

Projektgruppe:

Entwicklung eines 3D RPG Videospiels mittels prozeduraler Inhaltsgenerierung und Deep Reinforcement Learning

Jan Beier Nils Dunker Leonard Fricke Niklas Haldorn
Kay Heider Jona Lukas Heinrichs Mathieu Herkersdorf
Carsten Kellner Markus Mügge Thomas Rysch
Jannik Stadtler Tom Voellmer

29. September 2022



„PROJEKTGRUPPE: ENTWICKLUNG EINES 3D RPG VIDEOSPIELS MITTELS
PROZEDURALER INHALTSGENERIERUNG UND DEEP REINFORCEMENT
LEARNING“

SS 2022 - WS 2022/2023 · TU Dortmund

Inhaltsverzeichnis

1	Danksagungen	1
2	Einleitung	3
2.1	Motivation und Problemstellung	3
2.2	Zielsetzung und Vorgehensweise	3
2.3	Übersicht	5
3	Grundlagen	7
3.1	Reinforcement Learning	7
3.1.1	Episoden und Trajektorien	8
3.1.2	Policy-basierte & Value-basierte Algorithmen	8
3.1.3	Optimierung	9
3.1.4	Actor-Critic	9
3.1.5	PPO	11
4	Verwandte Arbeiten	13
5	Fachliches Vorgehen	15
5.1	Projektorganisation	15
5.1.1	Creature Animator	17
5.2	Creature Generation	18
5.3	Creature Animation	18
5.3.1	Trainingsumgebung	18
5.3.2	Training	20
5.4	Terraingeneration	20
6	Technische Umsetzung	21
6.1	Creature Animation	21
6.1.1	Trainingsumgebung	21
6.1.2	Erweiterung der Agent-Klasse	24
6.1.3	NeroRL & ML-Agents	26
6.1.4	LiDO3	28
7	Vorläufige Ergebnisse	31
7.1	Diskussion	31
8	Ausblick	33

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

Im Rahmen der Projektgruppe 649 nehmen insgesamt 12 Teilnehmer an der Planung und Entwicklung des prozedural generierten 3D Rollenspiels teil.

Für die Umsetzung des Projektes ist es vorgesehen die verschiedenen Bereiche des Rollenspiels aufzuteilen, sodass potentiell in Kleingruppen parallel an diesen gearbeitet werden kann. Diese Bereiche umfassen folgende Themengebiete: die Generierung von Spielleveln, die Evaluation der generierten Level, Generierung der Monster, Fortbewegung der Monster und zuletzt die Strategie bzw. Verhaltensweise der Monster. Innerhalb jeder dieser Bereiche sollen bestimmte Anforderungen, welche an die Gruppe vorgegeben sind, erfüllt sein:

Die Generierung von Spielleveln

- Die Generierung der Spiellevel sollte prozedural durchgeführt werden: z.B. KD-Trees, L-Systeme, Space-Partitioning
- Die Spiellevel enthalten Böden, Wände und auch Spawn-Points für Objekte und NPCs
- Gleichzeitig sollen die Komplexität, Größe und der Schwierigkeitsgrad der Level parametrisierbar sein

Evaluation der Generierten Level

- Die Level/Spielwelten sollen anhand vorbestimmter Metriken evaluiert werden, wie z.B.: Lösbarkeit der Level, Schwierigkeitsgrad
- Folgende Ansätze sind dafür vorgeschlagen: Imitation Learning, Deep Reinforcement Learning

Generierung der Monster

- In diesem Bereich sind viele Freiheiten gelassen worden, da die Generierung der Monster auf viele unterschiedliche Weisen durchgeführt werden kann

2 Einleitung

- Relevant dabei ist nur, dass die Erstellung von NPCs algorithmus-basiert ist und dass innerhalb von Unity die von Unity bereitgestellten Joints verwendet werden sollten
- Eine Orientierungshilfe dabei kann der Unity-MLAgents-Walker sein

Fortbewegung der Monster

- Animationen sollen nicht händisch erstellt werden
- Mit Hilfe von Deep Reinforcement Learning können die Monster lernen sich zu bewegen
- Der Agent wählt die Kräfte aus, welche auf seine Joints ausgeübt werden
- Je nach lösen einer spezifischen Aufgabe wird der Agent dann belohnt
- Dabei könnten Inverse Kinematiken nützlich sein

Strategie bzw. Verhaltensweise der Monster

- Klassische Ansätze (high level) wären hier die Behavior Trees oder auch State machines
- Währenddessen lernende Ansätze (low level) wären Imitation Learning und Deep Reinforcement learning

Um ein grundlegendes Verständnis aller Teilnehmenden für alle Bereiche zu schaffen, werden im ersten Monat in einer Seminarphase alle Bereiche auf Kleingruppen, bestehend aus 3 Teilnehmern, aufgeteilt und potentielle Ansätze für die Umsetzung der Gebiete für ein Videospiel erforscht. Während dieser Seminarphase werden dann nacheinander die Themen der Gruppen den restlichen Teilnehmenden vorgestellt. Somit ist eine Verständnisgrundlage für alle Beteiligten geschaffen, sodass basierend auf dem allgemeinen Kenntnisstand mit der eigentlichen Entwicklung des Spieles angefangen werden kann.

