Entwicklung eines physikbasierten Charaktercontrollers mit Unity ML Agents

Software-Engineering
Fakultät für Informatik
der Hochschule Heilbronn

Bachelor-Thesis

vorgelegt von

Simon Grözinger
Matrikelnummer: 205047

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1. Einleitung

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

2. Grundlagen

Dieses Kapitel behandelt die Grundlagen der verwendeten Technologien, Packeten 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.[2]

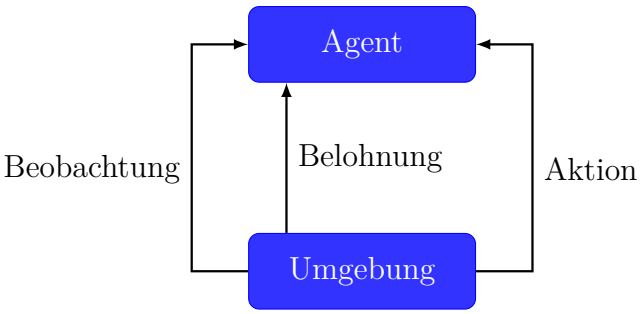

Abbildung 2.1.: Verstärkendes Lernen Ablauf

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, in dem maschinelle Lernalgorithmen und Funktionen für die Verwendung mit der Spieleumgebung Unity implementiert und kontinuierlich weiterentwickelt werden.

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-Netzwerkkommunikation, worüber Zustand der Simulationsumgebung in Unity, ausgewählte Aktionen des neuronalen Netzes in Python und weitere Werte für die Auswertung des Trainings ausgetauscht werden.[1]

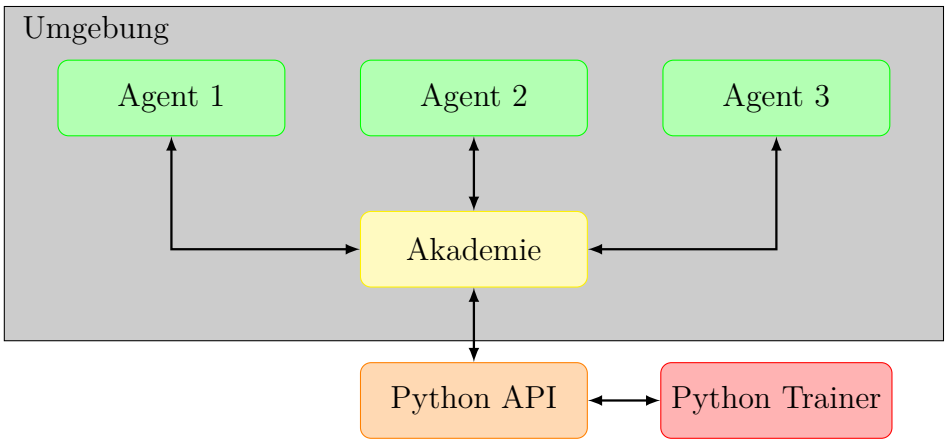

Abbildung 2.2.: Unity ML-Agents Aufbau

Die Umgebung in Abbildung 2.2 stellt den Aufbau der Unity Umgebung mit den ML-Agnents Komponenten dar. Das Unity-Paket enthält drei Grundlegenden Komponenten, die Akademie, Agenten und Sensoren. 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. Jeder

Agent braucht zusätzlich noch ein Verhalten, welches auf ein bereits trainiertes Modell oder auf eine Trainingskonfiguration verweist.

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 seperate Komponenten. Zahlenwerte sowie Vektoren und Quaternionen können jedoch auch direkt über eine Funktion im Agenten der Beobachtung angehängt werden.

Die Akademie ist eine einfach Instanz 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.[1]

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. Darurch sollten Codeausschnitte und Komponentenabbildungen in folgenden Kapiteln deutlich zu verstehen sein.

