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

4. Oktober 2022

„Projektgruppe: Entwicklung eines 3D RPG Videospiels mittels prozeduraler Inhaltsgenerieung und Deep Reinforcement Learning“

SS 2022 - WS 2022/2023 · TU Dortmund

Inhaltsverzeichnis

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

Die prozedurale Generierung von NPCs sowie das Erlernen von Bewegungen und das Animieren dieser stellt heutzutage in der Videospielindustrie eine zentrale Herausforderung dar. Nicht nur müssen Kreaturen effizient und somit oftmals zur Laufzeit generiert werden, sondern es müssen auch die Animationen welche durch maschinelles Lernen erlernt werden, auf ein möglichst breites Spektrum an Kreaturen anwendbar sein. Dabei ist es also essentiell, dass das Training der Kreaturen möglichst generalisiert stattfindet, sodass bei der Kreatur-Generierung eine große, sich bei den Körpermerkmalen variierende Menge der Kreaturen erzeugt werden kann, wodurch weniger Einschränkungen für die Körpereigenschaften der NPCs aufkommen. Somit sollten also die Animationen auf einen möglichst breiten Pool von Kreaturen anwendbar sein, ohne für zufällig erzeugte, potentiell neue, Körpermerkmale neu trainieren zu müssen.

Weiterhin muss bei der Generierung von Spielfiguren die relevante Herausforderung des automatischen Aufspannens eines Meshes über den bis zu diesem Zeitpunkt untexturierten Körper einer Kreatur gelöst werden; das sogenannte *Automatic Rigging*, welches ebenfalls zu dem prozeduralen Erzeugen von Kreaturen dazugehört. Es muss also hier das relevante Problem betrachtet werden, Meshes welche ggf. ebenfalls prozedural erzeugt werden, auf das breite Spektrum von verschiedenen Körperausprägungen des Kreaturen-Pools anwenden zu können, ohne dass es zu sehr von den Körperformen der Kreaturen abweicht.

Ferner könnte die prozedurale Generierung von Inhalten in einem Videospiel nicht nur zum Erzeugen von Kreaturen zum Einsatz kommen, sondern auch für die Spielwelt selbst.

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
- 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

Zunächst 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 Ergebnisse präsentiert und diskutiert (7.1).

Schließlich wird in Kapitel 8 zusammengefasst, auf welche (Forschungs-)Bereiche die Erkenntnisse dieser Studie ausgeweitet werden können.

3 Grundlagen

3.1 L-System

Eine zentrale Eigenschaft in der Natur ist die Selbstähnlichkeit von Strukturen auf makroskopischer und mikroskopischer Ebene [16]. Das heißt, es lassen sich dieselben geometrischen Strukturen in verschiedenen Größenordnungen wiederfinden. Beispielsweise ähneln die Knospen eines Blumenkohls auch seiner äußeren Struktur, wie in Abbildung 3.1[1] zu sehen ist.

Abbildung 3.1: Blumenkohl

Um diese Eigenschaften präzise und strukturell darzustellen kann ein Lindenmayer-System (L-System) [12] eingesetzt werden. Die Basis eines L-Systems bildet eine kontextfreie Grammatik $G = (N, T, S, P)$ mit einer Menge von Nicht-Terminalsymbolen N, einer Menge von Terminalsymbolen T, einem Startsymbol S und einer Menge von Produktionen P. Der Unterschied zwischen einem L-System und einer üblichen kontextfreien Grammatik besteht darin, dass in einem Ableitungsschritt eines L-Systems parallel alle Nicht-Terminale ersetzt werden. Zusätzlich werden nur eine feste Anzahl an Ableitungsschritten durchgeführt und es kann anstatt eines Startsymbols auch ein Startstring angegeben werden.

[1] Quelle: `https://commons.wikimedia.org/wiki/File:Blumenkohl-1.jpg`, Autor: Rainer Zenz, CC BY-SA 3.0 `http://creativecommons.org/licenses/by-sa/3.0/`, via Wikimedia Commons

3.1.1 Grafische Darstellung in 2D

Der resultierende String einer Ableitung kann grafisch als Zeichenvorschriften für turtle graphics interpretiert werden. In turtle graphics gibt es eine turtle, oder Zeichenkopf, der initial in eine Richtung zeigt und sich in zwei oder drei Dimensionen fortbewegen kann. Bei jeder Fortbewegung wird entlang der aktuellen Richtung der turtle und der hinterlegten Distanz, eine Strecke gezeichnet. Die Symbole der Grammatik werden hierbei als Fortbewegung um eine gewisse Distanz oder eine Drehung interpretiert. In Abbildung 3.2 ist eine Visualisierung dieses Konzeptes zu sehen. Hier ist die turtle initial nach oben gerichtet.

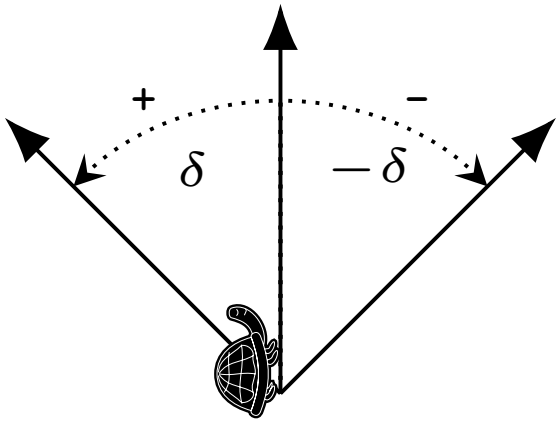


Abbildung 3.2: L-System 2D Rotation

Ohne Erweiterungen entstehen hierbei immer zusammenhängende Strukturen, die an einem Stück gezeichnet werden müssen; die turtle darf also nicht springen. Das heißt, Blätter, Äste oder Stängel sind nur schwer zu realisieren.

3.1.2 Erweiterungen

Um komplexere Strukturen abzubilden, werden zusätzliche Symbole eingesetzt und die Auswertung erweitert. Zwei übliche Erweiterung, die besonders zur Generierung von Vegetation nützlich sind, sind Bracketed L-Systeme und Stochastische L-Systeme.

Bracketed L-Systeme

Bracketed L-Systeme [16] werden zur Definition von Strukturen eingesetzt, bei denen die turtle ihre Position ändern muss, ohne dass ein Strich gezeichnet wird. Damit die generierte Struktur zusammenhängend bleibt, wird die Position und aktuelle Richtung der turtle auf einem Stack gespeichert und kann durch `push` und `pop` Operationen verwaltet werden. Die turtle kann sich nur fortbewegen ohne zu zeichnen, indem sie an eine vorherige Position zurückspringt. Die Menge der Terminalsymbole wird dabei um `[` für die `push` Operation und `]` für `pop` erweitert. Bei dem Lesen eines `[`, wird die aktuelle Position und Rotation der turtle auf dem Stack gespeichert. Wenn ein `]` gelesen wird,

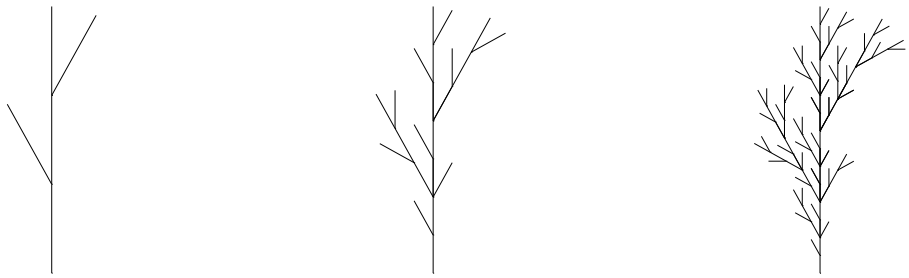
dann wird die turtle zu der Position und Rotation zurückgesetzt, die auf dem Stack als oberstes Element gespeichert worden war.

Mit dieser Erweiterung können insbesondere auch Äste von Bäumen oder Stängel von Blumen gezeichnet werden. Ein Beispiel für eine Art von Strauch kann mittels dieses L-Systems erzeugt werden:

- Nicht-Terminalsymbole $N = \{F\}$
- Terminalsymbole $T = \{\texttt{+}, \texttt{-}, \texttt{[}, \texttt{]}\}$
- Startstring $S = F$
- Produktionen:

$$F \rightarrow \; F\,\texttt{[+}F\texttt{]}\,F\,\texttt{[-}F\texttt{]}\,\texttt{[}F\texttt{]}$$

Hierbei beschreibt das Nicht-Terminalsymbol F eine Fortbewegung, `+` und `-` jeweils eine Drehung um 30° nach links bzw. rechts in 2D und `[, ]` beschreiben die `push` und `pop` Operationen. Für die ersten drei Ableitungsschritte sind die generierten Strukturen in Abbildung 3.3 dargestellt.



Ein Ableitungsschritt | Zwei Ableitungsschritte | Drei Ableitungsschritte

Abbildung 3.3: Darstellung der ersten drei Ableitungsschritte eines Bracketed L-Systems

Stochastische L-Systeme

Eine Eigenschaft von den bisher verwendeten L-Systemen ist, dass die Produktionen deterministisch ausgewertet werden und die resultierenden Strings eindeutig sind. Um realistische Pflanzen generieren zu können muss jedoch etwas Variation eingeführt werden; es wäre also vorteilhaft, wenn die Produktionen nicht-deterministisch ausgewertet werden. Dazu kann ein stochastisches L-System [16] verwendet werden. Hierbei kann es mehrere Produktionen mit gleichen linken Seiten geben. Jeweils über die Menge von Produktionen mit gleicher linken Seite wird eine Wahrscheinlichkeitsverteilung erstellt. Das heißt die Auswahl der Ersetzung eines Nicht-Terminalsymbols erfolgt zufällig.

Im Folgenden ist ein Beispiel für ein stochastisches Bracketed L-System gegeben:

- Nicht-Terminalsymbole $N = \{F\}$

- Terminalsymbole $T = \{\texttt{+}, \texttt{-}, \texttt{[}, \texttt{]}\}$
- Startstring $S = F$
- Produktionen:

$$F \xrightarrow{0.33} F\texttt{[+}F\texttt{]}F\texttt{[-}F\texttt{]}F$$
$$F \xrightarrow{0.33} F\texttt{[+}F\texttt{]}F$$
$$F \xrightarrow{0.34} F\texttt{[-}F\texttt{]}F$$

Hier gibt es für das Nicht-Terminal F drei Produktionen, zwei Produktionen werden mit einer Wahrscheinlichkeit von 33 % gewählt und eine Produktion wird mit 34 % gewählt. Drei resultierende Bilder dieses L-Systems sind in Abbildung 3.4 gegeben. Es ist gut erkennbar wie aus einem L-System deutlich unterschiedliche Strukturen generiert werden können, die jedoch alle ähnliche Grundstrukturen aufweisen.

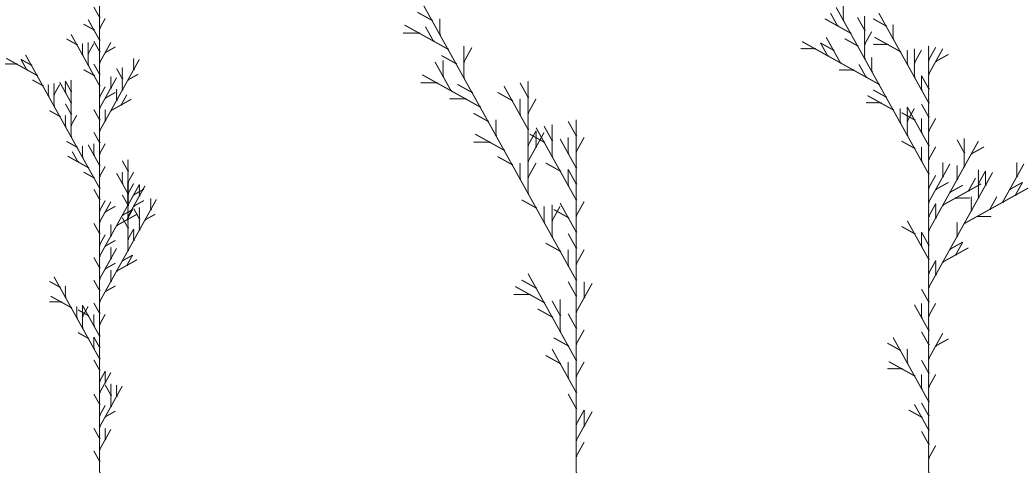


Abbildung 3.4: Drei Resultate desselben stochastischen L-Systems

Erweiterung in 3D

Die L-Systeme aus den vorherigen Abschnitten haben stets zweidimensionale Strukturen erstellt. Dabei wurden die Symbole `+` und `-` genutzt um die aktuelle Richtung der turtle in der Ebene zu rotieren. Für Strukturen in drei Dimensionen kann diese Idee weiterverwendet werden. Hierbei werden jeweils zwei Symbole für Rotationen um jede der drei Achsen eingeführt. Die Symbole `+` und `-` rotieren die turtle um die z-Achse, `&` und `^` rotieren um die x-Achse und `/` und `\` rotieren um die y-Achse.

