1 Danksagungen

Die erforderlichen Berechnungen wurden auf dem Linux-HPC-Cluster der Technischen Universität Dortmund (LiDO3) durchgeführt, in Teilen durch die Forschungsgroßgeräte-Initiative der Deutschen Forschungsgemeinschaft (DFG) unter der Projektnummer 271512359 gefördert.

2 Einleitung

2.1 Motivation und Problemstellung

2.2 Zielsetzung und Vorgehensweise

2.3 Übersicht

3 Grundlagen

(Die benutzten) Vortragsthemen vom Anfang hier als eigene Unterkapitel beschreiben.

3.1 Reinforcement Learning

Mit Reinforcement Learning (RL, deutsch: verstärkendes Lernen) werden bestimmte Varianten des maschinellen Lernens bezeichnet. Allgemein lernt beim Reinforcement Learning ein Agent eine Policy (deutsch: Strategie), um ein Problem zu lösen.
Abbildung 3.1 stellt den allgemeinen Kontrollfluss Zyklus eines Reinforcement Learning Prozesses dar. Der Agent ist in einer Umgebung (Environment) aktiv. In regelmäßigen Schritten erhält der Agent von der Umgebung einen aktuellen Zustand (State) in Form eines Vektors, sowie eine Belohnung (Reward), die entweder Null (d.h. keine Belohnung im aktuellen Schritt), oder ein positiver oder negativer Wert sein kann. Der Agent wendet die gelernte Policy auf den State an und berechnet damit eine Aktion. Die Aktion wird in der Umgebung ausgeführt. Danach übergibt die Umgebung den neuen State und die neue Belohnung an den Agenten.

3.1.1 Episoden und Trajektorien

Eine Episode beschreibt die Zeitspanne von der Initialisierung bis zur Terminierung der Umgebung. Eine Trajektorie beschreibt eine Reihe von Tupeln aus State, Aktion und Reward aus den einzelnen Schritten einer Episode. Die Gleichung 3.1 stellt die mathematische Schreibweise einer Trajektorie dar. Auf Basis gesammelter Trajektorien kann

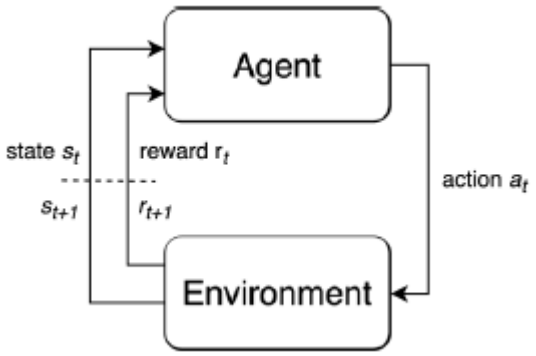

Abbildung 3.1: Reinforcement Learning Kontrollfluss [**FoundationsDeepRL**]

der Agent einen Lernvorgang durchführen und die Policy updaten.

$$\tau = (s_0, a_0, r_0), (s_1, a_1, r_1), \ldots, (s_n, a_n, r_n) \tag{3.1}$$

3.1.2 Policy-basierte & Value-basierte Algorithmen

Grundsätzlich wird im Reinforcement Learning zwischen Policy-basierten und Value-basierten Algorithmen unterschieden. Diese werden in den folgenden Abschnitten genauer beschrieben.

Policy-basierte Algorithmen

Bei Policy-basierten Algorithmen wählt eine Policy (π) die nächste Aktion ($a \sim \pi(a)$) aus, die in der Umgebung umgesetzt wird. Es können zwei Arten von Policies verwendet werden. Eine deterministische Policy bildet deterministisch einen Zustand s auf eine Aktion a ab (siehe Gleichung 3.2).

$$\pi(s) = a \tag{3.2}$$

In der Praxis werden fast ausschließlich stochastische Policies (siehe Gleichung 3.3) verwendet.

$$\pi(a|s) = \mathbb{P}_\pi \left[A = a | S = s\right] \tag{3.3}$$

Eine stochastische Policy ordnet für jede Aktion a eine Wahrscheinlichkeit unter Bedingung des Zustands s zu. Aus der entstehenden Verteilung wird dann die Aktion a gesamplet. Policy-basierte Algorithmen lernen die Funktion π, sodass das Objective maximiert wird. Policy-basierte Algorithmen können grundsätzlich auf beliebige Aktionstypen angewendet werden, kontinuierliche Aktionsräume sind möglich. Policy-basierte Algorithmen konvergieren garantiert zu einer lokalen, optimalen Policy. Allerdings sind Policy-basierte Algorithmen Sample-ineffizient und haben eine hohe Varianz.

Value-basierte Algorithmen

Bei Value-basierten Algorithmen wird eine Funktion zum Schätzen der Wertung eines Zustands bzw. eines Zustand-Aktion Tupels gelernt. Aus der Bewertung wird dann eine Policy generiert. Es gibt zwei grundlegende Value-Funktionen.

$$V^\pi(s) = \mathbb{E}_{s_0=s, \tau\sim\pi}\left[\sum_{t=0}^{T} \gamma^t r_t\right] \tag{3.4}$$

$$Q^\pi(s,a) = \mathbb{E}_{s_0=s, a_0=a, \tau\sim\pi}\left[\sum_{t=0}^{T} \gamma^t r_t\right] \tag{3.5}$$

Die V-Funktion ("Value-Function", siehe Gleichung 3.4) schätzt den Wert eines Zustandes anhand der diskontierten Rewards einer in diesem Zustand startenden Trajektorie. Die Q-Funktion ("Action-Value-Function", siehe Gleichung 3.5) schätzt den Wert eines Zustandes unter Durchführung einer Aktion anhand der diskontierten Rewards einer in

diesem Zustand startenden Trajektorie, die zuerst die gewählte Aktion durchführt. Bei reinen Value-basierten Algorithmen, wird bevorzugt die Action-Value-Funktion Q^π verwendet, da sich diese leichter in eine Policy umwandeln lässt. Value-basierte Algorithmen sind Sample-effizienter als Policy-based Algorithmen und haben eine niedrigere Varianz. Allerdings kann eine Konvergenz zu einem lokalen Optimum nicht garantiert werden. Die Standard-Methode für Value-basierte Algorithmen ist nur auf diskreten Aktionsräumen anwendbar. **??**

