



Projektgruppenbericht

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, Carsten Kellner, Markus Mügge, Thomas Rysch,
Jannik Stadtler, Tom Voellmer

24. März 2023

Betreuer:

Prof. Dr. Günter Rudolph
Nicolas Fischöder
Marco Pleines

Zusammenfassung

Bei der Entwicklung eines Videospiels kommen prozedurale Inhaltsgenerierung und Deep Reinforcement Learning immer häufiger zum Einsatz. Nicht nur können damit Zeit und Ressourcen gespart werden, sondern auch Flexibilität in dem Entwurf des Spieles aufrechterhalten werden. Die Wahl geeigneter Verfahren für die Kreaturen- und Inhaltsgenerierung und für das Animieren der Kreaturen bildet dabei oft eine zentrale Herausforderung. Im Rahmen einer Projektgruppe der TU-Dortmund, nehmen sich somit 11 Teilnehmer über einen Zeitraum von 2 Semestern der Aufgabe an, in insgesamt 3 Untergruppen, den Creature-Generatoren, World-Generatoren und Creature-Animatoren, einen Prototypen eines Videospiels mittels prozeduraler Inhaltsgenerierung und Deep Reinforcement Learning zu entwickeln. In dieser Ausarbeitung werden für die Kreaturen- und Inhaltsgenerierung, Methoden wie das L-System, Space-Partitioning, Metaball-Generierung, die Bone-Heat-Methode für automatisches Rigging und weiterhin eigen entwickelte, parametrische Methoden näher betrachtet. Währenddessen werden für das Animieren der Kreaturen ein RL-Framework namens NeroRL, Ansätze des ML-Agents von Unity und LiDO3 genutzt. Dabei werden während des Zeitraumes der Entwicklung sowohl technische Umsetzungen und die weitere Erforschung der jeweiligen Fachgebiete erörtert, als auch die organisatorischen Mittel und Entscheidungen der Mitglieder über den Verlauf der Implementierung des Spiels festgehalten.

Inhaltsverzeichnis

Zusammenfassung	i
1 Einleitung	1
1.1 Motivation und Problemstellung	1
1.2 Zielsetzung und Vorgehensweise	2
1.3 Übersicht	4
2 Grundlagen	5
2.1 L-System	5
2.1.1 Grafische Darstellung in 2D	6
2.1.2 Erweiterungen	6
2.1.3 Auswertung eines L-Systems	9
2.2 Dungeon Generierung mittels Space Partitioning	10
2.2.1 Space Partitioning	10
2.2.2 Platzierung der Räume und Korridore	12
2.3 Metaball	13
2.4 Reinforcement Learning	14
2.4.1 Begriffe des Reinforcement Learning	14
2.4.2 Policy-basierte & Value-basierte Algorithmen	15
2.4.3 Actor-Critic	17
2.4.4 Proximal Policy Optimization (PPO)	18
2.5 Unity Features: NavMeshes	20
3 Verwandte Arbeiten	23
3.1 Creature Generation using Genetic Algorithms and Auto-Rigging	23
3.2 Procedural Generation of 2D Creatures	23
3.3 Charakteranimation	24
4 Projektorganisation	27
4.1 Creature Generator	31
4.2 Creature Animator	32
5 Umsetzung	33
5.1 Creature Generation	33
5.1.1 Fachliche Umsetzung	33
5.1.2 Technische Umsetzung	44

Inhaltsverzeichnis

5.2	Creature Animation	50
5.2.1	Trainingsumgebung	50
5.2.2	Erweiterung der Agent-Klasse	54
5.2.3	Konkrete Implementationen der Agents	57
5.2.4	RL-Framework	59
5.2.5	LiDO3	64
5.3	World Generation	67
5.3.1	Übersicht	67
5.3.2	Generierung von Levels mit Space Partitioning	67
5.3.3	Levels	67
5.3.4	Generierung des Terrains	70
5.3.5	Terrain-Transformationen	73
5.3.6	L-System Vegetation	78
5.4	Spiel	81
5.4.1	Design	81
5.4.2	Implementation	82
6	Evaluation	87
6.1	Einschränkungen der Evaluation	87
6.1.1	Auswahl der Kreaturen	87
6.1.2	Konfiguration der Trainingsdurchläufe	88
6.2	Ergebnisse	89
6.2.1	Vierbeiner Kreatur	89
6.2.2	Zweibeiner Kreatur	92
6.3	Probleme	96
6.3.1	Stabilität des Skeletts	96
6.3.2	Dokumentation	96
6.3.3	Organisatorische Probleme	98
6.4	Diskussion	99
7	Fazit & Ausblick	103
	Themenallokation	105
	Abbildungsverzeichnis	109
	Tabellenverzeichnis	111
	Algorithmenverzeichnis	113
	Literaturverzeichnis	115

1 Einleitung

1.1 Motivation und Problemstellung

Das Modellieren einer Kreatur in der Videospielindustrie wird üblicherweise von Spieldesignern vom Skelett, über die Haut und die Texturen welche auf der Haut sichtbar sind vormodelliert. Dieser Prozess ist für eine Vielzahl von sich variierenden Spielcharakteren jedoch sehr mühsam und kostet viel Zeit. Daher spielt die prozedurale Generierung von NPCs, sowie das Erlernen von Bewegungen und das (automatisierte) Animieren dieser eine immer größer werdende Rolle und stellt damit auch eine zentrale Herausforderung dar. Kreaturen müssen oftmals zur Laufzeit und somit effizient generiert werden und weiterhin müssen die durch maschinelles Lernen angeeigneten Animationen auf ein möglichst breites Spektrum an Kreaturen anwendbar sein. Somit ist es relevant, dass das Training der Kreaturen möglichst generalisiert stattfindet, sodass bei der Kreaturen-Generierung eine große, sich bei den Körpermerkmalen variierende Menge der Kreaturen erzeugt werden kann, damit durch die Animation möglichst wenig Einschränkungen für die Körpereigenschaften auftreten. Somit sollten die Animationen auf einen möglichst breiten Pool von Kreaturen anwendbar sein, dass für Kreaturen mit neuen Körpermerkmalen nicht neu trainiert werden muss.

Eine zentrale Herausforderung bei der Generierung von Spielfiguren ist das automatische Aufspannen eines Meshes über den bis zu diesem Zeitpunkt untexturierten Körper der Kreatur; das sogenannte *Automatic Rigging*, welches ebenfalls zu dem prozeduralen Erzeugen von Kreaturen dazugehört. Es muss hier das Problem betrachtet werden Meshes, welche ebenfalls prozedural erzeugt werden, auf das breite Spektrum von verschiedenen Körpераusprägungen des Kreaturen-Pools anwenden zu können, ohne dass es zu sehr von den Körperperformen der Kreaturen abweicht.

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

In dieser Arbeit wird für die Generierung von Skeletten für Kreaturen, das Rigging, eine parametrisierbare Methode implementiert, die sich an dem Paper [17] von Jonathan A. Hudson orientiert. Das Erzeugen eines Meshes für die Skelette wird mit einer Metaball-Generierung 5.1.1 erreicht. Für das anschließende Zuordnen des Meshes zu den Komponenten des Skeletts, das Skinning, wird die Bone-Heat-Method [3] betrachtet.

Für das Animieren, das automatische Fortbewegen und Verhalten der Kreaturen, wird maschinelles Lernen verwendet; das Sammeln von Observationen des Verhaltens der

1 Einleitung

Kreaturen wird innerhalb von Unity in der Movement Arena mittels einer ML-Agents Agent [2] Schnittstelle umgesetzt. Das Verarbeiten und Training aus den resultierenden Observationen wird entweder durch ein Netz aus einer ONNX Datei oder durch NeroRL [29], ein Python Backend, realisiert.

Für die Generierung der Spiellevel und deren restlichen Inhalte wird sowohl ein Space-Partitioning Algorithmus [36], als auch ein Algorithmus basierend auf dem Lindenmayer-System (L-System) [30] entwickelt. Ergänzend werden hauseigene Systeme ausgeprägt, um Themengebiete wie das Placement der Inhalte oder auch Nav-Meshes auf der Spielwelt abzudecken.

1.2 Zielsetzung und Vorgehensweise

Bei der Entwicklung eines Spieles müssen viele Themengebiete zur Planung einkalkuliert werden, um während der Entwicklung eine Übersicht zu behalten und das Endziel stets zu berücksichtigen. Daher wurde sich für die Einteilung des Spieldesigns in folgende verschiedene Bereiche entschieden, an welchen parallel gearbeitet werden kann: 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 erfüllt sein:

Die Generierung von Spielleveln

- Die Generierung der Spiellevel sollte prozedural durchgeführt werden: z.B. durch 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

1.2 Zielsetzung und Vorgehensweise

- 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-ML-Agents-Walker sein

Fortbewegung der Monster

- Animationen sollen nicht händisch erstellt werden
- Die Monster sollen sich mit Hilfe von Deep Reinforcement Learning lernen 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

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 wurde entschieden die Entwicklung des Creature-Generation Tools abzuwarten, sodass mit der Fertigstellung anhand der Stärken des Tools das Spiel-design abgeleitet werden kann. Damit 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**.

1 Einleitung

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 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.

1.3 Übersicht

Zunächst werden in dem Kapitel 2 alle Grundlagen beschrieben in Bezug auf die angeführten Themenbereiche der Projektgruppe welche in der Seminarphase gegenseitig vorgestellt wurden 1.2. Basierend auf diesen Themengebieten werden dann in Kapitel 3 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 4 und 5) ausgemacht wurden. Sobald die Grundlagen über die Themen der späteren fachlichen und technischen Vorgehensweise geklärt sind, wird in Kapitel 4 die Entscheidungsgrundlage für die organisatorische Vorgehensweise beschrieben, welches sich während der Entwicklungsphase der verschiedenen Bereiche 1.2 erschlossen hat. Nachdem das organisatorische Prozedere bekannt ist, wird im Anschluss die fachliche Vorgehensweise und die Umsetzung in Kapitel 5 evaluiert. Anschließend werden in Kapitel 6.1 die Ergebnisse präsentiert und diskutiert in Kapitel 6.4. Schließlich wird in Kapitel 7 zusammengefasst, auf welche (Forschungs-)Bereiche die Erkenntnisse dieser Studie ausgeweitet werden können.

2 Grundlagen

2.1 L-System

In der Natur folgt der Aufbau vieler Pflanzen dem Prinzip der Selbstähnlichkeit. Das heißt, es lassen sich dieselben geometrischen Strukturen auf makroskopischer und mikroskopischer Ebene wiederfinden [36]. Beispielsweise ähneln die Knospen eines Blumenkohls auch seiner äußeren Struktur, wie in Abbildung 2.1 zu sehen ist.



Abbildung 2.1: Blumenkohl¹

Um diese Eigenschaften präzise und strukturell darzustellen kann ein Lindenmayer-System (L-System) [30] 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 [38] 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.

¹Quelle: <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

2.1.1 Grafische Darstellung in 2D

Der resultierende String einer Ableitung kann grafisch als Zeichenvorschriften für *Turtle graphics* [11] 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. Für die Drehung der Turtle werden die Terminalssymbole $+$ und $-$ verwendet. In Abbildung 2.2 ist eine Visualisierung dieses Konzeptes zu sehen. Hier ist die Turtle initial nach oben gerichtet.

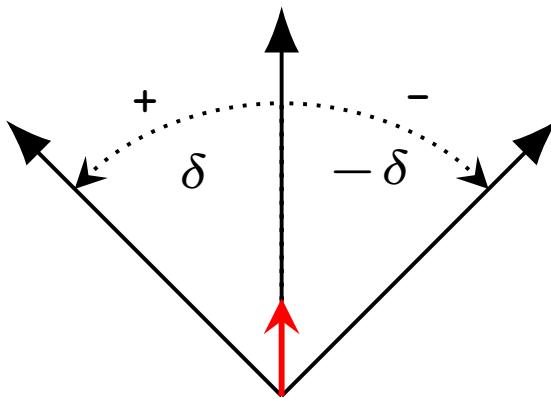


Abbildung 2.2: L-System 2D Rotation. Der rote Pfeil zeigt in die Richtung der Turtle.

Ohne Erweiterungen des L-Systems 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.

2.1.2 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 [36] 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

2.1 L-System

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. Die Turtle wird zu der Position und Rotation zurückgesetzt, die auf dem Stack als oberstes Element gespeichert ist, wenn ein] gelesen wird.

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

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

$$F \rightarrow F [+F] F [-F] [F]$$

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 2.3 dargestellt.



Abbildung 2.3: 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* [36] verwendet werden. Hierbei kann es mehrere Produktionen mit gleichen linken Seiten geben. Jeweils über die Menge von

2 Grundlagen

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

$$\begin{aligned} F &\xrightarrow{0.33} F [+F] F [-F] F \\ F &\xrightarrow{0.33} F [+F] F \\ F &\xrightarrow{0.34} F [-F] F \end{aligned}$$

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 2.4 gegeben. Es ist gut erkennbar wie aus einem L-System deutlich unterschiedliche Strukturen generiert werden können, die jedoch alle ähnliche Grundstrukturen aufweisen.

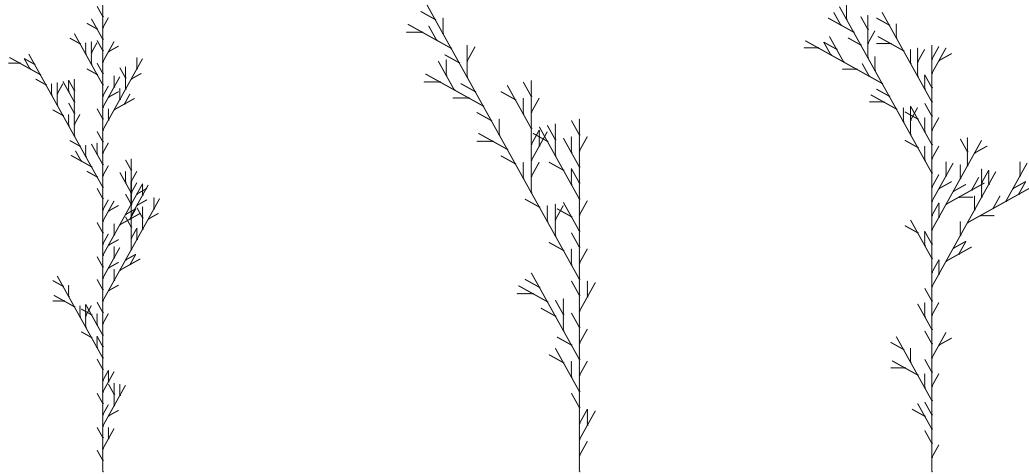


Abbildung 2.4: Drei Resultate desselben stochastischen L-Systems

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

2.1 L-System

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 Beispiel² für ein L-System, das einen dreidimensionalen Baum erzeugt ist im Folgenden gegeben:

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

$$\begin{aligned} A &\rightarrow [B] // [B] // B \\ B &\rightarrow \& FFFA \end{aligned}$$

Mit diesem L-System kann ein dreidimensionaler Baum wie in Abbildung 2.5 in Unity erzeugt werden. Es wurden 7 Ableitungsschritte durchgeführt und es wurde jeweils 28° um die Achsen rotiert.