Ein Beispiel[2] für ein L-System, das einen dreidimensionalen Baum erzeugt ist im Folgenden gegeben:

[2]L-System adaptiert von: `https://www.sidefx.com/docs/houdini/nodes/sop/lsystem.html#3d`

- Nicht-Terminalsymbole $N = \{F\}$
- Terminalsymbole $T = \{+, -, \&, \hat{}, /, \backslash, [,]\}$
- Startstring $S = FFFA$
- Produktionen:

$$A \rightarrow [B]////[B]////B$$
$$B \rightarrow \&FFFA$$

Mit diesem L-System kann ein dreidimensionaler Baum wie in Abbildung 3.5 in Unity erzeugt werden. Es wurden 7 Ableitungsschritte durchgeführt und es wurde jeweils 28° um die Achsen rotiert.

Abbildung 3.5: L-System 3D Baum

3.1.3 Auswertung eines L-Systems

Die Auswertung eines L-Systems wird in diesem Abschnitt beschrieben. Der Algorithmus 1 umfasst die, ausgehend vom Startstring, durchgeführten i Ableitungsschritte der Grammatik. Der Startstring wird Zeichen-für-Zeichen von links nach rechts durchlaufen und es wird für jede Iteration ein leerer String als Zwischenresultat angelegt. Für jedes gelesene Zeichen des Startstrings wird überprüft, ob es sich um ein Nicht-Terminalsymbol oder Terminalsymbol handelt. Im Falle eines Nicht-Terminalsymbols wird die Ersetzung gemäß der entsprechenden Produktion an das Zwischenresultat angehangen. Ein Terminalsymbol wird einfach in das Zwischenresultat übernommen. In jeder Iteration wird der Startstring vollständig durchlaufen. Dadurch werden alle Nicht-Terminale des Startstrings der jeweiligen Iteration ersetzt. Am Ende der n-ten Iteration wird das Zwischenresultat als neuen Startstring für die $n+1$-te Iteration gesetzt und der Prozess wiederholt sich für die festgelegten i Iterationen. Für ein stochastisches L-System muss die Wahl der Ersetzung in Zeile 6 gemäß der Wahrscheinlichkeiten für das Nicht-Terminalsymbol erfolgen.

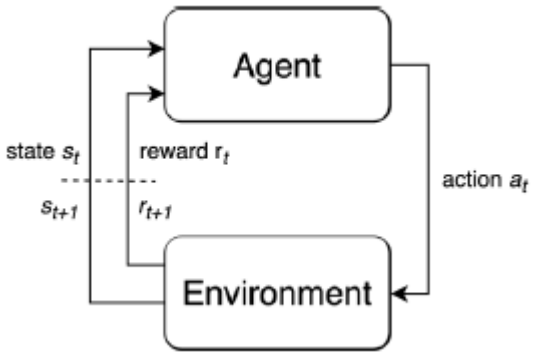


Abbildung 3.6: Reinforcement Learning Kontrollfluss [6]

Eingabe: Grammatik (N, T, S, P), Iterationen i
Ausgabe: String w nach i Ableitungsschritten

```
1:  w ← S
2:  for i Iterationen do
3:      temp ← ∅
4:      for c ∈ w do
5:          if c ∈ N then
6:              Wähle Ersetzung r aus Produktion mit (c → r) ∈ P
7:              Hänge r an temp an
8:          else
9:              Hänge c an temp an
10:         end if
11:     end for
12:     w ← temp
13: end for
14: return w
```

Algorithmus 1: L-System Auswertung

3.2 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 [6][5].

Abbildung 3.6 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.[6]

3.2.1 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.[6]

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

3.2.2 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). [6]

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

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

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

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. [5]

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. [6] 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. []

3.2.3 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 ($R_t(\tau) > 0$) werden wahrscheinlicher. [6]

3.2.4 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. [6]

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.

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

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.

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

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. [13]

3.2.5 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 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. [6]

Clipping

Es gibt zwei theoretische Varianten des PPO Algorithmus. Die erste Variante baut direkt auf dem Trust Region Policy Optimization Algorithmus (TRPO) [15] 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. [14]

Algorithmus Algorithmus 2 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

```
1 for episode=0...MAX do
2     for actor=1...N do
3         Sample Trajectory for T time steps with π_{ϑ_old}
4         Compute advantage estimates A_1,...,A_T with π_{ϑ_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. ϑ
9     ϑ_old = ϑ
```

Algorithmus 2: PPO Pseudocode Algorithmus [6]

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

4.1 Creature Generation using Genetic Algorithms and Auto-Rigging[7]

In seiner Thesis beschreibt Hudson einen möglichen Ansatz zur prozeduralen Generierung von Kreaturen, basierend auf Parametern in Form von Minima und Maxima für Dimensionen einzelner Körperteile.
Zunächst wird ein Torso generiert, der aus drei mit Hilfe der Parameter zufällig skalierten Segmenten besteht. Daran werden weitere Körperteile wie Arme, Beine, Kopf und Flügel jeweils an der richtigen Stelle platziert. Die Anzahl und Ausmaße dieser werden ebenfalls zufällig bestimmt.
Vorausgesetzt werden hierbei Annahmen über die prinzipelle Struktur des Skeletts. Hudson beschränkt sich hierbei auf humanoide Zweibeiner und ein allgemeines Vierbeiner Modell. Welche Positionen und Rotationen dabei genau verwendet werden wird nicht spezifiziert. Dieser Aspekt lässt weiter Raum für Optimierung offen und wirft die Frage danach auf, inwiefern sich auch diese Werte randomisieren lassen. Des weiteren spricht Hudson selbst die mögliche Erweiterung auf andere Typen von Kreaturen an.
Außerdem können die daraus erzeugten Kreaturen mit Hilfe eines genetischen Algorithmus kombiniert und mutiert werden, was für mehr Variation sorgen soll, während gleichzeitig gute Ergebnisse weiterentwickelt werden können.
Die Motivation besteht darin, einem Experten Vorlagen zur manuellen Modellierung einer glaubwürdigen Kreatur zu liefern. Dieser Ansatz müsste erweitert werden um ihn auf unser Ziel anzupassen, wobei die vollständig prozedurale Generierung im Vordergrund steht.

4.2 Procedural Generation of 2D Creatures [8]

In dieser Thesis wird unter anderem ein möglicher Ansatz zur Generierung eines Meshes um das zuvor erzeugte Skelett einer Kreatur. Dies geschieht mit Hilfe von Metaballs.
Metaballs bilden die implizite Definition einer Oberfläche, die jedem Punkt im Raum einen Wert $F(x, y, z)$ zuweist. Überschreitet dieser Wert einen festgelegten Schwellenwert(häufig 1), liegt der Punkt außerhalb des Meshes, liegt er darunter, liegt der Punkt innerhalb des Meshes. Die Oberfläche liegt dort wo F gleich dem Schwellenwert ist. F sezt sich aus den Gleichungen mehrerer Kugeln $i \in \{1, ..., N\}$ zusammen, sodass diese ineinander verschmelzen und eine Glatte Fläche bilden, wenn sie einander nahegelegen

sind.

$$F(x,y,z) = \sum_{i=1}^{N} f_i(x,y,z)$$

Es können beliebige Distanzfunktionen für die einzelnen Kugeln verwendet werden, Janno hat sich auf die folgende festgelegt.

$$f_i(x,y,z) = exp(\frac{B_i r_i^2}{R_i^2} - B_i)$$

mit $B_i = -0.5$. Dabei ist r_i die euklidische Distanz des Punktes zum Mittelpunkt der Kugel, und R_i der Radius der Kugel.
Die Idee zur Platzierung der Kugeln entlang der Kreatur ist, sie entlang jedes Segmentes mit einem Abstand zu positionieren, der garantiert, dass sie eine zusammenhängende Fläche bilden. Nimmt man einen gleichen Radius R von zwei Kugeln an, lässt sich der maximale Abstand dieser zu einem Punkt bestimmen, der genau den Schwellenwert(=1) erreicht.

$$2 \cdot f(x,y,z) = 1$$

Durch Umformen erhält man einen Wert für den Abstand zum Mittelpunkt, der verdoppelt den maximalen Abstand der beiden Kugeln ergibt. Im Fall der verwendeten Funktion bedeutet dies einen Abstand von $R \cdot 3.13$.
Die Kugeln werden anschließend mit gleichem Abstand entlang des Segmentes platziert, sodass dieser den maximalen Abstand nicht überschreitet.

4.3 Introduction

4.3.1 ml-Agents

Das Unity Machine Learning Agents Toolkit, kurz ml-Agents, ist ein von Unity entwickeltes und unter der Apache License 2.0 lizensiertes open-source Projekt, dass Implementierungen von beliebten Reinforment Learning Algorithmen zur Verfügung stellt. Das Toolkit wurde am 19. September 2017 vorgestellt und wurde seitdem stetig weiterentwickelt. Die aktuellste Version ist Release 19, welcher am 14. Januar 2022 veröffentlicht wurde. Diese Version bietet neben Implementierungen für die Deep Reinforcement Learning Algorithmen PPO, SAC und MA-POCA auch Implementierungen für die Imitation Learning Algorithmen BC und GAIL.

Neben den Implementierungen der Algorithmen stellt das Toolkit auch eine Python API zur Verfügung. Mit dieser API können eigene Agenten mit den zur Verfügung gestellten Algorithmen trainiert werden. Dabei unterstützt die API sowohl Diskrete als auch Kontinuierliche Aktions- und Beobachtungsräumne. Es unterstützt außerdem das Platzieren von mehreren Agenten in einer Umgebung. Diese können sowohl das selbe Verhalten lernen, um das Training zu beschleunigen, oder verschiedene Verhalten, zum Beispiel zum Trainieren der Charaktere in asymmetrischen Spielen.

Neben der nach der Python API erstellten Umgebung benötigt ml-Agents zum Starten des Trainingsvorgangs noch eine Konfigurations Datei. Diese wird für ml-Agents als eine yaml Datei zur Verfügung gestellt. Diese Datei beinhaltet eine Liste von Verhalten und jeder Agent in der Umgebung muss einem der Verhalten entsprechen. Dies ermöglicht es für verschiedene Verhalten verschiedene Konfigurationen für den Ablauf des Trainings und den Aufbau des Netzwerks anzugeben. Die Konfiguration definiert den zu verwendenden Algorithmus, die Anzahl der Schritte, die während des Trainings in der Umgebung ausgeführt werden sollen, und wie oft Checkpoints des Netzes gespeichert werden sollen. Zusätzlich beinhaltet es verschiedene Abschnitte, die Teilaspekte des Trainings beeinflussen.
Der Hyperparameter Abschnitt der Datei beschreibt die Parameter des Trainings, wie die Batchgröße, die Buffergröße und die globalen Parameter des ausgewählten Algorithmus, wie zum Beispiel die Lernrate oder das beta und epsilon, bei PPO, oder das tau, bei SAC.
Der Network Settings Abschnitt beschreibt die Größe und Anzahl der Schichten in dem Netzwerk. Zusätzlich kann hier angegeben werden, ob die Werte normalisiert werden sollen.

Sowohl die Checkpoints als auch das finale Netzwerk werden als onnx Datei gespeichert, die dann, zum Beispiel mit Unity Barracuda, geladen werden kann, um das Netzwerk zu verwenden.
Zum Analysieren und Bewerten des Trainings erhebt ml-Agents während des Trainings verschiedene Daten, wie den durchschnittlichen Reward und die Loss Werte der verschiedenen Netzwerke. Diese werden in einem Tensorboard gespeichert, welches in dem selben Ordner wie die onnx Datei gefunden werden kann.

4.3.2 neroRL

NeroRL ist ein unter der MIT Lizenz lizensiertes Reinforment Learning Framework, dass seinen Fokus auf verschiedene Varianten des Proximal Policy Optimization (PPO) Algorithmus legt. Es basiert auf dem Code von dem ml-Agents Toolkit und verwendet dessen Python API, um mit den Umgebungen zu kommunizieren.
NeroRL stellt neben der regulären Variante von PPO auch eine Implementierung mit Rekkurenz zur Verfügung. Zusätzlich kann bei allen Implementierungen der Agent entweder mit geteilten oder mit getrennten Netzwerken und Gradienten für den Aktor und den Kritiker trainiert werden.
Diese Implementierungen können verwendet werden, um in der bereits existierenden ObstacleTower Umgebung oder in beliebigen openAIGym Umgebungen zu trainieren. Durch das aufbauen auf dem ml-Agents Toolkit können zusätzlich auch Agenten in Umgebungen trainiert werden, die mit der Python API von ml-Agents erstellt wurden.
Allerdings stellt NeroRL noch einige zusätzliche Anforderungen an diese Umgebungen. So dürfen die Observationsräume nur Vektor und Visuelle Beobachtungen enthalten. Außerdem werden nur Diskrete und Multidiskrete Aktionsräume unterstützt. NeroRL kann also nicht verwendet werden, um Agenten in einer Umgebung mit kontinuierlichen Akti-

onsräumen zu trainieren. Des weiteren unterstützt neroRL nur genau einen Agenten pro Umgebung. Es ist also nicht möglich das Training zu beschleunigen, indem mehrere Agenten in einer Umgebung das selbe Verhalten lernen. Zum Beschleunigen des Trainings kann bei neroRL stattdessen über ein Attribut in der Konfigurationsdatei angegeben werden, dass mit mehreren Instanzen der Umgebung gleichzeitig trainiert werden soll. So würden bei neroRL zum Beispiel anstelle von einer Umgebung mit zehn Agenten zehn Instanzen der Umgebung mit je einem Agenten ausgeführt.