Ein besonderer Schwerpunkt soll dabei auf die Themen der **Generierung der Monster** und der **Fortbewegung der Monster** gelegt werden, da diese die Basis für die restlichen Aufgaben bilden sollten. Es wurde sich absichtlich dafür entschieden von Anfang an kein klares Design der späteren Spielwelt, Monster und des Spielercharakters zu definieren, sodass sich daraus keine Einschränkungen für die initiale Implementierungsphase ergeben sollten. Vielmehr sollte aus den Ergebnissen der initialen Phase das spätere Spieldesign abgeleitet werden. Damit also dieser Grundstein gelegt werden konnte, wurde sich dafür entschieden die **Generierung der Monster** und die **Fortbewegung der Monster** der Spielwelt als erstes zu priorisieren und somit die 12 Teilnehmer der Projektgruppe für die Anfangsphase in zwei Untergruppen aufzuteilen: die **Creature-Generator**, bestehend aus 7 Mitgliedern, und die **Creature-Animator**, bestehend aus 5 Mitgliedern. Es wird antizipiert, dass sich mit fortschreitender Zeit die Gruppen, sobald die Basis für das Spiel besteht, in weitere (kleinere) Gruppen aufteilen werden. Das Ziel ist somit, parallel an mehreren Themen gleichzeitig arbeiten zu können und

somit effizient Fortschritt zu machen. Währenddessen muss eine deutliche Kommunikation zwischen den (Unter-)Gruppen bestehen um die Themen miteinander abstimmen zu können, vor allem für die Anfangsphase zwischen den **Creature-Generator** und den **Creature-Animator**.

Für diese Anfangsphase wurde evaluiert, die Kreatur-Generierung so einfach wie möglich zu halten um das Beibringen von Bewegungen durch die **Creature-Animator** Gruppe so einfach wie möglich zu gestalten und die Freiheitsgrade auf einem Minimum zu halten. Dafür ist das Vorhaben zweibeinige, humanoide Kreaturen und auch vierbeinige Kreaturen erzeugen zu können. Zusätzliche Körperteile wie beispielsweise Flügel, mehrere Arme oder Beine und Ähnliche Extras werden weggelassen, damit also keine zusätzlichen Freiheitsgrade hinzukommen und das Lernen der Bewegung der Kreatur gegebenenfalls erschweren würden. Trotzdem soll später evaluiert werden, ob solche ergänzenden Körperteile hinzugefügt werden könnten, sobald die Basisstruktur, also das stabile Erzeugen und Lernen der Bewegungen der Kreaturen, besteht. Außerdem wird durch diesen Ansatz der Spielentwurf so schlicht wie möglich gehalten, sodass keine potentiellen Einschränkungen bezüglich des Spieldesigns existieren und damit die spätere Gestaltung des Spiels basierend auf den dann bestehenden Funktionalitäten entschieden werden kann.

2.3 Übersicht

Zuerst werden in dem zweiten Kapitel (3) alle Grundlagen beschrieben in Bezug auf die angeführten Themenbereiche der Projektgruppe welche in der Seminarphase gegenseitig vorgestellt wurden (2.2). Basierend auf diesen Themengebieten werden dann in Kapitel 4 verwandte Arbeiten und Literatur vorgestellt, welche sowohl während der Evaluation in der Seminarphase erforscht wurden, als auch in der späteren Erarbeitung weiterer Themen und Algorithmen (Kapitel 5 und 6) ausgemacht wurden. Sobald die Grundlagen über die Themen der späteren fachlichen und technischen Vorgehensweise geklärt sind, wird in Kapitel 5 das fachliche Vorgehen beschrieben, welches sich während der Entwicklungsphase der verschiedenen Bereiche (s. Kapitel 2.2) erschlossen hat. Nachdem die fachbezogene Vorgehensweise bekannt ist, wird im Anschluss die Umsetzung in Kapitel 6 evaluiert, wie die in Kapitel 5 beschriebenen Verfahren und Algorithmen technisch umgesetzt sind. Anschließend werden in Kapitel 7 die (**Zwischen-**)Ergebnisse präsentiert und diskutiert (7.1), **hier später für den Endbericht ausführen wie genau und wodrauf bezogen die Ergebnisse diskutiert werden**. Schließlich wird in Kapitel 8 zusammengefasst, auf welche (Forschungs-)Bereiche die Erkenntnisse dieser Studie ausgeweitet werden können. **Weiter ausführen?**