Abbildung 2.5: L-System 3D Baum

2.1.3 Auswertung eines L-Systems

In diesem Abschnitt wird die Auswertung eines L-Systems beschrieben. Ausgehend vom Startstring S werden i Ableitungsschritte der Grammatik ausgeführt. Der Algorithmus 1 beschreibt dieses Vorgehen. Zunächst wird der Startstring S in einem String w gespeichert. Der String w wird nun Zeichen-für-Zeichen von links nach rechts durchlaufen. Dabei wird einmal pro Iteration ein leerer String für das Zwischenresultat angelegt.

²L-System adaptiert von: <https://www.sidefx.com/docs/houdini/nodes/sop/lSystem.html#3d>

2 Grundlagen

Für jedes gelesene Zeichen des Strings w 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 String w vollständig durchlaufen. Dadurch werden pro Iteration alle Nicht-Terminale des Strings ersetzt. Am Ende der n -ten Iteration wird das Zwischenresultat als neuer String w 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.

Eingabe: Grammatik (N, T, S, P) , Iterationen i

Ausgabe: String w nach i Ableitungsschritten

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

Algorithmus 1: L-System Auswertung

2.2 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* [36].

2.2.1 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

2.2 Dungeon Generierung mittels Space Partitioning

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 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 2.6 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.

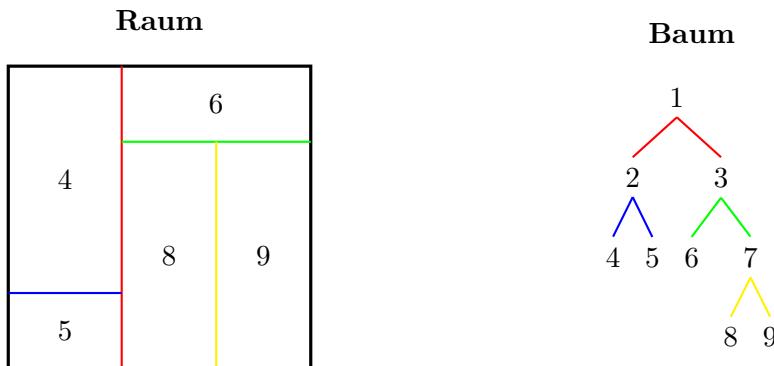


Abbildung 2.6: Space Partitioning mit k -d-Baum

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 2.7 zeigt die Partitionierung einer quadratischen Fläche mit einem Quadtree, wobei in jedem Schritt in vier gleich große Partitionen unterteilt wird.

2 Grundlagen

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.

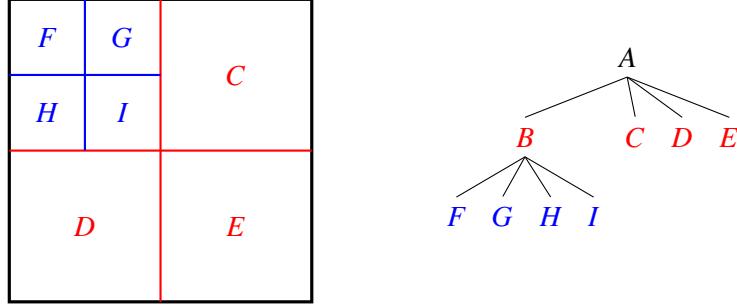


Abbildung 2.7: Space Partitioning mit Quadtrees

2.2.2 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 2.8 gegeben. Der blau markierte Korridor wurde beispielsweise im Knoten 0 platziert.

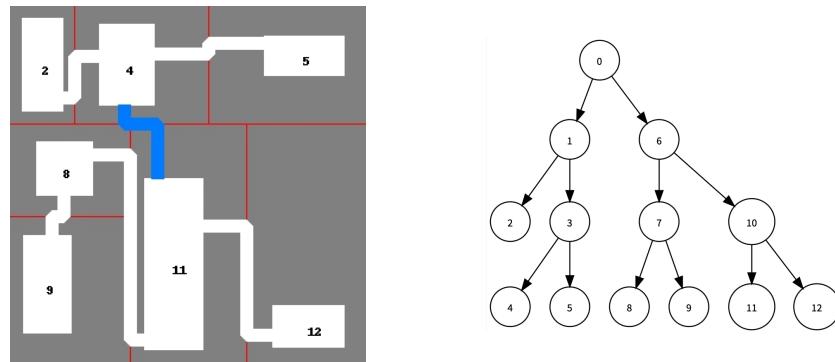


Abbildung 2.8: Prozedural generierter Dungeon mit k -d-Baum

2.3 Metaball

Ein Ansatz zur Erzeugung von visuell ansprechender, natürlich wirkender Geometrie ist der Metaball [5]. Die grundlegende Idee dahinter ist es mehrere Kugeln im Raum zu platzieren und diese ineinander „verschmelzen“ zu lassen, sodass sich daraus ein größeres Objekt ergibt (siehe 2.9).

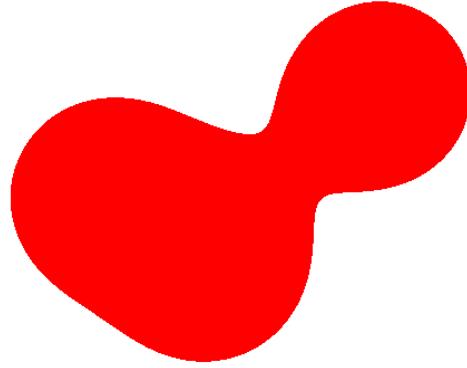


Abbildung 2.9: 2D Beispiel eines Metaballs bestehend aus drei Kugeln

Der Metaball M , bestehend aus n Kugeln, ist durch eine Funktion gegeben, die jedem Punkt \vec{x} im Raum einen Wert zuordnet.

$$M(\vec{x}) = \sum_{i=0}^n f_i(\vec{x})$$

Diese setzt sich aus den Funktionen jeder der n Kugeln zusammen. Für eine einzelne Kugel lässt sich dieser Wert als Abstand zu deren Zentrum interpretieren.

Zusätzlich benötigt man einen Schwellenwert (typischerweise ist dieser 1), der beschreibt, welcher Wert für M den Übergang zwischen innerhalb und außerhalb des Metaballs festlegt. Ist $M < \text{Schwellenwert}$ liegt der Punkt außerhalb des Metaballs, ist $M > \text{Schwellenwert}$ liegt er innerhalb. $M = \text{Schwellenwert}$ beschreibt somit die Oberfläche des Körpers.

Eine mögliche Metaball-Funktion f_i ist

$$f_i(\vec{x}) = \frac{1}{||\vec{c}_i - \vec{x}||} = \frac{1}{r_i(\vec{x})}$$

Wobei \vec{c}_i das Zentrum von Kugel i beschreibt und r_i den Abstand zwischen \vec{x} und \vec{c}_i .

Um aus dieser Funktion ein diskretes Mesh zu erzeugen, lässt sich der Raum mit Hilfe des Marching-Cubes-Algorithmus Abtasten [23] um die Oberfläche zu bestimmen.

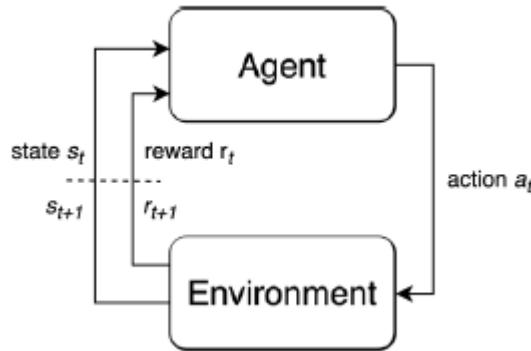


Abbildung 2.10: Reinforcement Learning Kontrollfluss [12]

2.4 Reinforcement Learning

Mit Reinforcement Learning (RL, deutsch: verst rkendes Lernen) werden bestimmte Varianten des maschinellen Lernens bezeichnet. Allgemein lernt beim Reinforcement Learning ein Agent (d.h. ein kontrollierbarer Akteur) eine Policy (deutsch: Strategie), um in einer Umgebung ein Problem zu l sen, das sequentielle Entscheidungen umfasst. Reinforcement Learning ist ein breites Forschungsgebiet. In diesem Grundlagenkapitel werden nur die Grundlagen des Reinforcement Learning beschrieben, die im Rahmen der Projektgruppe relevant sind. Eine 脦bersicht 脰ber weitere Varianten des Reinforcement Learning findet sich z.B. in den B chern von Graesser und Keng und Dong u.a.

2.4.1 Begriffe des Reinforcement Learning

Agent und Umgebung

Abbildung 2.10 stellt den allgemeinen Markov Entscheidungsprozess (MDP) eines Reinforcement Learning Prozesses dar. Der Agent ist in einer Umgebung (Environment) aktiv. In regelm igen Schritten sammelt der Agent in der Umgebung Observationen, bestehend aus einem aktuellen Zustand (State) s_t , sowie einer Belohnung (Reward) r_t , die entweder Null (d.h. keine Belohnung im aktuellen Schritt), oder ein positiver oder negativer Wert sein kann. Die durch den Agent gelernte Policy bildet den State auf eine Aktion a_t ab. Die Aktion wird in der Umgebung ausgefuhrt. Danach ergibt die Umgebung den neuen State s_{t+1} und die neue Belohnung r_{t+1} an den Agenten [12].

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

2.4 Reinforcement Learning

Reward aus den einzelnen Schritten einer Episode. Die Gleichung 2.1 stellt die mathematische Schreibweise einer Trajektorie dar. Eine Trajektorie enthält durch die Zeitschritte $[0; t]$ indizierte Tupel. Jedes Tupel beinhaltet den Ausgangszustand s_t , die ausgeführte Aktion a_t und den dadurch erhaltenen Reward r_t zum Zeitpunkt t . Gleichung 2.2 zeigt die Definition des Returns einer Trajektorie. Der Return ist der summierte Reward der Trajektorie. Auf Basis gesammelter Trajektorien kann der Agent einen Lernvorgang durchführen und die Policy aktualisieren [12].

$$\tau = (s_0, a_0, r_0), (s_1, a_1, r_1), \dots, (s_t, a_t, r_t) \quad (2.1)$$

$$R(\tau) = \sum_0^t r_t \quad (2.2)$$

2.4.2 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. Beide Verfahren haben das grundlegende Ziel, die maximale kumulative Belohnung (Return) zu erhalten. Es existieren weitere Unterscheidungen zwischen RL-Methoden. Dazu gehört unter anderem die Unterscheidung zwischen modellbasiertem RL und RL ohne Modell. Bei komplexen Umgebungen eignet sich primär das RL ohne Modell, wobei der Agent keine Kenntnisse über Zustandstransitions- und Belohnungsfunktion besitzt, da die Erstellung eines präzisen Modells zu komplex ist. Im Folgenden wird deswegen von modellfreiem RL ausgegangen [9].

Value-basierte Algorithmen

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

$$V^\pi(s) = \mathbb{E}_{s_0=s, \tau \sim \pi} \left[\sum_{t=0}^T \gamma^t r_t \right] \quad (2.3)$$

$$Q^\pi(s, a) = \mathbb{E}_{s_0=s, a_0=a, \tau \sim \pi} \left[\sum_{t=0}^T \gamma^t r_t \right] \quad (2.4)$$

Die V-Funktion ("Value-Function", siehe Gleichung 2.3) schätzt den Wert eines Zustandes anhand der diskontierten kumulativen Belohnung einer in diesem Zustand startenden Trajektorie, unter der Annahme, dass der aktuellen Policy gefolgt wird. Die Q-Funktion ("Action-Value-Function", siehe Gleichung 2.4) schätzt den Wert eines Zustandes unter

2 Grundlagen

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 [12]. Bei der intuitiven Umwandlung verwendet der Agent als nächste Aktion die Aktion mit dem maximalen Wert der Action-Value-Funktion. Die Berechnung des maximalen Wertes der Action-Value-Funktion ist in kontinuierlichen Aktionsräumen sehr aufwändig, sodass Value-basierte Algorithmen sind Proben-effizienter als Policy-based Algorithmen und haben eine niedrigere Varianz. Allerdings kann eine Konvergenz zu einem lokalen Optimum nicht garantiert werden und die Berechnung des maximalen Wertes der Action-Value-Funktion ist bei kontinuierlichen Aktionsräumen sehr aufwändig, sodass Value-basierte Algorithmen bei kontinuierlichen Aktionsräumen schlechter geeignet sind [24].

Policy-basierte Algorithmen

Bei Policy-basierten Algorithmen wählt eine Policy (π) die nächste Aktion ($a \sim \pi(s)$) 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 2.5). [12]

$$\pi(s) = a \quad (2.5)$$

In der Praxis werden fast ausschließlich stochastische Policies (siehe Gleichung 2.6) verwendet.

$$\pi(a|s) = \mathbb{P}_\pi [A = a | S = s] \quad (2.6)$$

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 gesampt. Policy-basierte Algorithmen lernen die Funktion π , sodass das Objective maximiert wird. Policy-basierte Algorithmen können grundsätzlich auf beliebige Aktionsarten angewendet werden, kontinuierliche Aktionsräume sind möglich. Policy-basierte Algorithmen konvergieren garantiert zu einer lokalen, optimalen Policy. Allerdings sind Policy-basierte Algorithmen Proben-ineffizient und haben eine hohe Varianz. [9]

REINFORCE Ein Beispiel für einen Policy-basierten Algorithmus ist REINFORCE. Der REINFORCE Algorithmus wird hier verwendet, um die Grundlagen der Policy Optimierung per Policy Gradient einzuführen.

$$\vartheta \leftarrow \vartheta + \alpha \nabla_\vartheta J(\pi_\vartheta) \quad (2.7)$$

Gleichung 2.7 zeigt die Optimierung einer Policy. Die optimierte Policy entsteht aus der alten Policy, indem eine Anpassung aus Gradient und Objective Funktion ($J(\pi_\vartheta)$)

2.4 Reinforcement Learning

aufaddiert werden.

$$\nabla_{\theta} J(\pi_{\theta}) = \mathbb{E}_{\tau \sim \pi_{\theta}} \left[\sum_{t=0}^T R_t(\tau) \nabla_{\theta} \log(\pi_{\theta}(a_t | s_t)) \right]; \quad R_t(\tau) = \sum_{t'=t}^T \gamma^{t'-t} r_{t'} \quad (2.8)$$

Dabei werden die Parameter $\pi_{\theta}(a_t | s_t)$ der Policy angepasst. Ungünstige Aktionen ($R_t(\tau) < 0$) werden weniger wahrscheinlich. Günstige Aktionen ($R_t(\tau) > 0$) werden wahrscheinlicher. [12]

2.4.3 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, die in den Abschnitten 2.4.2 und 2.4.2 beschrieben wurden. 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. Der Agent kann dadurch mit gesammelten Teilstücken von Episoden lernen, statt zwingend vollständige Episoden zu benötigen.