Zum Starten des Trainingsvorgangs muss dem Framework eine Konfigurationsdatei im yaml Format übergeben werden, welche alle relevanten Informationen für das Training enthält.
Der erste Abschnitt enthält Informationen über die Umgebung. Wenn für das Training in Build verwendet werden soll, dann wird der Pfad dazu hier angegeben. Außerdem können in diesem Abschnitt die Reset Parameter angegeben werden.
Der zweite Abschnitt enthält die Informationen über das Netzwerk. Dazu gehören neben der Anzahl der Ebenen und deren Größe auch die Aktivierungs und Kodierungsfunktionien. Der Pfad unter dem das Model gespeichert werden soll und der Abstand der Checkpoints wird ebenfalls in diesem Abschnitt festgelegt.
Der dritte Abschnitt beinhaltet Informationen zur Evaluierung eines Modells. Dazu gehört mit wie vielen Instanzen evaluiert werden soll und mit welchen Seeds.
Im vierten Abschnitt wird festgelegt wie viele parallele Instanzen der Umgebung zum trainieren verwendet werden sollen und wie viele Schritte jede Instanz pro Update machen soll. Das Produkt dieser beiden Werte ergibt die Batchgröße für das Training.
Der letzte Abschnitt enthält die Informationen zum eigentlichen Training. In diesem Abschnitt werden der zu verwendende Algorithmus und die Werte für die Hyperparameter von diesem festgelegt. Die Anzahl an durchzuführenden Updates und die Anzahl an Epochs pro Update werden ebenfalls in diesem Abschnitt festgelegt.

Zum Analysieren und Bewerten erhebt neroRL während des Trainings verschiedene Kenndaten, wie den durchschnittlichen Reward und die Loss Werte der verschiedenen Netzwerke. Diese werden in einem Tensorboard gespeichert, welches standardmäßig in einem summaries Ordner zu finden ist.

5 Fachliches Vorgehen

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 5.1.2 und **??** genauer beschrieben.

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

(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

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

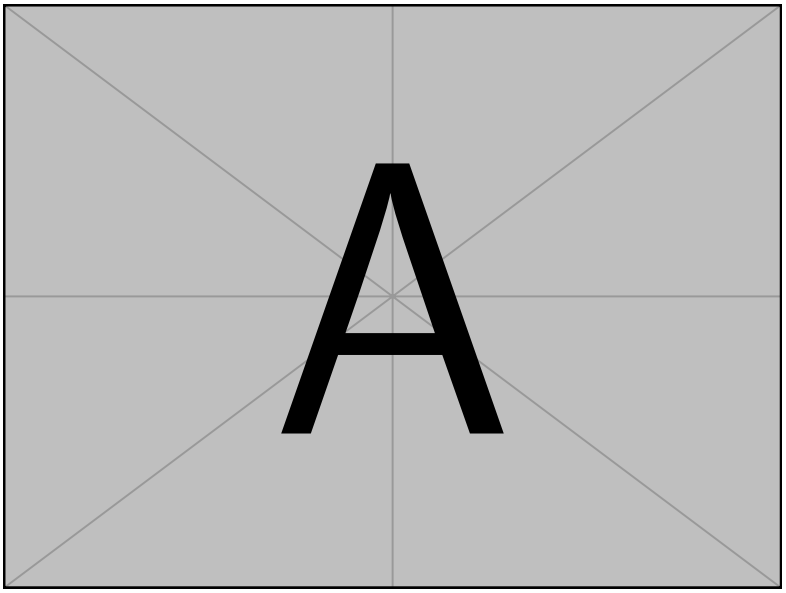


Abbildung 5.1: Stand des Status-Quo am (Datum)

- **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 Generator

In der ersten Phase hat sich die Creature-Generation Gruppe mit der Evaluation und Findung eines geeigneten Generator-Algorithmus beschäftigt. Zunächst wurden dafür in zwei temporären Untergruppen zwei verschiedene Ansätze erforscht:

Skelett → Skin Bei dem ersten Ansatz wurde mit einem L-System Parser experimentiert, zuerst Koordinaten zu erzeugen und das Skelett der Kreatur dann dort reinzulegen (s. Kapitel). Der Skin der Kreatur sollte dabei über Metaballs (Kapitel) und Marching Cubes (Kapitel) zusammengestellt werden.

Skin → Skelett Bei dieser Alternative sollte mit Hilfe von Metaballs und Marching Cubes zuerst ein Skin erzeugt werden, in welches dann ein Skelett durch Automatic Rigging reingelegt werden und durch Dual Quaternion Skinning animierbar gemacht werden sollte.

Während der Ausarbeitung hat sich die Creature-Generator Gruppe wöchentlich am Mittwoch getroffen und hat analog zum wöchentlichen Jour-Fixe aller Teilnehmer einen Regeltermin zum Besprechen von abgeschlossenen aber auch ausstehenden Aufgaben abgehalten. Eine Dokumentation davon wurde ebenfalls analog zum Jour-Fixe sowohl auf Discord als auch im GitHub-Wiki des Creature-Generation Repositories (Link) abgelegt.

Am Schluss der Evaluation beider Ansätze wurde sich für die erste Alternative entschieden: es sollte das L-System zum Erzeugen der initialen Koordinaten genutzt werden, um daraus dann das Skelett zu erstellen und anschließend mit Hilfe der Metaballs den Skin über das Skelett zu legen. Der Dual Quaternion Ansatz (Kapitel) wurde fortwirkend von Leonard Fricke weiterverfolgt, hatte jedoch zu diesem Zeitpunkt noch keine große Priorität, da das erste Ziel eine funktionierende Skelett Generierung zu erhalten war, um sich danach dem Skin-Mesh widmen zu können.

Während der weiteren Ausarbeitung hat sich jedoch ein Alternativansatz zum L-System von Jona Heinrichs gezeigt (Kapitel), um effizientere Skelette erzeugen zu können. Damit wurde die Entwicklung des L-Systems an dieser Stelle eingestellt. Somit wurde evaluiert, dass beide Autoren des L-Systems, Tom Voellmer und Kay Heider, von der Creature-Generation Gruppe in eine weitere Gruppe der Terrain-Generator abzuzweigen, da sich beide Teilnehmer während der Seminarphase mit der Terrain-Generation beschäftigt haben und somit das nächste Thema angegangen werden konnte.

Inzwischen wurden an die Creature-Animator Gruppe bereits erste Skelette als Blueprints bereitgestellt, sodass diese bereits ihr Training auf den bis zu diesem Zeitpunkt erzeugten Kreaturen prüfen konnten. Dabei hat sich Markus Mügge als Vermittler zwischen den Creature-Generatorn und Creature-Animatorn bereiterklärt und war somit für die Kommunikation und den Wissensaustausch beider Teams außerhalb der Jour-Fixe zuständig. Bei dieser Kommunikation beider Teams haben sich etliche Verbesserungen welche von den Creature-Animatorn angeführt wurden von den Creature-Generatorn umgesetzt.

Dabei hat Markus Mügge die finale und relevante Innovation eines Interface den Creature-Animatorn zur Verfügung gestellt, sodass Letztere nicht mehr von einzelnen Blueprints bzw. Paketen mit Kreaturen abhängig waren, welche von den Creature-Generatorn übermittelt werden mussten, sondern nun eigene Kreaturen on-demand erzeugen und ihr Training der Bewegung der Kreaturen untersuchen konnten.

„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

5.1.2 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.

5.2 Creature Generation

5.2.1 Parametrische Kreatur

Als Grundlage für die parametrische Generierung der Kreaturen dient das von Jon Hudson in seiner Thesis [7] beschriebene Modell, welches in Abschnitt **??** näher beschrieben wird.
Die Kreatur besteht aus mehreren Körperteilen, die separat generiert werden und jeweils ihre eigenen Parameter besitzen. Diese Körperteile sind Torso, Beine, Arme, Hals, Füße und Kopf, die jeweils aus einem oder mehreren Knochen bestehen. Alle in diesem

Abschnitt erwähnten Zufallsvariablen entstammen einer uniformen Verteilung.

Knochen Koordinaten

Um einfache lokale Transformationen zu ermöglichen, definieren wir ein einheitliches Koordinatensystem für alle Knochen. Der Ursprung dessen ist der **proximale Punkt**, dieser ist der der Körpermitte am nächsten gelegene Punkt und entspricht somit der Stelle, an der der knochen, gegebenenfalls mit einem Offset, an seinem Elternknochen befestigt ist. Der **distale Punkt** ist der Endpunkt des Knochens, also der am weitesten von der Körpermitte entfernte Punkt.
Die **proximale Achse** verläuft parallel zum Knochen, von der Körpermitte weg, also in Richtung des distalen Punktes. Die **ventrale Achse** liegt orthogonal zur proximalen Achse und zeigt in Richtung der Vorderseite des Knochens. Diese lässt sich prinzipiell beliebig definieren, in unserem Fall haben wir uns jedoch für Armknochen auf die dem Körper zugewandte Innenseite des Knochens festgelegt, für parallel zum Boden generierte Torsi auf die Unterseite und für alle weiteren Knochen auf die der Blickrichtung des Skeletts zugewandten Seite. Die **laterale Achse** liegt senkrecht auf den beiden zuvor definierten Achsen.

Generierung

Unsere Methode setzt voraus, dass zunächst gewisse Vorgaben zur generellen Struktur der Kreatur gemacht werden. In unserem konkreten Fall bedeutet dies, dass wir uns zunächst auf Zwei- und Vierbeiner beschränken. Der Zweibeiner orientiert sich an der menschlichen Anatomie und besitzt deshalb in jedem Bein jeweils zwei Knochen und einen einzelnen Fußknochen sowie zwei Arme. Die Beine eines Vierbeiners besitzen, angelehnt an die Skelette echter vierbeiniger Säugetiere, jeweils vier Knochen. Arme werden hier nicht generiert. Um Kreaturen zu erzeugen, die nicht einer dieser Strukturen entsprechen, wie beispielsweise Spinnentiere oder Echsen, müsste man weitere Skelette definieren.

Zuerst wird der **Torso** Generiert. Im Falle des Zweibeiners ist dieser nach oben gerichten, beim Vierbeiner parallel zum Boden. Als Parameter werden jeweils Minima und Maxima für die Länge und den Umfang des Torsos übergeben. Daraus wird zunächst ein zufälliger Wert für die Länge bestimmt. Da wir uns vorerst für einen Torso bestehend aus drei Knochen entschieden haben, wird diese Länge zufällig auf drei kleinere Längen aufgeteilt. Jeder dieser Knochen erhält dann einen zufälligen Umfang. Das unterste Torsosegment dient als Elternknochen der Kreatur, die anderen beiden Knochen werden dann nacheinander am distalen Punkt des vorangegangenen Knochens befestigt.
Anschließend werden die **Beine** paarweise generiert, wodurch nur symmetrische Kreaturen erstellt werden können. Auch hier wird zunächst die Länge zufällig bestimmt und dann auf zwei beziehungsweise vier Knochen aufgetelt. Auch Die Umfänge werden zufällig bestimmt, jedoch zusätzlich sortiert, sodass das das dickste Segment der Körpermitte am nächsten gelegen ist. In beiden Fällen werden die Beine gerade nach unten gereriert. Bei einem Zweibeiner wird am distalen Punkt des unteren Beinknochen in Richtung des-

sen ventraler Achse ein Fußknochen mit parametrisch zufällig bestimmter Größe erzeugt. Befestigt werden die Beine durch einen zusätzlichen Hüftknochen, der parallel zum Torso am proximalen Punkt des ersten Torsoknochens ansetzt und dessen proximale Achse entgegen derer seines Elternknochens gerichtet ist, also den Torso etwas verlängert. Es wird jeweils ein Bein links und rechts vom Mittelpunkt des Hüftknochens angebracht. Der Abstand entlang der lateralen Achse ergibt sich aus dem Umfang des Knochens. Im Falle des Vierbeiners wird analog dazu ein weiteres Beinpaar am distalen Punkt des letzten Torsoknochens generiert. Anschließend werden Torso und Beine so rotiert, dass sich alle Enden der Beine auf der selben Höhe befinden und die Beine noch immer gerade nach unten Zeigen.
Die **Arme** des Zweibeiners werden analog zu den Beinen mit Hilfe eines Schultersegmentes am distalen Punkt des obersten Torsoknochens erzeugt.
Der **Hals** wird als Verlängerung des Torsos generiert. Dabei wird sowohl die Länge und Dicke zufällig bestimmt als auch die Anzahl der Knochen. Beim Zweibeiner wird er am distalen Punkt des Schulterknochens befestigt und alle Knochen zeigen parallel zum Torso gerade nach oben. Beim Vierbeiner erfolgt das Befestigen am distalen Punkt des vorderen Hüftknochens. Dabei ist die Rotation des ersten Knochens ein zufälliger Wert zwischen der der proximalen und der der negativen ventralen Achse des Elternknochens. Jedes weitere Halssegment erhält eine zufällige Rotation um $\pm 20°$.
Der **Kopf** ist ein einzelner Knochen mit zufälliger Länge und Umfang, der als Verlängerung des letzten Halsknochens betrachtet werden kann.