3.1.3 Optimierung

Der Lernvorgang eines Reinforcement Learning Systems erfolgt über den Policy Gradient.

$$\vartheta \leftarrow \vartheta + \alpha \nabla_\vartheta J(\pi_\vartheta) \tag{3.6}$$

Gleichung 3.6 zeigt die Optimierung einer Policy. Die optimierte Policy entsteht aus der alten Policy, indem eine Anpassung aus Gradient und Objective Funktion ($J(\pi_\vartheta)$) aufaddiert werden.

$$\nabla_\vartheta J(\pi_\vartheta) = \mathbb{E}_{\tau \sim \pi_\vartheta} \left[\sum_{t=0}^{T} R_t(\tau) \nabla_\vartheta log(\pi_\vartheta(a_t|s_t)) \right] ;\ R_t(\tau) = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} \tag{3.7}$$

Dabei werden die Wahrscheinlichkeiten $\pi\vartheta(a_t|s_t)$ der Policy angepasst. Ungünstige Aktionen ($R_t(\tau) < 0$) werden weniger wahrscheinlich. Günstige Aktionen ($R_t(\tau) > 0$) werdeen wahrscheinlicher.

3.1.4 Actor-Critic

Die Actor-Critic Methode ist eine Methode des Reinforcement Learning. Die Actor-Critic Methode kombiniert die Policy-Gradient und die Value-Function Methoden. Der Actor lernt eine Policy (Policy Gradient Methode) und nutzt für das Lernsignal eine durch den Critic gelernte Value-Function (Value-Function Methode). Durch den Critic wird so eine dichte Bewertungsfunktion erzeugt, auch wenn die echten Rewards aus dem Environment nur selten (d.h. an wenigen Zeitpunkten) vorkommen.

Advantage-Actor-Critic (A2C)

Advantage-Actor-Critic ist eine Variante der Actor-Critic Methode, bei der statt einer Value-Funktion eine Advantage-Funktion verwendet wird.

$$A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t) \tag{3.8}$$

Die Advantage-Funktion bewertet Zustand-Aktion Tupel im Vergleich zum Durchschnitt des Zustandes. Dadurch wird der erreichbare Reward in Relation zur Ausgangssituation gesetzt. Ein kleiner Reward aus einem schlechten Zustand wird somit ebenso gut gewertet, wie ein großer Reward aus einem guten Ausgangszustand. Überdurchschnittliche

Aktionen haben also einen positiven Advantage, während unterdurchschnittliche Aktionen einen negativen Advantage haben. Durch die Verwendung der Advantage-Funktion ändert sich die Formel für den Gradienten. Die Gradienten Formel für die A2C Methode ist in Gleichung 3.9 dargestellt.

$$\nabla_\vartheta J(\pi_\vartheta) = \mathbb{E}_t \left[A_t^\pi(s_t, a_t) \nabla_\vartheta log(\pi_\vartheta(a_t|s_t)) \right] \tag{3.9}$$

Für die Berechnung der Advantage-Funktion werden V^π und Q^π benötigt. Da es sehr ineffizient ist, sowohl Q^π, als auch V^π zu lernen, lernt der Critic nur V^π und schätzt auf dieser Grundlage dann Q^π. Für die Schätzung kann z.B. Generalized-Advantage-Estimation verwendet werden.

Generalized-Advantage-Estimation (GAE)

Generalized-Advantage-Estimation (GAE) ist eine Methode, den Advantage anhand einer gelernten V^π Funktion zu schätzen. GAE verwendet einen exponentiell gewichteten Durchschnitt von n-step Forward Return Advantages.

$$A_{NSTEP}^\pi = \left[(\sum_{i=0}^{n} \gamma^i r_{t+i}) + \gamma^{n+1} V^\pi(s_{t+n+1}) \right] - V^\pi(s_t) \tag{3.10}$$

Gleichung 3.10 stellt die Berechnung von n-step Forward Return Advantages dar. Dabei werden n Schritte an tatsächlichen Rewards gewertet. Ab dem $n+1$ Schritt wird mit einer gelernten V^π Funktion geschätzt. Die tatsächlichen Rewards sorgen für eine hohe Varianz und haben dafür kein Bias, da sie direkt aus der Umgebung gesamplet werden. Die V^π Funktion stellt eine Erwartung über alle möglichen Trajektorien dar und weist damit eine geringe Varianz auf. Da die V^π Funktion gelernt wurde, entsteht dabei Bias. Mit kleinen Werten für n haben n-step Forward Return Advantages eine geringe Varianz bei hohem Bias, mit großen Werten für n haben n-step Forward Return Advantages eine hohe Varianz bei geringem Bias (Bias-Variance Trade-Off). Allgemeine Aussagen über die Wahl des n Parameters sind schwierig.
Generalized Advantage Estimation hebt die explizite Wahl von n aus, indem n-step Forward Return Advantages für verschiedene n Werte gewichtet. kombiniert werden.

$$A_{GAE(\gamma,\lambda)}^\pi(s_t, a_t) = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l};\ \delta_t = r_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t) \tag{3.11}$$

Die allgemeine Formel für GAE ist in Gleichung 3.11 abgebildet. Die Gewichtung wird durch die Decay-Rate $\gamma \in [0;1]$ und den Discount-Factor $\lambda \in [0;1]$ parametrisiert. In der Praxis wird statt der unendlichen Summe eine Summe bis zum Ende der verfügbaren Trajektorie verwendet.

3.2 PPO

Proximal Policy Optimization (PPO) ist ein Reinforcement Learning Algorithmus, der die Actor-Critic Methode umsetzt.

3.2.1 Performance Collapse

Ein Problem von Reinforcement Learning Algorithmen ist der Performance Collapse (Leistungseinbruch). Performance Collapse bedeutet, dass die Returns der Umgebung im Lernvorgang plötzlich einbrechen, da z.B. die Policy über ein lokales Optimum heraus optimiert wurde und der Gradient nun zu einem schlechteren lokalen Optimum konvergiert. Die Ursache für das Problem ist der indirekte Ansatz der Policies. Ein RL Algorithmus manipuliert die Parameter der Policy, um diese zu verändern. Kleine Veränderungen in den Parametern können dabei große Veränderungen in der Policy auslösen.