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.

$$A^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) - V^{\pi}(s_t) \quad (2.9)$$

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 2.10 dargestellt.

$$\nabla_{\theta} J(\pi_{\theta}) = \mathbb{E}_t \left[A_t^{\pi}(s_t, a_t) \nabla_{\theta} \log(\pi_{\theta}(a_t | s_t)) \right] \quad (2.10)$$

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. [12]

2 Grundlagen

Generalized-Advantage-Estimation (GAE)

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

$$A_{NSTEP}^\pi = \left[\left(\sum_{i=0}^n \gamma^i r_{t+i} \right) + \gamma^{n+1} V^\pi(s_{t+n+1}) \right] - V^\pi(s_t) \quad (2.11)$$

Gleichung 2.11 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 gesampt werden. Die V^π Funktion stellt eine Erwartung über alle möglichen Trajektorien dar und weist damit eine geringe Varianz auf. Da die V^π Funktion gelernt wurde, entsteht dabei Bias. Mit kleinen Werten für n haben n-step Forward Return Advantages eine geringe Varianz bei hohem Bias, mit großen Werten für n haben n-step Forward Return Advantages eine hohe Varianz bei geringem Bias (Bias-Variance Trade-Off). Allgemeine Aussagen über die Wahl des n Parameters sind schwierig.

Generalized Advantage Estimation hebt die explizite Wahl von n aus, indem n-step Forward Return Advantages für verschiedene n Werte gewichtet kombiniert werden.

$$A_{GAE(\gamma, \lambda)}^\pi(s_t, a_t) = \sum_{l=0}^{\infty} (\gamma \lambda)^l \delta_{t+l}; \quad \delta_t = r_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t) \quad (2.12)$$

Die allgemeine Formel für GAE ist in Gleichung 2.12 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. [33]

2.4.4 Proximal Policy Optimization (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 [12]. Da die Trainingsdaten mit der aktuellen Policy gesammelt werden, kann ein Performance Collapse dazu führen, dass die kollabierte Policy sich nicht mehr erholt.

Clipping

Es gibt zwei theoretische Varianten des PPO Algorithmus. Die erste Variante baut direkt auf dem Trust Region Policy Optimization Algorithmus (TRPO) [32] auf und verwendet eine adaptive KL Penalty, um die Schrittgröße einzuschränken. Diese wird hier nicht weiter beschrieben, da in der Praxis nur die zweite Variante von PPO verwendet wird, weil diese in Tests bessere Ergebnisse erzielt und leichter zu implementieren ist. Die zweite PPO Variante nutzt ein Clipped Surrogate Objective.

$$J^{CLIP}(\vartheta) = \mathbb{E}_t [\min(r_t(\vartheta)A_t, clip(r_t(\vartheta), 1 - \epsilon, 1 + \epsilon)A_t)] \quad (2.13)$$

Die Gleichung 2.13 zeigt das Clipped Surrogate Objective, das vom PPO Algorithmus maximiert wird. Eine Veränderung außerhalb des Bereichs $[1-\epsilon; 1+\epsilon]$ wird dabei geclipppt (abgeschnitten) und somit die Schrittgröße eingeschränkt.

$$J^{CLIP+VF+S}(\vartheta) = \mathbb{E}_t [J_t^{CLIP} - c_1 L_t^{VF} + c_2 S[\pi_\vartheta(s_t)]] \quad (2.14)$$