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. Reinforcement Learning ist ein breites Forschungsgebiet. In diesem Grundlagenkapitel werden nur die Grundlagen des Reinforcement Learning beschrieben, die im Rahmen der Projektgruppe eingesetzt wurden. Eine Übersicht über weitere Varianten des Reinforcement Learning findet sich z.B. in den Büchern [1][2].

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.[1]

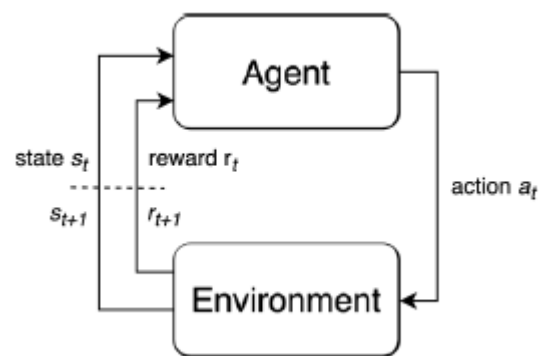


Abbildung 3.1: Reinforcement Learning Kontrollfluss [1]

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 der Agent einen Lernvorgang durchführen und die Policy updaten.[1]

$$\tau = (s_0, a_0, r_0), (s_1, a_1, r_1), \dots, (s_n, a_n, r_n) \quad (3.1)$$

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. Diese Policy bzw. ihre Parameter werden direkt durch den Algorithmus gelernt. 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). [1]

$$\pi(s) = a \quad (3.2)$$

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

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

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. [2]

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.

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

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

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. [1] 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.

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

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.

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

Dabei werden die Wahrscheinlichkeiten $\pi_{\vartheta}(a_t | s_t)$ der Policy angepasst. Ungünstige Aktionen ($R_t(\tau) < 0$) werden weniger wahrscheinlich. Günstige Aktionen ($R_t(\tau) > 0$) werden wahrscheinlicher. [1]

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-Function eine Advantage-Funktion verwendet wird.

$$A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t) \quad (3.8)$$

3 Grundlagen

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.

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

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. [1]

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.

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

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 gesampelt 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.

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

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]

3.1.5 PPO

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

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 hinaus 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. [1]

Clipping

Es gibt zwei theoretische Varianten des PPO Algorithmus. Die erste Variante baut direkt auf dem Trust Region Policy Optimization Algorithmus (TRPO) [4] 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.

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

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.

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

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. [5]

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.

3 Grundlagen

```
1 for episode=0...MAX do
2   for actor=1...N do
3     Sample Trajectory for T time steps with  $\pi_{\vartheta_{old}}$ 
4     Compute advantage estimates  $A_1, \dots, A_T$  with  $\pi_{\vartheta_{old}}$ 
5     Collect all states in batch of size  $N*T$ 
6     for epoch=1...K do
7       for minibatch of size m in batch do
8         Optimize Clipped Surrogate Objective L w.r.t.  $\vartheta$ 
9    $\vartheta_{old} = \vartheta$ 
```

Algorithmus 1: PPO Pseudocode Algorithmus [1]

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

Um koordiniert das Ziel der Projektgruppe zu erreichen, spielt die Projektorganisation bei einer Gruppe von 12 Teilnehmern eine wichtige Rolle. Im Laufe der Projektgruppe wurden Untergruppen für verschiedene Bereiche gebildet. Außerdem übernahmen einige Teilnehmer Rollen im Rahmen der Projektleitung. Die primäre Gruppenaufteilung ist in Tabelle 5.1 abgebildet. Initial wurde die Projektgruppe in die Untergruppen Creature-Generation und Creature-Animation aufgeteilt.

- **Creature-Generation:** Die Creature-Generation Gruppe wendet verschiedene Methoden der prozeduralen Inhaltsgenerierung an. Damit sollen Skelette für Kreaturen generiert werden und auf Basis dieser Skelette Skinning hinzugefügt werden. Außerdem ist die Creature-Generation Gruppe für die prozedurale Generierung einer Spielwelt verantwortlich.
- **Creature-Animation:** Die Creature-Animation Gruppe wendet Reinforcement Learning an, um Bewegungsmodelle für die von der Creature-Generation Gruppe generierten Kreaturen zu trainieren.