3.2.2 Clipping

Es gibt zwei theoretische Varianten des PPO Algorithmus. Die erste Variante baut direkt auf dem Trust Region Policy Optimization Algorithmus (TRPO) auf und verwendet eine Adaptive KL Penalty, um die Schrittgröße einzuschränken. In der Praxis wird jedoch nur die zweite Variante von PPO verwendet, da diese in Tests bessere Ergebnisse erzielt und leichter zu implementieren ist. Die zweite Variante nutzt ein Clipped Surrogate Objective.

$$J^{CLIP}(\vartheta) = \mathbb{E}_t \left[\min(r_t(\vartheta)A_t, clip(r_t(\vartheta), 1-\epsilon, 1+\epsilon)A_t \right] \tag{3.12}$$

Die Gleichung 3.12 zeigt das Clipped Surrogate Objective, das vom PPO Algorithmus maximiert wird. Eine Veränderung außerhalb des Bereichs $[1-\epsilon; 1+\epsilon]$ wird dabei geclippt (abgeschnitten) und somit die Schrittgröße eingeschränkt.

$$J^{CLIP+VF+S}(\vartheta) = \mathbb{E}_t \left[J_t^{CLIP} - c_1 L_t^{VF} + c_2 S[\pi_\vartheta(s_t)] \right] \tag{3.13}$$

Die für die Clipped Surrogate Objective Variante des PPO Algorithmus verwendete Loss Funktion wird in Gleichung 3.13 gezeigt. Zusätzlich zum Clipped Surrogate Objective wird der Loss der Value-Funktion und ein Entropie Bonus zur Steigerung der Exploration berechnet. In der Praxis wird das Objective negiert, damit Gradient Descent statt Gradient Ascent durchgeführt werden kann.

Algorithm 1 PPO Pseudocode Algorithmus

```
for episode=0...MAX do
    for actor=1...N do
        Sample Trajectory for T time steps with π_{ϑ_old}
        Compute advantage estimates A_1,...,A_T with π_{ϑ_old}
    Collect all states in batch of size N*T
    for epoch=1...K do
        for minibatch of size m in batch do
            Optimize Clipped Surrogate Objective L w.r.t. ϑ
    ϑ_old = ϑ
```

Algorithmus Algorithmus 1 zeigt den Pseudocode des PPO Algorithmus. Für jede Episode wird zuerst pro Actor eine Trajektorie in der Umgebung gesamplet. Dabei werden die Aktionen jeweils durch die aktuelle Policy $\pi_{\vartheta_{old}}$ bestimmt. Für diese Trajektorie werden die zugehörigen Advantages mit GAE berechnet. Mit den gesampleten Trajektorien wird die Policy in Epochen trainiert. In jeder Epoche werden Minibatches aus dem gesampleten Batch generiert. Mit den Daten der Minibatches wird das Clipped Surrogate Objective optimiert.

4 Verwandte Arbeiten

Beschreiben warum andere Quellen nicht ausreichend waren und weshalb der eigene Ansatz jene fehlende Themen ergänzt.

5 Fachliches Vorgehen

Keine technischen Details (wie z.B. Implementierung)

5.1 Projektorganisation

Wie sind wir vorgegangen... auf alles bezogen. Wer hat an welchen Kapiteln mitgearbeitet.

5.1.1 Creature Animator

Die Gruppe der Creature Animator hat sich in zwei Untergruppen aufgeteilt. In der ersten Phase hat sich die erste Untegruppe damit beschäftigt, den ML-Agents Walker in eine neue Trainingsumgebung einzubauen und die Skripte dynamischer zu gestalten, damit diese in der zweiten Arbeitsphase verwendet und erweitert werden konnten. Währenddessen versuchte die andere Untergruppe den ML-Agents Walker das Schlagen beizubringen. Die beiden Untergruppen haben sich wöchentlich mittwochs getroffen, um von ihren Fortschritten und Problemen zu berichten. Dabei wurden die Ergebnisse in Protokollen festgehalten, welche in einem GitHub Wiki abgelegt wurden.

In der zweiten Phase, welche nach der Bereitstellung der ersten generierten Kreaturen von der Creature Generator Gruppe begann, veränderten sich die Aufgabenbereiche der beiden Untergruppen. Die „Schlagen“-Gruppe arbeitete seit dem an einer Erweiterung von Nero-RL, sodass Nero-RL anstelle von ML-Agents zum Trainieren der Kreaturen genutzt werden kann. Die Aufgabe der „Trainingsumgebung“-Gruppe war es den neuen Kreaturen das Fortbewegen beizubringen und der Creature Generator Gruppe Feedback zu den Kreaturen zu geben. Dabei arbeiteten die Gruppenmitglieder an verschiedenen kleineren Aufgaben. Jan beschäftigte sich mit dem Training und dem Finden und Ausprobieren neuer Rewardfunktionen, Nils arbeitete an der dynamischen Generierung von Arenen und dem Landen von Konfigurationeinstellungen aus Dateien und Carsten testete verschiedene Parameter aus und implementierte das Erstellen von NavMeshes zur Laufzeit. In der zweiten Phase lösten „On-Demand“-Treffen die regelmäßigen Treffen zwischen den beiden Untergruppen ab, um mehr Zeit zum Arbeiten an den Aufgaben zu haben. Zudem wurden anstelle der Treffen nur noch die wichtigsten Punkte protokolliert. Ansonsten wurden Probleme und Fehler direkt als Issue in den entsprechenden GitHub Repositories hinterlegt.

„Trainingsumgebung/Movement“-Gruppe	„Schlagen/Nero-RL“-Gruppe
Carsten Kellner	Jannik Stadtler
Jan Beier	Niklas Haldorn
Nils Dunker	

Tabelle 5.1: Die zwei Untergruppen und ihre Mitglieder

5.2 Creature Generation

5.3 Creature Animation

5.3.1 Trainingsumgebung

Als Grundlage für die eigene Trainingsumgebung diente die Trainingsumgebung des ML-Agent Walkers. Bei genauerer Betrachtung der Trainingsumgebung des ML-Agent Walkers stellte sich sehr schnell raus, dass diese für unsere Anforderungen zu statisch war, da es zum Beispiel nicht möglich war, die verwendete Kreatur einfach gegen eine andere Kreatur auszutauschen. Zudem mussten neue Arenen aufwendig per Hand erstellt werden und die Kreaturen konnten nur auf einer flachen Ebene trainiert werden. Daher wurde eine eigene dynamischere Trainingsumgebung erstellt, die vor allem die zuvor genannten Punkte umsetzt. Sowohl die Anzahl der Arenen als auch die Kreatur können in der neuen Trainingsumgebung einfach eingestellt werden. Somit können die Arenen vollständig zur Laufzeit generiert werden. Zudem besteht die Möglichkeit neben flachen Terrain auch unebenes Terrain zu verwenden.