Die für die Clipped Surrogate Objective Variante des PPO Algorithmus verwendete Loss Funktion wird in Gleichung 2.14 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. [34]

```

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

Algorithmus 2: PPO Pseudocode Algorithmus [12]

Algorithmus Algorithmus 2 zeigt den Pseudocode des PPO Algorithmus. Für jede Episode wird zuerst pro Actor eine Trajektorie in der Umgebung gesampt. Dabei werden die Aktionen jeweils durch die aktuelle Policy $\pi_{\vartheta_{old}}$ bestimmt. Für diese Trajektorie werden die zugehörigen Advantages mit GAE berechnet. Mit den gesampleten Trajektorien wird die Policy in Epochen trainiert. In jeder Epoche werden Minibatches aus dem gesampleten Batch generiert. Mit den Daten der Minibatches wird das Clipped Surrogate Objective optimiert.

2 Grundlagen

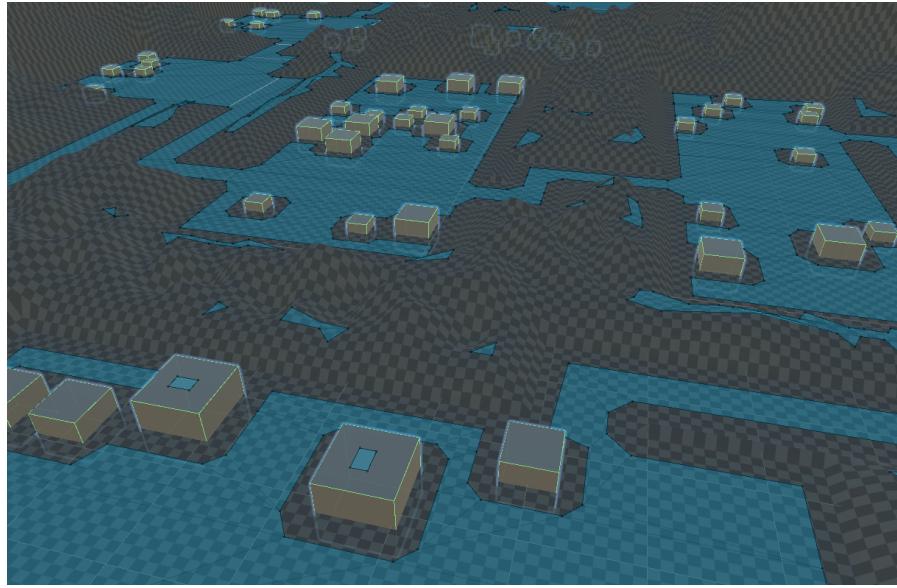


Abbildung 2.11: Prozedural generierte Umgebung mit **NavMesh** (in blau)

2.5 Unity Features: NavMeshes

Um die zu erlernenden Aufgaben zu vereinfachen, bietet es sich an, Wegfindungsalgorithmen zu nutzen (siehe Abschnitt 5.2.3). Unity bietet dafür das sogenannte **NavMesh** an.

Ein **NavMesh** stellt die begehbar Fläche der Szene dar. Ein Agent, welcher sich auf dem **NavMesh** befinden, kann eine Anfrage mit einem gewünschten Zielpunkt an das **NavMesh** senden. Das **NavMesh** generiert dann eine Liste an Punkten auf dem **NavMesh**. Von einem Punkt am Index i zu einem Punkt $i + 1$ existiert dabei immer eine gerader Weg ohne Hindernisse. Außerdem existiert ein gerader Weg von der Position des Agenten zum Zeitpunkt der Generierung des Pfades zum Punkt am Index 0. Der letzte Punkt des Pfades ist der gewünschte Zielpunkt.

Es gibt diverse Möglichkeiten, ein **NavMesh** in Unity benutzen. Wir haben uns dazu entschieden, unserem **Terrain**-Objekt die Komponente **NavMeshSurface** zu geben.

Zu Beginn müssen wir der Komponente einige Informationen über unseren Agent mitteilen. Diese Informationen umfassen:

- Den *Agent Radius*, um den minimalen Abstand zu Hindernissen bestimmen zu können. Sollte dieser Wert zu groß gewählt sein, kann es unter Umständen passieren, dass der Agent durch enge Passagen nicht durchlaufen kann, obwohl dieser eigentlich klein genug dafür wäre.
- Die *Agent Height*

2.5 Unity Features: NavMeshes

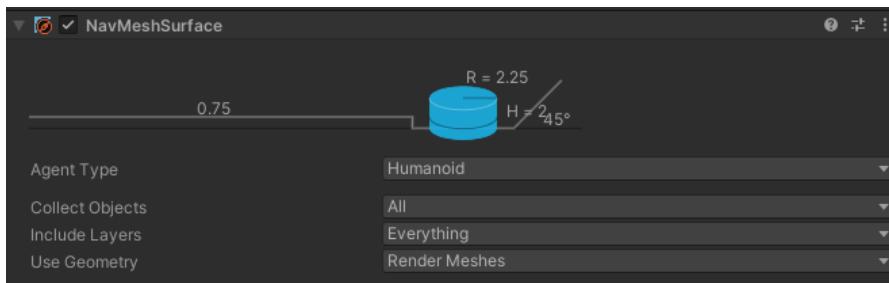


Abbildung 2.12: Auszug aus den Einstellungen für die `NavMeshSurface`-Komponente.

- Die *Max Slope*, welche die maximale Steigung, die der Agent bewältigen kann, beschreibt.
- Die *Step Height*, welche den maximalen Höhenunterschied mit beliebiger Steigung beschreibt, die der Agent unabhängig von der angegebenen *Max Slope* bewältigen kann. Dies ist dazu notwendig, damit der Agent zum Beispiel eine Treppe hochlaufen kann, da die einzelnen Treppenstufen meistens eine Steigung von 90° besitzen.

Im nächsten Schritt sammelt das Skript sämtliche geometrischen `GameObjects` der Szene, um zusammen mit den Informationen über den Agent die begehbar Fläche zu approximieren (auch *Baking* genannt).

Ein `NavMesh` lässt sich auch während der Laufzeit des Programmes generieren. In unserem Anwendungsfall ist dies essentiell, da wir die Umgebung erst zur Laufzeit erstellen. Eine Vorabgenerierung des `NavMeshes` wäre daher unmöglich.

3 Verwandte Arbeiten

3.1 Creature Generation using Genetic Algorithms and Auto-Rigging

In seiner Thesis beschreibt Hudson [17] 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 prinzipielle 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.

3.2 Procedural Generation of 2D Creatures

In dieser Thesis [19] 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

3 Verwandte Arbeiten

setzt sich aus den Gleichungen mehrerer Kugeln $i \in \{1, \dots, N\}$ 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, Jan-no hat sich auf die folgende festgelegt.

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

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.

3.3 Charakteranimation

Die Animation bzw. Fortbewegung von Charaktermodellen spielt sowohl in der Forschung, als auch in praxisnahen Bereichen, wie der Videospielentwicklung, eine wichtige Rolle. Bei der Entwicklung von Videospielen werden primär feste (d.h. Key-Frame basierte) oder prozedural generierte Bewegungsanimationen verwendet [15]. Diese erfordern tiefgehendes Wissen über die zugrundeliegenden Modelle und Umgebungen und sind nicht flexibel einsetzbar. Statt einer physikbasierten Steuerung wird häufig eine kinematische Steuerung verwendet, da diese signifikant leichter zu implementieren ist und intuitivere Bewegung durch Nutzer ermöglicht. Das gewünschte Verhalten wird von den Entwicklern festgelegt. In den letzten Jahren werden vermehrt auch physikbasierte Charakteranimationen eingesetzt, da diese natürlicher mit der Umgebung interagieren können, insbesondere bei unerwarteten Umgebungsänderungen [10].

Charakteranimationen auf Basis von maschinellem Lernen ermöglichen es, in relativ kurzer Zeit viele verschiedenartige Animationen für viele diverse Charaktermodelle zu entwickeln, da keine aufwändige manuelle Definition der Animationen durch Entwickler

3.3 Charakteranimation

oder Grafikdesigner notwendig ist [22]. Aufgrund der hohen Komplexität der Bewegungsabläufe und der hochdimensionalen Aktionsräume, werden für das Lernen von Charakteranimationen Methoden aus dem Bereich des Deep Learning, insbesondere des Reinforcement Learning und Imitation Learning verwendet [28, 25]. Imitation Learning verwendet Generative Adversarial Networks, um die Bewegungsabläufe aus einem Datensatz einer Experten Strategie oder eines Motion Capture Systems zu imitieren, es wird also ein entsprechender Datensatz als Grundlage benötigt [14, 28]. Reinforcement Learning kann in komplexen Umgebungen eingesetzt werden, um dort die Aktionen eines Agenten (d.h. die Animation eines Charakters) zu optimieren. Dabei werden meist spezialisierte Umgebungen und Belohnungsfunktionen für einzelne Charaktermodelle entwickelt, die dann für das Training der Animationen verwendet werden [25].

4 Projektorganisation

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.

Um ein grundlegendes Verständnis aller Teilnehmenden für alle Bereiche zu schaffen, werden zu Beginn der Projektgruppe 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 nacheinander die Themen der Gruppen den restlichen Teilnehmenden vorgestellt. Somit wird eine Verständnisgrundlage für alle Beteiligten geschaffen, sodass basierend auf dem allgemeinen Kenntnisstand mit der eigentlichen Entwicklung des Spieles angefangen wird.

Um anschließend koordiniert das Ziel der Projektgruppe zu erreichen, spielt die Projektorganisation bei einer Gruppe von 12 Teilnehmern eine wichtige Rolle. Im Laufe der Projektgruppe werden Untergruppen gebildet und arbeiten parallel an den verschiedenen Bereichen. Außerdem übernehmen einige Teilnehmer Rollen im Rahmen der Projektleitung. Die primäre Gruppenaufteilung ist in Tabelle 4.1 abgebildet. Initial wird 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 4.1 und 4.2 genauer beschrieben.

4 Projektorganisation

(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
Markus Mügge		Carsten Kellner	
Thomas Rysch			

Tabelle 4.1: Primäre Gruppenaufteilung der Projektgruppe

Projektleitung Aus den initialen Gruppen der Creature-Generation und der Creature-Animation Gruppe sind jeweils zwei Projektmanager einberufen. Diese sind in Tabelle 4.1 gepunktet unterstrichen. Außerdem ist Thomas Rysch als Projektleiter ernannt. Der Projektleiter soll eine Moderator- und Organisationsrolle einnehmen, die Gruppen und Themen zusammen- und im Überblick behalten und somit potentielle Konflikte frühzeitig erkennen und auflösen. Die Projektmanager übernehmen eine analoge Aufgabe im Rahmen ihrer jeweiligen Gruppen und bilden somit gemeinsam mit dem Projektleiter eine Struktur in der die Übersicht über aktuelle Themen einfach beibehalten werden kann.

Jour-Fixe Wöchentlich findet ein Jour-Fixe statt um gemeinsam über die aktuellen Entwicklungen zu sprechen und Herausforderungen gemeinsam anzugehen und zu lösen. Im Anschluss an das Jour-Fixe treffen sich die vier Projektmanager und der Projektleiter, bereiten für den nächsten Termin des Jour-Fixe die zu besprechenden Themen vor und klären gegebenenfalls organisatorische Einzelheiten um somit die restliche Gruppe der Teilnehmer zu entlasten. Während der Jour-Fixe wird von jedem der Teilnehmer abwechselnd eine Dokumentation des Treffens manifestiert um somit Ergebnisse aber auch ToDo's festzuhalten und einsehen zu können. Für das wöchentliche Treffen wird ein Tagesprotokoll genutzt, welches von dem Projektleiter durch Bildschirmübertragung an die Teilnehmer präsentiert und durchgesprochen wird. Das Tagesprotokoll kann jederzeit durch jeden Teilnehmer modifiziert und ergänzt werden, falls zu besprechende Themen durch den jeweiligen Teilnehmer aufgekommen sind. Die Struktur des Tagesprotokolls befasst sich als aller erstes mit spontanten Ergänzungen der Teilnehmenden, falls noch spontan ein Besprechungspunkt auf die Agenda mitaufgenommen werden sollte. Dann wird mit der Reihenfolge des Teilnehmenden, welcher an der Reihe ist die Dokumentation für das jeweilige Treffen mitzuschreiben, weiterverfahren. Danach werden inhaltlich zuest organisatorische Punkte durchgesprochen und schließlich die aktuell relevanten Entwicklungsbereiche der einzelnen Teams. Währenddessen macht sich der Projektleiter Notizen zu aktuellen und neuen inhaltlich relevanten Themen, um diese dann in der an das Jour-Fixe anschließenden Management-Runde zu besprechen.

Organisationswerkzeuge Für die weitere Organisation (der Entwicklung und Implementierung) der Projektgruppe werden verschiedene Tools und Hilfsmittel eingesetzt,

die im Folgenden beschrieben werden.

- **Discord:** Discord soll als Basis-Kommunikationsmittel genutzt werden. Hier werden für die entsprechenden Phasen und Themenbereiche, sowie Gruppen eigene Text- und Sprach-Kanäle erstellt. Hier werden auch aktuelle (organisatorische-) Themen aufgegriffen und besprochen. Der wöchentliche Jour-Fixe Termin ist ebenfalls in Discord abgehalten.
- **GitHub:** GitHub wird 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 wird 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 wird eine GitHub-Organisation gegründet (Link zur Organisation). Pro Gruppe bzw. Themenbereich soll zunächst ein Repository erstellt werden. 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 Ressourcenallokierung gewährleisten zu können, wird Google-Kalender genutzt, in welchem alle Teilnehmenden ihre jeweiligen Universitäts- und Urlaubsbezogenen Termine eintragen.
- **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, wird ein Status-Quo Dokument entwickelt. Dieses stellt bereits fertiggestellte, momentan laufende und potentiell in Zukunft zu behandelnde Projekte und Themengebiete übersichtlich dar. Dafür wird mit Hilfe des Tools Graphviz (Fig. 4.1) ein simpler Graph erzeugt. Dieser wird zudem am Anfang jedes Monats durch die Management-Runde aktuell gehalten. Abbildung 4.1 zeigt den Status-Quo zum Zeitpunkt der Erstellung des Abschlussberichtes.
- **Projekt-Timeline:** Zu Beginn der Projektgruppe wird eine Projekt-Timeline erstellt, welche für das jeweilige Semester die kurz- und langfristigen Aufgaben der entsprechenden Gruppen in tabellarischer Form festhalten soll. Da dieses Hilfsmittel jedoch in dem ersten Semester keine Verwendung fand, wird es zu Anfang der zweiten Hälfte der Projektgruppe als obsolet eingestuft und verworfen. Die kurz- und langfristigen Aufgaben werden stattdessen über das Status-Quo Dokument und die wöchentlichen Jour-Fixe Dokumentationen festgehalten.

4 Projektorganisation

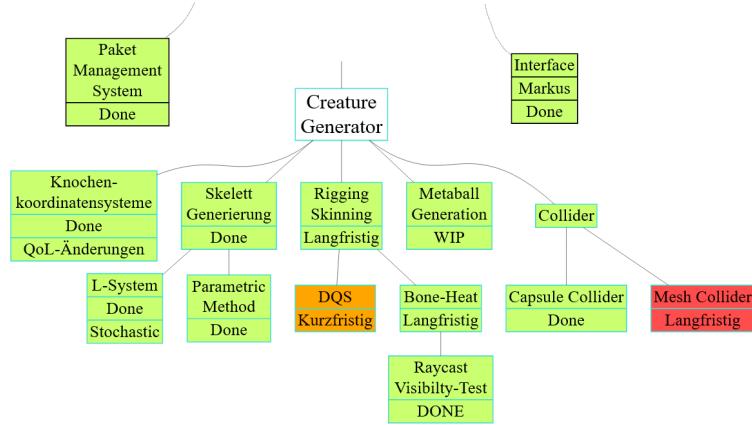


Abbildung 4.1: Stand des (fertigen) Status-Quo am 26.02.2023, Teil der Creature-Generator

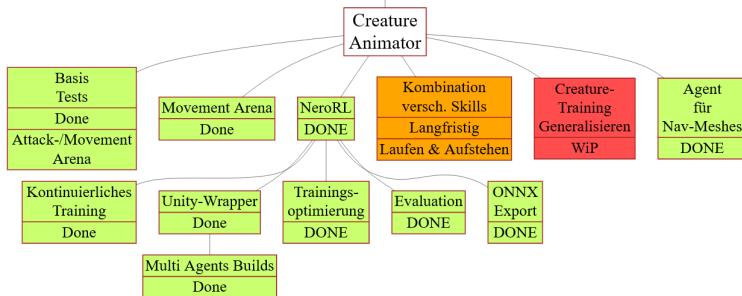


Abbildung 4.2: Stand des (fertigen) Status-Quo am 26.02.2023, Teil der Creature-Animator

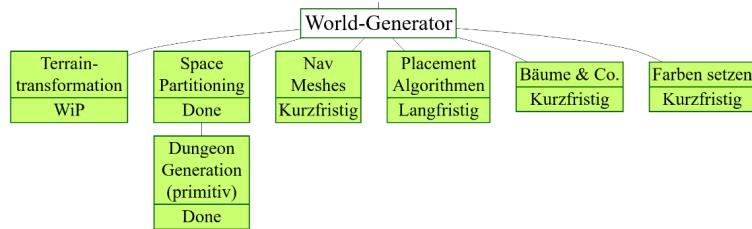


Abbildung 4.3: Stand des (fertigen) Status-Quo am 26.02.2023, Teil der World-Generator

4.1 Creature Generator

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

Skelett → Skin Bei dem ersten Ansatz wird mit einem L-System Parser experimentiert, zuerst Koordinaten zu erzeugen und das Skelett der Kreatur dann dort reinzulegen (s. Kapitel 5.1.1). Der Skin der Kreatur sollte dabei über Metaballs (Kapitel 5.1.1) und Marching Cubes [23] zusammengestellt werden.

Skin → Skelett Bei dieser Alternative soll 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 soll.

Während der Ausarbeitung trifft sich die Creature-Generator Gruppe wöchentlich am Mittwoch und hält analog zum wöchentlichen Jour-Fixe aller Teilnehmer einen Regeltermin zum Besprechen von abgeschlossenen aber auch ausstehenden Aufgaben ab. Eine Dokumentation davon wird 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 soll 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 5.1.1) wird fortwirkend von Leonard Fricke weiterverfolgt, hat jedoch zu diesem Zeitpunkt noch keine große Priorität, da das erste Ziel eine funktionierende Skelett Generierung zu erhalten ist, um sich danach dem Skin-Mesh widmen zu können. Während der weiteren Ausarbeitung wurde jedoch ein Alternativansatz von Jona Heinrichs entwickelt (Kapitel 5.1.1), um im Vergleich zum L-System noch effizientere Skelette erzeugen zu können. Damit wird die Entwicklung des L-Systems an dieser Stelle eingestellt. Somit wird entschieden, dass beide Autoren des L-Systems, Tom Voellmer und Kay Heider, von der Creature-Generation Gruppe in eine weitere Gruppe der Terrain-Generator abgezweigt werden, da sich beide Teilnehmer während der Seminarphase mit der Terrain-Generation beschäftigt haben und somit das nächste Thema angehen. 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-Generatoren und Creature-Animatoren bereiterklärt und ist somit für die Kommunikation und den Wissensaustausch beider Teams außerhalb der Jour-Fixe zuständig. Bei dieser Kommunikation beider Teams werden etliche Verbesserungen, welche von den Creature-Animatoren angeführt wurden, von den Creature-Generatoren umgesetzt. Dabei hat Markus Mügge die

4 Projektorganisation

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

Tabelle 4.2: Die zwei Untergruppen und ihre Mitglieder

finale und relevante Innovation eines Interface den Creature-Animatorn zur Verfügung gestellt, sodass Letztere nicht mehr von einzelnen Blueprints bzw. Paketen mit Kreaturen abhängig sind, welche von den Creature-Generatoren übermittelt werden müssten, sondern nun eigene Kreaturen on-demand erzeugen und ihr Training der Bewegung der Kreaturen untersuchen können.

4.2 Creature Animator

Die Gruppe der Creature Animator hat sich in zwei Untergruppen aufgeteilt. In der ersten Phase beschäftigt sich die erste Unterguppe damit, 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 können. Währenddessen versuchte die andere Unterguppe dem ML-Agents Walker das Schlagen beizubringen. Die beiden Unterguppen treffen sich wöchentlich mittwochs, um von ihren Fortschritten und Problemen zu berichten. Dabei werden die Ergebnisse in Protokollen festgehalten, welche in einem GitHub Wiki abgelegt sind.