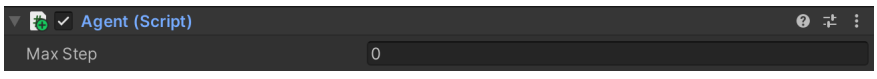

Abbildung 2.3.: Unity ML-Agents Agenten Komponente

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.

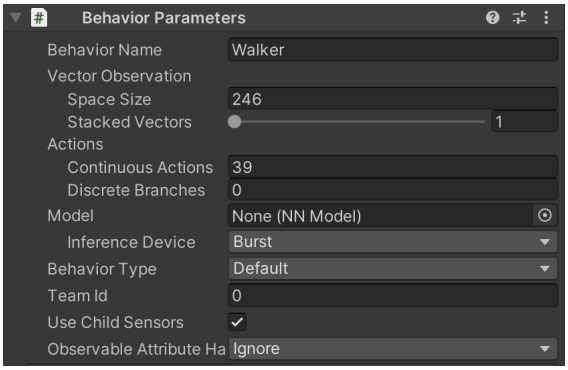

Abbildung 2.4.: Unity ML-Agents Verhalten Parameter Komponente

In Abbildung 2.5 ist die Verhaltens Parameter Komponente zusehen. Das Verhalten legt die Ein- und Ausgangsgröße des Modells fest und weist dem Agenten ein Verhalten zu.