Zunächst wurde der ML-Agents Walker zum Testen der neuen Trainingsumgebung verwendet, damit die Kreatur als Fehlerquelle ausgeschlossen werden konnte. Nachdem die Tests mit dem ML-Agents Walker erfolgreich waren, wurde der ML-Agents Walker durch eine neue Kreatur ersetzt, welche mit Hilfe des L-Systems zuvor generiert wurde. Die neue Kreatur wurde ausgiebig in der neuen Trainingsumgebung getestet und mit dem ML-Agents Walker verglichen. Durch das dadurch gewonnene Feedback konnte die Creature Generator Gruppe Anpassungen und Verbesserungen an der Kreatur vornehmen.

Trotz den vielen Änderungen an der Kreatur war es nicht möglich, die Kreatur aus dem L-System zum Laufen zu bringen. Aus diesem Grund ersetzte eine andere Methode, welche von Jona und Markus umgesetzt wurde, das L-System. Mit der neuen Methode wurde auch der Creature-Generator direkt in die Trainingsumgebung eingebunden. Dies ermöglichte es, schnell verschiedene Kreaturen zu testen. Die Bugreports und Featurerequests zum Generator werden direkt als Issue in das entsprechende GitHub Repository geschrieben und werden gegebenenfalls im Jour Fixe oder über Discord besprochen.

LiDO3

Das RL-Training benötigt viele Rechenressourcen. Die ersten Trainingstests mit der ML-Agents-Walker-Umgebung haben gezeigt, dass eine Trainingsdauer von über einen Tag

auf aktueller Hardware zu erwarten ist. Deswegen muss das Training auf einen Server laufen. Als besondere Anforderungen benötigen die Server eine Nvidia Grafikkarte, um mit CUDA[1] pytorch[2] zu beschleunigen.

Aufgrund der Einschränkungen stand nur LiDO3[3], der HPC der TU Dortmund, da andere Rechenknoten wie z. B. Noctua 2[4] von der Universität Paderborn Grafikarten nur für Forschungsprojekte mit bestimmter Reichweite zur Verfügung stellen. Der Zugang zu LiDO3 wurde durch unsere PG-Betreuer gestellt.

Konfiguration

Da die Trainingungebung auf den ML-Agent-Walker basiert, waren viele Konfiguration fest-codiert. Zuerst wurden diese über den Unity-Inspektor änderbar gemacht. Als mit dem aktiven Training auf LIDO begonnen wurde, stellte sich diese Methode als nicht flexible genug heraus. Die Trainingsumgebung musste für jede Änderung neu gebaut werden. Deshalb wurde ein neues System geschrieben, welches über Dateien die Konfiguration dynamisch lädt.

5.3.2 Training

Generalisierung

- PPO
- ML-Agents
- Nero?

5.4 Terraingeneration

[1] https://developer.nvidia.com/cuda-zone
[2] https://pytorch.org/
[3] https://www.lido.tu-dortmund.de/cms/de/home/
[4] https://pc2.uni-paderborn.de/hpc-services/available-systems/noctua2

6 Technische Umsetzung

Bei der Erläuterung der Wahl der Hierarchie für Knochen nur deskriptiv darauf eingehen; keine Details oder Begründung erforderlich. **Dies übernimmt die Creature-Generator Gruppe.** Eingehen auf Status Quo.

6.1 Creature Animation

6.1.1 Trainingsumgebung

Im Folgenden soll der Aufbau der Trainingsumgebung beschrieben werden, welche es erlaubt verschiedenste Kreaturen ohne große Anpassungen zu trainieren. Die Umgebung ist dabei aus den folgenden Klassen aufgebaut:

- `DynamicEnviormentGenerator`
  - `TerrainGenerator`
  - Verschiedenen Konfigurationsdateien
  - `DebugScript`
- allen anderen modifizierten ML-Agents Skripten

In diesen Abschnitt wird nur auf den Aufbaue des `DynamicEnviormentGenerator` sowie dessen Hilfsklassen und nicht auf die ML-Agent-Skripte eingegangen. Die Hilfsklassen sind der `TerrainGenerator`, `GenericConfig` und dessen Implementierungen sowie das `DebugScript`. Erstere ist verantwortlich für die Generierung des Terrains, die Config-Dateien laden dynamisch die Einstellungen aus einer Datei und das letzte Skript beinhaltet hilfreiche Debug-Einstellungen. Die grundsätzliche Idee der Trainingsumgebung stammt von dem ML-Agents-Walker. Da an diesem keine Versuche mit Unterschiedlichen Umgebungen und Kreaturen durchgeführt wurden, ist der Aufbau des Projekts nicht dynamisch genug.

Dynamic Enviorment Generator

Zur dynamischen Umsetzung der Trainingsarena werden alle Objekte zur Laufzeit erstellt. Die Generierung der Arena läuft dann wie folgt ab:

1. Erstellen von n Arenen, wobei n eine zu setzende Variable ist.
2. Füge ein Ziel für die Kreatur in die Arena ein

3. Generiere die Kreatur

Die einzelnen (Teil)-Arenen bestehen aus einem Container-Objekt unter dem ein Terrain und vier Wall-Prefabs angeordnet sind. Diese Prefabs und weitere Elemente wie Texturen werden dynamisch aus einem Ressourcen-Ordner geladen, damit möglichst wenige zusätzliche Konfigurationen den Editor verkomplizieren. Das Terrain wird mit leeren Terraindaten vorinitialisiert und später befüllt. Hierbei kann die Position des Container-Objects in der Szenen wie folgt berechnet werden:

$$\begin{pmatrix} \lceil \frac{\text{Anzahl der Arenen}}{\sqrt{\text{Anzahl der Arenen}}} \rceil \\ 0 \\ \text{Anzahl der Arenen} \mod \sqrt{\text{Anzahl der Arenen}} \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{6.1}$$

Alle anderen Objektpositionen müssen danach neu im lokalen Koordinatensystem gesetzt werden. Da die Unity-Standard-Texturen sehr hell sind sind, werden die Texturen bei der Initialisierung mit ML-Agents-Texturen, welche dunkler sind, getauscht. An das Terrain werden zuletzt Collider und ein `TerrainGenerator`-Skript angefügt.