In der zweiten Phase, welche nach der Bereitstellung der ersten generierten Kreaturen von der Creature Generator Gruppe beginnt, verändern sich die Aufgabenbereiche der beiden Unterguppen. Die „Schlagen“-Gruppe arbeitet seitdem 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 ist es den neuen Kreaturen das Fortbewegen beizubringen und der Creature Generator Gruppe Feedback zu den Kreaturen zu geben. Dabei arbeiten die Gruppenmitglieder an verschiedenen kleineren Aufgaben. Jan Beier beschäftigt sich mit dem Training und dem Finden und Ausprobieren neuer Rewardfunktionen, Nils Dunker arbeitet an der dynamischen Generierung von Arenen und dem Laden von Konfigurationseinstellungen aus Dateien und Carsten testet verschiedene Parameter und implementiert das Erstellen von NavMeshes zur Laufzeit. In der zweiten Phase lösen „On-Demand“-Treffen die regelmäßigen Treffen zwischen den beiden Unterguppen ab, um mehr Zeit zum Arbeiten an den Aufgaben zu haben. Zudem sollen anstelle der Treffen nur noch die wichtigsten Punkte protokolliert werden. Ansonsten werden Probleme und Fehler direkt als Issue in den entsprechenden GitHub Repositories hinterlegt.

5 Umsetzung

5.1 Creature Generation

Im folgenden Kapitel wird zuerst eine organisatorische Übersicht über die Themengebiete des PG649 Creature Generators gegeben, die anschließend im technischen Detail erläutert werden. Ziel des Kapitels ist es zuerst eine Verständnisgrundlage zu schaffen, damit die nachfolgenden technischen Merkmale, Design-Entscheidungen und Kompromisse nachvollziehbar sein werden.

5.1.1 Fachliche Umsetzung

Parametrische Kreatur

Als Grundlage für die parametrische Generierung der Kreaturen dient das von Jon Hudson in seiner Thesis [17] beschriebene Modell, welches in Abschnitt 5.1.2 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 Kombination aus Gleich- und Normalverteilungen.

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

5 Umsetzung

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 gerichtet, 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 aufgeteilt. Auch die Umfänge werden zufällig bestimmt, jedoch zusätzlich sortiert, sodass das dickste Segment der Körpermitte am nächsten gelegen ist. In beiden Fällen werden die Beine gerade nach unten generiert. Bei einem Zweibeiner wird am distalen Punkt des unteren Beinknochen in Richtung dessen 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

5.1 Creature Generation

vorderen Hüftknochens. Dabei ist die Rotation des ersten Knochens ein zufälliger Wert zwischen der proximalen und der negativen ventralen Achse des Elternknochens. Jedes weitere Halssegment erhält eine zufällige Rotation um $\pm 20^\circ$.

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 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.

L-System Kreatur

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 Gelenke (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.

Das Skelett einer Kreatur die durch ein L-System erzeugt wurde, ist in Abbildung 5.1 dargestellt. Das L-System, aus dem diese Skelettstruktur entstanden ist, ist im Folgenden gegeben:

- Startstring:

$$S = [|C|+(90)[AH]-(180)[AH]+(90)T+(90)[G-(90)L]-(180)[G+(90)L]$$

5 Umsetzung

- Iterationen: 2
- Produktionen:

$A \rightarrow F(0.25)F(0.25)$	Arme
$L \rightarrow F(0.35)F(0.3)^{\wedge}V$	Beine
$T \rightarrow F(0.4)F(0.4)$	Torso
$C \rightarrow F(0.3)$	Kopf
$G \rightarrow F(0.2)$	Hüfte
$V \rightarrow F(0.2)$	Füße
$H \rightarrow F(0.15)$	Hände

Die Werte in den Klammern hinter einigen Symbolen sind Argumente für die Fortbewegung und Drehung der Turtle. Bei dem Auswerten eines F bewegt sich die Turtle normalerweise um eine Einheit in die aktuelle Richtung. Der Wert in den Klammern hinter dem F überschreibt dies, sodass bspw. $F(0.2)$ die Turtle nur um 0.2 Einheiten fortbewegen lässt. Ähnlich ist dies bei den Terminalsymbolen $+$ und $-$, wo das Argument den Grad δ angibt, um wie viel sich die Turtle in der Ebene drehen soll.

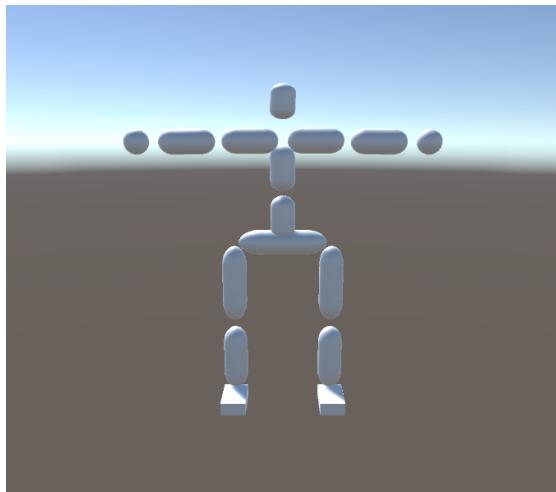


Abbildung 5.1: Mittels L-System generiertes Skelett

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 2.1.2 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 das jeweilige Tupel erzeugt hat. Dies ist notwendig, da die Produktions 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 5.1 zeigt ein über ein L-System generiertes Skelett.

Mesh-Generierung

Metaball Zur Generierung der Geometrie der Kreatur verwenden wir, angelehnt an den Ansatz von Madis Janno [19] (siehe Abschnitt 2.3), eine modifizierte Form von Metaballs. Dabei werden genau so viele Bälle entlang eines Generierten Segments platziert wie benötigt werden um zu garantieren, dass dadurch in jedem Fall ein zusammenhängendes Mesh entsteht. Tatsächlich ist jedoch der Einfluss nahegelegener Kugeln aufeinander so groß, dass dadurch nicht nur die minimale Berührung der einzelnen Kugeln entsteht, sondern diese glatt ineinander übergehen. Wie auch bei Janno 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:

$$f_i(\vec{x}) = \exp\left(B_i - \frac{B_i r_i^2(\vec{x})}{R_i^2}\right).$$

Für Metaball i ist R_i dessen Radius, B_i ein Parameter zur Einstellung der "Blobbiness", welcher beeinflusst wie sehr sich Metaballs miteinander verbinden. Wir verwenden den Wert $B_i = 0.5$. $r_i(\vec{x})$ ist der Abstand des Punktes \vec{x} zum Zentrum c_i von Metaball i , also:

$$r_i(\vec{x}) = \|\vec{x} - \vec{c}_i\| = \sqrt{(x_x - c_{ix})^2 + (x_y - c_{iy})^2 + (x_z - c_{iz})^2}.$$

5 Umsetzung

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 Körper. Dazu ersetzen wir die Bälle durch Kapseln, also Zylinder mit jeweils durch eine Halbkugel abgerundeten Enden, weshalb wir diese auch als „Metakapseln“ bezeichnen. 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 Verbindungsgeraden zwischen Start- und Endpunkt.

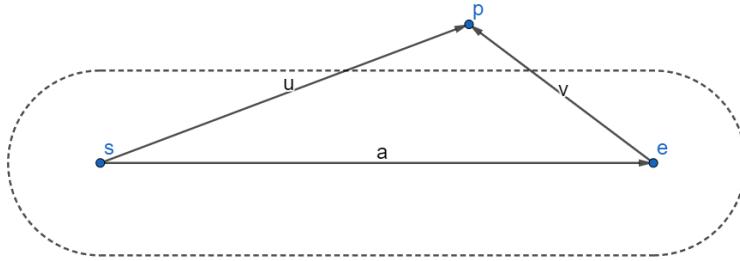


Abbildung 5.2: Beispiel Metakapsel; Die gestrichelte Linie enthält alle Punkte mit $r = R$

Um diese Form zu erzeugen muss lediglich die Funktion $r(\vec{x})$ zur Berechnung des Abstandes angepasst werden. 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 5.2). Die Distanz lässt sich dann folgendermaßen bestimmen:

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

Es werden drei Fälle unterschieden. Liegt der Punkt p im Falle von Abbildung 5.2 links von s , beziehungsweise rechts von e , ist die Distanz von p zum Segment einfach der

5.1 Creature Generation

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 entlang der Segmente platziert werden. Grundsätzlich lassen sich auf diese Weise beliebig geformte Objekte, die mit einer solchen Funktion definiert werden, einbinden. Da die Kapsel nur zylindrisch geformte Segmente darstellen lässt, die in alle Richtungen den gleichen Radius besitzen, haben wir eine weitere Modifikation vorgenommen, die es erlaubt die Metakapsel entlang der ventralen oder lateralen Achse zu strecken beziehungsweise zu stauchen. Dadurch ist es uns möglich Körperteile mit unterschiedlichen Abmessungen entlang der verschiedenen Achsen darzustellen und dennoch unnatürlich wirkende Formen wie beispielsweise einen Quader zu vermeiden. Dazu bestimmen wir bei der Berechnung von $r(\vec{x})$ zunächst den Vektor, der die kürzeste Verbindung von \vec{x} und dem Segment darstellt, anstatt direkt dessen Länge auszuwerten. Diesen Vektor skalieren wir anschließend entlang der gewünschten Achse. Die Länge des skalierten Vektors ist dann die neue Distanz.

Mesh Indexing Zur weiteren 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 des 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 des Meshes, welches durch Marching Cubes entsteht nicht numerisch stabil genug 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 [6].

Eine Delaunay-Triangulierung maximiert das Minimum von allen Winkeln in der Triangulierung [7]. Diese Triangulierung vermeidet Dreiecke die extrem lang und dünn sind. Eine Triangulierung erfüllt die Delaunay-Bedingung genau dann, wenn der Umkreis jedes

5 Umsetzung

Dreiecks leer ist. Der Umkreis eines Dreiecks ist der Kreis, der alle Ecken des Dreiecks schneidet.

Die Triangulierung, die bereits durch den Marching Cubes Algorithmus entstanden ist, kann durch die „Edge-Flipping“ Technik in eine Delaunay Triangulierung transformiert werden [7]. Dazu iteriert der Algorithmus durch alle Kanten des Meshes und prüft für die beiden anliegenden Dreiecke der Kante, ob die gegenüberliegenden Winkel α und β kleiner oder gleich 180° sind. Die Bedingung $\alpha + \beta \leq 180^\circ$ ist äquivalent zu der Delaunay-Bedingung der beiden Dreiecke. Wenn die Delaunay-Bedingung nicht erfüllt ist, können die Dreiecke durch eine „flip“ Operation korrigiert werden. Dabei wird die gemeinsame Kante BD der Dreiecke $\triangle ABD$ and $\triangle BCD$ in eine Kante AC mit Dreiecken $\triangle ABC$ and $\triangle ACD$ geändert. Für das gesamte Mesh kann so lange die ‘flip’ Operation ausgeführt werden, bis alle Dreiecke delaunay sind.

Die „edge-flip“ Operation ist auf der Mesh-Datenstruktur, welche in 3D-Engines wie Unity verwendet wird, relativ aufwendig, da diese keine Möglichkeit bietet, die zugehörigen Dreiecke einer Kante zu finden. In diesem Projekt wurde daher die Polygon Mesh Processing Bibliothek (PMP) [37] verwendet. Diese enthält bereits viele Methoden um 3D-Meshes auf komplexe Weise zu modifizieren. Insbesondere wird eine optimierte Datenstruktur für das Mesh genutzt, welche es erlaubt schnell über alle Kanten zu iterieren und diese ohne großen Aufwand zu flippen.

Automatisches Rigging

Um die Knochen-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 Knochen in der Szene bewegt. Jeder Vertex kann an beliebig vielen Knochen angehängt werden. Durch eine Gewichtung wird bestimmt wie sehr ein Vertex durch die Transformation eines Knochens 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 Knochen plaziert sind, können wir hier davon ausgehen, dass das Mesh das Skelett bereits „sinnvoll“ umhüllt.

Während der Entwicklung wurde zunächst eine triviale Methode als Zwischenlösung verwendet. Dabei wurden alle Vertices nur an den Knochen mit der geringsten Distanz angehängt. Diese Methode ist jedoch visuell nicht brauchbar, da bei Bewegung des Skelettes an den Gelenken zwischen den Knochen Löcher und verschiedene andere Artefakte

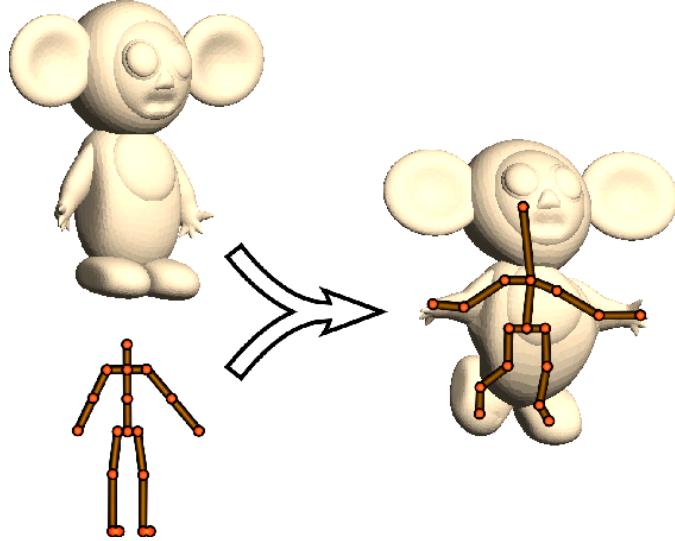


Abbildung 5.3: Beispiel für ein korrekt eingebettetes Skelett in einem Mesh [3].

entstehen. Es wird also eine Lösung benötigt die es ermöglicht Vertices an mehrere Knochen anzuhängen und die Gewichte so bestimmt, dass das Mesh an den Übergängen zwischen den Knochen möglichst natürlich deformiert wird.

Bone-Heat Methode Ein bereits bekannter Algorithmus zur automatischen Gewichtsberechnung und Verknüpfung des Meshes ist die Bone-Heat Methode [3]. Dieser berücksichtigt mehrere zusätzliche 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 Knochen sollte ungefähr proportional zu der Distanz des Gelenks zur Oberfläche des Meshes sein. Ein Algorithmus, welcher die Gewichte alleine aus der Distanz der Knochen 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 Knochen verwendet. Wie in Abbildung 5.4 zu sehen ist, wird ein Knochen auf 1° gesetzt und der andere auf 0° . Bei Deformation der beiden Knochen 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 [3, S. 6]. Die Gewichte für Knochen i werden berechnet durch:

5 Umsetzung

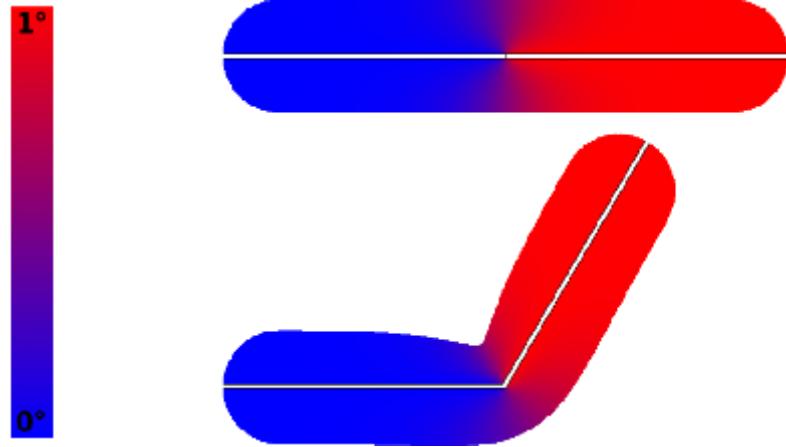


Abbildung 5.4: Temperatur-Gleichgewicht für zwei Knochen [3].

$$\begin{aligned}
 \frac{\partial \mathbf{w}^i}{\partial t} &= \Delta \mathbf{w}^i + \mathbf{H} (\mathbf{p}^i - \mathbf{w}^i) = 0 \\
 \iff -\Delta \mathbf{w}^i + \mathbf{H} \mathbf{w}^i &= \mathbf{H} \mathbf{p}^i \\
 \iff (-\Delta + \mathbf{H}) \mathbf{w}^i &= \mathbf{H} \mathbf{p}^i.
 \end{aligned} \tag{5.1}$$

Dabei ist Δ der diskrete Laplace-Beltrami Operator auf der Oberfläche des Meshes, welcher mit der Kotangens-Methode approximiert wird [6]. \mathbf{p}^i ist ein Vektor mit $p_j^i = 1$ wenn der nächste Knochen zum Vertex j der Knochen i ist. Sonst ist $p_j^i = 0$. \mathbf{H} ist eine Diagonalmatrix in der H_{jj} die Hitze des nächstgelegenen Knochen von Vertex j ist. Sei $d(j)$ die Distanz zum nächstgelegenen Knochen von Vertex j , dann wird $H_{jj} = c/d(j)^2$ gesetzt. Allerdings nur wenn das Geradensegment von dem Vertex zu dem Knochen vollständig in dem Volumen des Modells enthalten ist. Wenn der Knochen von dem Vertex aus also nicht sichtbar ist, wird $H_{jj} = 0$ gesetzt. Dies verhindert, dass beispielsweise Vertices des Arms an den Torso angehängt werden. Wenn mehrere Knochen 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 für die Vertices wurde in diesem Projekt durch ein Signed-Distance-Field (SDF) realisiert, welches wir vorab mit einem Geometry-Shader generieren [27]. Damit werden effiziente Raycast-Operationen in dem Mesh möglich. Der Parameter c wird in dem Paper [3] 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 5.1 ist es wichtig die Eigenschaften des diskreten Laplace-Beltrami Operators

5.1 Creature Generation

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 5.5).

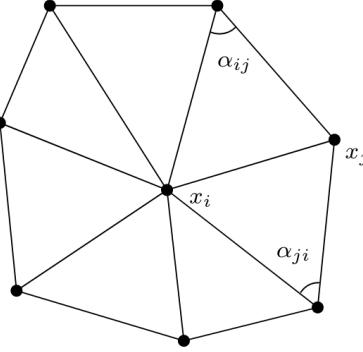


Abbildung 5.5: Berechnung der α -Winkel für eine innere Kante [6].

$$v(x_i, x_j) = \begin{cases} \cot \alpha_{ij} + \cot \alpha_{ji} & \text{für innere Kanten} \\ \cot \alpha_{ij} & \text{für äußere Kanten} \end{cases} \quad (5.2)$$

Aus den Kotangens-Gewichten aus Gleichung 5.2 kann nun in Formel 5.3 der Laplace-Beltrami Operator in seiner Matrix-Form berechnet werden [8, 6]. Dabei beschreibt $N(x_i)$ die Menge der benachbarten Vertices von x_i . Da für das Lösen der Gleichung 5.1 lediglich die Matrix $(-\Delta + \mathbf{H})$ benötigt wird, kann diese in der Implementierung direkt als eine Matrix erstellt werden. Die Normalisierungsfaktoren aus [8] entfallen für diese Anwendung.

$$\Delta_{ij} = \begin{cases} 0 & i \neq j, x_j \notin N(x_i) \\ v(x_i, x_j) & i \neq j, x_j \in N(x_i) \\ -\sum_{x_j \in N(v_i)} v(x_i, x_j) & i = j \end{cases} \quad (5.3)$$

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 pro Reihe in der Matrix [8]. Ein solches System lässt sich, wie in [8] gezeigt, sehr effizient lösen, wenn es sich um eine SPD-Matrix (symmetrisch positiv definit) handelt. Die Matrix ist durch die Konstruktion in Gleichung 5.3 symmetrisch. Für die Lösung dieses Systems wurde in diesem Projekt die Bibliothek Eigen [13] verwendet, welche zahlreiche Methoden zur Matrix-Berechnung bereitstellt. Es wurde der Matrix-Solver **SparseLU** verwendet, welcher auf dünn besetzte SPD-Matrizen spezialisiert ist.

5 Umsetzung

Durch die Lösung des Systems für jeden Knochen i erhalten wir einen Vektor \mathbf{w}^i aus Gleichung 5.1. Wir hängen jeden Vertex mit dem entsprechenden Gewicht an diesen Knochen, falls das Gewicht größer als 0 ist. Für die Verwendung der Gewichte in Unity mussten diese außerdem pro Vertex absteigend sortiert und normalisiert werden, sodass sie sich zu 1 aufsummieren.

Skinning