Die weitere Organisation innerhalb der Untergruppen wird in den Abschnitten 5.1.1 und ?? genauer beschrieben.

Projektleitung Aus den initialen Gruppen, der Creature-Generation und der Creature-Animation Gruppe, wurden jeweils zwei Projektmanager einberufen. Diese sind in Ta-

(Creature-)Generation		(Creature-)Animation	
Creature-Generation	Terrain-Generation	Creature-Animation	neroRL
Leonard Fricke	Kay Heider	Jan Beier	Niklas Haldorn
Jona Lukas Heinrichs	Tom Voellmer	Nils Dunker	Jannik Stadtler
Mathieu Herkersdorf		Carsten Kellner	
Markus Mügge			
Thomas Rysch			

Tabelle 5.1: Primäre Gruppenaufteilung der Projektgruppe

belle 5.1 gepunktet unterstrichen. Außerdem wurde Thomas Rysch als Projektleiter ernannt. Der Projektleiter sollte darin eine Moderator- und Organisationsrolle einnehmen, die Gruppen und Themen zusammen- und im Überblick behalten und somit potentielle Konflikte frühzeitig erkennen und auflösen. Die Projektmanager übernehmen eine analoge Aufgabe im Rahmen ihrer jeweiligen Gruppen und bilden somit gemeinsam mit dem Projektleiter eine Struktur in der die Übersicht über aktuelle Themen einfach beibehalten werden kann.

Jour-Fixe Wöchentlich findet ein Jour-Fixe statt um gemeinsam über die aktuellen Entwicklungen zu sprechen und Herausforderungen gemeinsam anzugehen und zu lösen. Im Anschluss an das Jour-Fixe treffen sich die vier Projektmanager und der Projektleiter, bereiten für den nächsten Termin des Jour-Fixe die zu besprechenden Themen vor und klären gegebenenfalls organisatorische Einzelheiten um somit die restliche Gruppe der Teilnehmer zu entlasten. Während der Jour-Fixe wird von jedem der Teilnehmer abwechselnd eine Dokumentation des Treffens manifestiert um somit Ergebnisse aber auch ToDo's festhalten und einsehen zu können. Für das wöchentliche Treffen wurde ein Tagesprotokoll (Fig.) genutzt, welches von dem Projektleiter durch Bildschirmübertragung an die Teilnehmer präsentiert und durchgesprochen wurde. Das Tagesprotokoll konnte jederzeit durch jeden Teilnehmer modifiziert und ergänzt werden, falls zu besprechende Themen durch den jeweiligen Teilnehmer aufgekommen sind.

Organisationswerkzeuge Für die weitere Organisation (der Entwicklung und Implementierung) der Projektgruppe wurden verschiedene Tools und Hilfsmittel eingesetzt, die im Folgenden beschrieben werden.

- **Discord:** Discord wurde als Basis-Kommunikationsmittel genutzt. Hier wurden für die entsprechenden Phasen und Themenbereiche, sowie Gruppen eigene Text- und Sprach-Kanäle erstellt (Fig. (?)). Hier wurden auch aktuelle (organisatorische) Themen aufgegriffen und besprochen. Der wöchentliche Jour-Fixe Termin wurde ebenfalls in Discord abgehalten.
- **GitHub:** GitHub wurde als Entwicklungs-, Repository- und Speicherplattform genutzt. Die Dokumentationen des wöchentlichen Jour-Fixe sind im GitHub-Wiki abgelegt und zusätzlich in einem Discord Textkanal gespeichert. Somit wurde implizit auch durch die Verfügbarkeit auf zwei Plattformen ein Backup der Dokumentationen realisiert, falls im worst-case eine der beiden Plattformen ausfallen sollte. Für die Projektgruppe wurde eine GitHub-Organisation gegründet (Link zur Organisation). Pro Gruppe bzw. Themenbereich wurde zunächst ein Repository erstellt. Nach Fertigstellung einzelner Versionen werden diese als Pakete exportiert und in einem Main-Repository (Link) zum fertigen Spiel zusammengestellt.
- **Google-Kalender:** Um insbesondere in der vorlesungsfreien Zeit eine reibungslose Organisation und Ressourcenallozierung gewährleisten zu können, wurde Google-Kalender genutzt, in dem sich alle Teilnehmenden in einem Kalender gesammelt

