



HOCHSCHULE HEILBRONN

ENTWICKLUNG EINES PHYSIKBASierten CHARAKTERCONTROLLERS MIT UNITY ML AGENTS

Software-Engineering
Fakultät für Informatik
der Hochschule Heilbronn

Bachelor-Thesis

vorgelegt von

Simon Grözing
Matrikelnummer: 205047

Inhaltsverzeichnis

1. Einleitung	6
2. Grundlagen	7
2.1. Verstärkendes Lernen	7
2.2. ML-Agents	8
2.2.1. Aufbau	8
2.2.2. Komponenten	9
2.2.3. Programmierschnittstellen	11
2.3. Unity Physik	13
3. Analyse	17
3.1. Szenenaufbau	17
3.2. Physikkomponenten und -konfiguration	18
3.3. Agent implementierung	18
4. Charaktercontroller	21
4.1. Nutzersteuerung	21
4.1.1. Versuch 1	21
4.1.2. Versuch 2	22
4.1.3. Versuch 3	22
4.2. Modell Anpassungen	23
4.2.1. Versuch 4	23
4.3. Mixamo Charakter	26
5. Fazit	27
A. Anhang 1	28

Abbildungsverzeichnis

2.1.	Verstärkendes Lernen Ablauf	7
2.2.	Unity ML-Agents Aufbau	8
2.3.	Unity ML-Agents Agenten Komponente	9
2.4.	Unity ML-Agents Verhalten Parameter Komponente	10
2.5.	Unity ML-Agents Entscheidung Anfragen Komponente	10
2.6.	Unity ML-Agents Physik Festkörper	13
2.7.	Unity ML-Agents Physik Kollisionskomponenten	14
2.8.	Unity ML-Agents Physik Gelenk	15
3.1.	Walker-Demo Szenenaufbau	17
3.2.	Agent Konfiguration	19
4.1.	Walker Demo Match Velocity Belohnungsfunktion	24
4.2.	Neue Exponential Match Velocity Belohnungsfunktion	24
4.3.	Vergleich vorwärts laufen Exp Belohnung gegen Walker Demo Belohnung . .	25
4.4.	Vergleich der Walker Demo Belohnungsfunktion unter verschiedenen Zielge- schwindigkeiten	25

Tabellenverzeichnis

3.1. Walker Agent Körperteile	18
---	----

Listings

2.1. Agent Funktionen	11
2.2. Trainer Konfigurationsdatei	12
4.1. Nutzersteuerung erster Prototyp	21
4.2. Nutzersteuerung berechnung mit Weltachsen	22
4.3. Erweiterung der Nutzersteuerung mit separater Blickrichtung	22

1. Einleitung

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

2. Grundlagen

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

2.1. Verstärkendes Lernen

Der Begriff 'Verstärkendes Lernen' beschreibt eine Art von Problemstellung und die dafür geeigneten Problemlösungsmethoden im Bereich des Maschinellen Lernens. Die grundlegenden Bestandteile einer Trainingsumgebung sind der Agent und die Umgebung, in der der Agent seine Aktionen ausführt. Der Ansatz ist in vielerlei Hinsicht vergleichbar mit dem Lernvorgang von Menschen. Ein Baby lernt das Krabbeln ohne direkte Anweisungen, nur durch die Wahrnehmung der Umgebung, das Verhalten der Umgebung in Relation zu seinen Bewegungen und die mit den Bewegungen einhergehenden Belohnungen. Auf dieselbe Art lernt der Agent beim Verstärkenden Lernen von jedem Zustand die Aktion auszuführen, um die Belohnung zu maximieren. Die Belohnung können dabei positiv oder negativ sein. Im Fall des Babys sind die Belohnungen Faktoren wie Schmerz, Hunger, Müdigkeit oder gestillte Neugier. Der Agent hingegen erhält eine numerische Belohnung.[3]

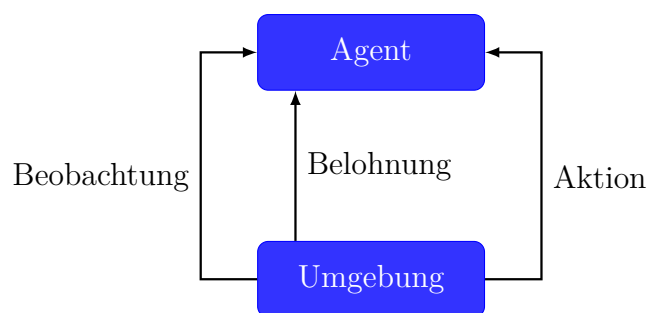


Abbildung 2.1.: Verstärkendes Lernen Ablauf

Die Abbildung 2.1 zeigt die Verbindungen zwischen dem Agent und der Umgebung. Der Agent erhält als Input einen Zustand oder meist 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 abgefragt. Bei rundenbasierten Spielen kann dieser Vorgang jedoch auch nur einmal pro Runde stattfinden.

2.2. MI-Agents

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

2.2.1. Aufbau

Die Implementierung ist in zwei Teile unterteilt. 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-Netzwerkcommunication. Während des Trainings tauschen die Umgebungen den Zustand der Simulationsumgebung, die ausgewählten Aktionen des neuronalen Netzes und weitere für die Auswertung des Trainings relevante Daten aus.[1]

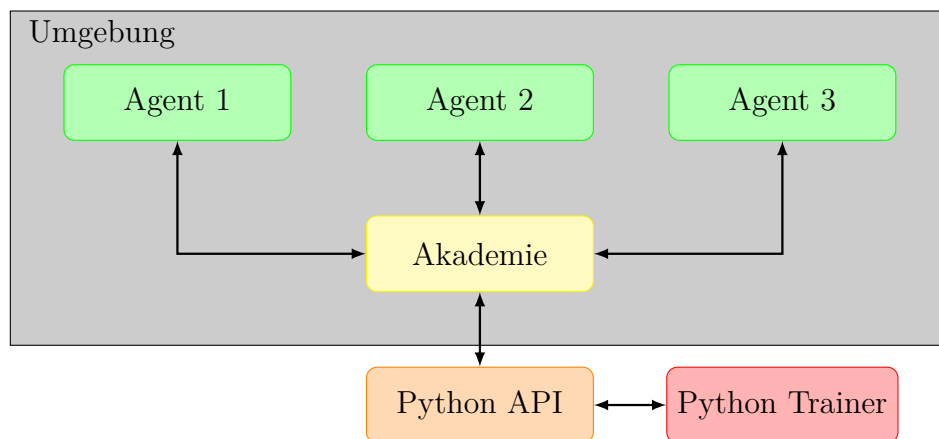


Abbildung 2.2.: Unity ML-Agents Aufbau

Die Umgebung in Abbildung 2.2 stellt den Zusammenhang der ML-Agents Komponenten dar. Das Unity-Paket enthält zwei Grundlegenden Komponenten, die Akademie und Agenten. Um eine Szene für das verstärkende Lernen einzurichten benötigt die Szene mindestens

einen Agenten. Objekte in der Unity Szene werden, mit dem hinzufügen einer Agenten Komponente als Agent für das Verstärkende Lernen konfiguriert.

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, für Beobachtungen basierend auf Raycasts sowie Kamerabildern existieren separate Komponenten. Zahlenwerte sowie Vektoren und Quaternionen können jedoch auch direkt über eine Funktion im Agenten der Beobachtung angehängt werden.

Die Akademie ist eine Einzelinstanz welche beim starten der Unity Umgebung einmalig erstellt und von allen Agenten referenziert wird. Zuständig ist die Akademie für das steuern des Trainingsablaufs sowie das Aufbauen der Kommunikationspipeline zwischen der Unity- und Pythonumgebung.

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. Über die Python-API kann der Python-Trainer auf Beobachtungen zugreifen, Aktionen ausführen und anhand der Belohnungssignale die Gewichtung der neuronalen Netze anpassen, um das Verhalten des Agenten zu optimieren. Nachdem die Konfigurationsparameter von der Unity-Instanz an die Python-Umgebung übertragen wurden, wird basierend darauf ein Python-Trainer erstellt. Der Python-Trainer initialisiert dann die neuronalen Netze und berechnet deren Gewichtungen mithilfe der Lernalgorithmen.

2.2.2. Komponenten

In diesem Kapitel werden die Grundlegenden Komponenten des Unity ML-Agents Packets, welche in der Arbeit verwendet wurden erklärt. Dadurch sollten Codeausschnitte und Komponentenabbildungen in folgenden Kapiteln deutlich zu verstehen sein.



Abbildung 2.3.: Unity ML-Agents Agenten Komponente

Abbildung 2.3 zeigt die Basiskomponente des Agenten. Die Agenten Komponente stellt alle grundlegenden Funktionen des verstärkenden Lernens bereit und implementiert die Verbindung zur Akademie und dem Verhalten des Agenten. Ohne das überschreiben der Funktionen

ist die Agentenklasse jedoch ohne Funktion. Die genauen Methoden zur Implementierung eigener Agentenklassen werden näher in Kapitel 2.2.3 behandelt. Das einzige Feld zur Konfiguration ist Max Step, welches die maximale Anzahl der Schritte innerhalb einer Episode festlegt.

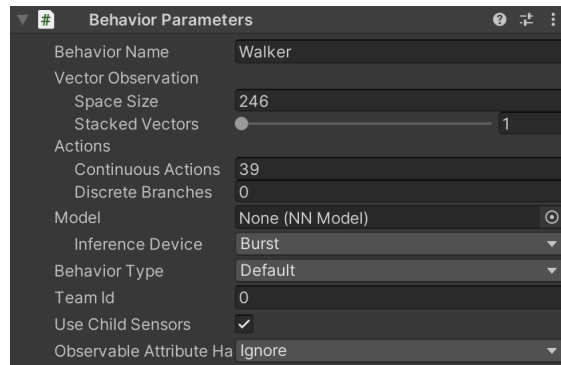


Abbildung 2.4.: Unity ML-Agents Verhalten Parameter Komponente

In Abbildung 2.5 ist die Verhaltens Parameter Komponente zusehen. Das Verhalten legt die Ein- und Ausgangsgröße des Modells fest. Für das Training muss ein gleichnamiges Verhalten in der Konfigurationsdatei definiert sein. Um ein bereits trainiertes Verhalten zu verwenden muss im Modell Feld die Modelldatei referenziert werden.

- 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

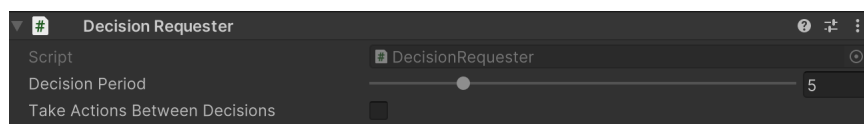


Abbildung 2.5.: Unity ML-Agents Entscheidung Anfragen Komponente

Die Komponente in Abbildung 2.5 fragt in regelmäßigen Abständen Entscheidungen an. Das bedeutet es wird eine Beobachtung erstellt. Darauf wird die Beobachtung als Eingangswert

für das neuronale Netz genutzt und dann eine Aktion vom neuronalen Netz ausgewählt. Während dem training wird diese Beobachtung zusammen mit der darauf ausgeführten Aktion und der resultierenden Belohnung im Trainingsspeicher abgelegt. Die "Decision Period" gibt an in welchem Interval 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.

2.2.3. Programmierschnittstellen

Die Agenten Komponente enthält einige Funktionen die für das Training eines Agenten implementiert werden müssen. In den folgenden Absätzen werden die Funktionen anhand Beispielen näher erklärt.

```
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 Methoden wird festgelegt welche Daten dem Agent für das Training bereit stehen 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 ein Reward in jedem FixedUpdate vergeben über die AddReward Methode die auch Teil der Agenten-Komponente ist. Der Reward kann

aber an jeder Stelle im Code vergeben werden, der Code dient hier nur als ein Beispiel.

```

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      lambda: 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

Die Trainings Konfigurationsdatei (siehe Listing 2.2) enthält mehrere Teile. Der Hyperparameter Teil enthält die Hyperparameter des Maschinellen Lernalgorithmuses (Zeile 5-13), danach folgt der `network_settings` Teil welcher die Konfiguration des Neuronalelennetzes festlegt (Zeile 14-18). Anschließend folgen noch Konfigurationen für die Belohnungssignale im Bereich `reward_signals` (Zeile 19-22) und Einstellungen für die Speicherung der Daten sowie der länge des Trainings (Zeile 23-27). Ganz am Ende der Konfigurationsdatei (Zeile 28-29) befinden sich noch Umgebungsparameter welche erweitert und während dem Training ausgelesen werden können.

2.3. Unity Physik

Unity ermöglicht mit der eingebauten Physikengine weitestgehend realistische Berechnung von Kollisionen, Schwerkraft und weiteren Kräften.

Die Festkörper (Rigidbody) Komponente ermöglicht es 3D Objekte als nicht verformbares Objekt physikalisch zu simulieren.

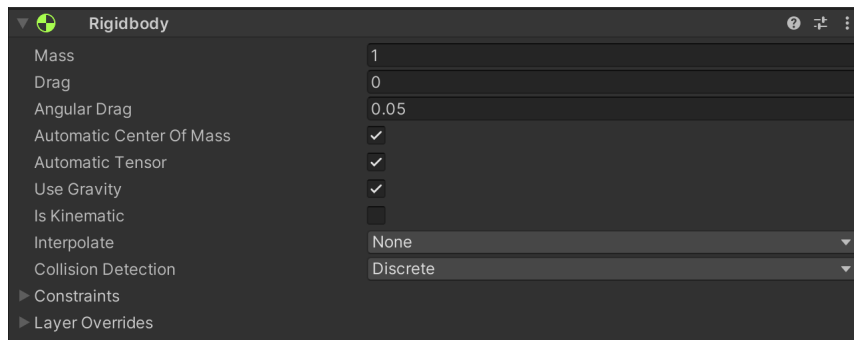


Abbildung 2.6.: Unity ML-Agents Physik Festkörper

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

Um Kollisionen zwischen Objekten zu berechnen benötigen diese zusätzlich eine Kollisionskomponente. Zur Optimierung werden zur Berechnung der Kollisionen die Körper vereinfacht dargestellt. Komplexe 3D Modelle werden als Kugel, Kapsel oder Box vereinfacht.

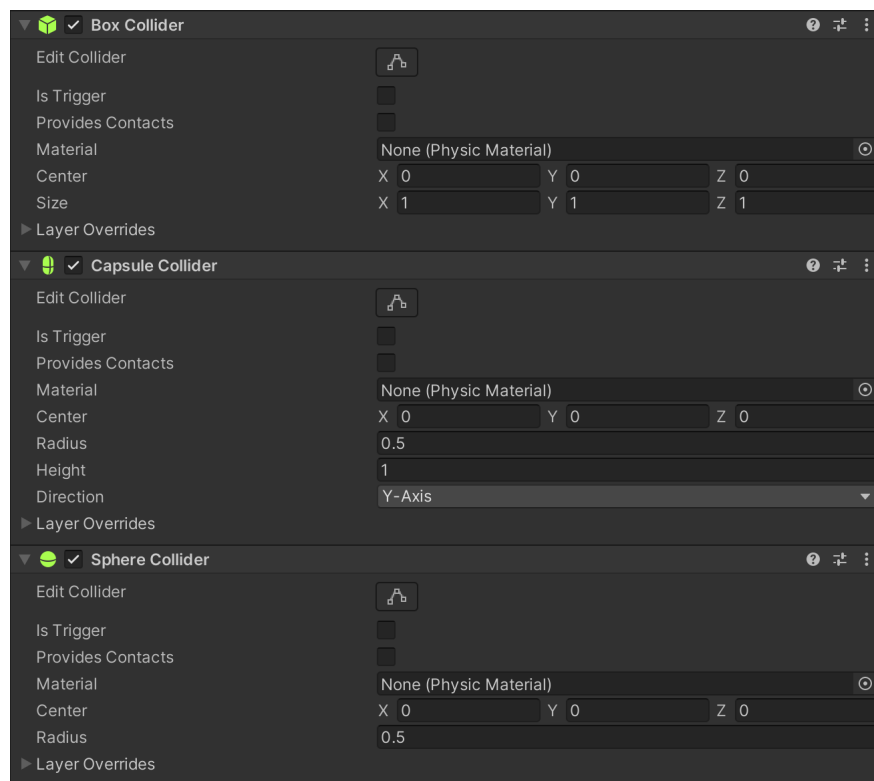


Abbildung 2.7.: Unity ML-Agents Physik Kollisionskomponenten

Festkörper können mit Gelenken zu komplexeren Körperstrukturen verbunden werden. Die Konfigurierbare Gelenkkomponente (Configurable Joint) ermöglicht es ein Gelenk mit freier Bewegung und Rotation auf allen 3 Achsen zu simulieren. Im Kontext dieser Arbeit wird das Gelenk dabei auf Rotation beschränkt und als Kugelförmiges Gelenk verwendet.

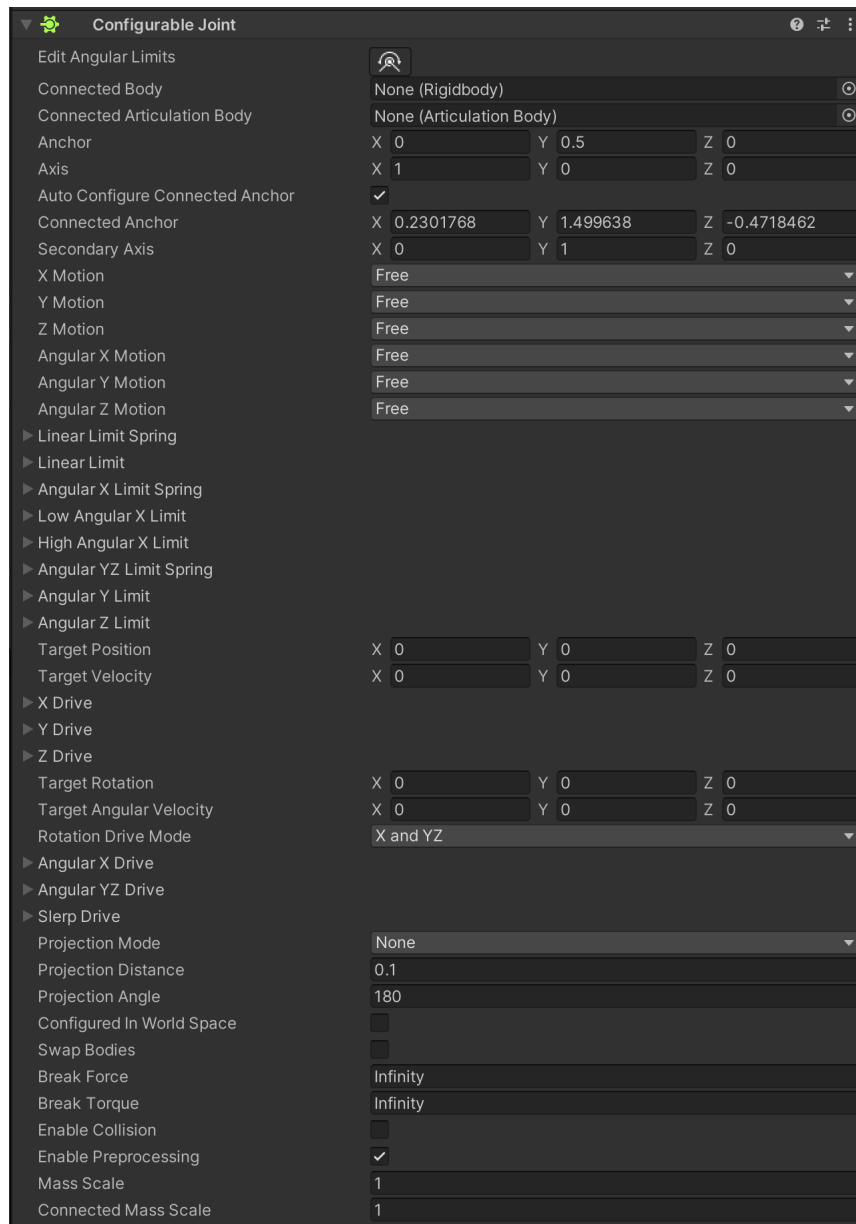


Abbildung 2.8.: Unity ML-Agents Physik Gelenk

- Connected Body: bestimmt mit welchem Körper das Gelenke verbunden ist
- Anchor: legt fest an welchem Punkt die Verbindung zum verbundenen Körper besteht
- Axis: legt die Hauptbewegungs- und Rotationsachse fest
- Secondary Axis: legt die sekundäre Bewegungs- Rotationsachse fest

- Angular X Y Z Motion: bestimmt ob das Gelenk Rotation zwischen den Körpern auf der X Y Z Achse zulässt
- Target Position: bestimmt das Ziel zu welchem das Gelenk sich bewegen soll
- Angular X Y Z Limit: ermöglicht das festlegen von Winkellimits für die Rotationsbewegungen
- X Y Z und Slerp Drive: bestimmen die Stärke der Federkraft welche das Gelenk in die Zielposition bewegt

3. Analyse

Zusätzlich zu den maschinellen Lernkomponenten stellt Unity auch Demonstrationsumgebungen bereit, in denen verschiedene Lösungen für gängige Verstärkungslernprobleme implementiert sind. In der Walker-Demo wird ein physisch simulierter Charakter darauf trainiert, zu einem Zielwürfel zu laufen. Diese Demo-Umgebung implementiert bereits einige Grundlagen für die Steuerung eines physisch simulierten Charakters. Aus diesem Grund wird in dieser Arbeit die Walker-Demo als Basis für die Entwicklung genutzt. In diesem Kapitel wird daher die Walker-Demo analysiert, um in den folgenden Kapiteln darauf aufzubauen.

3.1. Szenenaufbau

Die Szene besteht aus einem quadratischen Spielfeld mit Boden und Wänden die der Charakter nicht verlassen kann (siehe Abbildung 3.1). Der Läufer startet jede Trainingsepisode in der Mitte des Spielfelds. Das Ziel wird zu Beginn zufällig platziert. Bei jedem Erreichen des Ziels wird die Zielposition erneut zufällig bestimmt.

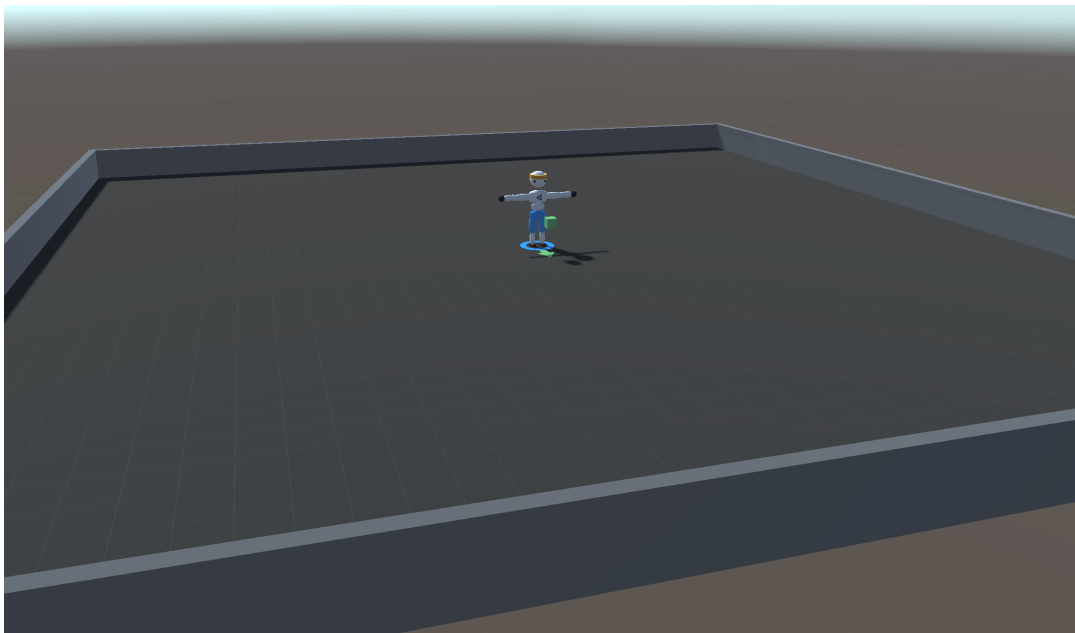


Abbildung 3.1.: Walker-Demo Szenenaufbau

3.2. Physikkomponenten und -konfiguration

Der Körper besteht aus 11 Kapseln, drei Kugeln und 2 Quadern, jeder dieser Formen hat eine Festkörper und eine Kollisions Physikkomponente. Zwischen den Körperteilen werden die Gelenke als Kugelgelenke simuliert. Die genaue Physikkonfiguration der Körperteile werden veranschaulicht in Tabelle 3.1

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

3.3. Agent implementierung

In diesem Kapitel wird die Agenten Implementierung näher Analysiert.

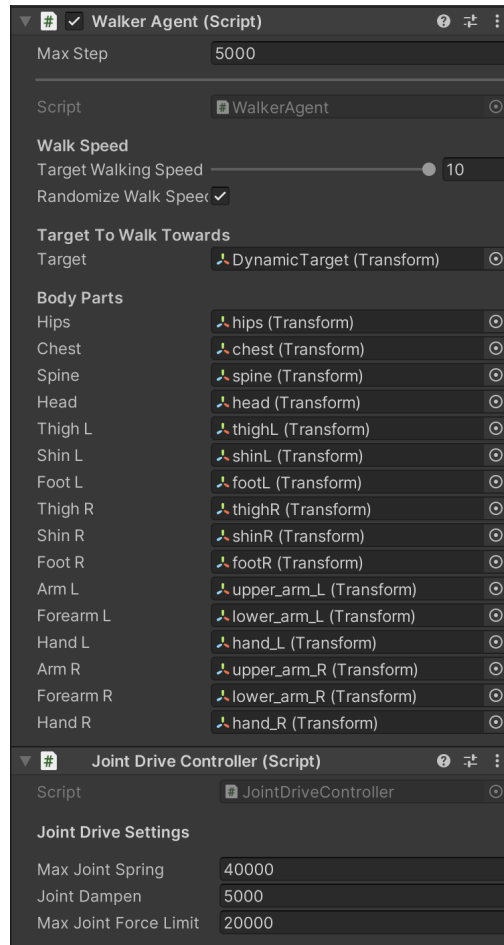


Abbildung 3.2.: Agent Konfiguration

Abbildung 3.2 zeigt die Agentenkomponente im Inspektor. Um die Komponente zu nutzen müssen hier die Körperteile des Walkers referenziert werden. Zusätzlich kann eine Zielgeschwindigkeit festgelegt werden und ob die Geschwindigkeit variieren soll während dem Training. Als letztes muss auch das Zielobjekt referenziert werden.

$$Q_{H\ddot{u}fte} = Quaternion(H\ddot{u}ftr\ddot{u}hrung - Zielr\ddot{u}hrung)$$

$$Q_{Blick} = Quaternion(Blickr\ddot{u}hrung - Zielr\ddot{u}hrung)$$

$$S = \{$$

$$|Zielgeschwindigkeit - Durchschnittsgeschwindigkeit|,$$

$$Durchschnittsgeschwindigkeit,$$

$$Zielgeschwindigkeit,$$

$$Q_{H\ddot{u}fte},$$

$$Q_{Blick},$$

$$Zielposition,$$

K_1, K_2, \dots, K_n
 $\}$

$K = \{$
Bodenkontakt,
Geschwindigkeit,
Rotationsgeschwindigkeit,
Position – Hüftposition.
Quaternion(LokaleRotation),
Gelenkstärke
 $\}$

$A = \{$
Rotationswinkel₁, *Rotationswinkel₂*, ..., *Rotationswinkel_n*,
Gelenkstärke₁, *Gelenkstärke₂*, ..., *Gelenkstärke_n*
 $\}$

$V_\delta = \text{Clip}(|\vec{\text{Geschwindigkeit}} - \vec{\text{Zielgeschwindigkeit}}|, 0, |\vec{\text{Zielgeschwindigkeit}}|)$
 $R_V = (1 - (V_\delta/|\vec{\text{Zielgeschwindigkeit}}|)^2)^2$
 $R_L = (\vec{\text{Zielrichtung}} \cdot \vec{\text{Blickrichtung}}) + 1) \cdot 0.5$
 $R = R_V \cdot R_L$

Initialisierung bzw Episodereset erklären Orientation Object erklären Zielsetzung erklären
 Rewardfrequenz erwähnen

4. Charaktercontroller

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

4.1. Nutzersteuerung

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

4.1.1. Versuch 1

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

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

Listing 4.1: Nutzersteuerung erster Prototyp

Diese implementierung ermöglicht grundsätzlich das der Läufer über die Tastatureingabe gesteuert werden kann, hat aber noch einige Probleme. Das positionieren des Ziels relativ

zur Hüfte hat als Konsequenz das jede Bewegung der Hüfte Einfluss auf die Laufrichtung hat, wodurch sich der Läufer nicht stabil steuern lässt.

4.1.2. Versuch 2

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

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

Listing 4.2: Nutzersteuerung berechnung mit Weltachsen

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

4.1.3. Versuch 3

Die Lösung ist eine Kombination aus den Ideen der ersten beiden Versuche. Die Laufrichtung soll weiterhin relativ zum Läufer bestimmen werden gleichermaßen aber von der wechselhaften Bewegung des Läufers entkoppeln sein. Das hinzufügen einer separaten Rotationskomponente für die Bestimmung der Blickrichtung erfüllt hier die Kriterien.

Zu beginn wird die Blickrichtung mit der Vorwärtskomponente der Hüftrotation gleichgesetzt. Ausgehend von der Startrichtung wird dann über horizontalen Mausinput die Richtung angepasst (siehe Listing 4.3).

```
1 void Start()
2 {
3     //Root Position als Startposition festhalten
4     startForward = root.forward;
5     startRight = root.right;
6 }
7 void FixedUpdate()
8 {
9     //Einlesen Tastatur Input
10    float inputHor = Input.GetAxis("Horizontal");
11    float inputVert = Input.GetAxis("Vertical");
```

```

12
13 //Einlesen Maus Input
14 float mouseX = Input.GetAxis("Mouse X");
15 rotAngle += mouseX;
16
17 //Berechnung der Rotation
18 Quaternion rotation = Quaternion.AngleAxis(rotAngle,
19     rotationAxis);
20
21 //Anwendung der Rotation auf Richtungsvektoren
22 Vector3 directionForward = rotation * startForward;
23 Vector3 directionRight = rotation * startRight;
24
25 //Setzen der Zielposition
26 transform.position = root.position + directionForward *
    inputVert + directionRight * inputHor;

```

Listing 4.3: Erweiterung der Nutzersteuerung mit separater Blickrichtung

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

4.2. Modell Anpassungen

Das trainierte Modell der Walker Demo beherrscht jedoch nur die Fortbewegung in Blickrichtung. Durch diese Einschränkung lässt sich von der Steuerung mit WASD nur die W Komponente nutzen. Eine weitere große Einschränkung ist, dass der Läufer nicht darauf trainiert ist stehen zu bleiben. Das resultiert darin dass der Läufer fällt sobald der Nutzer keinen Tastaturinput gibt. Dieses Kapitel beschäftigt sich mit den Einschränkungen der Walker-Demonstration im Bezug auf unterschiedliche Bewegungsrichtungen.

4.2.1. Versuch 4

Im ersten Schritt wird getestet wie der Walker angepasst werden kann um die fehlenden Bewegungsabläufe in separaten Modellen zu erlernen. Für das stehenbleiben wird die Zielgeschwindigkeit auf 0 gesetzt während das Ziel auf der Startposition befindet. Die Belohnungsfunktion der Demo hat dabei das Problem das durch die Zielgeschwindigkeit geteilt wird.

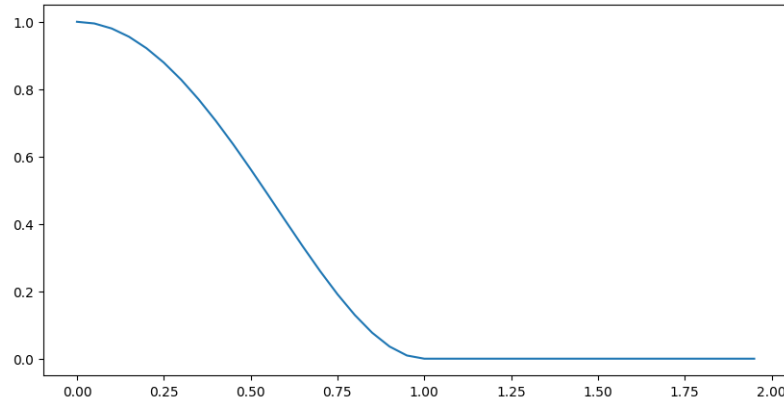


Abbildung 4.1.: Walker Demo Match Velocity Belohnungsfunktion

Die Abbildung 4.1 zeigt die Original Belohnungsfunktion ($R_V = (1 - (V_\delta / |\vec{\text{Zielgeschwindigkeit}}|)^2)^2$) für die Zielgeschwindigkeit von 1.

Um das zu vermeiden wurde das trainieren mit einer anderen Belohnungsfunktion getestet.

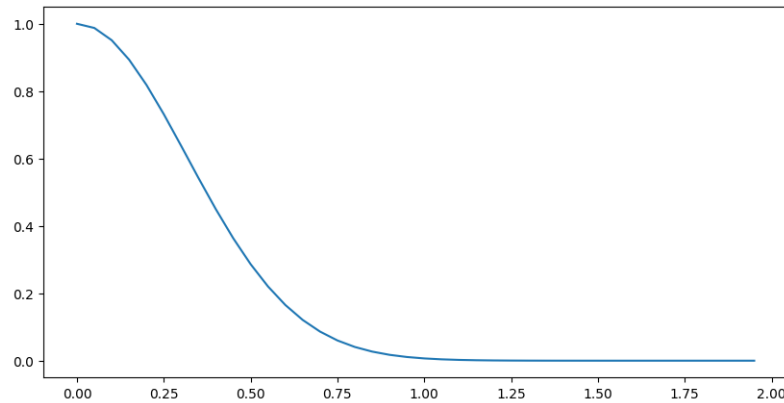


Abbildung 4.2.: Neue Exponential Match Velocity Belohnungsfunktion

Die Abbildung 4.2 zeigt die neue Belohnungsfunktion ($R1_V = \exp(5 \cdot (-V_\delta^2))$). Die neue Belohnungsfunktion ist inspiriert von den Belohnungsfunktionen des Papers "DeepMimic: Example-Guided Deep Reinforcement Learning of Physics-Based Character Skills".[2]

Der Walker konnte mit der neuen Belohnungsfunktion lernen auf der Stelle zu stehen. Mit zufälliger Zielgeschwindigkeit zu einem Ziel zu laufen wie im Ursprünglichen Verhalten konnte damit jedoch nicht zufriedenstellend erlernt werden.

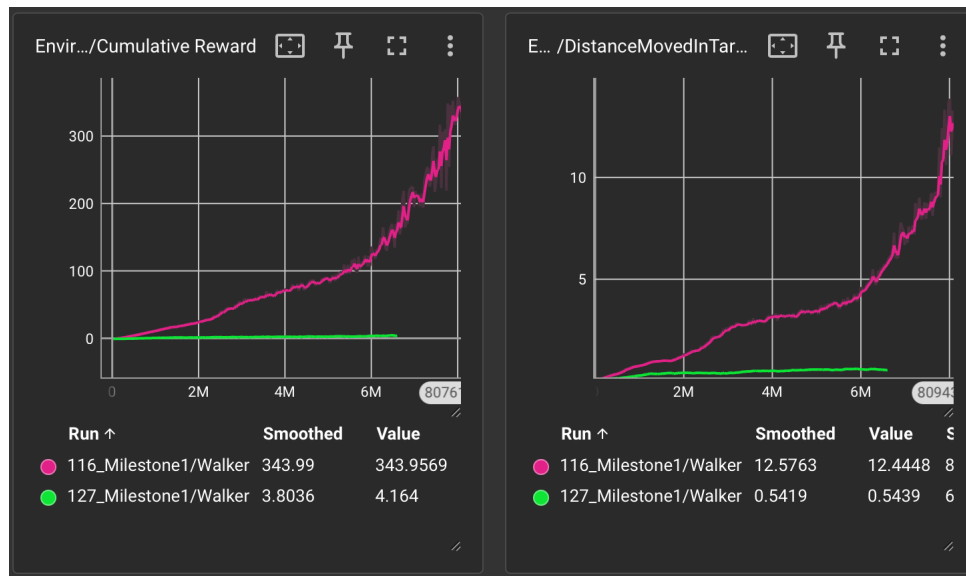


Abbildung 4.3.: Vergleich vorwärts laufen Exp Belohnung gegen Walker Demo Belohnung