Nachdem das Mesh durch den Prozess des Riggings mit den Vertex-Gewichten an das Skelett angehängt wurde, muss noch definiert werden, wie das Mesh die Transformationen des Skeletts umsetzt. Für diesen Prozess des „Skinnings“ existieren im allgemeinen zwei unterschiedliche Ansätze.

Die einfache Technik ist das Linear Blend Skinning (LBS). Hier werden die Vertices wie in Formel 5.4 transformiert. Der Vertex \mathbf{v} wird zunächst mit allen Knochen-Transformationen \mathbf{C}^j transformiert und dann wird der gewichtete Durchschnitt mit den entsprechenden Vertex-Gewichten w^j verwendet um \mathbf{v}' zu bestimmen.

$$\mathbf{v}' = \sum_{j=0}^n w^j \mathbf{C}^j \mathbf{v} \quad (5.4)$$

LBS hat allerdings den Nachteil, dass einige Transformationen an den Gelenken unnatürlich „aufrollen“. Dieser sogenannte „Candy-Wrapper“ Effekt ist in Abbildung 5.6 (links) zu erkennen.

Um dieses Problem zu beheben kann alternativ die Dual Quaternion Technik verwendet werden. Hier wird anstatt einer Linear-Kombination über die transformierten Vertices eine Linear-Kombination über duale Quaternionen berechnet [21]. Diese erlauben es bei der Transformation der Vertices die Skalierung des Mesh-Volumens beizubehalten, da duale Quaternionen sowohl Rotation, als auch Verschiebung gleichzeitig kodieren. Die Implementierung des DQ-Skinnings ist mit kleinen Anpassungen aus dem Projekt [31] entstanden. In Abbildung 5.6 kann man auf der rechten Seite erkennen, das DQS die Probleme von LBS erfolgreich verhindert.

5.1.2 Technische Umsetzung

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

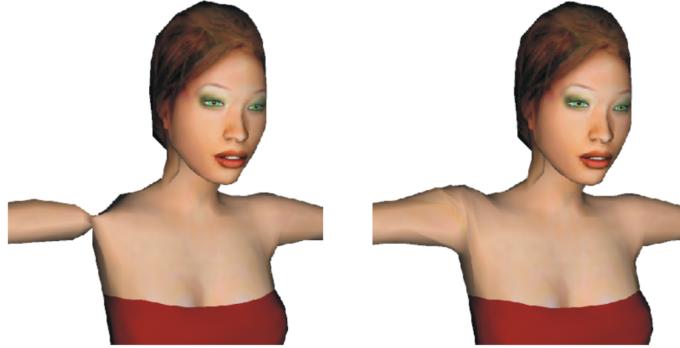


Abbildung 5.6: Vergleich von Linear Blend Skinning (links) und Dual Quaternion Skinning (rechts) [21].

Packages/com.pg649.creaturegenerator/Runtime. Der **Assets**-Ordner enthält lediglich Debug-Skripte, die nicht exportiert werden sollen.

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 Kreatur, 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 ggf. 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.

5 Umsetzung

Bone Definition

Eine der ersten Datenstrukturen, die von dem Generator erzeugt werden, ist ein Baum von `BoneDefinitions`. 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, parametrischen Generatoren ist nicht durchsetzt mit Konventionen und Annahmen über Koordinatensysteme; Stattdessen 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.

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 `ParametricCreatureSettings` zufällig tatsächliche Längen, Dicken, and Anzahlen in Form einer `BipedSettingsInstance` 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 `BoneDefinitions`, wie in Kapitel 5.1.1 beschrieben.

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`.

Skeleton Assembler

Der `SkeletonAssembler` baut aus der `SkeletonDefinition` einen Baum aus Unity `GameObjects`, 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 `GameObjects` 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 Knochens ab.
- ein `Bone`, der Metadaten, wie z.B. Länge oder Kategorie des Knochens, enthält, aber auch die Farbe des Knochens für die spätere Darstellung im Spiel

Die Farbe des Knochens wird zufällig im HSV Farbraum generiert und dann in das in der Computergrafik übliche RGB Format konvertiert. Der HSV Farbraum erleichtert es zufällig Farben in bestimmten Farbtönen und Helligkeiten zu generieren, in unserem Fall zombieartige Grüntöne.

Die Masseberechnung für den `Rigidbody` verdient ebenfalls besonderes Augenmerk. Der `SkeletonAssembler` berechnet die Masse des `Rigidbodies` grundsätzlich aus dem Volumen des an dem ihm angebrachten `Colliders` und der Dichte des Knochens, die der `SkeletonAssembler` einem `DensityTable` entnimmt. In der Praxis führen zu große Unterschiede in der Masse von zwei durch einen `ConfigurableJoint` verbundenen `Rigidbodies` zu numerischer Instabilität in der Physiksimulation, weshalb der `SkeletonAssembler` optional in jedem `Rigidbody` die selbe fixe Masse setzen kann.

5 Umsetzung

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 für fast alle Knochen 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. Lediglich für Knochen die im **BoneTree** direkt unterhalb eines Hüft- oder Bein-Knochens liegen wird eine lineare Bewegung entlang der der z-Achse (proximalen Achse) erlaubt und eine entsprechende Feder konfiguriert, um die zuvor beschriebenen Stoßdämpfer zu modellieren.

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.

Der **projectionMode** des Joints wird auf **PositionAndRotation** 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.

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 Scawk [35].

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.

5.2 Creature Animation

5.2.1 Trainingsumgebung

Im Folgenden soll der Aufbau der Trainingsumgebung beschrieben werden, welche es erlaubt verschiedenste Kreaturen ohne große Anpassungen zu trainieren. Im Laufe der Projektgruppe wurden drei unterschiedliche Versionen der Trainingsumgebung genutzt.

Vorherige Versionen

Die erste Version ist eine leicht abgeänderte Form der ML-Agents-Beispielumgebung für den Walker. Dabei handelt es sich um eine Unity-Szene mit einer fest vorgebenden Anzahl an Arenen und Kreaturen, welche trainiert werden können.

Da sich hier schnell Limitationen in Bezug auf die Anpassbarkeit herausstellten wurde die Umgebung geändert, so dass die Anzahl der Arenen dynamisch generiert werden und Kreaturen als Prefab und per Seed über den `CreatureGenerator` platziert werden können.

Die `DynamicEnviormentGenerator`-Umgebung ist dabei aus den folgenden Klassen aufgebaut:

- `DynamicEnviormentGenerator`
 - `TerrainGenerator`
 - Verschiedenen Konfigurationsdateien
 - `DebugScript`
- großteils alle modifizierten ML-Agents Skripten

Da die Umgebung weiterhin als zusätzliche Szene in dem Projekt existiert¹ und die aktuelle Version der Trainingsumgebung auf dem `DynamicEnviormentGenerator` aufbaut, wird im Folgenden auf den Aufbau des `DynamicEnviormentGenerator` sowie dessen Hilfsklassen 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.

¹Es sei dabei angemerkt, dass die Szene nicht aktiv getestet wird und deswegen nicht garantiert werden kann, dass mit weiteren Änderungen die Funktionalität bestehen bleibt.

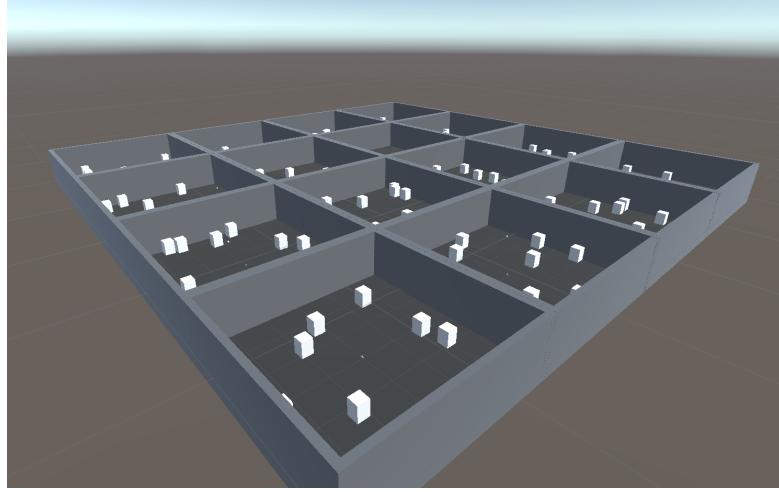


Abbildung 5.7: Ein Beispiel der generierten Trainingsumgebung mit mehreren Arenen.

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, abgebildet in der Grafik 5.7, 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:

$$\begin{pmatrix} \lceil \frac{\text{Anzahl der Arenen}}{\sqrt{\text{Anzahl der Arenen}}} \rceil \\ 0 \\ \text{Anzahl der Arenen} \mod \sqrt{\text{Anzahl der Arenen}} \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \quad (5.5)$$

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

In Schritt 2. der Arenagenerierung muss beachtet werden, dass nach dem Erstellen des Zielobjekts das `WalkTargetScript` hinzugefügt wird. Am Ende des Erstellungsprozesses

5 Umsetzung

wird der Walker erstellt. Hierzu wird ein von den Creature-Generator-Team bereitgestelltes Paket² benutzt. Das Paket stellt eine Klasse bereit, welche mit zwei Skript-Objekte konfiguriert wird. Zusätzlich wird ein seed übergeben, welcher reproduzierbare Kreaturen erlaubt. Die erstellte Kreatur muss danach mit den entsprechenden ML-Agent-Skripten versehen werden. Hierzu wird ein `WalkerAgent` Objekt als String übergeben. Dies ermöglicht es, mehrere unterschiedliche Agent-Skripte durch eine Änderung im Editor zu setzen. Somit können Reward-Funktion und Observation für zwei unterschiedliche Trainingsversuche getrennt, in eigenen Dateien, entwickelt werden.

TerrainGenerator Da ein typisches 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 `DynamicEnvironmentGenerator` 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.

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`³ hinzu. Die Konfigurationsmöglichkeiten sind in der Grafik 5.8 zu sehen.

Aktuelle Version – Advanced Environment Generator

Über den Produktzyklus des `DynamicEnvironmentGenerator` sind insbesondere gegen Ende der Projektgruppe mehrere Probleme aufgefallen. Da durch häufige Updates die Codequalität nachlässt und einige zuvor bedachte Optionen durch Zeitmangel nicht mehr relevant waren wurde entschieden, die Umgebung durch eine neue zu ersetzen.

²<https://github.com/PG649-3D-RPG/Creature-Generation>

³<https://docs.unity3d.com/ScriptReference/JsonUtility.html>

5.2 Creature Animation

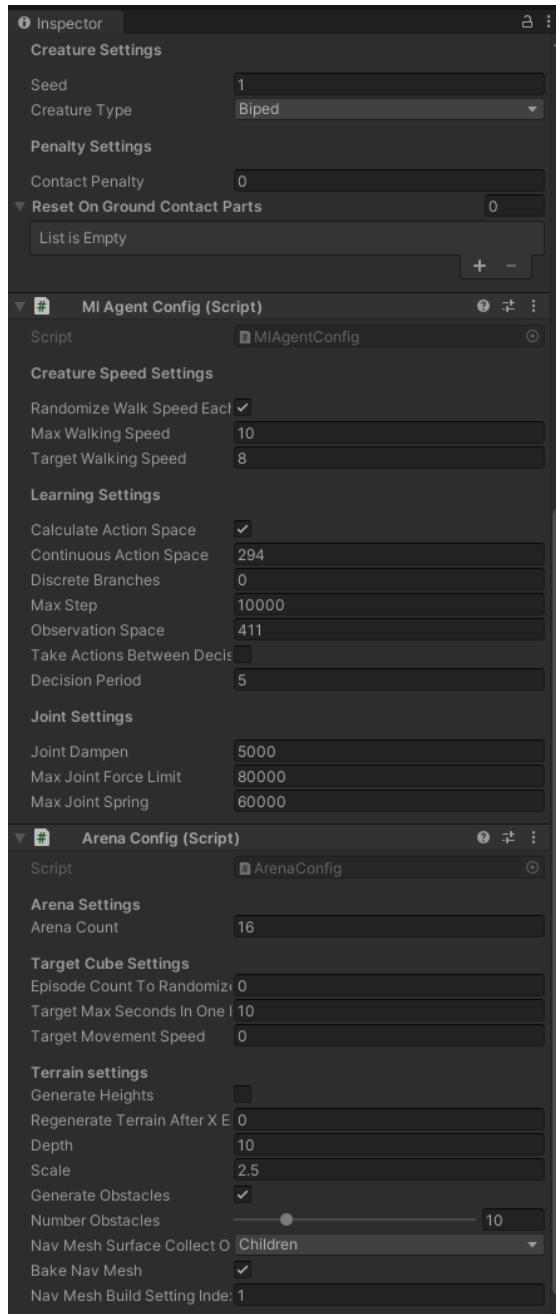


Abbildung 5.8: Konfigurationsmöglichkeiten des `DynamicEnviormentGenerator` im Unity-Editor.

5 Umsetzung

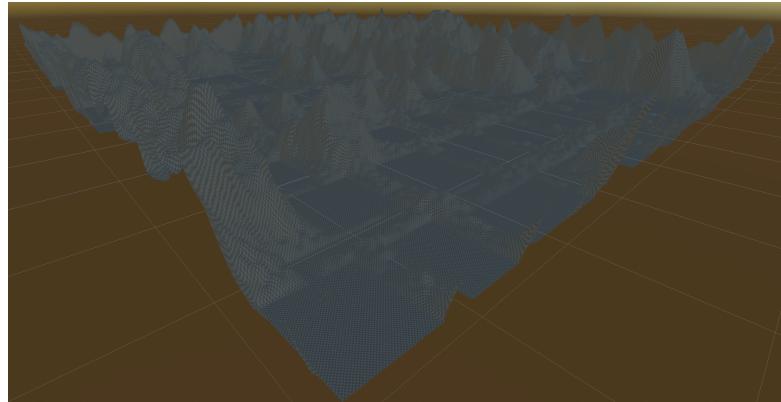


Abbildung 5.9: Ansicht der generierten Arena durch den AdvancedEnvironmentGenerator.

Dabei wurde insbesondere auf eine starke Nähe zum Spiel geachtet, damit nur wenige Modifikationen für dieses erstellt werden müssen.

Die neue Umgebung wird im folgenden als `AdvancedEnvironmentGenerator` bezeichnet und befindet sich in einer neuen Szene mit den Namen „`AdvancedMovementScene`“. Bedingt durch die konzeptionelle Weiterentwicklung der vorherigen Versionen mussten in den restlichen Skripten kaum Anpassungen durchgeführt werden. Eine Hauptänderung ist die geringere Anzahl an Verweisen auf den neuen Generator. Da der `DynamicEnvironmentGenerator` in seinen ersten Versionen eine Art Gottklasse war, verwiesen die meisten anderen Skripte darauf. Dies führte schnell zu unübersichtlichen Abhängigkeiten.

Die Generator-Klasse selbst konnte deutlich gekürzt werden, da eine Generierung von Arenen nicht mehr benötigt wird. Zu dem Zeitpunkt der Erstellung existierte bereits der `TerrainGenerator` für das finale Spiel, welche ab dieser Version für die Trainingsumgebung verwendet wird. Weiterhin ermöglichte dies eine Entfernung von vielen Einstellungen für das Terrain aus dem Generator-Objekt.

In der Abbildung 5.9 ist das Endergebnis des `AdvancedEnvironmentGenerator` dargestellt. Da es im realen Spiel zu Kollisionen zwischen verschiedenen Kreaturen kommen kann, wurde für die Trainingsumgebung nur eine zusammenhängende Arena verwendet. Insgesamt stimmen die Bedingungen für das Lernen und dem Spiel somit überein, da wie im Spiel kleine Korridore und Arenen mit ggf. mehreren Kreaturen genutzt werden.

5.2.2 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

5.2 Creature Animation

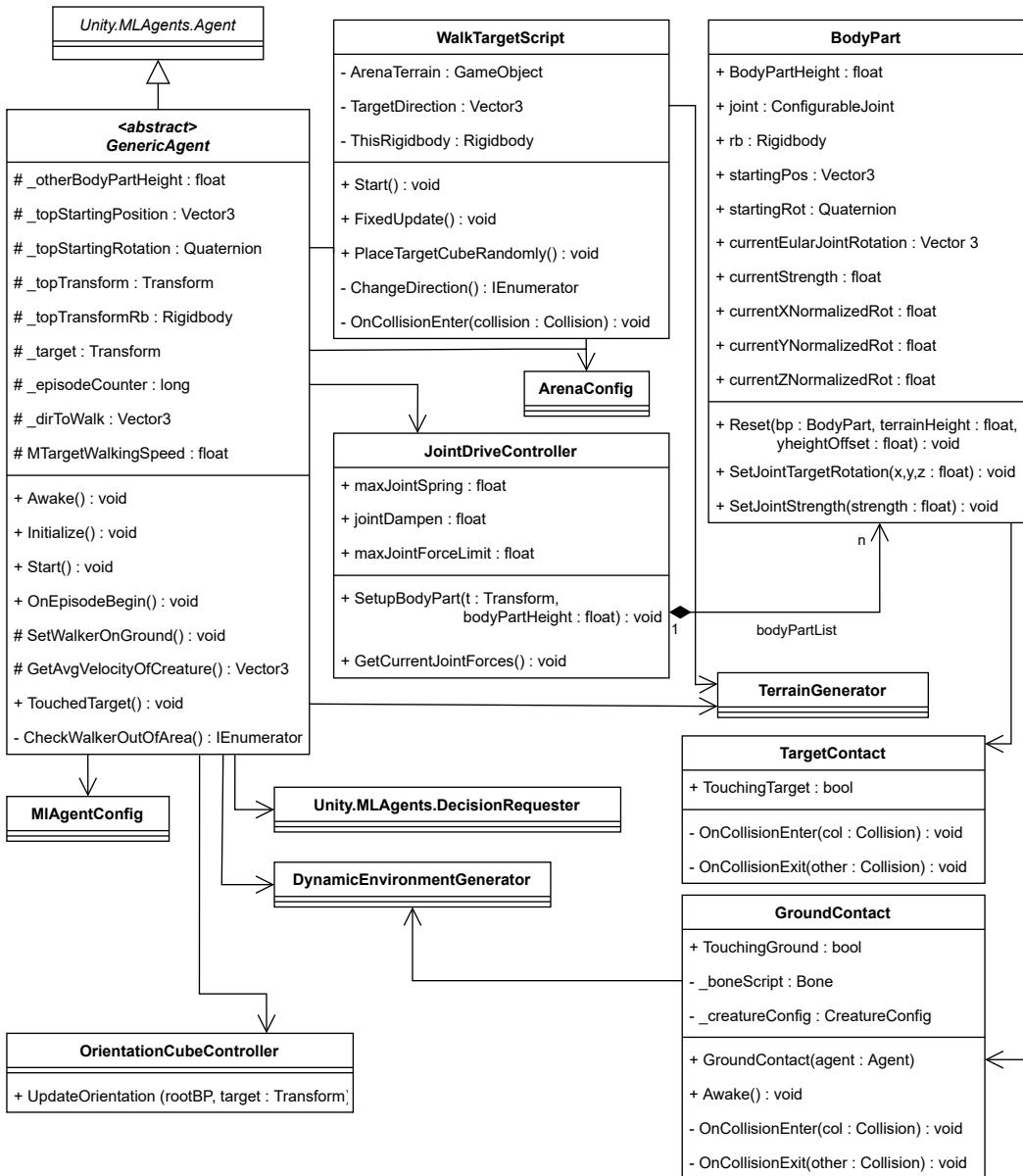


Abbildung 5.10: UML-Diagramm der GenericAgent-Klasse und weitere relevante Klassen

5 Umsetzung

und die Funktionalität dieser Klassen erklärt. Zur Veranschaulichung befindet sich in Abbildung 5.10 ein UML-Diagramm. Der Aufbau dieser Klassen orientiert sich dabei sehr stark an der Implementierung des ML-Agents Walker.

GenericAgent

Die kontrollierende Instanz einer konkreten Trainingsumgebung ist die **GenericAgent**-Klasse. Diese ist dazu in der Lage, mit dem Modell des ML-Frameworks zu interagieren, also sowohl Beobachtungen der Trainingsumgebung weiterzugeben 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 durchführt. Schließlich muss die **GenericAgent**-Klasse noch die Rewards für die Trainingsumgebung verteilen. Zu diesem Zweck ist die Klasse als *abstract* definiert, da diese Rewardfunktionen stark von der Aufgabe des Agents abhängig sind. So benötigt zum Beispiel ein Agent, welcher ein bestimmtes Ziel möglichst schnell erreichen soll eine andere Reward-Funktion als ein Agent, welcher sich möglichst gut vor dem Spieler verstecken soll. Verschiedene Agents können so ohne Redundanz einfach als eine Erweiterung der **GenericAgent**-Klasse implementiert werden. In Abschnitt 5.2.3 findet sich eine Auswahl von konkreten Erweiterungen dieser abstrakten Klasse, welche wir im Laufe des Projekts verwendet haben.

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 wrapt 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 Initialisierung stellt der **JointDriveController** nur noch das Verbindungsstück zwischen der **GenericAgent**-Klasse und der verschiedenen **BodyPart**-Instanzen dar.

OrientationCubeController

Da sich das Target des Agenten potentiell überall innerhalb einer großen (und weitgehend unbekannten) Ingame-Umgebung befinden kann, ist es hilfreich, die gezielte Laufrichtung des Agenten an eine einheitliche Position zu platzieren. Hierfür besitzt jeder

Agent einen sogenannten *OrientationCube*, welcher an einer festen Position relativ zum Agenten steht und sich lediglich in die Richtung des Targets dreht. So kann der Agent (und infolgedessen das ML-Modell) einfach den OrientationCube referenzieren, um die Laufrichtung zu bestimmen. Der `OrientationCubeController` stellt dafür die Reorientierungsfunktion des OrientationCubes bereit.

WalkTargetScript

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

5.2.3 Konkrete Implementationen der Agents

Um eine bessere Vorstellung von den tatsächlich benutzten Agents zu bekommen, wird in diesem Abschnitt eine Auswahl solcher Agents vorgestellt.

AgentWalker

Der `AgentWalker` stellt den Startpunkt unserer Entwicklung eigener Agents dar. Die Aufgabe dieses Agents ist es, in die Richtung eines `Targets` zu laufen, am besten aufrecht auf zwei Beinen. Zu diesem Zweck besteht die Reward-Funktion grundsätzlich aus zwei Teilen: Einerseits sollte der Agent (beispielhaft repräsentiert durch sein Torso-Element) möglichst gut in Richtung des Targets orientiert sein. Andererseits sollte der Agent natürlich auch vorwärts laufen. Wenn der Agent beide diese Aufgaben zusammen bewältigen kann, bewegt der Agent sich auf einer geraden Linie in Richtung des Targets. Entsprechen wird eine Belohnung bestehend aus dem Produkt dieser beiden Teilbelohnungen vergeben.

Basierend auf dieser Reward-Funktion lassen sich kleine Anpassungen vornehmen, um Overfitting zu vermeiden oder bestimmtes Verhalten zu fördern. Zum Beispiel kann die Reward-Funktion eine bestimmte Laufgeschwindigkeit fordern, an die sich der Agent möglichst gut anpassen soll. Wenn nun im Laufe des Trainings diese gewünschte Laufgeschwindigkeit geändert wird, kann so gegen Overfitting vorgegangen werden. Auch können weitere Komponenten in das Produkt der Reward-Funktion aufgenommen werden, wie zum Beispiel eine Teilbelohnung, die der Kopfhöhe entspricht, um so einen aufrechten Gang anzuregen.

5 Umsetzung

Walking	$(1 - (\frac{\text{currentSpeed}}{\text{desiredSpeed}})^2)^2 \cdot (\frac{1}{2}(\text{currentDir} \cdot \text{desiredDir} + 1))$
Standup	$r_{hhhr} + r_{hhhp} + r_{thr} + r_{trr}$ $r_{hhhr} = \begin{cases} \text{clamp}(hh_n/hh_s, 0, 1)^2 & \text{if } hh_n > hh_{max} \vee hh_s - hh_n < 0.25 \\ 0 & \text{else} \end{cases}$ $r_{hhp} = \begin{cases} -0.1 & \text{if } hh_n < hh_{n-1} \wedge hh_s - hh_n > 0.25 \\ 0 & \text{else} \end{cases}$ $r_{thr} = \begin{cases} \text{clamp}(0.2^{\delta_{th}}, 0, 1) & \text{if } \delta_{th} < \delta_{thmin} \vee \delta_{th} < 0.125 \\ 0 & \text{else} \end{cases}$ $\delta_{th} = th_s - th_n $ $r_{trr} = \begin{cases} \text{clamp}(0.2^{\delta_{tr}}, 0, 1) & \text{if } \delta_{tr} < \delta_{trmin} \vee \delta_{tr} < 0.4 \\ 0 & \text{else} \end{cases}$ $\delta_{tr} = \begin{pmatrix} (tr_s - tr_n).x \% 360 \\ 0 \\ (tr_s - tr_n).z \% 360 \end{pmatrix}$

Tabelle 5.1: Varianten des **AgentNavMesh** mit Reward-Funktionen