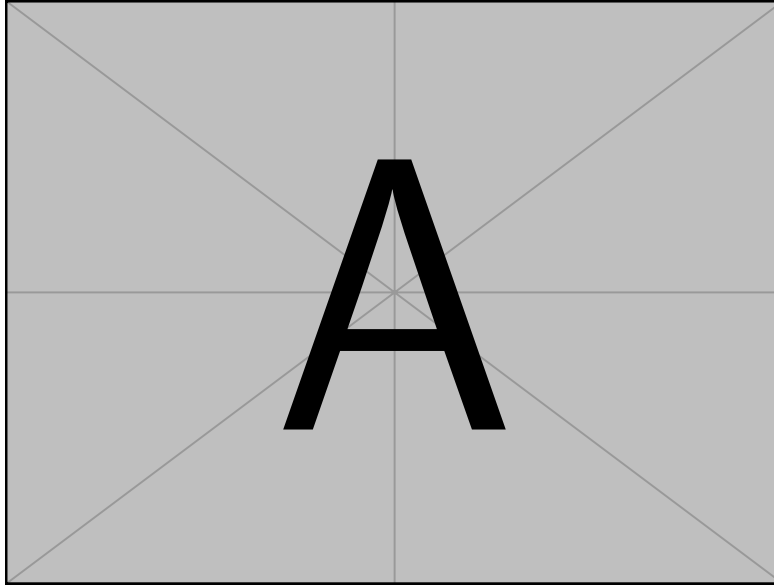


Abbildung 5.1: Stand des Status-Quo am (Datum)

haben und ihre jeweiligen Universitäts- und Urlaubsbezogenen Termine eingetragen haben.

- **Status-Quo-Übersicht:** In der Anfangszeit der Projektgruppe hat sich herausgestellt, dass nur schwierig der Überblick über den Gesamtfortschritt der Projektgruppe zu halten ist. Um den Fortschritt insbesondere auch außerhalb der Projektorganisation zu festzuhalten, wurde ein Status-Quo Dokument entwickelt. Dieses stellt bereits fertiggestellte, momentan laufende und potentiell in Zukunft zu behandelnde Projekte und Themengebiete übersichtlich dar. Dafür wurde mit Hilfe des Tools Graphviz ein simpler Graphen erzeugt der zudem am Anfang jedes Monats durch die Management-Runde aktuell gehalten wurde. Abbildung 5.1 zeigt den Status-Quo zum Zeitpunkt der Erstellung des Abschlussberichtes.
- **Projekt-Timeline:** Zu Beginn der Projektgruppe wurde eine Projekt-Timeline erstellt, welche für das jeweilige Semester die kurz- und langfristigen Aufgaben der entsprechenden Gruppen in tabellarischer Form festhalten sollte. Da dieses Hilfsmittel jedoch in dem ersten Semester keine Verwendung fand, wurde es zu Anfang der zweiten Hälfte der Projektgruppe als obsolet eingestuft und verworfen. Die kurz- und langfristigen Aufgaben wurden stattdessen über das Status-Quo Dokument und die wöchentlichen Jour-Fixe Dokumentationen festgehalten.

5.1.1 Creature Animator

Die Gruppe der Creature Animator hat sich in zwei Untergruppen aufgeteilt. In der ersten Phase hat sich die erste Unterguppe damit beschäftigt, den ML-Agents Walker in eine

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

Tabelle 5.2: Die zwei Untergruppen und ihre Mitglieder

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 Laden von Konfigurationseinstellungen 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.

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 umgesetzt. 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 CUDA¹ pytorch² zu beschleunigen.

Aufgrund der Einschränkungen stand nur LiDO3³, der HPC der TU Dortmund, da andere Rechenknoten wie z. B. Noctua 2⁴ 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 Trainingsumgebung 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.

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>

5.3.2 Training

Generalisierung

- PPO
- ML-Agents
- Nero?

5.4 Terraingeneration

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:

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

In diesen Abschnitt wird nur auf den Aufbau des `DynamicEnvironmentGenerator` 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 Environment 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:

$$\begin{pmatrix} \left\lceil \frac{\text{Anzahl der Arenen}}{\sqrt{\text{Anzahl der Arenen}}} \right\rceil \\ 0 \\ \text{Anzahl der Arenen} \bmod \sqrt{\text{Anzahl der Arenen}} \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \quad (6.1)$$

Alle anderen Objektpositionen müssen danach neu im lokalen Koordinatensystem gesetzt werden. Da die Unity-Standard-Texturen sehr hell 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 Paket¹ benutzt. Das Paket stellt eine Klasse bereit, welche mit zwei Skript-Objekten konfiguriert wird. Zusätzlich wird ein seed übergeben, welcher reproduzierbare Kreaturen erlaubt. Die erstellte 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 Spielterrain 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 **DynamicEnvironmentGenerator** gespeichert. Was unübersichtlich war und zu ständigen Neubauen des Projektes führte. Deshalb wurde eine **GenericConfig**

¹<https://github.com/PG649-3D-RPG/Creature-Generation>

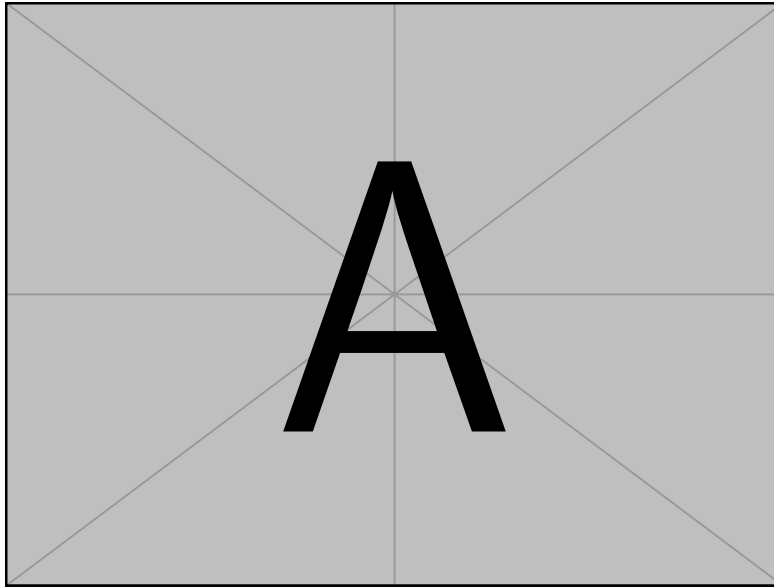


Abbildung 6.1: Konfigurationsmöglichkeiten des DynamicEnvironmentGenerator im Unity-Editor.

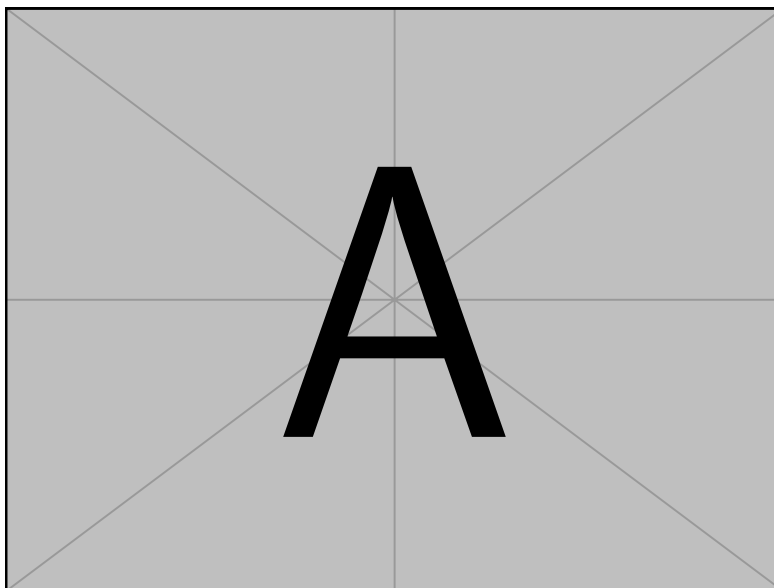


Abbildung 6.2: Ein Beispiel der generierten Trainingsumgebung mit mehreren Arenen.

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-Diagramm. Der Aufbau dieser Klassen orientiert sich dabei sehr stark an die Implementierung des ML-Agents Walker.

`GenericAgent`

Die kontrollierende 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 als 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 durchführt. 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 bestimmtes 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>

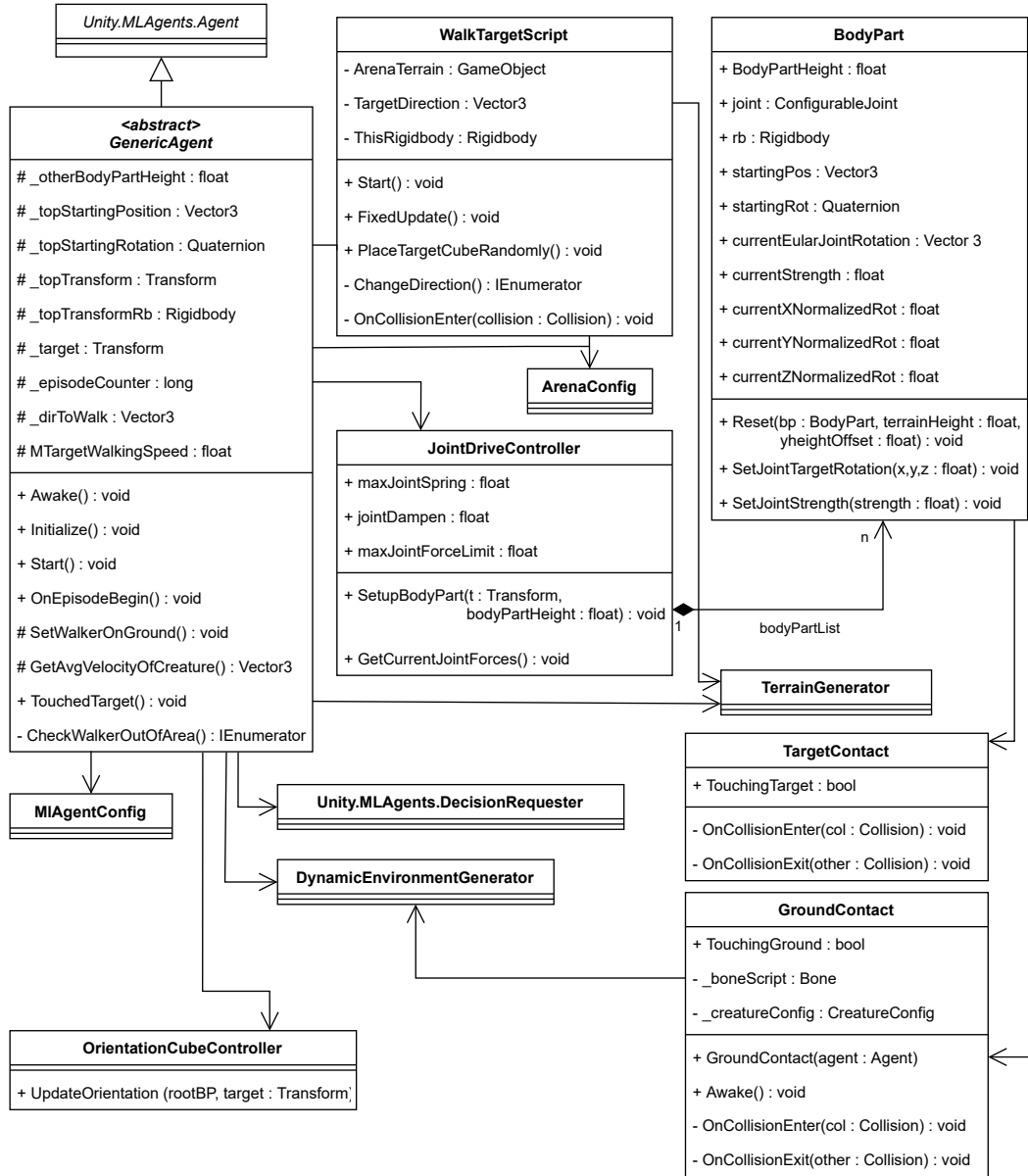


Abbildung 6.3: UML-Diagram der GenericAgent-Klasse und weitere relevante Klassen

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 multi-diskrete 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)

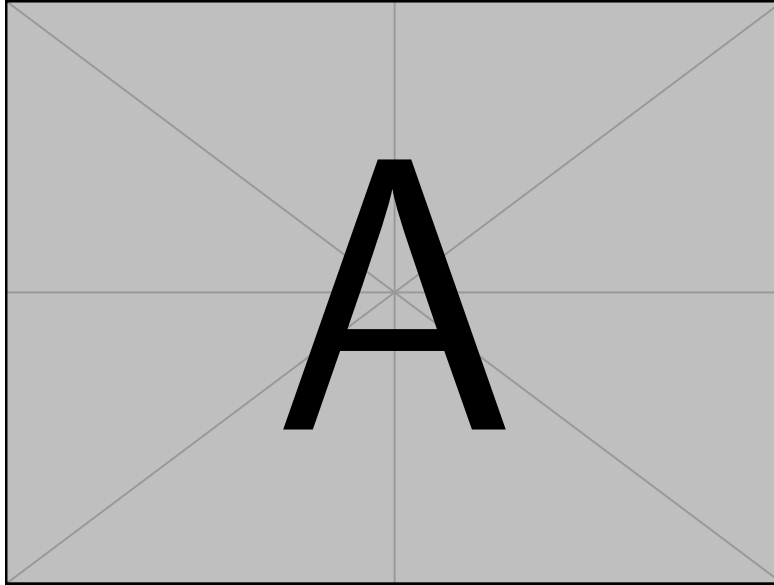


Abbildung 6.4: Parallel ausgeführte Multi-Agent Builds

gelernt. Daraus wird eine Normalverteilung generiert, aus der dann ein tatsächliches Aktionstupel gesampelt 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. Abbildung 6.4 zeigt vier parallel ausgeführte Multi-Agent-Builds des MLAGents Walker Projekts, die jeweils 10 Agenten beinhalten. Insgesamt können so mit 40 Instanzen parallel Erfahrungen gesammelt werden. 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 stabilisieren 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[6];

```
#!/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=t.x86_64