Die glaubwürdigkeit und Variation der Kreaturen ist dadurch also auch stark von den gewählten Parametern Abhängig. Diese werden von Hand gesetzt, wobei größere Wertebereiche auch für größere Unterschiede zwischen den Kreaturen sorgen, aber gleichzeitig die Kontrolle über das Ergebnis einschränken. Die Beschränkungen der Bewegungsradii der Gelenke werden momentan händisch mit erfahrungsgemäß guten Werten gesetzt. Ziel ist es allerdings auch diese in Zukunft soweit möglich prozedural mit Hilfe von Parametern zu erzeugen, um verschiedene Bewegungsmuster zu ermöglichen.

5.2.2 L-System Creature

Der erste Ansatz zur Generierung von Kreaturen verwendet ein L-System, um die Struktur des Skelettes zu formalisieren. Dabei wird jede gezeichnete Strecke der turtle als ein Knochen interpretiert. Hierdurch entsteht stets ein zusammenhängendes Skelett, da sich die turtle nicht fortbewegen kann ohne zu zeichnen. An Positionen wo die turtle anhält werden Joints gelegt. Mit diesem Ansatz können nicht nur Skelette von Zweibeinern und Vierbeinern erzeugt werden, sondern auch Kreaturen mit beliebig vielen Armen oder Beinen.

Generierung der Skelettstruktur

Um eine Kreatur mithilfe des L-Systems zu generieren wird zunächst die grundlegende Struktur im Startstring definiert. Hierbei werden ausgehend von der initialen Richtung

der turtle, Komponenten des Skeletts wie der Kopf, Arme, der Torso und die Beine definiert. Für jede dieser Körperteile kann ein eigenes Nicht-Terminalsymbol verwendet werden, um das jeweilige Skelett der Komponente zu definieren. Mit diesem Ansatz kann jedes Körperteil unabhängig von allen anderen entworfen werden. Mithilfe eines stochastischen L-Systems können diese zudem zufällige Ausprägungen haben. Beispielsweise ist es einfach, unterschiedlich lange Körperteile zufällig zu generieren. Allgemein ist es auch möglich eine zufällige Anzahl an Armen und Beinen zu erzeugen, womit direkt sehr unterschiedliche Kreaturen erzeugt werden können, die aber stets die gleiche Grundstruktur haben. Der Entwurf der einzelnen Körperteile wird durch die Produktionen des L-Systems definiert.

Der resultierende String des L-Systems wird von der turtle von links nach rechts, den Regeln entsprechend, durchlaufen. Die turtle bewegt sich dabei im dreidimensionalen Raum, wie in Abschnitt 3.1.2 beschrieben. Für jede Fortbewegung der turtle wird jeweils ihre Startposition und die Endposition im dreidimensionalen Raum als ein Tupel gespeichert. Die Tupel können als Strecken interpretiert werden, die daraufhin als Knochen dargestellt werden. Am Ende bildet die Liste der Tupel von Start- und Endpunkten die Skelettstruktur. Um die Joints sinnvoll definieren zu können muss festgehalten werden, welche Produktion ein jeweiliges Tupel erzeugt hat. Dies ist notwendig, da die Produktionen die Kategorien der Körperteile definieren und die Joints dementsprechend angepasst generiert werden müssen. Ausgehend von dieser Liste werden die Knochen erstellt und mit den Joints zu einem vollständigen Skelett zusammengesetzt.

Skelett aus L-System generieren

Ausgehend von einer Liste von Tupeln, die jeweils Start und Endpunkte einer von der Turtle gezeichneten Strecke enthalten wird das Skelett mit Unity Klassen generiert. Zunächst wird die Liste in eine Baumstruktur überführt. Dabei entspricht ein Knoten im Baum einem Knochen im Skelett. Zwei Knoten stehen in einer Eltern-Kind-Beziehung im Baum, wenn die zugehörigen Strecken sich schneiden. Dabei wird gefordert, dass der Endpunkt der zum Elternknoten gehörigen Strecke dem Startpunkt der zum Kindknoten gehörigen Strecke gleicht. Da angenommen wird, dass es zu jedem Startpunkt eines Tupels in der Liste ein weiteres Tupel gibt, dessen Endpunkt dem Startpunkt gleicht lässt sich der Baum so definieren. Der Baum wird traversiert und zu jedem Knoten wird ein GameObject erstellt. Dieses wird mit einem Rigidbody, einem CapsuleCollider und einem primitiven Mesh versehen. Die Knochen werden in Kategorien, wie zum Beispiel Arm, Bein oder Kopf eingeteilt. Zwei Knochen werden mit einem Joint verbunden, wenn sie im Baum in einer Eltern-Kind-Beziehung stehen. Für die erlaubten Rotationen der Joints wurden Standardwerte je nach Kategorie festgelegt. Abbildung 5.2 zeigt ein über ein L-System generiertes Skelett.

5.2.3 Metaballs

Zur Generierung der Geometrie der Kreatur verwenden wir, angelehnt an den Ansatz von Madis Janno [8] (siehe **??**), eine modifizierte Form von Metaballs. Wie auch bei Janno

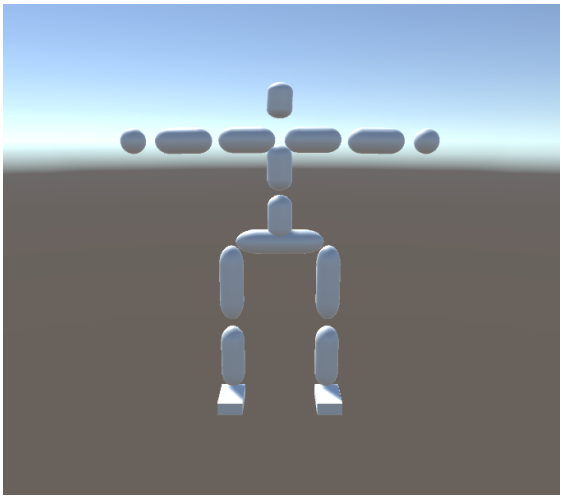


Abbildung 5.2: Mittels L-System generiertes Skelett

lässt sich unsere Methode mit beliebigen Metaball-Funktionen durchführen. Aufgrund der guten Ergebnisse haben wir uns jedoch vorerst auf die gleiche, zuerst von Ken Perlin beschriebene, Falloff-Funktion festgelegt.

$$f_i(x,y,z) = exp(B_i - \frac{B_i r_i^2}{R_i^2} - B_i)$$

Für Metaball i ist r_i der Abstand des Punktes $(x,y,z)^T$ zu dessen Zentrum, also:

$$r_i = ||(x,y,z)^T - (x_i,y_i,z_i)^T||_2 = \sqrt{(x-x_i)^2 + (y-y_i)^2 + (z-z_i)^2}$$

R_i ist der Radius von Metaball i und B_i ein Parameter zur Einstellung der "Blobbiness". Wir verwenden Werte mit $B_i < 0.5$.
Die generierten Kreaturen bestehen aus mehreren Segmenten (Knochen) mit jeweils einem Start- und Endpunkt sowie einer Dicke, die als Radius der darauf platzierten Metaballs verwendet werden kann. Entlang dieser Segmente soll dann das Mesh erzeugt werden. Die von Janno beschriebene Methode berechnet dafür, abhängig von der gewählten Falloff-Funktion, die minimale Anzahl an Metabällen für ein Segment und platziert diese gleichmäßig entlang dessen. Das Problem, welches sich daraus bei unseren Experimenten ergeben hat, liegt darin, dass mit höherer Komplexität der Kreaturen und einer damit einhergehenden steigenden Anzahl an Segmenten, der Einfluss von benachbarten Segmenten nicht gut kontrollieren lässt und diese teilweise ineinander verschmelzen.

Unser Ansatz um dieses Problem zu umgehen ist es, die Anzahl der einzelnen Metabälle drastisch zu reduzieren. Anstatt einer beliebig großen Zahl an Bällen entlang jedes Segments, erzeugen wir jeweils nur einen einzigen. Dazu ersetzen wir die Bälle durch Kapseln, also Zylinder mit jeweils durch eine Halbkugel abgerundeten Enden. Möglich

macht uns dies eine Modifikation der Falloff-Funktion, beziehungsweise der darin verwendeten Distanz. Wir berechnen hierbei nicht den Abstand zum Zentrum einer Kugel, sondern zu der Verbindungslinie zwischen Start- und Endpunkt.

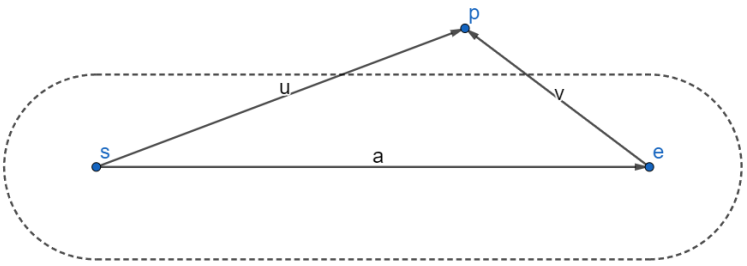


Abbildung 5.3: Beispiel Metakapsel; Die gestrichelte Linie enthält alle Punkte mit $r = R$

Sei s der Startpunkt, e der Endpunkt, a der Vektor $e - s$, p ein Punkt, dessen Abstand berechnet werden soll, $u = p - s$ und $v = p - e$ (Siehe Abbildung 5.3). Die Distanz lässt sich dann folgendermaßen bestimmen:

$$r = \begin{cases} ||p - s||, & \text{falls } a \cdot u < 0 \\ ||p - e||, & \text{falls } a \cdot v > 0 \\ \left|\left| \frac{a \times u}{||a||} \right|\right|, & \text{sonst} \end{cases}$$

Es werden drei Fälle unterschieden. Liegt der Punkt p im Falle von Abbildung 5.3 links von s, beziehungsweise rechts von e, ist Die Distanz von p zum Segment einfach der euklidische Abstand zum jeweiligen Punkt. Ob dies der Fall ist, lässt sich mit Hilfe der Skalarprodukte $a \cdot u$ beziehungsweise $a \cdot v$ überprüfen. Ansonsten berechnet man die Distanz von Punkt p zur Geraden, die durch s und e verläuft.

Da dies nur eine Erweiterung der Metaball-Funktion ist, lassen sich diese Kapseln weiterhin mit anderen Metabällen kombinieren. So können auch Körperteile erstellt werden, die nicht aus solchen Segmenten bestehen, oder Details aus kleineren Metabällen entland der Segmente platziert werden.

5.2.4 Mesh Generation

Marching Cubes

TODO: wollen wir hier noch etwas genauer drauf eingehen, oder reicht der Teil von Jona???

Mesh Indexing

Zur weitere Verarbeitung des Meshes beim automatischen Rigging, ist es wichtig, dass das Mesh korrekt indiziert ist, sodass keine doppelten Vertices existieren. Außerdem können

so die ausgehenden Kanten der Vertices bestimmt werden, welche ebenfalls später für das automatische Rigging benötigt werden. Während dem Marching Cubes Algorithmus werden die entstehende Dreiecke in eine gesonderte Datenstruktur hinzugefügt, welche für jeden Vertex prüft, ob dieser bereits im Vertex-Buffer existiert. Falls ja, wird kein neuer Vertex eingefügt, sondern das Dreieck referenziert den Vertex über den Index-Buffer. Neue Vertices werden dem Vertex-Buffer hinzugefügt und das entsprechende Dreieck referenziert den neuen Vertex. Die Anzahl der Vertices wird dadurch außerdem signifikant verringert, was Performance-Vorteile beim Rendering und bei der Weiterverarbeitung des Meshes ermöglicht.

Delaunay Triangulation

In der Entwicklung des automatischen Riggings des Modells wurde festgestellt, dass die Triangulierung, die durch Marching Cubes entsteht nicht geeignet ist, um mit der Bone-Heat Methode die Vertex-Gewichte zu berechnen. Für die Lösung des Matrix-Systems der Bone-Heat Methode ist es wichtig, dass die Triangulierung die Delaunay-Bedingung erfüllt.