Ein offensichtliche Limitation dieses Ansatzes ist es, dass der Agent sich nur in einer geraden Linie zum Target bewegen kann. Sollten sich Hindernisse auf diesem Weg befinden, kann der Agent damit erst einmal nicht umgehen, da ihm Informationen über solche Aspekte der Umgebung fehlen.

AgentNavMesh mit Varianten

Anstatt dem Agent mehr Informationen über die Umgebungen zu geben, ist es auch möglich, eine Reihe von Targets für den Agenten zu erstellen, sodass es immer einen geraden Weg ohne Hindernisse von einem dieser Targets (bzw. dem Agent) zum nächsten gibt. Das letzte Target dieser Reihe soll dann das ursprüngliche Target sein. Eine solche Reihe von Targets lässt sich gut mit „NavMeshes“ berechnen. Alle weiteren Eigenschaften übernimmt der **AgentNavMesh** vom **AgentWalker**.

Ein großer Vorteil dieses Vorgehens ist es, dass die gewünschte Aufgabe in kleinere Teilprobleme aufgeteilt wird, die individuell betrachtet deutlich einfacher zu lösen sind. Obwohl unsere Umgebungen viele Hindernisse und Kurven an zufälligen Positionen beinhalten kann, bestehen die Teilprobleme immer noch aus dem verhältnismäßig simplen Problem, geradeaus in eine Richtung zu laufen.

Diese Implementation des Agents ist die Basis aller im Endprodukt enthaltenen Agents. Einige der benutzten Reward-Funktionen werden in Tabelle 5.1 zusammengefasst.

Die Standup Reward-Funktion wird verwendet, um das Aufstehen aus einer liegenden Position zu trainieren. Die Funktion besteht aus vier Komponenten, die unter bestimmt-

ten Bedingungen einen Wert annehmen. r_{hh_r} bezeichnet den Reward für die Kopfhöhe (HeadHeight). Dieser ergibt sich aus der Differenz der aktuellen Kopfhöhe hh_n und der Kopfhöhe im stehenden Ausgangszustand hh_s . Der Reward wird nur vergeben, wenn die aktuelle Kopfhöhe größer ist als die maximale Kopfhöhe der aktuellen Episode hh_{max} , oder die aktuelle Kopfhöhe nah an der stehenden Kopfhöhe ist. Dadurch wird sichergestellt, dass durch wiederholtes Heben und Senken des Kopfes im nicht-stehenden Zustand kein hoher Reward verteilt wird. r_{hh_p} bezeichnet eine Bestrafung (negativer Reward) von -0.1 für die Kopfhöhe. Diese wird verteilt, wenn die aktuelle Kopfhöhe hh_n die Kopfhöhe des vorherigen Zeitschritts hh_{n-1} unterschreitet und die aktuelle Kopfhöhe nicht nah an der stehenden Kopfhöhe ist. Dies motiviert den Agenten, den Kopf nicht zu senken und stabil zu stehen. r_{thr} bezeichnet den Reward für die Höhe des obersten Knochens in der Hierarchie des Skeletts des Agents (TopTransform, Rücken). Dieser ergibt sich aus der Skalierung der Abweichung δ_{th} der aktuellen Höhe des Knochens th_n von dessen Startzustand th_s . Der Reward wird nur verteilt, wenn die aktuelle Differenz kleiner ist, als die minimale Differenz in der aktuellen Episode δ_{thmin} , oder die aktuelle Differenz sehr gering ist, der Knochen also nah der Ausgangshöhe ist. r_{trr} bezeichnet den Reward für die Rotation des obersten Knochens in der Hierarchie des Skeletts des Agents. Dieser wird analog zum Reward für die Höhe des Knochens r_{thr} berechnet. Die aktuelle Abweichung der Rotation δ_{tr} betrachtet die Rotation des Knochens entlang der x und z Achsen, die y Achse wird ignoriert, da diese die Blickrichtung kodiert. Die Abweichung entlang der x und z Achse zwischen aktueller Rotation tr_n und Startrotation tr_s wird jeweils auf den Bereich von 0 bis 360 Grad skaliert. Der Reward wird nur verteilt, wenn die aktuelle Rotation geringer ist, als die minimale Rotation der aktuellen Episode δ_{trmin} , oder die Rotation nah an der Ausgangsrotation ist. Durch die beiden Reward für den Rückenknochen des Skeletts wird der Agent dazu motiviert, seine aufrechte Ausgangssituation wiederherzustellen. Indem die Rewards jeweils nur dann verteilt werden, wenn eine Verbesserung im Vergleich zum Minimum der aktuellen Episode entsteht, oder der Ausgangszustand annähernd erreicht ist, wird verhindert, dass für wiederholtes Heben und Senken des Rückenknochens ein großer Reward verteilt wird. Dadurch wird ein stabiles Halten der Ausgangsposition ermutigt, sobald diese wieder erreicht wurde.

5.2.4 RL-Framework

Zum Trainieren der Kreaturen wird ein RL-Framework verwendet, welches mit den für die Kreaturen definierten Agenten interagiert. Das Projekt in Unity gebaut wird und somit stehen verschiedene Frameworks zur Verfügung, die sich in implementierten Algorithmen und Trainingsoptionen unterscheiden. Da sich zu Beginn des Projekts bereits auf die Verwendung des Proximal Policy Optimization (PPO) Algorithmus geeinigt wurde, sind zwei Frameworks in die engere Auswahl gekommen.

ML-Agents Das Unity Machine Learning Agents Toolkit [20], kurz ML-Agents, ist ein von Unity entwickeltes Projekt zum Trainieren von Agenten und stellt, wie in diesem

5 Umsetzung

Projekt benötigt, unter anderem eine Implementierung von PPO zur Verfügung. Zusätzlich definiert das Toolkit auch eine Python API, die verwendet werden kann um eigene Agenten mit den zur Verfügung gestellten Algorithmen trainieren zu lassen. Dabei unterstützt die API neben Diskreten und Kontinuierlichen Aktions- und Beobachtungsräumen auch das Platzieren mehreren Agenten in einer Unity Szene.

Während der Trainings angelegte Checkpoints und der finale Zustand des Netzwerks werden als onnx Dateien gespeichert, die dann geladen werden können, um das trainierte Netzwerk zu verwenden. Zum Analysieren und Bewerten des Trainings erhebt ML-Agents verschiedene Daten, wie den durchschnittlichen Reward und die Loss Werte der verschiedenen Netzwerke und speichert diese in einem Tensorboard.

neroRL Bei neroRL [29] handelt es sich um ein Reinforcement Learning Framework, das seinen Fokus auf verschiedene Varianten des Proximal Policy Optimization (PPO) Algorithmuses legt. Der Agent kann dabei entweder mit geteilten oder getrennten Netzwerken und Gradienten für den Aktor und den Kritiker trainiert werden. Dabei unterstützt es zum Trainieren neben openAIGym Umgebungen auch solche, die mit der Python API von ML-Agents erstellt wurden. Die Beobachtungsräume in diesem Umgebungen dürfen jedoch nur Vektor und Visuelle Beobachtungen enthalten und für die Aktionräume werden nur Diskrete und Multidiskrete Formate unterstützt. Des Weiteren kann in jeder Szene nur ein Agent platziert werden und zur Beschleunigung des Trainings müssen stattdessen mehrere Instanzen der Szene gleichzeitig trainiert werden.

Während der Trainings angelegte Checkpoints und der finale Zustand des Netzwerks werden als pt Dateien abgespeichert und können als solche nicht direkt in eine Unity Umgebung geladen werden. Zum Analysieren und Bewerten erhebt neroRL während des Trainings verschiedene Kenndaten, wie den durchschnittlichen Reward und die Loss Werte der verschiedenen Netzwerke, und speichert diese in einem Tensorboard.

Auswahl des Frameworks

Beide vorgestellten Frameworks verfügen im Hinblick auf dieses Projekt über Vor- und Nachteile.

ML-Agents unterstützt bereits von sich aus kontinuierliche Aktionsräume, welche zum Bewegen der Kreaturen benötigt werden. Außerdem erlaubt es mehrere Agenten in einer Szene zu platzieren, im folgenden als Multi-Agent-Szene bezeichnet, und so das Training zu beschleunigen. Die vorherigen Tests mit der Walker Testumgebung in Unity [1] haben gezeigt, dass selbst eine zum Laufen optimierte zweibeinige Kreatur recht lange braucht, bis sie stabil und ohne umfallen laufen kann. Bei einer prozedural generierten Kreatur ist zu erwarten, dass der Prozess länger dauert und darum sollten alle Möglichkeiten das Training zu beschleunigen verwendet werden. Ein Problem mit ML-Agents liegt in der Auswertung der Ergebnisse des Trainings. Im Rahmen dieser Ausarbeitung ist es

5.2 Creature Animation

notwendig sich kritisch mit den Ergebnissen auseinander zu setzen und für den Teil der CreatureAnimation sind vor allem die Statistiken des Trainings relevant. ML-Agents erhebt zwar im Rahmen des Trainingsvorgangs 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 verwendeten Daten bereits Durchschnitte sind und daher am Ende der Durchschnitt des Durchschnittes betrachtet wird, was keine guten Ergebnisse liefert. Ein weiterer Punkt ist, dass in der Konfiguration von ML-Agents eine maximale Anzahl von Checkpoints angegeben werden muss. Sollten im Laufe des Trainings mehr angelegt werden als dieses Limit erlaubt, überschreibt das Programm stattdessen die ältesten. Der im Zweifel wichtigste Nachteil von ML-Agents ist jedoch, dass es sich bei dem Toolkit um ein sehr großes Projekt handelt, welches aus vielen einzelnen Modulen besteht. Aufgrund der Komplexität des Projekts wäre es sehr schwierig die zuvor genannten Nachteile zu beseitigen und es so an die Bedürfnisse der Projektgruppe anzupassen.

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 die neroRL Konfiguration nur den Abstand zwischen Checkpoints und legt dann so viele Checkpoints an, wie nötig. Im Gegensatz zu ML-Agents gibt es keine maximale Anzahl und somit werden keine Checkpoints überschrieben. Zudem bietet neroRL eine Auswertungsfunktion, die nach dem Ende des Trainings verwendet werden kann, um bessere Daten zum Verlauf des Trainings zu erhalten. Dabei wird jeder angelegte Checkpoint des Netzes für eine festgelegte Anzahl an Episoden verwendet, um den Verlauf des Trainings und die Qualitätsunterschiede zwischen den Checkpoints darzustellen. Die Nachteile von neroRL sind, dass nur Diskrete und Multidiskrete Aktionräume unterstützt werden, welche zum Animieren der Kreaturen, nach vorläufigen Tests nicht ausreichen. Außerdem erlaubt es nicht mehrere Agenten in einer Szene zu platzieren. Für das Training skaliert es aber am besten, wenn mehrere Agenten pro Szene in Kombination mit mehreren parallelen Instanzen der Szene verwendet werden. Im Folgenden wird jeweils eine zum Trainieren verwendete Instanz der Szene als Build bezeichnet, da dies der Unity Terminologie entspricht. Eine Instanz einer Multi-Agent Szene ist entsprechend ein Multi-Agent Build.

Es ist also eindeutig, dass unabhängig davon welches Framework gewählt wird dieses noch angepasst werden muss, bevor es im Projekt verwendet werden kann. Die endgültige Entscheidung war es das neroRL Framework zu verwenden. Dieses auf kontinuierliche Aktionen und Umgebungen mit mehren Agenten zu erweitern erschien einfacher, als in ML-Agents das Anlegen der Statistiken und Checkpoints so zu ändern, dass es für die notwendigen Analysen geeignet ist. 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 dies bei Fragen zu ML-Agents der Fall wäre.

5 Umsetzung

Anpassung an neroRL

Bevor neroRL zum Trainieren verwendet werden kann müssen zunächst die zuvor aufgezählten Probleme mit dem Framework beseitigt werden.

Kontinuierliche Aktionsräumen Zunächst muss die Implementierung so erweitert werden, dass sie auch Umgebungen mit kontinuierlichen Aktionsräumen unterstützt. Um die Implementierung so einfach und übersichtlich wie möglich zu halten wurde entschieden, in diesem Schritt die Optionen für Diskrete und Multidiskrete 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. Zum Umsetzen der kontinuierlichen Aktionsräume muss die Ausgabe des Aktor Netzwerkes angepasst werden und diese wird in der `ContinuousActionPolicy` Klasse definiert. Diese 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 gesammelt werden kann. Als Verteilung wird die `Normal` Implementierung einer Gaußschen Normalverteilung aus PyTorch verwendet.

Multi-Agent Builds In dem nächsten Schritt muss die Implementierung so erweitert werden, dass neroRL auch Multi-Agent Szenen unterstützt und nicht das Trainieren mit nur parallelen Builds. Abhängig von den verfügbaren Rechenressourcen lassen sich höchstens 16 Builds parallel ausführen (LiDo3 cgpu01 Knoten mit 2 Intel Xeon E5-2640v4 und 2 NVIDIA Tesla K40). Es wurde entschieden, dass vorhandene Verhalten mit mehreren parallelen Builds zu erweitert, sodass es parallele Multi-Agent Builds unterstützt. Die resultierende Implementierung sollte also in der Lage sein besser zu skalieren, da nicht nur die Anzahl der Agenten in jeder Szene erhöht werden kann, sondern auch die Anzahl an Instanzen der Szene die parallel zum Trainieren verwendet werden. Diese Steigerung an Effizienz sollte mehr als ausreichend sein, um den zeitlichen Overhead durch das Verarbeiten mehrerer Agenten pro Umgebung auszugleichen.

Das von neroRL implementierte System verwendet zum Sammeln der Erfahrungen Buffer, die ermöglichen jede Erfahrung genau ihrem Build und dem Timestep an dem sie erstellt wurden zuzuordnen. Die Buffer mussten also erweitert werden, um zwischen den verschiedenen Agenten in einem Build zu unterscheiden, damit die Erfahrungen weiterhin zugeordnet werden können. Nach diesen Anpassungen hatten die Buffer 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

w \ a	8	16	24	32
1	80-81	53-54	42-43	35
2	47-48	39-41	32-34	31
4	38-39	30-31	41	31
8	34-35	29-30	42-43	40-41
12	34	30-31	47-48	66-67
16	36	33	55-56	100

Abbildung 5.11: Benötigte Sekunden zum Sammeln eines Batches der Größe 65536, Li-Do3 cgpu01 Knoten, 20 Kerne, 48GB Ram, 2 GPUs, Übersicht nach 5-6 Updates

nicht synchronisiert sind. Sobald genug Daten für die definierte Batchgröße gesammelt wurden, werden diese von den verschiedenen Builds und Agenten eingesammelt und in einem Batch kombiniert, welcher dann an den Trainingsalgorithmus übergeben wird. Ein vorläufiger Test mit der angepassten Implementierung und verschiedenen Kombinationen von Agenten und Builds ergab die in Abbildung 5.11 zu sehenden Ergebnisse. Diese zeigen, dass das Hinzufügen von Agenten und Builds den Samplingprozess stärker beschleunigt, als nur die Anzahl von einem der beiden zu erhöhen. Außerdem scheint es eine Sättigung zu geben ab der das Hinzufügen weiterer Agenten und Builds das Sammeln eines Batches wieder verlangsamt.

Optimierung der Trainingsqualität 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. Erste Tests belegten diese Befürchtung und lieferten unzureichende Ergebnisse. Beim Training der ml-Agent Walker Umgebung mit ML-Agents wurde nach ca. 30 Millionen Schritten ein durchschnittlicher Reward von ca. 2000 erreicht. Nach einem Test dieser Umgebung mit neroRL wurde auch nach über 150 Millionen Schritten nur ein Reward von ca. 220 erreicht. 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, wurden verschiedene Optimierungen implementiert:

5 Umsetzung

- **Normalisierung der Observationen:** Die Observationen werden durch den Sampler automatisch normalisiert, um das Training zu stabilisieren und vor ausreichenden 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. [16]). Unsere Implementierung von Tanh-Squashing wurde jedoch verworfen, da diese konsistent das Exploding-Gradients-Problem ausgelöst hat und das Training somit fehlgeschlagen ist.

5.2.5 LiDO3

Für die Trainingsdurchläufe wird das HPC-Cluster der TU-Dortmund genutzt. Zugänge hierfür wurden von unseren Projektgruppenbetreuern zur Verfügung gestellt. Um auf LiDO3 zu arbeiten wird mithilfe 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 erfordert 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[18].

In dem Beispieldskript 1 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 2 ist die Einrichtung einer funktionierenden Umgebung dargestellt.