```

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-
```

Für die Python-Installation wurde auf Anaconda³ 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 Grafikkarten bietet. Es besteht die Möglichkeit PyTorch zu bauen um die Unterstützung 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

Themenallokation

1. Einleitung: Thomas
2. Grundlagen
 - 2.1. Reinforcement Learning: Jannik
 - 2.2. Cellular Automata: Markus
 - 2.3. L-Systeme: Kay
3. Verwandte Arbeiten
 - 3.1. RL Frameworks
 - 3.1.1. ML-Agents: Niklas
 - 3.1.2. NeroRL: Niklas
4. Fachliches Vorgehen:
 - 4.1. Projektorganisation: Carsten, Thomas, Jannik, Leonard
 - 4.2. Creature Generation
 - 4.2.1. Metaballs: Jona
 - 4.2.2. Jonas Creature Generation Method: Jona, Markus
 - 4.2.3. L-System Creature Generation: Tom, Kay
 - 4.3. Creature Animation
 - 4.3.1. Trainingsumgebung: Carsten
 - 4.3.2. Training: Als Gruppe überlegen
 - 4.4. Terraingeneration
 - 4.4.1. Dungeon Generierung: Tom
 - 4.4.2. Space Partitioning: Tom
 - 4.4.3. Perlin Noise (Terrain-Flächen): Kay
 - 4.4.4. Vegetation via L-System: Kay
5. Technische Umsetzung
 - 5.1. Creature Animation: Nils, Carsten, Jan
 - 5.1.1. Trainingsumgebung: Nils
 - 5.1.2. Erweiterung der Agent Klasse

8 *Ausblick*

5.1.3. LiDO3

6. Vorläufige Ergebnisse

6.1. Diskussion

7. Ausblick

Abbildungsverzeichnis

3.1	Reinforcement Learning Kontrollfluss [1]	7
5.1	Stand des Status-Quo am (Datum)	17
6.1	Konfigurationsmöglichkeiten des <code>DynamicEnviornmentGenerator</code>	23
6.2	Beispiel der generierten Trainingsumgebung	23
6.3	UML-Diagram der <code>GenericAgent</code> -Klasse und weitere relevante Klassen	25
6.4	Parallel ausgeführte Multi-Agent Builds	27

Tabellenverzeichnis

5.1	Primäre Gruppenaufteilung der Projektgruppe	15
5.2	Die zwei Untergruppen und ihre Mitglieder	18

Algorithmenverzeichnis

1	PPO Pseudocode Algorithmus [1]	12
---	--------------------------------	----

Literatur

- [1] L. Graesser und W. Keng, Foundations of Deep Reinforcement Learning: Theory and Practice in Python, Ser. Addison-Wesley Data & Analytics Series. Pearson Education, 2019, ISBN: 9780135172483.
- [2] H. Dong, Z. Ding, S. Zhang, H. Yuan, H. Zhang, J. Zhang, Y. Huang, T. Yu, H. Zhang und R. Huang, Deep Reinforcement Learning: Fundamentals, Research, and Applications, S. Z. Hao Dong Zihan Ding, Hrsg. Springer Nature, 2020, <http://www.deeppreinforcementlearningbook.org>.
- [3] J. Schulman, P. Moritz, S. Levine, M. Jordan und P. Abbeel, High-Dimensional Continuous Control Using Gen 2015. DOI: 10.48550/ARXIV.1506.02438. Adresse: <https://arxiv.org/abs/1506.02438>.
- [4] J. Schulman, S. Levine, P. Moritz, M. I. Jordan und P. Abbeel, Trust Region Policy Optimization, 2015. DOI: 10.48550/ARXIV.1502.05477. Adresse: <https://arxiv.org/abs/1502.05477>.
- [5] J. Schulman, F. Wolski, P. Dhariwal, A. Radford und O. Klimov, „Proximal Policy Optimization Algorithms,“ CoRR, Jg. abs/1707.06347, 2017. arXiv: 1707.06347. Adresse: <http://arxiv.org/abs/1707.06347>.
- [6] (). LiDO3, Adresse: https://www.lido.tu-dortmund.de/cms/de/LiD03/LiD03_first_contact_handout.pdf.