TODO: Delaunay Edge-flipping Algorthmus erklären

5.2.5 Automatisches Rigging

Um die Bone-Struktur des generierten Skelettes mit dem Mesh zu verknüpfen, muss bei der 3D-Modellierung der Prozess des Riggings durchlaufen werden. Dieser beschreibt wie sich jeder einzelne Vertex des Meshes mit den Bones in der Szene bewegt. Jeder Vertex kann an beliebig vielen Bones angehängt werden. Durch eine Gewichtung wird bestimmt wie sehr ein Vertex durch der Transformation eines Bones mitbewegt wird. Da sowohl das Skelett, als auch das gesamte Mesh prozedural generiert werden, kann das Rigging nicht wie üblich in einem Modellierungs-Tool wie Blender [4] manuell durchgeführt werden, sondern muss zur Laufzeit des Spiels während der Generierung der Kreaturen geschehen.

Für ein erfolgreiches Rigging ist es wichtig, dass das Skelett bereits sinnvoll in dem Mesh eingebettet ist. Da hier das Mesh aus Metaballs generiert wird, welche um die Bones plaziert sind, können wir hier davon ausgehen, dass das Mesh das Skelett bereits sinnvoll umhüllt.

Während der Entwickelung wurde zunächst eine triviale Methode als Zwischenlösung verwendet. Dabei wurden alle Vertices nur an den Bone mit der geringsten Distanz angehängt. Diese Methode ist jedoch visuell unbrauchbar, da bei Bewegung des Skeletts an den Gelenken zwischen den Bones Löcher und verschiedene andere Artefakte entstehen. Es wird also eine Lösung benötigt, die es ermöglicht Vertices an mehrere Bones anzuhängen und die Gewichte so bestimmt, dass das Mesh an den Übergängen zwischen den Bones möglichst natürlich deformiert wird.

Bone-Heat Methode

Ein bereits bekannter Algorithmus zur automatisch Gewichtsberechnung und Verknüpfung des Meshes ist die Bone-Heat Methode [1]. Dieser berücksichtig mehrere zusätzliche

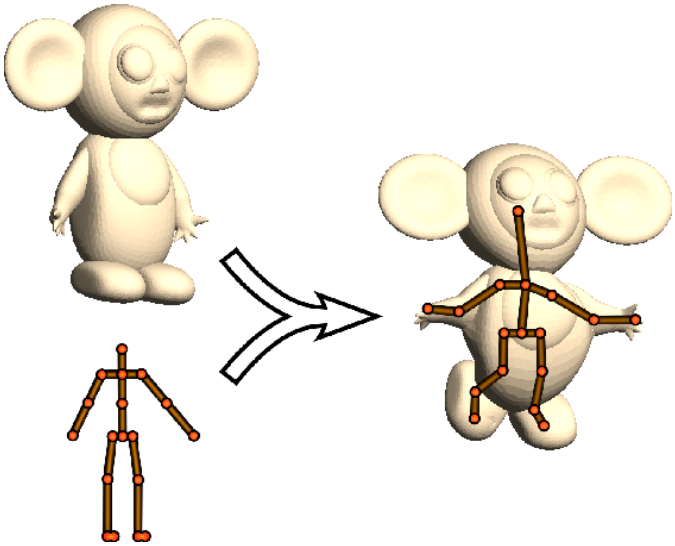


Abbildung 5.4: Beispiel für ein korrekt eingebettetes Skelett in einem Mesh [1].

Eigenschaften für die Gewichte. Zuerst sollen die Gewichte unabhängig von der Auflösung des Meshes sein. Außerdem müssen die Gewichte sich sanft über den Verlauf der Oberfläche verändern. Die Breite des Übergangs zwischen zwei Bones sollte ungefähr proportional zu der Distanz des Gelenks zur Oberfläche des Meshes sein. Ein Algorithmus, welcher die Gewichte alleine aus der Distanz der Bones zu den Vertices berechnet, kann oft schlechte Ergebnisse liefern, da er die Geometrie des Modells ignoriert. Zum Beispiel können Teile des Torsos mit einem Arm verbunden werden. Die Bone-Heat Methode behandelt stattdessen das innere Volumen des Modells als einen wärmeleitenden Körper. Es werden für jeden Vertex die Gleichgewichts-Temperatur berechnet und diese als Gewichte für die Bones verwendet. Wie in Abbildung 5.5 zu sehen ist, wird ein Bone auf 1° gesetzt und der andere auf 0°. Bei Deformation der beiden Bones mit den Gewichten aus dem Temperatur-Gleichgewichts entsteht an dem Gelenk eine natürlich aussehende Verformung der Oberfläche.

Das Temperatur-Gleichgewicht des Volumens zu berechnen wäre sehr aufwändig und langsam, deswegen wird das Gleichgewicht nur über die Oberfläche des Meshes berechnet. Die Gewichte für Bone i werden berechnet durch:

$$\begin{aligned} & \frac{\partial \mathbf{w}^i}{\partial t} = \Delta \mathbf{w}^i + \mathbf{H}\left(\mathbf{p}^i - \mathbf{w}^i\right) = 0 \\ \iff & -\Delta \mathbf{w}^i + \mathbf{H}\mathbf{w}^i = \mathbf{H}\mathbf{p}^i \\ \iff & \left(-\Delta + \mathbf{H}\right)\mathbf{w}^i = \mathbf{H}\mathbf{p}^i \end{aligned} \tag{5.14}$$

Dabei ist Δ der diskrete Laplace-Beltrami Operator auf der Oberfläche des Meshes, welcher mit der Kotangens-Methode approximiert wird [2]. $\mathbf{p}^i$ ist ein Vektor mit $p^i_j = 1$ wenn der nächste Bone zum Vertex j der Bone i ist. Sonst ist $p^i_j = 0$. $\mathbf{H}$ ist eine Diagonalmatrix wobei H_{jj} die Hitze des nächsten Bones von Vertex j ist. Sei $d(j)$ die

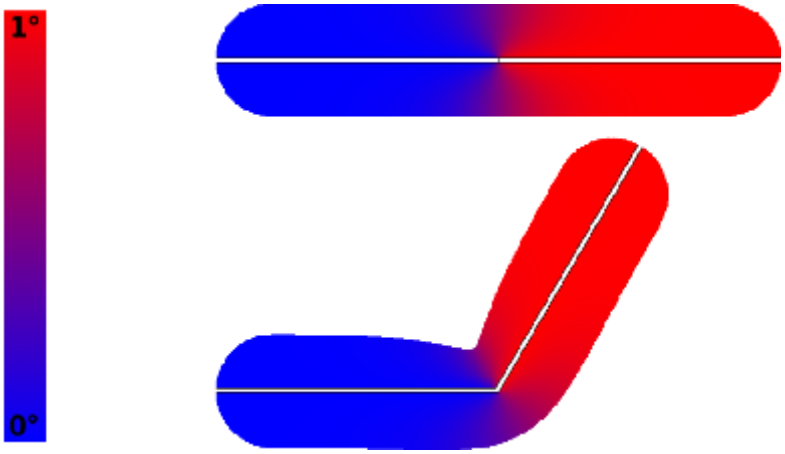


Abbildung 5.5: Temperatur-Gleichgewicht für zwei Bones [1].

Distanz zum nächsten Bone von Vertex j, dann wird $H_{jj} = c/d(j)^2$ gesetzt. Allerdings nur wenn das Geradensegment von dem Vertex zu dem Bone vollständig in dem Volumen des Modells enthalten ist. Wenn der Bone von dem Vertex aus also nicht sichtbar ist, wird $H_{jj} = 0$ gesetzt. Dies verhindert, dass beispielsweise Vertices am Arm and den Torso angehängt werden. Wenn mehrere Bones nahezu die gleiche Distanz zu dem Vertex haben und sichtbar sind, werden ihre Anteile an der Temperaturverteilung gleich berücksichtigt. p_j wird dann $1/k$ und $H_{jj} = kc/d(j)^2$

Der benötigte Sichtbarkeits-Test wird durch ein Signed-Distance-Field (SDF) realisiert, welches wir vorab mit einem Geometry-Shader generieren. Damit werden effiziente Raycast-Operationen in dem Mesh möglich. (TODO: passendes Zitat finden) Der Parameter c wird in dem Paper [1] auf 1 gesetzt, um natürlichere Ergebnisse zu erreichen. Für $c \approx 0.22$ würde der Algorithmus Gewichte berechnen die ähnlicher zu dem Temperatur-Gleichgewicht über dem tatsächlichen Volumen des Meshes sind.

Lösen des Laplace-Beltrami Systems

Für die effiziente Lösung des Matrix-Systems in Formel 5.14 ist es wichtig die Eigenschaften des diskreten Laplace-Beltrami Operators Δ näher zu betrachten. Der Operator wird durch die Kotangens-Methode approximiert, indem von jedem Vertex x_i für jede ausgehende Kante ein Gewicht $v(x_i, x_j)$ aus den gegenüberliegenden Winkel α_{ij} und α_{ji} berechnet wird (siehe Abbildung 5.6).

$$v(x_i, x_j) = \begin{cases} \cot \alpha_{ij} + \cot \alpha_{ji} & \text{for interior edges} \\ \cot \alpha_{ij} & \text{for boundary edges} \end{cases} \tag{5.15}$$

Aus den Kotangens-Gewichten aus Gleichung 5.15 kann nun in Formel 5.16 der Laplace-Beltrami Operator in seiner Matrix-Form berechnet werden [3, 2]. Dabei bescheibt $N(x_i)$ die Menge der benachbarten Vertices von x_i. Da für das Lösen der Gleichung 5.14 lediglich die Matrix $(-\Delta + \mathbf{H})$ benötigt wird, kann diese in der Implementierung direkt als eine

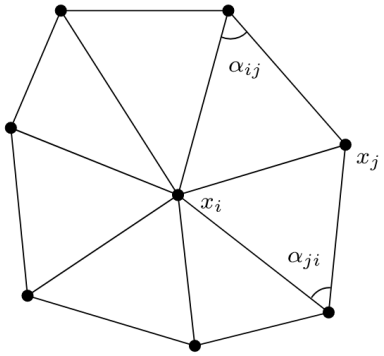


Abbildung 5.6: Berechnung der α-Winkel für eine innere Kante [2].

Matrix erstellt werden. Die Normalisierungsfaktoren aus [3] wurden hier weggelassen, da sie für diese Anwendung sowieso wegfallen.

$$\Delta_{ij} = \begin{cases} 0 & i \neq j, x_j \notin N(x_i) \\ v(x_i, x_j) & i \neq j, x_j \in N(x_i) \\ -\sum_{x_j \in N(v_i)} v(x_i, x_j) & i = j \end{cases} \tag{5.16}$$

Da die Matrix $\mathbf{H}$ eine Diagonalmatrix ist und der Operator Δ eines Vertex durch seine direkten Nachbarn lokal definiert ist, ist die zu lösende Matrix extrem dünn besetzt. Durch die Euler-Charakteristik für Dreiecks-Netze ergeben sich nur ungefähr 7 Einträge in der Matrix pro Reihe [3]. Ein solches System lässt sich, wie in [3] gezeigt, sehr effizient lösen, wenn es sich um eine SPD-Matrix (symetrisch positiv definit) handelt. Die Matrix ist durch die Konstruktion symetrisch.

TODO: Delaunay Bedingung erklären und Verbindung zu Definitheit erklären

TODO: Berechnung und Normalisierung der finale Vertex-Gewichte

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.

[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?

5.3.3 Auswahl des RL Frameworks

In den Abschnitten 4.3.1 und 4.3.2 wurden bereits zwei RL Frameworks vorgestellt, die beide zum Trainieren der generierten Kreaturen in Betracht gezogen wurden. Beide Frameworks haben Vor- und Nachteile im Bezug auf diese Aufgabe, was eine Entscheidung erschwert hat.

Ml-Agents unterstützt bereits von sich aus kontinuierliche Aktionsräume, welche bei der Bewegung der Kreaturen benötigt werden. Außerdem erlaubt es mehrere Agenten in einer Umgebung zu platzieren, um das Trainieren zu beschleunigen. Die vorherigen Tests mit der Walker Testumgebung in Unity [11] haben gezeigt, dass selbst eine zum Laufen optimierte zweibeinige Kreatur recht lange braucht, bis zuverlässig laufen kann. Bei einer prozedural generierten Kreatur ist zu erwarten, dass es noch länger dauert und darum sollten alle Möglichkeiten das Training zu beschleunigen verwendet werden. Das Problem mit ml-Agents liegt in der Auswertung der Ergebnisse. Im Rahmen dieser Ausarbeitung ist es auch notwendig sich kritisch mit den Ergebnissen der Projektgruppe auseinander zu setzten und für den Teil der CreatureAnimation sind vor allem die Statistiken des Trainings relevant. Ml-Agents erhebt zwar im Rahmen des Trainings Statistiken und speichert diese in einem Tensorboard ab, allerdings werden nur wenige Statistiken erhoben. Zusätzlich speichert das Framework nicht direkt die erhobenen Werte ab, sondern in jedem Schritt nur den Durchschnitt der Werte. Für eine Analyse würde dies bedeuten, dass die Ursprünglichen Werte bereits die Durchschnitte sind und daher am Ende der Durchschnitt des Durchschnittes betrachtet wird, was keine guten Ergebnisse liefert.

Die Vorteile von neroRL sind zum einen, dass es mehr Informationen in das Tensorboard schreibt und die direkten Werte dort einträgt, ohne diese vorher zu verarbeiten. Außerdem verlangt neroRL nur den Abstand zwischen Checkpoints und legt dann so viele Checkpoints an, wie nötigt, ohne alte zu überschreiben, wenn die maximale Anzahl erreicht ist. Es liefert die Möglichkeit das Training zu beschleunigen, indem mehrere Umgebungen mit jeweils einem Worker parallel ausgeführt und trainiert werden. Die Nachteile von neroRL sind, dass nur Diskrete und Multidiskrete Aktionsräume unterstütze werden, welche zum Animieren der Kreaturen voraussichtlich nicht ausreichen werden. Außerdem erlaubt es nicht mehrere Agenten in einer Umgebung zu platzieren, was besser skaliert als die Variante mit mehreren Umgebungen.

Es ist also eindeutig, dass, unabhängig davon welches Framework gewählt wird, das ausgewählte Framework noch angepasst werden muss, bevor es verwendet werden kann. Im Hinblick darauf wurde sich für das neroRL Framework entschieden. Dieses Framework auf kontinuierliche Aktionen und Umgebungen mit mehren Agenten zu erweitern erschien einfacher, als ml-Agents auf parallele Umgebungen zu erweitern und das Anlegen der Statistiken so zu ändern, dass sie für die notwendige Analyse geeignet sind. Ein weiterer Einfluss auf diese Entscheidung war, dass Marco Pleines, der Entwickler von neroRL als Betreuer in der Projektgruppe mitwirkt. Bei Fragen oder Problemen während der Anpassung können also deutlich schneller Feedback und Hilfestellungen gegeben werden, als es bei ml-Agents der Fall wäre.

5.3.4 Anpassung von neroRL

Bevor neroRL zum Trainieren verwendet werden kann müssen zunächst die in Kapitel 5.3.3 genannten Probleme beseitigt werden.

Zuerst muss die Implementierung so erweitert werden, dass sie auch Umgebungen mit kontinuierlichen Aktionsräumen unterstützt. Dazu muss neben dem Samplingprozess auch die Klasse zum Starten des Trainings angepasst werden, damit diese beim Überprüfen der Umgebung auch für kontinuierliche Aktionsräume korrekt die Form erkennt. Um die Implementierung so einfach und übersichtlich wie möglich zu halten, wurde entschieden in diesem Schritt die Optionen für Diskrete und Multi diskrete Aktionsräume komplett aus der Implementierung zu entfernen. Da im Rahmen dieser Projektgruppe nur kontinuierliche Aktionsräume benötigt werden, genügt auch ein Framework, welches nur diese unterstützt.

In dem nächsten Schritt muss die Implementierung so erweitert werden, dass mehrere Agenten mit dem selben Verhalten in einer Umgebung vorhanden sein können und sie alle mit in das Training einbezogen werden. Hier wurde entschieden das vorhandene Verhalten mit mehreren parallelen Umgebungen zu erweitert, anstatt es zu ersetzten. Die resultierende Implementierung sollte also in der Lage sein das Training noch weiter zu beschleunigen indem nicht nur die Anzahl der Agenten in jeder Umgebung erhöht werden kann, sondern auch die Anzahl an Umgebungen die parallel zum Trainieren verwendet werden. Diese sollte mehr als ausreichend sein, um den zeitlichen Overhead durch das verarbeiten mehrerer Agenten pro Umgebung auszugleichen.

In einem letzten Schritt muss noch die Implementierung und Optimierung des verwendeten PPO Algorithmus betrachtet werden. Die aktuelle Implementierung ist für Diskrete Aktionen bestimmt und das Netzwerk kann daher nur aus einer stark begrenzten Anzahl an Aktionen auswählen. Bei einem kontinuierlichen Raum der für die Aktionen nur eine Ober- und Untergrenze festlegt gibt es deutlich mehr mögliche Aktionen. Daher kann es sein, dass die aktuelle Implementierung von PPO nicht genügt, um nach dem Training zufriedenstellende Ergebnisse zu liefern. In diesem Fall müssten zusätzliche Optimierungsverfahren, wie z. B. Squashing oder Normalisierung, recherchiert und in den

Algorithmus integriert werden.

5.4 Terraingeneration

5.4.1 Generierung der Terrain-Flächen

Um ein Terrain mit natürlich erscheinenden Höhenunterschieden zu generieren, wird eine Perlin-Noise als Basis verwendet. Hierbei entstehen sanfte Höhenunterschiede, die für eine kleine Wölbung des Terrains sorgen. Mithilfe einer Perlin-Noise lassen sich auf das Terrain auch Flächen setzen, welche stärkere Höhenunterschiede haben. Diese Idee wird verwendet um eine Begrenzung des Terrains in Form von Gebirgen am Rande des Terrains zu generieren. Auch Hindernisse lassen sich mit diesem Ansatz generieren.

Allgemein wird das Platzieren der einzelnen Objekte wie Gebirge, Hindernisse, Vegetation oder Creature-Spawn-Points durch Zonen[5] ermöglicht. Diese Zonen bestimmen die Kategorie des platzierten Objektes an einer Koordinate und halten fest, ob überhaupt ein Objekt an einer Position gesetzt wurde. Dadurch lässt sich auch bestimmen, welcher Bereich des Terrains noch frei ist, oder durch Objekte belegt wurde. Dies ist hilfreich um festzulegen, wo beispielsweise zusätzliche Monster generiert werden können. Auch lassen sich die Zonen nutzen um abhängig von der Umgebung der Zone, passende Vegetation zu generieren oder Texturen zu setzen. Ein Vorteil dieses Ansatzes ist es, dass leicht neue Kategorien von Objekten eingeführt werden können und diese regelbasiert in die Umgebung eingefügt werden können.

5.4.2 Dungeon Generierung mittels Space Partitioning

In diesem Abschnitt wird eine Methode beschrieben Dungeons mithilfe von Space Partitioning prozedural zu generieren. Dungeons sind Level, die aus mehreren Räumen bestehen, die durch Korridore miteinander verbunden sind. Die Räume haben eine rechteckige Grundfläche, deren Breite und Tiefe durch Space Partitioning Algorithmen bestimmt sind. Das hier beschriebene Vorgehen orientiert sich an dem Buch *Procedural Content Creation in Games* [16].

Space Partitioning

Ein Space Partitioning Algorithmus unterteilt einen Raum in Partitionen. In der Regel handelt es sich bei den Ausgangsräumen um zwei- oder dreidimensionale Räume. Hier wird angenommen, dass der zu partitionierende Raum eine rechteckige (zweidimensionale) Fläche ist. Die Partitionierung der Räume erfolgt mithilfe von k-d-Bäumen oder Quadtrees. Beide Algorithmen speichern die Partitionierung des Raumes in einer Baumstruktur und arbeiten rekursiv.

Bei der Partitionierung eines zweidimensionalen Raumes mit einem k-d-Baum wird in jedem Schritt eine Achse zufällig ausgewählt, an der die Eingabefläche in zwei Teile partitioniert wird. Anschließend wird ein Punkt ausgewählt, der die Grenze zwischen

[5] inspiriert von: `https://youtu.be/h9tLcD1r-6w?t=2416`

den beiden Partitionen bestimmt. Dabei stellt die Eingabefläche einen Knoten im Baum dar, an den zwei Kindknoten für die Partitionen angehangen werden. So entsteht ein Binärbaum, in dem die Wurzel die Ausgangsfläche enthält und ein innerer Knoten oder Blattknoten eine der zwei Partitionen des Elternknotens enthält. Der Algorithmus wird so lange rekursiv angewandt, bis ein Abbruchkriterium erfüllt ist. Als Abbruchkriterium kann z.B. eine minimale Größe einer Partition gefordert werden. Alternativ kann die Höhe des Baumes durch einen maximalen Wert beschränkt werden. Die Blätter des Baumes enthalten die Partitionen der Ausgangsfläche. Abbildung 5.7 illustriert die Partitionierung einer quadratischen Fläche mit einem k-d-Baum. Als Abbruchkriterium wurde hier eine minimale Größe der Partitionen gewählt, daher wurden die Partitionen 4 und 5 nicht weiter unterteilt und der Baum ist nicht vollständig.

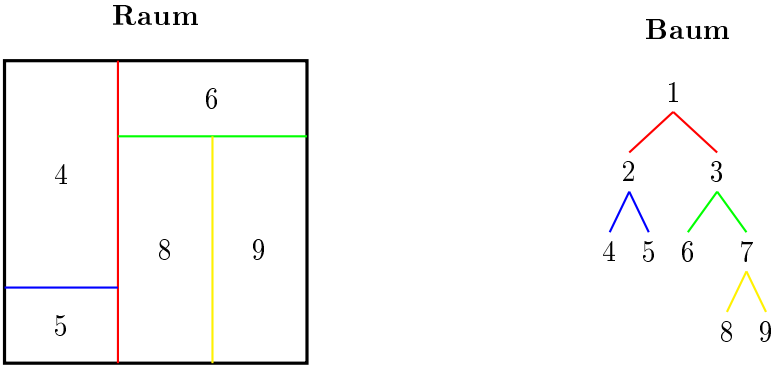


Abbildung 5.7: Space Partitioning mit k-d-Baum

Ein ähnlicher Algorithmus zur Unterteilung rechteckiger zweidimensionaler Flächen in Partitionen verwendet Quadtrees. Hier wird eine Fläche in jedem Schritt in vier Partitionen unterteilt. Damit hat jeder Knoten im Baum entweder keine Kinder oder genau vier Kinder. Bei der Partitionierung einer Fläche kann für jede Achse ein Punkt zufällig gewählt werden, an dem die Fläche partitioniert wird. Alternativ kann eine Fläche auch in vier gleich große Partitionen unterteilt werden. Letztere Methode kann genutzt werden, um einen im Vergleich zu den zufälligen Methoden gleichmäßigeren Dungeon zu generieren. Eine weitere Möglichkeit zur Erstellung der Partitionen an einem Knoten ist es die Fläche zunächst in vier gleich große Partitionen zu unterteilen, den Algorithmus jedoch nur auf einer, zwei oder drei der Partitionen rekursiv aufzurufen. Die restlichen Partitionen werden in diesem Fall direkt zu Blättern im Baum. Als Abbruchkriterien können die gleichen, wie bei k-d-Bäumen verwendet werden. Abbildung 5.8 zeigt die Partitionierung einer quadratischen Fläche mit einem Quadtree, wobei in jedem Schritt in vier gleich große Partitionen unterteilt wird.

Für beide Algorithmen lassen sich weitere Methoden zur Partitionierung der Ausgangsfläche, sowie weitere Abbruchkriterien finden. Weiterhin existieren Verallgemeinerungen bzw. Methoden für dreidimensionale Räume, wie z.B. Octrees.

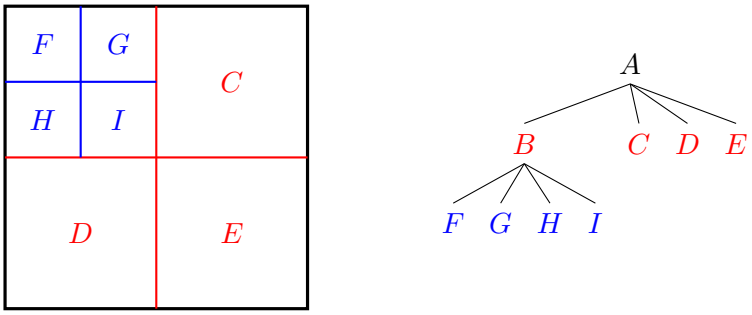


Abbildung 5.8: Space Partitioning mit Quadtrees

Platzierung der Räume und Korridore

Ausgehend von einer in einer mittels Space Partitioning erstellten Baumstruktur vorliegenden Partitionierung einer rechteckigen Fläche können nun Räume in den Partitionen platziert werden. Da die Partitionen zunächst direkt aneinander anliegen, werden die Räume mit einem durch Parameter beschränkten, zufällig gewählten Abstand zu der linken, rechten, oberen und unteren Begrenzung platziert. Nach der Platzierung der Räume wird der Baum traversiert, um Räume durch Korridore miteinander zu verbinden. Dazu wird in jedem Knoten ein Korridor generiert, der zwei Räume in den Partitionen der Teilbäume der Kinder miteinander verbindet. Der Korridor verläuft dabei durch eine in dem Knoten gespeicherte Unterteilungsebene, wodurch sichergestellt wird, dass Korridore sich nicht überschneiden. Ein Beispiel für einen prozedural generierten Dungeon mit zugehörigem k-d-Baum ist in Abbildung 5.9 gegeben. Der blau markierte Korridor wurde beispielsweise im Knoten 0 platziert.

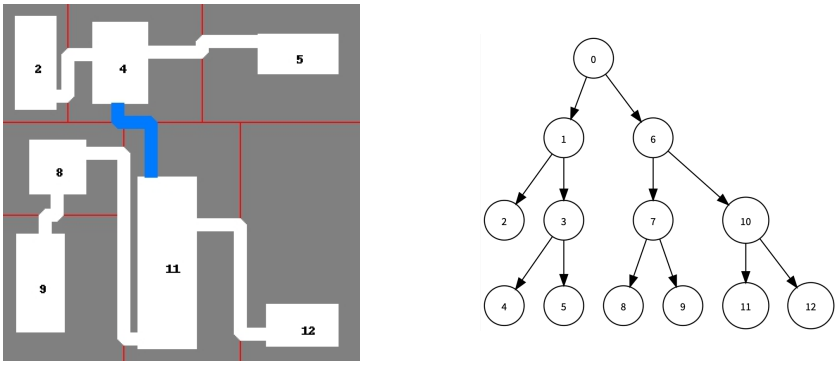


Abbildung 5.9: Prozedural generierter Dungeon mit k-d-Baum

6 Technische Umsetzung

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:

$$\begin{pmatrix} \lceil \frac{\text{Anzahl der Arenen}}{\sqrt{\text{Anzahl der Arenen}}} \rceil \\ 0 \\ \text{Anzahl der Arenen} \mod \sqrt{\text{Anzahl der Arenen}} \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{6.17}$$

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` 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.

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

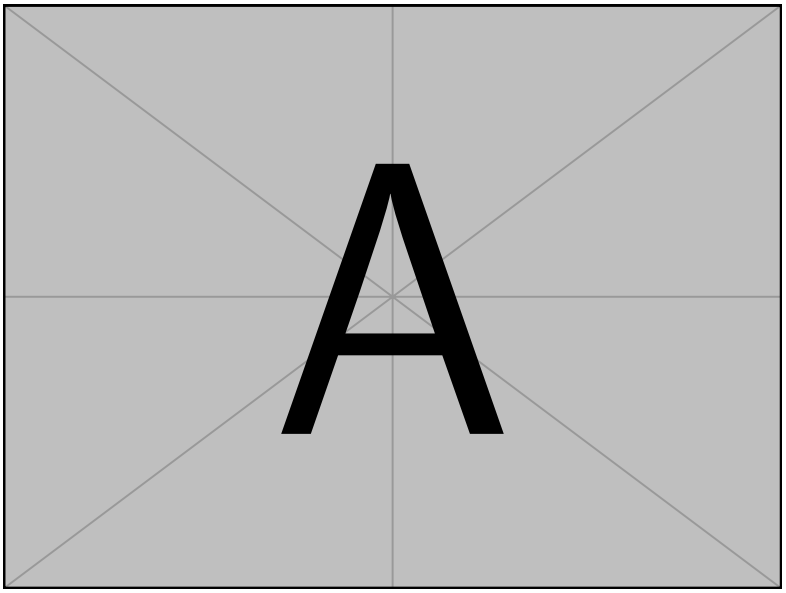


Abbildung 6.1: Konfigurationsmöglichkeiten des `DynamicEnviormentGenerator` im Unity-Editor.

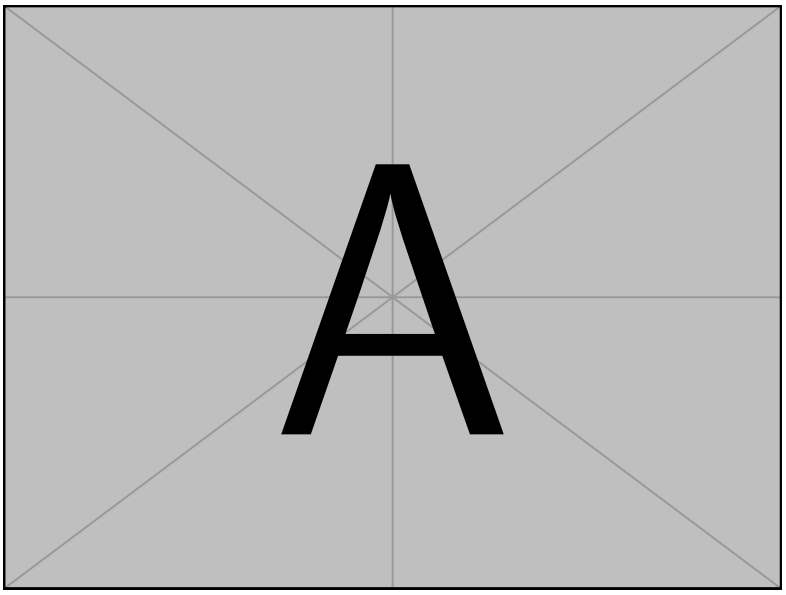


Abbildung 6.2: Ein Beispiel der generierten Trainingsumgebung mit mehreren Arenen.

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 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 In-

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

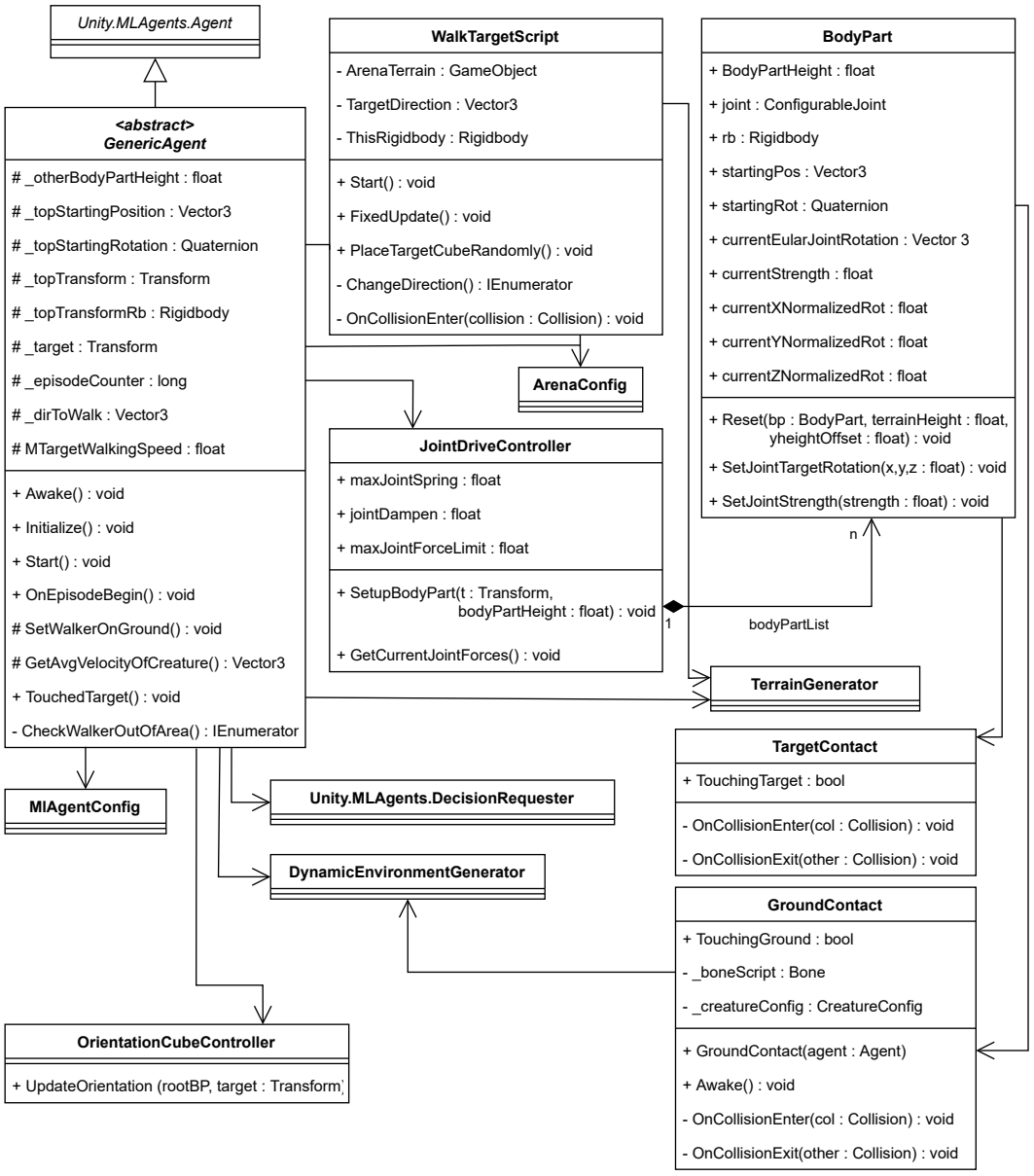


Abbildung 6.3: UML-Diagram der GenericAgent-Klasse und weitere relevante Klassen

itialisierung 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.

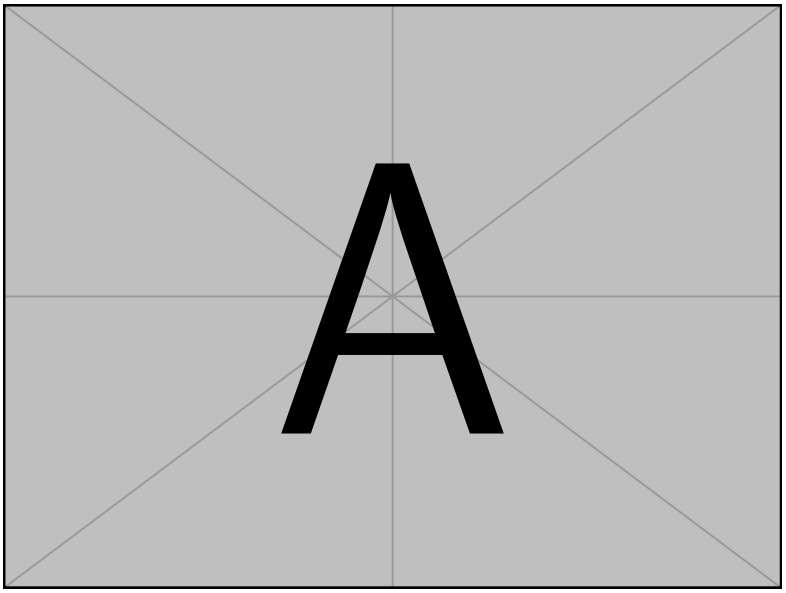


Abbildung 6.4: Parallel ausgeführte Multi-Agent Builds

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 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[9];

```
#!/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-linux_x86_
```

Für die Python-Installation wurde auf Anaconda[3] zurückgegriffen. Die installierte

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

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.

6.2 Creature Generation

Das folgende Kapitel ist eine Tour durch den PG649 Creature Generator und wird die wichtigsten Datenstrukturen und Klassen erläutern. Ziel der Tour ist es sowohl eine Hilfe beim Lesen des Quellcodes zu sein, als auch Design-Entscheidungen und Trade-Offs zu erläutern.

6.2.1 Unity Package

Der Generator wird als unabhängiges Unity Package entwickelt. So kann der Generator einfach in KI-Lernumgebungen und das spätere Spiel eingebunden werden und es wird eine saubere API für den Generator ermutigt.

Die Dateistruktur des Generators unterscheidet sich damit von einem typischen Unity Projekt. Statt im `Assets`-Ordner, liegen alle hier erläuterten Klassen in `Packages/com.pg649.creaturegenerator/Runtime`. Der `Assets`-Ordner enthält lediglich Debug-Skripte, die nicht exportiert werden sollen.

6.2.2 Konfiguration

Die Klassen `CreatureGeneratorSettings` und `ParametricCreatureSettings` enthalten alle Konfigurationsmöglichkeiten für den Generator. Sie sind der Hauptweg mit dem Nutzer mit dem Generator interagieren und bilden somit den Anfang der Tour.

Die Klasse `CreatureGeneratorSettings` enthält Einstellungen, die das Verhalten des Generators und der generierten Kreaturen bestimmen. Dazu gehören Einstellungen die zum Beispiel das generieren eines Meshes für die Kreatur abschalten, Einstellungen für das physikalische Verhalten der Kreature, sowie Einstellungen für Debug-Optionen. Die individuellen Einstellungen sind in der Klasse selbst dokumentiert und werden hier nicht einzeln aufgeführt.

Die Klasse `ParametricCreatureSettings` enthält Einstellungen, die das Aussehen der generierten Kreaturen bestimmen. Die Einstellungen definieren Intervalle für die erlaubte Länge, Dicke, und ggbf. Anzahl für Knochen bestimmter Kategorien. Wieder sind die individuellen Einstellungen in der Klasse selbst dokumentiert.

Beide Konfigurationsklassen sind sogenannte `ScriptableObjects`. Sie können von Unity serialisiert und als Assets gespeichert werden. So können für das spätere Spiel Einstellungen für verschiedene Kreaturen genau so mit exportiert werden wie beispielsweise Shader. Außerdem ist es möglich die Einstellungen mittels `git` zu versionieren.

6.2.3 Bone Definition

Eine der ersten Datenstrukturen, die von dem Generator erzeugt werden, ist ein Baum von `BoneDefinition`s. Dieser Baum bildet die abstrakteste Darstellung eines Skeletts und dient als generisches Ziel für die Generatoren der verschiedenen Kreaturen-Typen. Jede `BoneDefinition` enthält die Details eines Knochens, d.h. um was für einen Knochen es sich handelt (Arm, Bein, etc.), die Länge und Dicke des Knochens, die Ausrichtung seines lokalen Koordinatensystems, und Informationen dazu, wie der Knochen für das finale Skelett an seinem Eltern-Knochen angebracht werden soll.

Letztere Informationen werden `AttachmentHint` genannt und erlauben es Knochen relativ zur Größe des Elternknochens zu positionieren, sie um einen absoluten Vektor zu verschieben, die ventrale Achse des Koordinatensystems auszurichten, und zuletzt den Knochen in eine gewünschte Ausgangspose zu rotieren. So können humanoide Kreaturen beispielsweise in die typische T-Pose gebracht werden.

Es gibt drei Gründe das lokale Koordinatensystem eines jeden Knochens explizit anzugeben:

- Quellcode außerhalb der, später beschriebenen, paramatrischen Generatoren ist nicht durchsetzt mit Konventionen und Annahmen über Koordinatensysteme; Statdessen sind die gewählten Koordinatensysteme explizit.
- es erlaubt die Wahl von semantisch bedeutungsvollen Koordinatenachsen (Proximal, Ventral, Lateral)
- es erlaubt unterschiedlichen parametrischen Generatoren eigene Konventionen für ihre Koordinatensysteme zu wählen.

6.2.4 Parametrische Generatoren

Parametrische Generatoren haben die Aufgabe anhand der `ParametricCreatureSettings` `BoneDefinition`-Bäume zu generieren. Das Package enthält momentan Generatoren für zwei verschiedene Typen von Kreaturen: `BipedGenerator` und `QuadrupedGenerator`. Einstiegspunkte in die Generatoren sind jeweils die `BuildCreature` Methoden.

Beide Generatoren erzeugen zunächst aus den Intervallen in den `ParametricCreature-Settings` zufällig tatsächliche Längen, Dicken, and Anzahlen in Form einer `Biped-SettingsInstance` bzw. `QuadrupedSettingsInstance`. Die generierte Einstellungs-Instanz ist später Teil der Metadaten die zusammen mit der Kreatur zur Verfügung gestellt werden und wird während des Lern-Prozesses genutzt. Zu diesem Zweck implementieren sie das `ISettingsInstance`-Interface. Die Werte anfangs zu generieren erleichtert es außerdem die Symmetrie der Kreatur sicherzustellen.

Um die Kreaturen später trainieren zu können, müssen sie mehrfach generierbar sein. Beide Generatoren akzeptieren deshalb einen Seed für den Zufallsgenerator. Dabei ist zu beachten, dass der selbe Seed in der selben Version des Packages die selbe Kreatur erzeugen wird. Der Aufwand die Stabilitäts-Garantie auch über Package Versionen hinweg zu garantieren, wurde für nicht nötig gehalten und wurde nicht betrieben.

Nachdem die Parameter der Knochen finalisiert wurden, konstruieren beide Generatoren einen Baum aus `BoneDefinition`s.

Explain tree structure if not done in chapter 5

6.2.5 Skeleton Definition

Die Ausgabe der Parametrischen-Generatoren ist eine `SkeletonDefinition`, bestehend aus dem `BoneDefinition`-Baum, der Einstellungs-Instanz, und einem `LimitTable`. Die `LimitTable`-Klasse ist dabei eine Tabelle, die festhält um welche Koordinatenachsen und wie weit sich jeder Knochen rotieren darf.

Die `SkeletonDefinition` dient dann im nächsten Schritt als Eingabe für den `Skeleton-Assembler`.

6.2.6 Skeleton Assembler

Der `SkeletonAssembler` baut aus der `SkeletonDefinition` einen Baum aus Unity `GameObject`s, der dann in Szenen als Ragdoll verwendet werden kann. Einstiegspunkt dafür ist die Methode `Assemble`.

In einem ersten Durchgang wird für jede `BoneDefinition` des Baumes ein `GameObject` erstellt. Jedes dieser `GameObject`s wird mit mehreren Komponenten ausgestattet:

- ein `Rigidbody`, damit physikalische Kräfte auf den Knochen wirken können
- ein `Collider`, damit der Knochen mit anderen Objekten kollidieren kann. Die Form des `Colliders` hängt vom Typen des Knochen ab.
- ein `Bone`, der Metadaten, wie z.B. Länge oder Kategorie des Knochens, enthält

Der Wurzel-Knochen wird zusätzlich mit einer `Skeleton`-Komponente ausgestattet, die weitere Metadaten über das Skelett als ganzes enthält und einfaches iterieren über alle Knochen erlaubt.

Die Nicht-Wurzel Knochen werden entsprechend ihres `AttachmentHints` positioniert. Lediglich die Rotation in die Ausgangspose wird noch nicht angewandt, da dies die Ausrichtung der ventralen Achse aller Knoten unterhalb des momentanen Knoten beeinflussen würde.

Optional wird an dieser Stelle ein weiteres `GameObject` unter jeden Knoten gehangen, welches ein Mesh enthält, dass den `Collider` des Knochens visualisiert.

In einem weiteren Durchgang wird das Skelett zunächst in seine Ausgangspose rotiert. Danach werden Eltern-Kind Paare von Knochen mittels Unitys `ConfigurableJoint`-Komponente verbunden. Die Reihenfolge ist hier essentiell, da die Joints die Position der Knochen zum Zeitpunkt der Erstellung der Joints als Ruheposition ansehen.

Die `ConfigurableJoints` erlauben das setzen einer Ziel-Position und Ziel-Rotation und errechnen dann selbstständig die nötigen Kräfte, die auf ihren verbundenen Körper wirken müssen, um diese zu erreichen. Die Machine-Learning Verfahren produzieren Ziel-Rotationen für jeden Joint. Die Joints sind damit Herzstück des Bewegungssystems und ihre Konfiguration wird daher später näher erläutert.

Zuletzt wird noch der Wurzel-Knochen markiert und die von dem parametrischen Generator erzeugte Einstellungs-Instanz in der `Skeleton`-Komponente hinterlegt.

Configurable Joints

Die lineare Bewegung der Joints wird vollständig gesperrt. Dazu werden die `xMotion`, `yMotion`, `zMotion` Felder auf `Locked` gesetzt. Die Joints halten nun, soweit physikalisch möglich, ihre Position relativ zum Eltern-Knochen.

Die Rotation der Joint wird entsprechend der `LimitTable` eingeschränkt. Die `angularXMotion`, `angularYMotion`, `angularZMotion` Felder werden entsprechend auf `Locked` oder `Limited` gesetzt, und die dazugehörigen `angularLimits` werden ausgefüllt. Dabei gibt es zwei Dinge zu beachten.

Zum einen erlauben die Joints nur für die x-Achse die Angabe eines minimalen und maximalen Winkels, Rotationen um die y- und z-Achse können nur symmetrisch eingeschränkt werden. Allerdings haben die Joints ein eigenes Koordinatensystem separat von dem des Knochens. Der `LimitTable` enthält deshalb gegebenenfalls außerdem Informationen darüber welche Koordinatenachse des Knochens als x-Achse des Joints fungieren soll.

Zum anderen müssen Knochen behandelt werden, die zueinander gespiegelt sind. So muss zum Beispiel der eine Arm eines Zweibeiners im Uhrzeigersinn rotieren, um nach vorne bewegt zu werden, der andere aber gegen den Uhrzeigersinn. Die `Bone`-Komponenten enthalten deshalb das Feld `Mirrored`, was angibt ob der Knochen gespiegelt ist. Ist der Knochen gespiegelt, so werden die Koordinatenachsen des Joint-Koordinatensystems mit -1 multipliziert. So genügt ein einzelner Eintrag im `LimitTable` für beide Versionen des Knochens.

Zuletzt wird noch der `projectionMode` des Joints auf `PostionAndRotation` gestellt, um den Joint zu zwingen die gesetzten Rotations-Limits einzuhalten und für Debug-Zwecke wird der `slerpDrive` initialisiert, damit der Joint Kraft aufwenden kann.

6.2.7 Mesh Generator

Das Mesh der Kreatur wird mit Hilfe der Klasse `Metaball` erzeugt Einem Metaball-Objekt können einzelne Körper über die Methode `AddBall` hinzugefügt werden. Diese werden durch die Klasse `Ball` realisiert. Um andere Körper als Kugeln erstellen zu können, muss eine Klasse erstellt werden, die von `Ball` erbt und die Distanzfunktion anpasst, wie wir es für die `Capsule` getan haben. Die Definitionen verschiedener Falloff-Funktionen sind ebenfalls in der Ball-Klasse implementiert. Die jeweils verwendete Funktion wird als Enumeration über einen Parameter übergeben.
Die Klasse `Metaball` enthält außerdem eine statische Methode `BuildFromSkeleton`, die

das Mesh für ein gesamtes Skelett erzeugt. Dabei wird für jeden Knochen eine korrekt skalierte Metakapsel an der richtigen Position erstellt.
Aus der daraus resultierenden impliziten Definition der Oberfläche wird anschließend das diskrete Mesh generiert, dies ist durch die Klasse `MeshGenerator` und die darin enthaltene Methode `Generate` realisiert. Über das Klassenattribut `gridResolution` lässt sich die Auflösung des zur Abtastung verwendeten Gitters einstellen. Unsere Implementierung des Marching Cubes Algorithmus basiert auf einem existierenden Projekt des GitHub-Users Scrawk [10] .

6.2.8 Creature Generator

Der oben beschriebene Ablauf des Creature-Generators ist implementiert in der Klasse `CreatureGenerator`, die zugleich das öffentliche Interface des Generators ist. Die Methoden `ParametricBiped` und `ParametricQuadruped` erstellen jeweils die passende `SkeletonDefinition` und übergeben sie an die Methode `Parametric`, die daraus die vollständige Kreatur generiert.

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
     - 4.2.4. Mesh Generation: Leonard
     - 4.2.5. Automatisches Rigging: Leonard
   - 4.3. Creature Animation
     - 4.3.1. Trainingsumgebung: Carsten
     - 4.3.2. Training: Niklas
   - 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~~ für Endbericht, bisher nicht implementiert
5. Technische Umsetzung
   - 5.1. Creature Generation: Markus

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

Literatur

[1] Ilya Baran und Jovan Popović. „Automatic Rigging and Animation of 3D Characters“. In: ACM Transactions on Graphics. Bd. 26. 3. Association for Computing Machinery, 2007, S. 72–80.

[2] Alexander Bobenko und Boris Springborn. „A Discrete Laplace–Beltrami Operator for Simplicial Surfaces“. In: Discrete & Computational Geometry 38 (2007), S. 740–756.

[3] Mario Botsch, David Bommes und Leif Kobbelt. „Efficient Linear System Solvers for Mesh Processing“. In: Proceedings of the 11th IMA International Conference on Mathematics of Surfaces Berlin, Heidelberg: Springer-Verlag, 2005, S. 62–83.

[4] Blender Online Community. Blender - a 3D modelling and rendering package. Blender Foundation. Stichting Blender Foundation, Amsterdam, 2018. URL: http://www.blender.org.

[5] Hao Dong u. a. Deep Reinforcement Learning: Fundamentals, Research, and Applications. Hrsg. von Shanghang Zhang Hao Dong Zihan Ding. http://www.deepreinforcementlearningbook.org. Springer Nature, 2020.

[6] L. Graesser und W.L. Keng. Foundations of Deep Reinforcement Learning: Theory and Practice in Python. Addison-Wesley Data & Analytics Series. Pearson Education, 2019. ISBN: 9780135172483.

[7] Jonathan A. Hudson. „Creature Generation using Genetic Algorithms and Auto-Rigging“. In: 2013.

[8] Madis Janno. „Procedural Generation of 2D Creatures“. In: 2018.

[9] LiDO3. URL: https://www.lido.tu-dortmund.de/cms/de/LiDO3/LiDO3_first_contact_handout.pdf.

[10] Marching Cubes. 7. Mai 2022. URL: https://github.com/Scrawk/Marching-Cubes.

[11] mlAgents: Beispiel Umgebungen. https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Learning-Environment-Examples.md. Accessed: 24-09-2022.

[12] Przemyslaw Prusinkiewicz und Aristid Lindenmayer. The algorithmic beauty of plants. The virtual laboratory. Springer, 1990. ISBN: 978-0-387-94676-4.

[13] John Schulman u. a. High-Dimensional Continuous Control Using Generalized Advantage Estimation. 2015. DOI: 10.48550/ARXIV.1506.02438. URL: https://arxiv.org/abs/1506.02438.

[14] John Schulman u. a. „Proximal Policy Optimization Algorithms“. In: CoRR abs/1707.06347 (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

[15] John Schulman u. a. Trust Region Policy Optimization. 2015. DOI: 10.48550/ARXIV.1502.05477. URL: https://arxiv.org/abs/1502.05477.

[16] Noor Shaker, Julian Togelius und Mark J. Nelson. Procedural Content Generation in Games -. Berlin, Heidelberg: Springer, 2016. ISBN: 978-3-319-42716-4.