```

1  #!/bin/bash -l
2  #SBATCH -C cgpu01
3  #SBATCH -c 20
4  #SBATCH --mem=40G
5  #SBATCH --gres=gpu:2
6  #SBATCH --partition=long
7  #SBATCH --time=48:00:00
8  #SBATCH --job-name=pg_k40
9  #SBATCH --output=/work/USER/log/log_%A.log
10 #SBATCH --signal=B:SIGQUIT@120
11 #SBATCH --mail-user=OUR_MAIL@tu-dortmund.de
12 #SBATCH --mail-type=ALL
13 #-----
14 GAME_NAME="GAME_NAME"
15 GAME_PATH="/work/USER/games/$GAME_NAME"
16 module purge
17 module load nvidia/cuda/11.1.1
18 source /work/USER/anaconda3/bin/activate
19 conda activate /work/mmarplei/grudelpg649/k40_env
20 chmod -R 771 $GAME_PATH
21 cd $GAME_PATH
22 srun mlagents-learn /work/smnidunk/games/config/Walker.yaml
   ↳ --run-id=$GAME_NAME --env=t.x86_64 --num-envs=6 --no-graphics

```

Listing 1: Skript zur Ausführung von ML-Agents auf LIDO3.

```

1 module purge
2 module load nvidia/cuda/11.1.1
3 source <anaconda3-path>/bin/activate
4 conda activate <env_to_install>
5 conda install torchvision torchaudio cudatoolkit=11.1 -c pytorch
6 python -m pip install mlagents==0.29.0 --force-reinstall
7 python -m pip install
   ↳ /work/mmarplei/grudelpg649/torch-1.10.0a0+git3c15822-cp39-cp39-linux_x86_64.whl
   ↳ --no-deps --force-reinstall

```

Listing 2: Installationsskript für Python und ML-Agents auf LIDO3.

5 Umsetzung

```
1 GAME_NAME="GameMNSNero"
2 GAME_PATH="/work/<Username>/games/$GAME_NAME"
3 module purge
4 module load nvidia/cuda/11.1.1
5 module load gcc/11.1.0
6 source /work/<Username>/anaconda3/bin/activate
7 conda activate /work/mmarplei/grudelpg649/Jannik/neroEnv
8 chmod -R 771 $GAME_PATH
9 cd /work/<Username>/neroRL
10 srun python -m neroRL.train --config
    ↳ /work/<Username>/games/config/Nero.yaml --run-id Generated2B
    ↳ --worker-id=6780 --env=$GAME_PATH/t
```

Listing 3: Änderung des ML-Agents-Skripts für die Ausführung von NeroRL.

Für die Python-Installation wurde auf Anaconda⁴ zurückgegriffen. Die installierte Anaconda-Arbeitsumgebung kann für die folgenden Schritte genutzt werden, indem die Slurm-Skripte diese am Anfang laden. CUDA kann als Kernelmodul in verschiedenen Versionen geladen werden oder per Anaconda installiert werden.

Problematisch ist die Installation von PyTorch, da ab Version 1.5 die Installationsbinärdateien keine Unterstützung für die von LiDO3 genutzten NVIDIA Tesla K40 Grafikkarten bietet. