x + 1f) * 0.5f;
115     y = (y + 1f) * 0.5f;
116     z = (z + 1f) * 0.5f;
117
118     var xRot = Mathf.Lerp(joint.lowAngularXLimit.limit,
119                           joint.highAngularXLimit.limit, x);
120     var yRot = Mathf.Lerp(-joint.angularYLimit.limit,
121                           joint.angularYLimit.limit, y);
122     var zRot = Mathf.Lerp(-joint.angularZLimit.limit,
123                           joint.angularZLimit.limit, z);
124
125     joint.targetRotation = Quaternion.Euler(xRot, yRot, zRot);
126 }
127
128 public void SetJointStrength(float strength)
129 {
130     var rawVal = (strength + 1f) * 0.5f *
131         physicsConfig.maxJointForceLimit;
132     var jd = new JointDrive
133     {
134         positionSpring = physicsConfig.maxJointSpring,
135         positionDamper = physicsConfig.jointDampen,
136         maximumForce = rawVal
137     };
138     joint.slerpDrive = jd;
139     currentStrength = jd.maximumForce;
140 }
```

```

136
137     void OnCollisionEnter(Collision col)
138     {
139         if (col.transform.CompareTag("ground"))
140         {
141             touchingGround = true;
142             if (triggerTouchingGroundEvent)
143             {
144                 onTouchedGround.Invoke();
145             }
146         }
147         if (col.transform.CompareTag("target"))
148         {
149             onTouchedTarget.Invoke();
150         }
151     }
152
153     void OnCollisionExit(Collision other)
154     {
155         if (other.transform.CompareTag("ground"))
156         {
157             touchingGround = false;
158         }
159     }
160
161     void FixedUpdate()
162     {
163         //calculate joint power
164         if (joint)
165         {
166             Vector3 currentTorque = joint.currentTorque;
167             Vector3 angularVel = rb.angularVelocity;
168
169             power = Mathf.Abs(Vector3.Dot(currentTorque,
170                               angularVel));
171         }
172     }

```

Listing B.1: Körperteil Skript

B.2. Angepasstes Agenten Skript

```

1 using System;
2 using System.Collections.Generic;
3 using UnityEngine;
4 using UnityEngine.Events;
5 using Unity.MLAgents;

```

```
6  using Unity.MLAgents.Actuators;
7  using Unity.MLAgents.Sensors;
8  using Random = UnityEngine.Random;
9
10 /*
11 * Walker Agent 1
12 * brain of the walker controlling the bodyparts and implementing
13 * the rl loop
14 */
15 public class WalkerAgent1 : Agent
16 {
17     [Header("Walk Speed")]
18     public float minWalkingSpeed = 0.1f;
19     public float maxWalkingSpeed = 10f;
20     public bool randomizeWalkSpeedEachEpisode;
21     public float targetWalkingSpeed = 0.1f;
22
23     [Header("Target To Walk Towards")]
24     public Transform target;
25     public UnityEvent onTouchedTarget = new UnityEvent();
26
27     [Header("Bodyparts")]
28     public Transform root;
29     public Transform head;
30     public bool randomizeRotationOnEpisode = true;
31     [HideInInspector] public List<Bodypart> bodyparts = new
32         List<Bodypart>();
33
34     [Header("Debug Log Stats")]
35     public bool logStats = false;
36
37     //This will be used as a stabilized model space reference point
38     //for observations
39     //Because ragdolls can move erratically during training, using
39     //a stabilized reference transform improves learning
40     protected OrientationCubeController1 walkOrientationCube;
41     public EnvironmentParameters m_ResetParams;
42     public StatsRecorder statsRecorder;
43
44     /*
45     * Environment stats
46     */
47     protected Vector3 previousPos;
48     protected float distanceMovedInTargetDirection;
49     protected float lastReachedTargetTime = 0f;
50     protected int reachedTargets;
51
52     public override void Initialize()
53     {
54         //init orientation object
```

```
53     GameObject orientationObject = new
54         GameObject("OrientationObject");
55     orientationObject.transform.parent = transform;
56     walkOrientationCube =
57         orientationObject.AddComponent<OrientationCubeController1>();
58     walkOrientationCube.root = root;
59     walkOrientationCube.target = target;
60
61     //change to auto setup each body part
62     foreach (Bodypart bp in
63         root.GetComponentsInChildren<Bodypart>())
64     {
65         bp.Initialize();
66         bp.onTouchedGround.AddListener(OnTouchedGround);
67         bp.onTouchedTarget.AddListener(OnTouchedTarget);
68         bodyparts.Add(bp);
69     }
70
71
72     /// <summary>
73     /// Loop over body parts and reset them to initial conditions.
74     /// </summary>
75     public override void OnEpisodeBegin()
76     {
77         //Reset all of the body parts
78         foreach (Bodypart bp in bodyparts)
79         {
80             bp.ResetTransform();
81         }
82
83         //Random start rotation to help generalize
84         if (randomizeRotationOnEpisode)
85         {
86             root.rotation = Quaternion.Euler(0, Random.Range(0.0f,
87                 360.0f), 0);
88         }
89
90         //Set our goal walking speed
91         targetWalkingSpeed =
92             randomizeWalkSpeedEachEpisode ?
93                 Random.Range(minWalkingSpeed, maxWalkingSpeed) :
94                 targetWalkingSpeed;
95
96         //record walking speed stats
97         RecordStat("Environment/WalkingSpeed", targetWalkingSpeed);
98
99         //record then reset distance moved in target direction
```

```
97     RecordStat("Environment/DistanceMovedInTargetDirection",
98                 distanceMovedInTargetDirection);
99     distanceMovedInTargetDirection = 0f;
100    previousPos = root.position;
101
102    //record then reset targets reached
103    RecordStat("Environment/ReachedTargets", reachedTargets);
104    reachedTargets = 0;
105
106    /// <summary>
107    /// Add relevant information on each body part to observations.
108    /// </summary>
109    public void CollectObservationBodyPart(Bodypart bp,
110                                            VectorSensor sensor)
111    {
112        //GROUND CHECK
113        sensor.AddObservation(bp.touchingGround); // Is this bp
114        touching the ground
115
116        //Get velocities in the context of our orientation cube's
117        space
118        //Note: You can get these velocities in world space as well
119        //      but it may not train as well.
120        sensor.AddObservation(walkOrientationCube.transform.InverseTransform
121                               position);
122
123        //return when bodypart joint has no free rotation axis
124        if (bp.dof.sqrMagnitude <= 0) return;
125
126        sensor.AddObservation(bp.rb.transform.localRotation);
127        sensor.AddObservation(bp.currentStrength /
128                               bp.physicsConfig.maxJointForceLimit);
129
129    /// <summary>
130    /// Loop over body parts to add them to observation.
131    /// </summary>
132    public override void CollectObservations(VectorSensor sensor)
133    {
134        var cubeForward = walkOrientationCube.transform.forward;
135
136        //velocity we want to match
137        var velGoal = cubeForward * targetWalkingSpeed;
138        //ragdoll's avg vel
139        var avgVel = GetAvgVelocity();
```

```
140
141     //current ragdoll velocity. normalized
142     sensor.AddObservation(Vector3.Distance(velGoal, avgVel));
143     //avg body vel relative to cube
144     sensor.AddObservation(walkOrientationCube.transform.InverseTransform
145     //vel goal relative to cube
146     sensor.AddObservation(walkOrientationCube.transform.InverseTransfor
147
148     //rotation deltas
149     sensor.AddObservation(Quaternion.FromToRotation(root.forward,
150         cubeForward));
150     sensor.AddObservation(Quaternion.FromToRotation(head.forward,
151         cubeForward));
152
153     //Position of target position relative to cube
154     sensor.AddObservation(walkOrientationCube.transform.InverseTransfor
155
156     foreach (Bodypart bp in bodyparts)
157     {
158         CollectObservationBodyPart(bp, sensor);
159     }
160
161     public override void OnActionReceived(ActionBuffers
162         actionBuffers)
163     {
164         var continuousActions = actionBuffers.ContinuousActions;
165         int i = -1;
166
167         foreach (Bodypart bp in bodyparts)
168         {
169             if (bp.dof.sqrMagnitude <= 0) continue;
170             float targetRotX = bp.dof.x == 1 ?
171                 continuousActions[++i] : 0;
172             float targetRotY = bp.dof.y == 1 ?
173                 continuousActions[++i] : 0;
174             float targetRotZ = bp.dof.z == 1 ?
175                 continuousActions[++i] : 0;
176             float jointStrength = continuousActions[++i];
177             bp.SetJointTargetRotation(targetRotX, targetRotY,
178                 targetRotZ);
179             bp.SetJointStrength(jointStrength);
180         }
181
182         public virtual void FixedUpdate()
183         {
184             distanceMovedInTargetDirection +=
185                 GetDistanceMovedInTargetDirection();
186
187         var cubeForward = walkOrientationCube.transform.forward;
```

```
183
184     // Set reward for this step according to mixture of the
185     // following elements.
186     // a. Match target speed
187     //This reward will approach 1 if it matches perfectly and
188     // approach zero as it deviates
189     var matchSpeedReward =
190         GetMatchingVelocityReward(cubeForward *
191             targetWalkingSpeed, GetAvgVelocity());
192     RecordStat("Reward/MatchingVelocityReward",
193             matchSpeedReward);
194
195     //Check for NaNs
196     if (float.IsNaN(matchSpeedReward))
197     {
198         throw new ArgumentException(
199             "NaN in moveTowardsTargetReward.\n" +
200             $" cubeForward: {cubeForward}\n" +
201             $" root.velocity: {bodyparts[0].rb.velocity}\n" +
202             $" maximumWalkingSpeed: {maxWalkingSpeed}"
203         );
204     }
205
206     // b. Rotation alignment with target direction.
207     //This reward will approach 1 if it faces the target
208     // direction perfectly and approach zero as it deviates
209     var headForward = head.forward;
210     headForward.y = 0;
211     // var lookAtTargetReward = (Vector3.Dot(cubeForward,
212     //     head.forward) + 1) * .5F;
213     var lookAtTargetReward = (Vector3.Dot(cubeForward,
214         headForward) + 1) * .5F;
215     RecordStat("Reward/LookAtTargetReward", lookAtTargetReward);
216
217     //Check for NaNs
218     if (float.IsNaN(lookAtTargetReward))
219     {
220         throw new ArgumentException(
221             "NaN in lookAtTargetReward.\n" +
222             $" cubeForward: {cubeForward}\n" +
223             $" head.forward: {head.forward}"
224         );
225     }
226
227     AddReward(matchSpeedReward * lookAtTargetReward);
228 }
229
230 //Returns the average velocity of all of the body parts
231 //Using the velocity of the hips only has shown to result in
232 // more erratic movement from the limbs, so...
233 //...using the average helps prevent this erratic movement
```

```
225     public Vector3 GetAvgVelocity()
226     {
227         Vector3 velSum = Vector3.zero;
228
229         foreach (Bodypart bp in bodyparts)
230         {
231             velSum += bp.rb.velocity;
232         }
233
234         var avgVel = velSum / bodyparts.Count;
235         return avgVel;
236     }
237
238     //normalized value of the difference in avg speed vs goal
239     //walking speed.
240     public float GetMatchingVelocityReward(Vector3 velocityGoal,
241                                             Vector3 actualVelocity)
242     {
243         //distance between our actual velocity and goal velocity
244         float upperLimit = Mathf.Max(0.1f, targetWalkingSpeed);
245         var velDeltaMagnitude =
246             Mathf.Clamp(Vector3.Distance(actualVelocity,
247                                         velocityGoal), 0, upperLimit);
248
249         //return the value on a declining sigmoid shaped curve that
250         //decays from 1 to 0
251         //This reward will approach 1 if it matches perfectly and
252         //approach zero as it deviates
253         float matchingVelocityReward = Mathf.Pow(1 -
254             Mathf.Pow(velDeltaMagnitude / upperLimit, 2), 2);
255         return matchingVelocityReward;
256     }
257
258     protected float GetDistanceMovedInTargetDirection()
259     {
260         //calculate the displacement vector
261         Vector3 currentPos = root.position;
262         Vector3 displacement = currentPos - previousPos;
263
264         //project the displacement vector onto the goal direction
265         //vector
266         float movementInTargetDirection = Vector3.Dot(displacement,
267                                                       walkOrientationCube.transform.forward);
268
269         //update the previous position for the next frame
270         previousPos = currentPos;
271         return movementInTargetDirection;
272     }
273
274     protected void OnTouchedTarget()
275     {
```

```

267     if (lastReachedTargetTime + 0.1f <= Time.time)
268     {
269         lastReachedTargetTime = Time.time;
270         reachedTargets++;
271         onTouchedTarget.Invoke();
272     }
273 }
274
275 protected void OnTouchedGround()
276 {
277     //check that the episode did not start in the last step to
278     //remove duplicate calls
279     if (Academy.Instance.StepCount < 1) return;
280     SetReward(-1f);
281     EndEpisode();
282 }
283
284 protected void RecordStat(string path, float value)
285 {
286     if (logStats) Debug.Log($"{path}: {value}");
287     statsRecorder.Add(path, value);
288 }
```

Listing B.2: Agenten Skript

B.3. Angepasstes Zielsteuerung Skript

```

1  using UnityEngine;
2  using Random = UnityEngine.Random;
3  using Unity.MLAgents;
4  using UnityEngine.Events;
5
6  public class TargetController1 : MonoBehaviour
7  {
8      [Header("Target Config")]
9      public float spawnRadius; //The radius in which a target can be
10     randomly spawned.
11     public bool setRandomStartPos = true;
12
13     private Vector3 m_startingPos; //the starting position of the
14     target
15
16     void OnEnable()
17     {
18         m_startingPos = transform.position;
19         if (setRandomStartPos)
20             {
```

```

19         MoveTargetToRandomPosition();
20     }
21 }
22
23 /// <summary>
24 /// Moves target to a random position within specified radius.
25 /// </summary>
26 public void MoveTargetToRandomPosition()
27 {
28     var newTargetPos = m_startingPos + (Random.insideUnitSphere
29         * spawnRadius);
30     newTargetPos.y = m_startingPos.y;
31     transform.position = newTargetPos;
32 }
```

Listing B.3: Zielsteuerung Skript

B.4. Angepasstes Orientierungsobjekt Skript

```

1 using UnityEngine;
2
3 /// <summary>
4 /// Utility class to allow a stable observation platform.
5 /// </summary>
6 public class OrientationCubeController1 : MonoBehaviour
7 {
8     public Transform root;
9     public Transform target;
10
11     //Update position and Rotation
12     public void UpdateOrientation()
13     {
14         var dirVector = target.position - transform.position;
15         dirVector.y = 0; //flatten dir on the y. this will only
16             // work on level, uneven surfaces
17         var lookRot =
18             dirVector == Vector3.zero
19                 ? Quaternion.identity
20                 : Quaternion.LookRotation(dirVector); //get our
21                     // look rot to the target
22
23         //UPDATE ORIENTATION CUBE POS & ROT
24         transform.SetPositionAndRotation(root.position, lookRot);
25     }
26
27     void FixedUpdate()
28     {
```

```

27     UpdateOrientation();
28 }
29 }
```

Listing B.4: Orientationsobjekt Skript

B.5. Implementierung Spherecast

```

1 void CheckLookAtTarget()
2 {
3     RaycastHit hit;
4     if (Physics.SphereCast(head.position, lookAtTargetMargin,
5         head.forward, out hit, 9f) &&
6         hit.collider.gameObject.CompareTag("lookTarget"))
7     {
8         if (isLookingAtTarget)
9         {
10             if (startetLookingAtTarget + durationLookAtTarget <=
11                 Time.fixedTime)
12             {
13                 onLookedAtTarget.Invoke();
14                 isLookingAtTarget = false;
15             }
16             startetLookingAtTarget = Time.fixedTime;
17             isLookingAtTarget = true;
18         }
19     }
20     else
21     {
22         isLookingAtTarget = false;
23     }
24 }
```

Listing B.5: Implementierung von Spherecast um Blickkontakt überprüfen zu können

B.6. Implementierungen der Belohnungen für eine verbesserte Gehbewegung

```

1 float leftFootDistance = Vector3.Distance(footL.position,
2     target.position);
```

```

2 float rightFootDistance = Vector3.Distance(footR.position,
3     target.position);
4 if (!leftForward && leftFootDistance < rightFootDistance)
5 {
6     leftForward = true;
7     switchTime = Time.fixedTime;
8 }
9 else if (leftForward && rightFootDistance < leftFootDistance)
10 {
11     leftForward = false;
12     switchTime = Time.fixedTime;
13 }
14 float switchDeltaTime = Time.fixedTime - switchTime;
15 float footSwitchReward = -Mathf.Clamp((switchDeltaTime / 4) - 0.3f,
16     0f, 1f);
17 if (logStats && footSwitchReward < 0) Debug.Log($"foot switch
18     reward: {footSwitchReward}, switched: {switchDeltaTime} seconds
19     ago");
20 RecordStat("Reward/FootSwitchReward", footSwitchReward);
21
22 float leftHandDistance = Vector3.Distance(handL.position,
23     target.position);
24 float rightHandDistance = Vector3.Distance(handR.position,
25     target.position);
26 float rootDistance = Vector3.Distance(root.position,
27     target.position);
28 float pendulumSum = 0f;
29 if (leftForward)
30 {
31     pendulumSum = (rootDistance - rightHandDistance) +
32         (leftHandDistance - rootDistance);
33 }
34 else
35 {
36     pendulumSum = (rootDistance - leftHandDistance) +
37         (rightHandDistance - rootDistance);
38 }
39 float armPendulumReward = Mathf.Clamp((pendulumSum / 2) - 0.5f,
40     -1f, 0f);
41 RecordStat("Reward/ArmPendulumReward", armPendulumReward);
42
43 float totalPower = GetTotalPower();
44 float powerSaveReward = -Mathf.Clamp(totalPower / 3000 - 0.05f, 0f,
45     1f);
46 if (logStats) Debug.Log($"power save reward: {powerSaveReward},
47     total: {totalPower}");
48 RecordStat("Reward/PowerSaveReward", powerSaveReward);
49
50 AddReward(Mathf.Min(footSwitchReward, powerSaveReward,
51     armPendulumReward));

```

Listing B.6: Implementierung von zusätzliche Belohnungen für Gehbewegungsanpassungen

B.7. Implementierung von Deep Mimic Körperhaltungsbelohnung

```

1  private float CalculatePoseReward()
2  {
3      float sum = 0f;
4      //sum over all bodyparts
5      for (int i = 0; i < bodyparts.Count; i++)
6      {
7          Bodypart bp = bodyparts[i];
8          ReferenceBodypart rbp =
9              referenceController.referenceBodyparts[i];
10         float angle = 0f;
11         if (i == 0)
12         {
13             Vector3 bodypartUp = bp.transform.up;
14             Vector3 referenceUp = rbp.transform.up;
15             bodypartUp.y = 0;
16             referenceUp.y = 0;
17             bodypartUp.Normalize();
18             referenceUp.Normalize();
19             angle = Vector3.Angle(referenceUp, bodypartUp);
20         }
21         else
22         {
23             angle = Quaternion.Angle(rbp.transform.localRotation,
24                                     bp.rb.transform.localRotation);
25         }
26         sum += angle;
27     }
28     float avg = sum / (bodyparts.Count - 1);
29     float poseReward = -(avg / 180f);
30     return poseReward;
31 }
```

Listing B.7: Implementierung von Deep Mimic Körperhaltungsbelohnung

B.8. Initialisierung der Körperhaltung von Animation

```

1  public void ResetReference()
2  {
3      transform.position = startingPos;
4      //call the Play method of the Animator to start playing the
5      //animation at a specific point
6      float randomPhase = Random.Range(phaseStartMin, phaseStartMax);
7      animator.Play(animationName, -1, randomPhase);
```

```
7     //reset reference bodyparts on next frame when animation has
8     //started from random phase
9     StartCoroutine(ResetBodypartsOnNextFrame());
10 }
11 IEnumerator ResetBodypartsOnNextFrame()
12 {
13     //wait for the next frame
14     yield return null;
15
16     // Code to be executed on next frame
17     int i = 0;
18     foreach (Bodypart bp in bodyparts)
19     {
20         Transform referenceBone =
21             referenceController.referenceBodyparts[i].transform;
22         bp.ResetTransform(referenceBone.position,
23                           referenceBone.rotation);
24         i++;
25     }
26 }
```

Listing B.8: Implementierung Initialisierung des Charakters mit Körperhaltung aus Animation