- 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

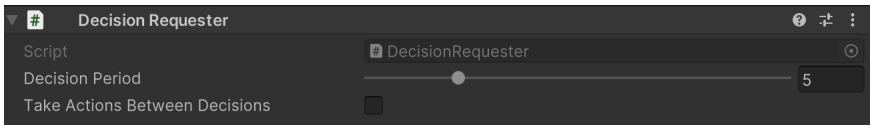

Abbildung 2.5.: Unity ML-Agents Entscheidung Anfragen Komponente

Die Komponente in Abbildung 2.5 fragt in regelmäßigen Abständen Entscheidungen an. Das bedeutet es wird eine Beobachtung erstellt, 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

```
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        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

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 Neuronalennetzes 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.

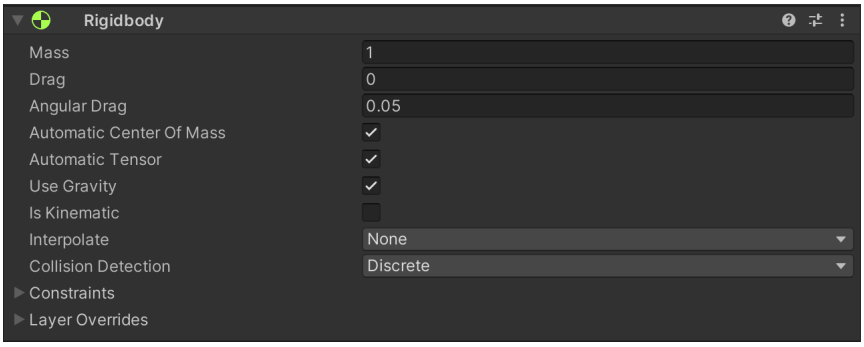

Abbildung 2.6.: Unity ML-Agents Physik Festkörper

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

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

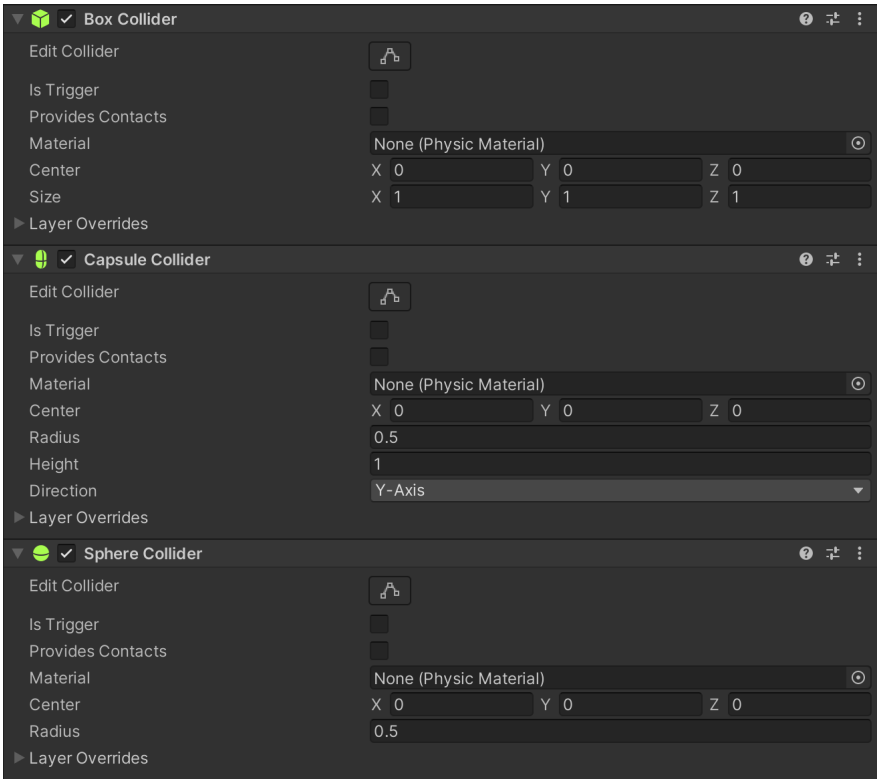

Abbildung 2.7.: Unity ML-Agents Physik Kollisionskomponenten

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.

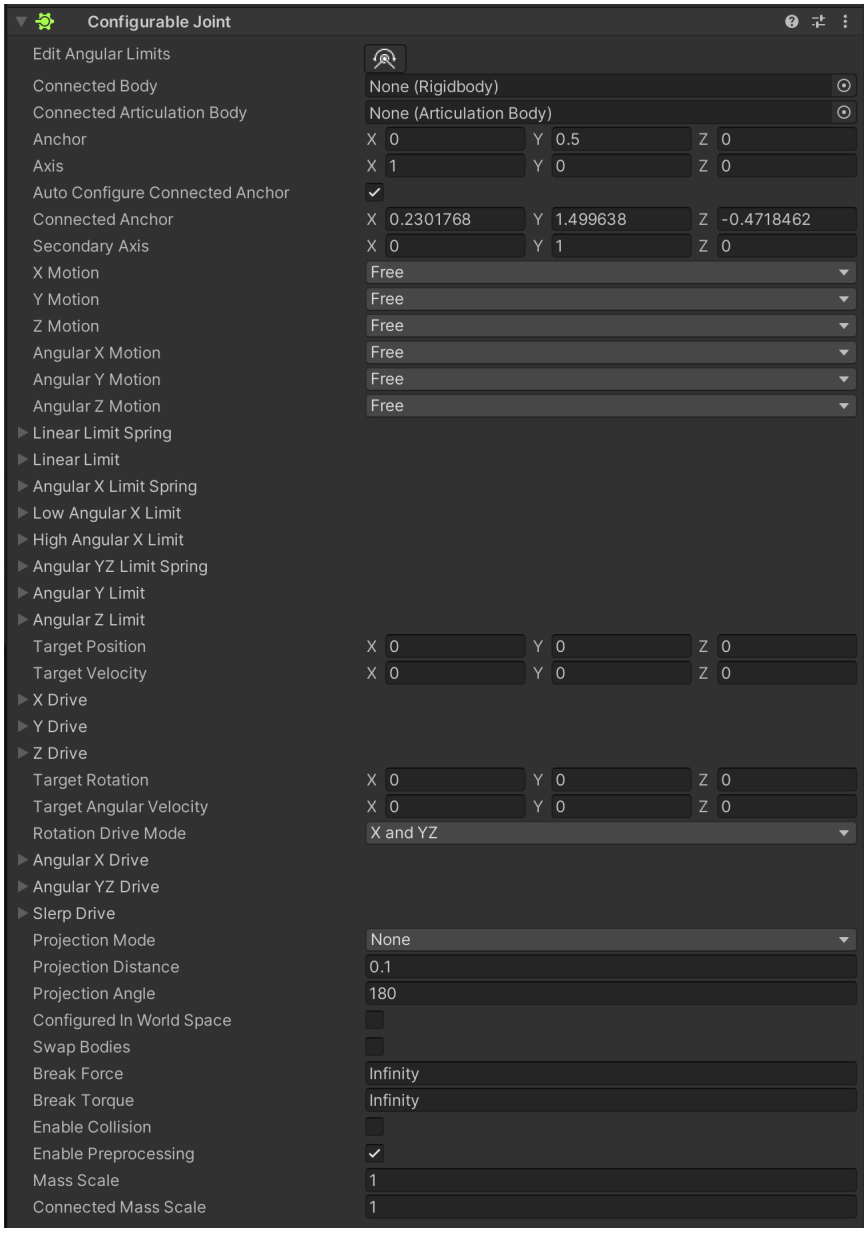

Abbildung 2.8.: Unity ML-Agents Physik Gelenk

- 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 Beweungs- 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 Grundlage 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.

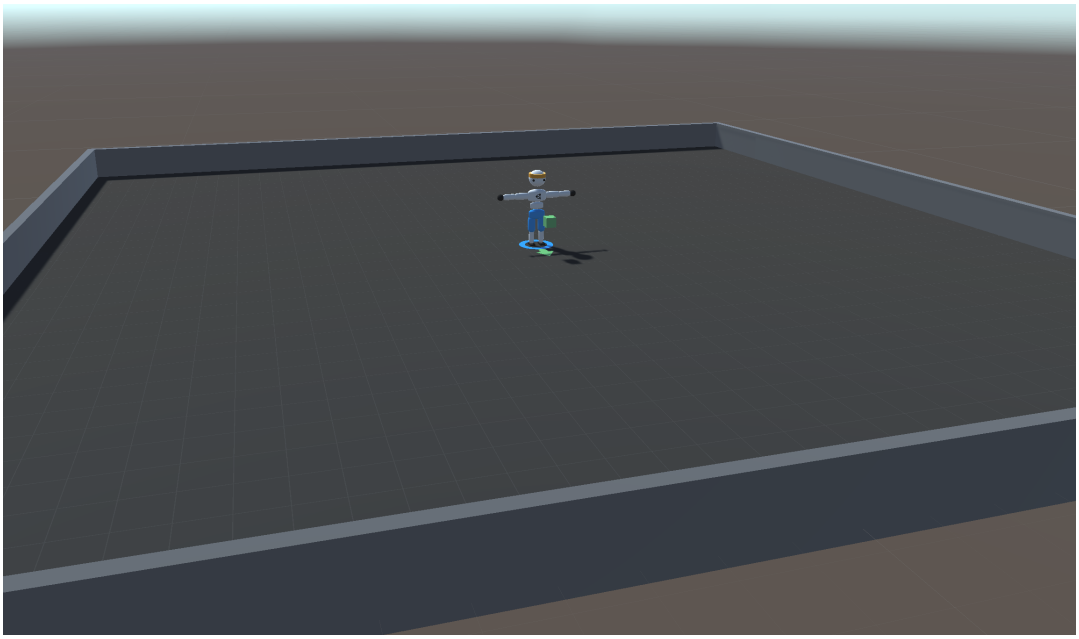

Abbildung 3.1.: Walker-Demo Szenenaufbau

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.

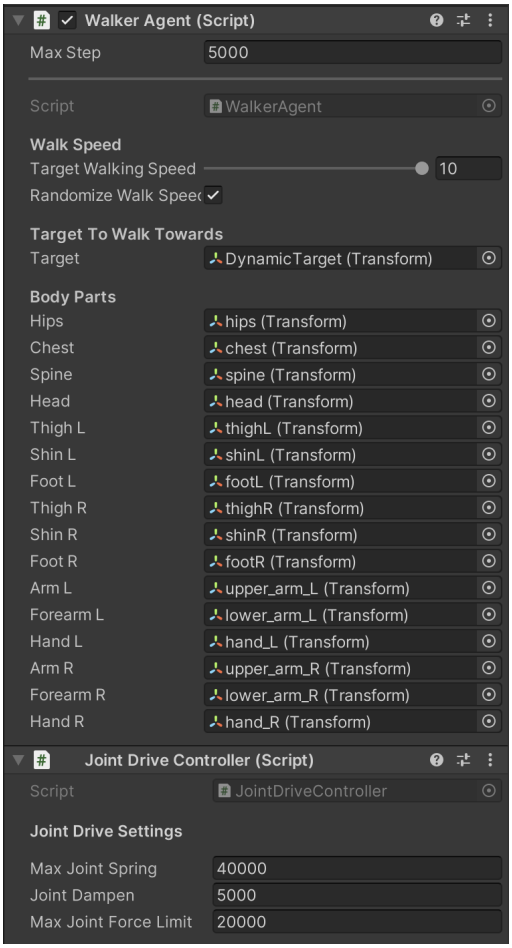

Abbildung 3.2.: Agent Konfiguration

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.

$$Q_{Hüfte} = Quaternion(Hüft\vec{r}ichtung - Zielr\vec{i}chtung)$$
$$Q_{Blick} = Quaternion(Blickr\vec{i}chtung - Zielr\vec{i}chtung)$$

$$S = \{$$
$$|Zielgeschw\vec{i}ndigkeit - Durchschnittsg\vec{e}schwindigkeit|,$$
$$Durchschnittsg\vec{e}schwindigkeit,$$
$$Zielgeschw\vec{i}ndigkeit,$$
$$Q_{Hüfte},$$
$$Q_{Blick},$$
$$Zielp\vec{o}sition,$$

$K_1, K_2, ..., K_n$
}

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

$A = \{$
$Rotationswinkel_1, Rotationswinkel_2, ..., Rotationswinkel_n$,
$Gelenkstärke_1, Gelenkstärke_2, ..., Gelenkstärke_n$
}

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

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.

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

Listing 4.1: Nutzersteuerung erster Prototyp

Die Funktion in Listing 4.1 liest den Input über das Unity InputSystem. Das Ziel wird relativ zur Hüfte gesetzt. Dieser Ansatz funktioniert grundsätzlich, hat aber noch ein grundlegendes Probleme und einige Einschränkungen. Das Problem ist dass die Zielposition in Abhängigkeit der Hüftrotation berechnet wird. Das hat zur folge das Schwankungen des Walkers Einfluss auf die Laufrichtung haben.

```
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.

Die Lösung für das Problem ist das einführen einer separaten Rotationskomponente für die Blickrichtung. Zu beginn wird die Blickrichtung mit der Vorwärtskomonente der Hüftrotation gleichgesetzt. Ausgehend von der Startrichtung wird dann über horizontalen Mausinput die Richtung angepasst (siehe Listing 4.3).

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

Listing 4.3: Erweiterung der Nutzersteuerung mit separater Blickrichtung

Mit dieser Implementierung lässt sich der Walker intuitiv steuern. 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. Die Vorwärtsbewegung in Kombination mit der Maussteuerung der Blickrichtung ermöglicht es sich nahezu überall hinzubewegen. Eine weitere große Einschränkung ist, das der Walker

nicht darauf trainiert ist stehen zu bleiben. Das resultiert darin das der Walker fällt sobald der Nutzer keinen Tastaturinput gibt.

4.2. Modell Anpassungen

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 Spherecast) -> funktioniert nicht schlecht aber bei längerem training hört der Agent auf das Blickziel zu erreichen

4.3. Mixamo Charakter

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

5. Fazit

Text

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