Die Abbildung 4.3 zeigt mit der grünen Linie die Leistung der neuen Belohnungsfunktion und mit der pinken Linie die Leistung der Walker Demo Belohnungsfunktion. Nachfolgender Vergleich der Belohnungsfunktionen zeigt das die Ursprüngliche Belohnungsfunktion durch das Teilen mit der Zielgeschwindigkeit die Sensitivität der Funktion je nach Zielgeschwindigkeit beeinflusst. Daraus folgt das bei steigender Zielgeschwindigkeit eine größere Abweichung der Geschwindigkeit geduldet wird. Diese Anpassung verbessert die Generalisierung zwischen den wechselnden Geschwindigkeiten um ein vielfaches.

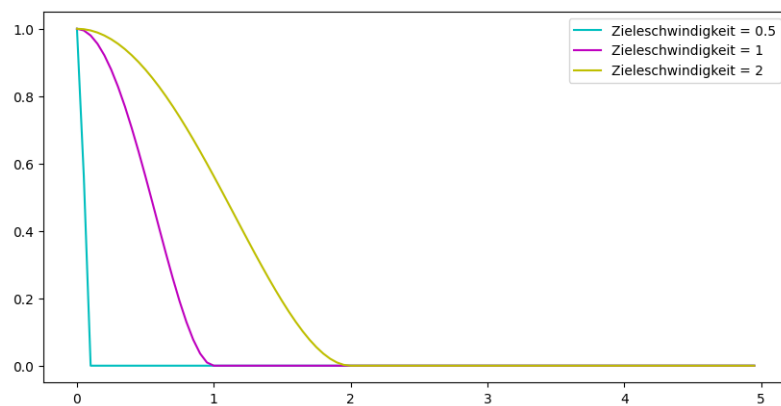


Abbildung 4.4.: Vergleich der Walker Demo Belohnungsfunktion unter verschiedenen Zielgeschwindigkeiten

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

chem die Funktion eine Belohnung von 0 annimmt, wurde auf ein minimum von 0.1 beschränkt. Somit konnte sicher gestellt werden das im Bereich der normalen Fortbewegung keine Veränderung auftritt. Bei allen Zielgeschwindigkeiten unter 0.1 wurde somit 0.1 als maximale Abweichung genutzt.

$$V_{\delta} = \text{Clip}(|\vec{Geschwindigkeit} - \vec{Zielgeschwindigkeit}|, 0, |\vec{Zielgeschwindigkeit}|)$$

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

stehen bleiben: wenn Distance < slowDownDistance dann je nach Distanz die Geschwindigkeit verringern -> agent bleibt vor ziel stehen bzw. kreist um ziel

extra Laufrichtungen: -getrennte Modelle und Modell live wechseln problem mit seitwärts laufen und vermutlich schlechter Übergang bei Wechsel -ein Modell mit Laufrichtung als one hot encoding in Beobachtung mit Lektion für verschiedene Laufrichtung -> kann nicht von vorwärtslaufen auf andere Richtung generalisieren bzw. anpassen (vergisst vorheriges verhalten) schränkt Bewegung auf feste Bewegungsrichtungen ein (vorwärts, rechts, links, rückwärts) -extra Ziel für Blickrichtung: -Ziel zufällig gesetzt mit Winkel Begrenzung von agent zu ziel -> Winkel ändert sich bei Bewegung -Blickziel setzen bei Episodenwechsel oder Ziel erreicht -> ändert sich zu häufig das Agent verhalten nicht lernt -Blickziel und Laufziel nur neu setzen wenn Durchschnittliche Blickbelohnung > Grenzwert -> zu schwer bzw. dauert zu lange, Agent veralten schon zu sehr vertieft um es groß zu ändern -Walker lernt auf Boden zu schauen da Blickrichtung nach unten näher an Blickrichtung Ziel ist wenn sich das Ziel hinter dem Walker befindet -Extra Belohnung für aufrechte Blickrichtung -Blickziel wird jedes Physikupdate neu gesetzt um Winkel gleich zu behalten -Blickziel neu setzen wenn bestimmte Zeit auf Ziel geschaut (mit Sphercast) -> funktioniert nicht schlecht aber bei längerem training hört der Agent auf das Blickziel zu erreichen

4.3. Mixamo Charakter

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

5. Fazit

Text

A. Anhang 1

Literaturverzeichnis

- [1] Arthur Juliani u. a. “Unity: A general platform for intelligent agents”. In: *arXiv preprint arXiv:1809.02627* (2020). URL: <https://arxiv.org/pdf/1809.02627.pdf>.
- [2] Xue Bin Peng u. a. “Deepmimic: Example-guided deep reinforcement learning of physics-based character skills”. In: *ACM Transactions On Graphics (TOG)* 37.4 (2018), S. 1–14.
- [3] Richard S Sutton und Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.