In Schritt 2. der Arenagenerierung muss beachtet werden, dass nach dem Erstellen des Zielobjekts das `WalkTargetScript` hinzugefügt wird. Am Ende des Erstellungsprozesses wird der Walker erstellt. Hierzu wird ein von den Creature-Generator-Team bereitgestelltes Paket[1] benutzt. Das Paket stellt ein Klasse bereit, welche mit zwei Skript-Objekte konfiguriert wird. Zusätzlich wird ein seed übergeben, welcher reproduzierbare Kreaturen erlaubt. Die erstelle Kreatur muss danach mit den entsprechenden ML-Agent-Skripten versehen werden. Hierzu wird ein `WalkerAgent` Objekt als String übergeben. Dies ermöglicht es, mehrere unterschiedliche Agent-Skripte durch eine Änderung im Editor zu setzen. Somit können Reward-Funktion und Observation für zwei unterschiedliche Trainingsversuche getrennt, in eigenen Dateien, entwickelt werden.

TerrainGenerator

Da ein typisches Spieleterrain im Gegensatz zum ML-Agents-Walker-Terrain nicht flach ist, wurde ein neues Objekt erstellt, welches sowohl die Generierung von Hindernissen, als auch eines unebenen Bodens erlaubt. Um ein möglichst natürlich erscheinendes Terrain zu erzeugen wird ein Perlin-Noise verwendet. Dieses spiegelt jeweils die Höhe des Terrains an einen spezifischen Punkt wider. Im späteren Projektverlauf wurde dieses Skript durch den Terraingenerator des dazugehörigen Teams ersetzt.

Konfigurationsobjekte

Da sich die statische Konfiguration des ML-Agents-Walker als problematisch erwies, wurde die Konfiguration über die Laufzeit des Projekts dynamischer gestaltet. Zuerst wurden alle Konfigurationen im `DynamicEnviormentGenerator` gespeichert. Was unübersichtlich war und zu ständigen neubauen des Projektes führte. Deshalb wurde eine `GenericConfig`

[1] https://github.com/PG649-3D-RPG/Creature-Generation

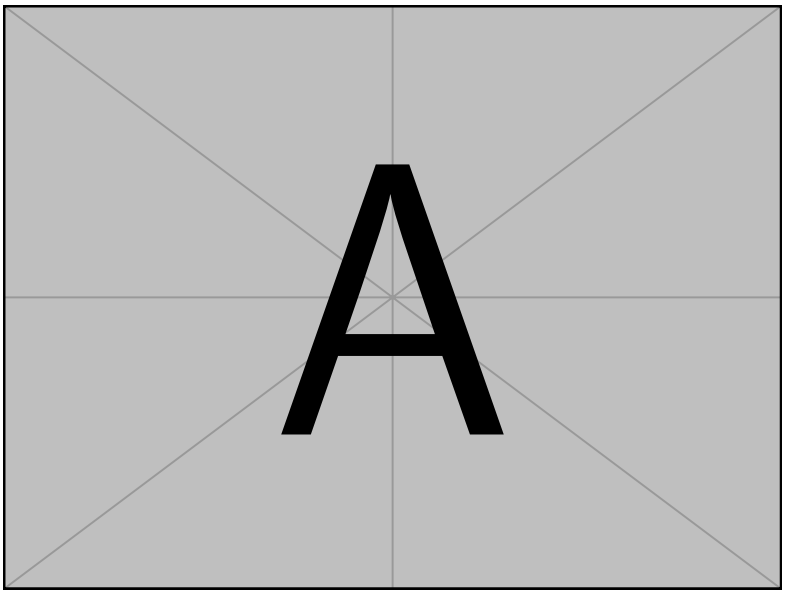

Abbildung 6.1: Konfigurationsmöglichkeiten des `DynamicEnviormentGenerator` im Unity-Editor.

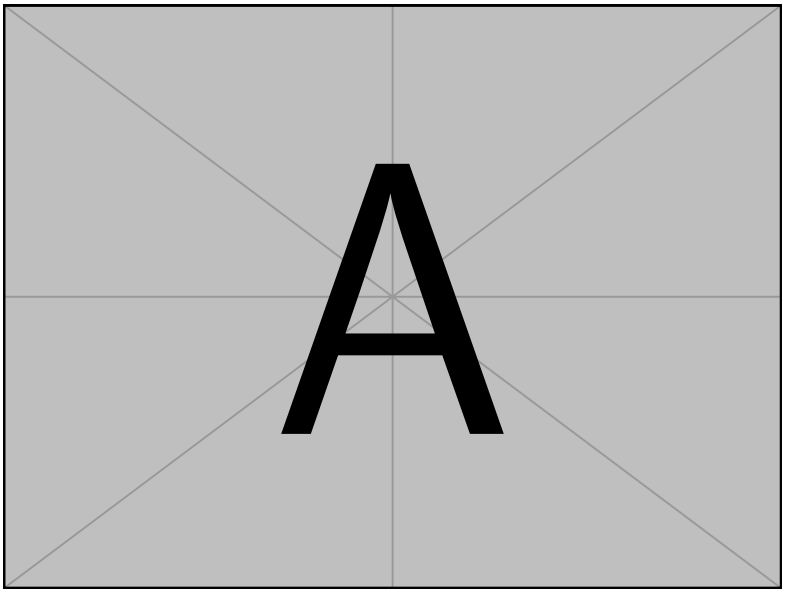

Abbildung 6.2: Ein Beispiel der generierten Trainingsumgebung mit mehreren Arenen.

Klasse eingeführt, welche die im Editor eingestellten Optionen für die einzelnen Teilbereiche Terrain, Arena und ML-Agent in Json-Format in den Streaming-Asset-Ordner speichert. Da dieser Ordner beim bauen des Projekts in das fertige Spiel übertragen wird, sind diese Konfigurationen automatisiert dort vorhanden.

Im Fall, dass das Spiel ohne Editor gestartet wird, was meist beim Training der Fall ist, lädt das generische Objekt aus den Json-Dateien die Einstellungen und ersetzt die Editorkonfiguration damit. Hierdurch ist ein ändern der Konfiguration des Spiels ohne neu-erstellen der Binärdateien ermöglicht. Diese Konfigurationsart fügt Abhängigkeiten zu dem Unity eigenen JsonUtility[2] hinzu.

texttt im Titel?

6.1.2 Erweiterung der Agent-Klasse

Als eine Erweiterung der `Agent`-Klasse von ML-Agents stellt die `GenericAgent`-Klasse das Verbindungsstück zwischen dem ML-Framework und der Unity-Engine dar. Im Folgenden wird der Aufbau der Klasse `GenericAgent` sowie derer Hilfsklassen `JointDriveController`, `BodyPart`, `OrientationCubeController` und `WalkTargetScript` erläutert und die Funktionalität dieser Klassen erklärt. Zur Veranschaulichung befindet sich in Abbildung 6.3 ein UML-Diagram. Der Aufbau dieser Klassen orientiert sich dabei sehr stark an die Implementierung des ML-Agents Walker.

GenericAgent

Die kontrollende Instanz einer konkreten Trainingsumgebung ist die `GenericAgent`-Klasse. Diese ist dazu in der Lage, mit dem Modell des ML-Frameworks zu interagieren, also sowohl Beobachtungen der Trainingsumgebung weiterzugeben also auch die Ausgaben des Modells anzunehmen (und zu verarbeiten). Außerdem ist die Klasse für die Instandhaltung der Trainingsumgebung verantwortlich, indem sie Events der Umgebung verarbeitet (z.B. das Erreichen des Targets oder das Verlassen des zugänglichen Bereiches) und ggf. spezifizierte Routinen wie das Zurücksetzen der Umgebung durführt. Schließlich muss die `GenericAgent`-Klasse noch die Rewards für die Trainingsumgebung verteilen. Zu diesem Zweck ist die Klasse als *abstract* definiert, da diese Rewardfunktionen stark von der Aufgabe des Agents abhängig sind. So benötigt zum Beispiel ein Agent, welcher ein bestimmtest Ziel möglichst schnell erreichen soll eine andere Reward-Funktion als ein Agent, welcher sich möglichst gut vor dem Spieler verstecken soll. Verschiedene Agents können so ohne Redundanz einfach als eine Erweiterung der `GenericAgent`-Klasse implementiert werden.

Mehr auf Reward-Funktionen eingehen oder erst bei konkreten Agents?

JointDriveController und BodyPart

Um die Ingame-Repräsentation (also die generierte Creature) des Agents zu kontrollieren, besitzt der `GenericAgent` einen `JointDriveController`. Bei der Initialisierung der Trainingsumgebung wrappt der `JointDriveController` die verschiedenen Unity-Transforms

[2] https://docs.unity3d.com/ScriptReference/JsonUtility.html

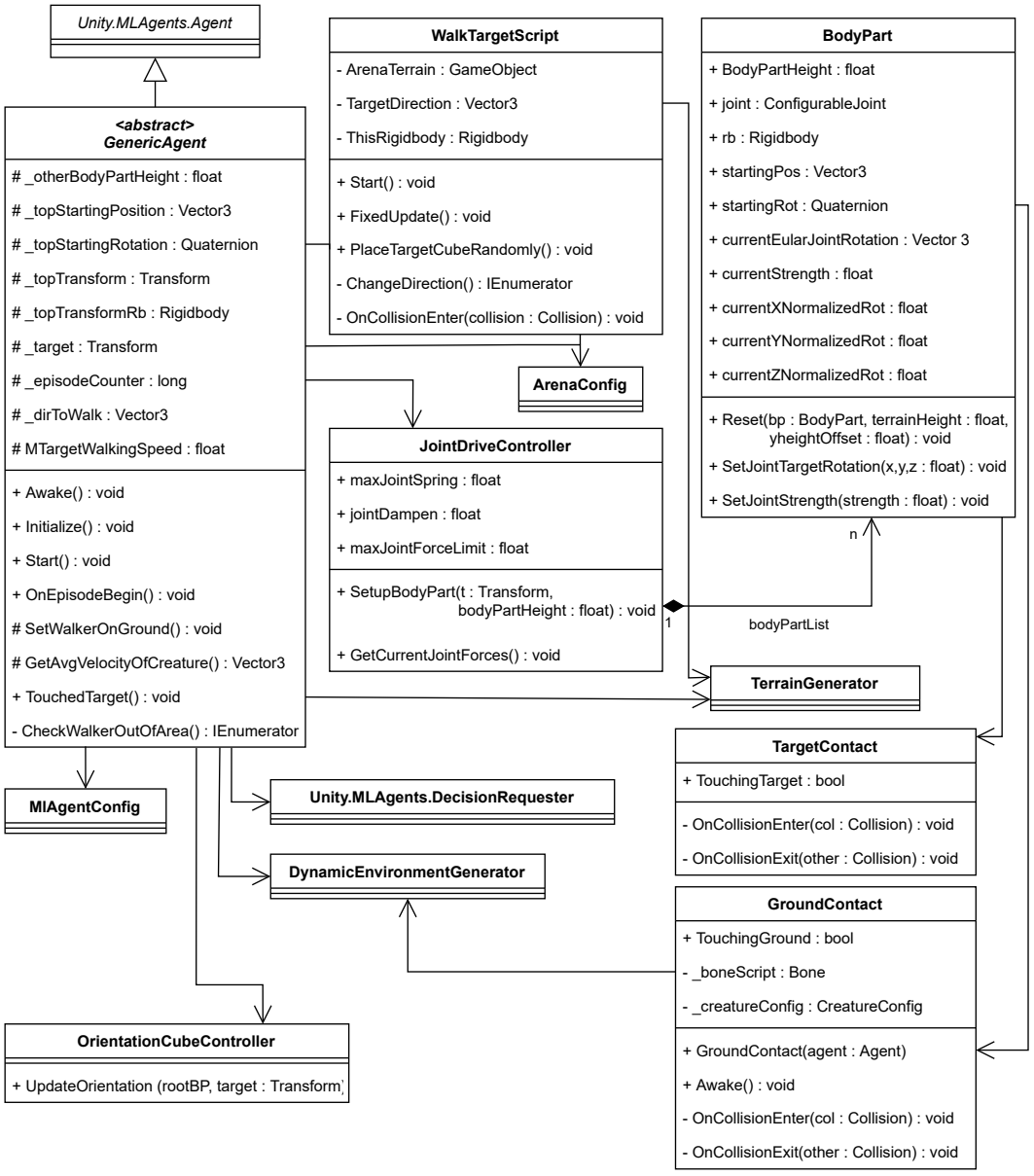

Abbildung 6.3: UML-Diagram der GenericAgent-Klasse und weitere relevante Klassen

der Creature in Instanzen der Hilfsklasse `BodyPart`. Die Klasse `BodyPart` gibt uns leichten Zugang zu häufig benötigten Funktionalitäten, wie zum Beispiel das Zurücksetzen oder Steuern des Transform. Auch besitzt ein `BodyPart` nützliche Informationen über das jeweilige Transform, welche dem ML-Modell weitergegeben werden können. Nach der Initialisierung stellt der `JointDriveController` nur noch das Verbindungsstück zwischen der `GenericAgent`-Klasse und der verschiedenen `BodyPart`-Instanzen dar.

OrientationCubeController

Da sich das Target des Agenten potentiell überall innerhalb einer großen (und weitgehend unbekannten) Ingame-Umgebung befinden kann, ist es hilfreich, die gezielte Laufrichtung des Agenten an eine einheitliche Position zu platzieren. Hierfür besitzt jeder Agent einen sogenannten *OrientationCube*, welcher an einer festen Position relativ zum Agenten steht und sich lediglich in die Richtung des Targets dreht. So kann der Agent (und infolgedessen das ML-Modell) einfach den OrientationCube referenzieren, um die Laufrichtung zu bestimmen. Der `OrientationCubeController` stellt dafür die Reorientierungsfunktion des OrientationCubes bereit.

WalkTargetScript

Größtenteils unabhängig vom Agenten agiert das Target mithilfe des `WalkTargetScript`. Die Hauptaufgabe des Scripts ist es, das Target zu steuern (sowohl Neuplatzierung bei einem Reset, als auch normale Bewegungen innerhalb einer Episode) und beim Eintreten eines CollisionEvents zwischen dem Target und dem Agenten den Agenten zu notifizieren. Da zurzeit das Target nur aus einer Kugel besteht, ist komplizierteres Verhalten nicht notwendig.

6.1.3 NeroRL & ML-Agents

Für das Training der Kreaturen kommt das Python-basierte Machine Learning Framework neroRL zum Einsatz. Die Auswahl ist in Kapitel **??** beschrieben. neroRL verbindet sich mit der Unity-seitig implementierten ML-Agents API und implementiert den PPO Algorithmus (siehe Kapitel **??**) auf Basis von PyTorch. Kapitel **??** beschreibt für die technische Umsetzung relevante Probleme, die die ursprüngliche Version von neroRL mit sich bringt. In diesem Kapitel wird die technische Umsetzung der Veränderungen an neroRL beschrieben, durch die das Framework für unser Projekt einsetzbar ist.

Kontinuierliche Aktionsräume

In der ursprünglichen Version von neroRL werden ausschließlich diskrete und multidiskrete Aktionsräume unterstützt. Unsere Kreaturen benötigen kontinuierliche Aktionen. Die Klasse `ContinuousActionPolicy` implementiert einen Head (Kopf) für das neuronale Netzwerk, der kontinuierliche Aktionen umsetzt. Für jede Aktion des zugrundeliegenden Aktionsraumes werden μ (der Mittelwert) und σ (die Standardabweichung)

gelernt. Daraus wird eine Normalverteilung generiert, aus der dann ein tatsächliches Aktionstupel gesamplet werden kann. Als Verteilung wird die `Normal` Implementierung einer Gaußschen Normalverteilung aus PyTorch verwendet.

Multi-Agent Builds

Abhängig von den verfügbaren Rechenressourcen lassen sich maximal ca. 16 Builds parallel ausführen (LiDo3 cgpu01 Knoten mit 2 Intel Xeon E5-2640v4 und 2 NVIDIA Tesla K40). Um mehr Daten parallel zu sammeln, haben wir neroRL so erweitert, dass in einem Build mehrere Umgebungen mit jeweils einem Agenten parallel laufen können. Die von neroRL verwendeten Buffer Systeme verwenden dafür die Dimensionen $[worker][agent][timestep][content]$. Über $[worker]$ und $[agent]$ werden alle Daten genau einem Agenten in einem der ausgeführten Builds zugeordnet. Für die $[timestep]$ Dimension wird außerdem für jeden Build und Agenten festgehalten, welches der aktuelle Zeitschritt ist. Da Umgebungen zu unterschiedlichen Zeitpunkten terminieren und zurückgesetzt werden, ist es möglich, dass die Zeitschritte zwischen verschiedenen Agenten nicht synchronisiert sind. Sobald genug Daten für die definierte Batchgröße gesampelt sind, werden diese von den verschiedenen Builds und Agenten eingesammelt und in einem Batch kombiniert, das dann an den Trainingsalgorithmus übergeben wird.

Optimierung der Trainingsqualität

Nach Fertigstellung der Implementierung der Unterstützung von kontinuierlichen Aktionen zeigen die ersten Trainingsergebnisse unzureichende Ergebnisse. Während das Training der ML-Agents Walker Beispielumgebung mit ML-Agents nach ca. 30 Millionen Schritten einen durchschnittlichen Reward von ca. 2000 erreicht, erreicht neroRL auch nach über 150 Millionen Schritten nur einen Reward von ca. 220. Eine qualitative visuelle Bewertung des gelernten Verhaltens zeigt, dass der Walker sich zwar gezielt auf die erste Position des Ziels zubewegt, allerdings nach dem Verschieben des Ziels nicht in der Lage ist, sich umzudrehen. Um die Qualität des Trainings mit neroRL zu steigern, werden verschiedene Optimierungen verwendet.

- **Normalisierung der Observationen:** Die Observationen werden durch den Sampler automatisch normalisiert, um das Training zu stabiliseren und vor ausreißenden Werten zu schützen.

- **Squashing:** Grundsätzlich können die aus dem Netzwerk gesampleten kontinuierlichen Aktionen im Intervall $[-\infty; \infty]$ liegen. Auf Unity liegen die Aktionen im Intervall $[-1; 1]$ und werden dementsprechend abgeschnitten. In einigen Fällen hat sich Tanh-Squashing als eine Methode erwiesen, um die Trainingsqualität mit kontinuierlichen Aktionen zu steigern und die Aktionen auf das Intervall $[-1; 1]$ zu projizieren, anstatt diese abzuschneiden (vgl. []). Unsere Implementierung von Tanh-Squashing wurde jedoch verworfen, da diese konsistent das Exploding-Gradients-Problem ausgelöst hat und das Training somit fehlgeschlagen ist.

6.1.4 LiDO3

Wie bereits erwähnt, werden die Berechnungen jeweils auf den HPC der TU Dortmund ausgeführt. Um auf LiDO3 zu arbeiten wird mit Hilfe eines Gatewayservers auf das Cluster zugegriffen. Der Zugriff ist ausschließlich über das TU Dortmund Netzwerk möglich. Über den Gatewayserver kann ein Zugriff auf die Rechenressourcen direkt über die Shell oder über Skripte angefordert werden. Da die Shell-Methode einen dauerhaften Login erfordern würde, wird mit Skripten gearbeitet. Diese bestehen aus Konfigurationen für LiDO3 und den eigentlich Programmteil, welcher ausgeführt werden soll. LiDO3 nutzt als Jobmanager Slurm, weshalb die Skripte die Slurm-Syntax nutzen. Eine ausführliche Beschreibung die LiDO3 Konfiguration findet sich im Benutzerhandbuch[1];

```
#!/bin/bash -l
#SBATCH -C cgpu01
#SBATCH -c 20
#SBATCH --mem=40G
#SBATCH --gres=gpu:2
#SBATCH --partition=long
#SBATCH --time=48:00:00
#SBATCH --job-name=pg_k40
#SBATCH --output=/work/USER/log/log_%A.log
#SBATCH --signal=B:SIGQUIT@120
#SBATCH --mail-user=OUR_MAIL@tu-dortmund.de
#SBATCH --mail-type=ALL
#-------------------------------------

GAME_NAME="GAME_NAME"
GAME_PATH="/work/USER/games/$GAME_NAME"


module purge
module load nvidia/cuda/11.1.1

source /work/USER/anaconda3/bin/activate
conda activate /work/mmarplei/grudelpg649/k40_env

chmod -R 771 $GAME_PATH
cd $GAME_PATH

srun mlagents-learn /work/smnidunk/games/config/Walker.yaml --run-id=$GAME_NAME --env
```

In dem Beispielskript 6.1.4 sind Anweisungen an die LiDO-Umgebung jeweils mit einem Kommentarzeichen gefolgt von *SBATCH* gekennzeichnet. Die Konfiguration wird

so gewählt, dass eine maximale Laufzeit mit exklusiven Ressourcenrechten auf den Rechenknoten besteht. Zusätzlich muss sichergestellt werden, dass eine Grafikkarte zur Verfügung steht. Diese stehen auf den *cgpu01*-Rechenknoten mit jeweils 20 CPU-Kernen und 48 Gigabyte RAM zur Verfügung. Die maximale Laufzeit des Prozesses ist bei den GPU-Knoten auf *long* begrenzt, was 48 Stunden entspricht. Es wird jeweils ein Log mitgeschrieben, aus dem der Trainingsfortschritt gelesen werden kann und bei besonderen Ereignissen eine Mail geschickt, um sofort benachrichtigt zu werden, falls der Job fertig ist oder fehlschlägt.

Kompatibilitätsprobleme

Um das beschriebene Skript auszuführen, muss auf LiDO3 eine ML-Agents-Umgebung installiert werden. Dabei handelt es sich um ein Python Umgebung, mit PyTorch und CUDA. In dem Slurm-Skript 6.1.4 ist die Einrichtung einer funktionierenden Umgebung dargestellt.

```
// LIDO UMGEBUNGSVARIABLEN
module purge
module load nvidia/cuda/11.1.1

source <anaconda3-path>/bin/activate
conda activate <env_to_install>
conda install torchvision torchaudio cudatoolkit=11.1 -c pytorch
python -m pip install mlagents==0.29.0 --force-reinstall
python -m pip install /work/mmarplei/grudelpg649/torch-1.10.0a0+git3c15822-cp39-cp39-linux_x86_
```

Für die Python-Installation wurde auf Anaconda[3] zurückgegriffen. Die installierte Anaconda-Arbeitsumgebung kann für die folgenden Schritte genutzt werden, indem die Slurm-Skripte diese am Anfang laden. CUDA kann als Kernelmodul in verschiedenen Versionen geladen werden oder per Anaconda installiert werden.

Problematisch ist die Installation von PyTorch, da ab Version 1.5 die Installationsbinärdateien keine Unterstützung für die von LiDO3 genutzten NVIDIA Tesla K40 Grafikarten bietet. Es besteht die Möglichkeit PyTorch zu bauen um die Unterstürzung zu erhalten. Dies musste für unsere Arbeitsumgebung nicht gemacht werden, da die PG-Betreuer ein Paket mit einer für LiDO funktionierenden PyTorch-Version von einer vorherigen PG zur Verfügung stellen konnten. Wie in 6.1.4 dargestellt müssen zuerst die Abhängigkeiten von PyTorch, dann ML-Agents und zuletzt die spezielle PyTorch Version installiert werden, da sonst die Abhängigkeiten Probleme bereiten.

[3] https://www.anaconda.com/

7 Vorläufige Ergebnisse

Unterkapitel nach Erkenntnissen. Metrik nach der bewertet wird erörtern. Objektiv ohne Wertung der Ergebnisse.

7.1 Diskussion

Diskussion der Ergebnisse in Bezug auf die initiale Zielsetzung.

8 Ausblick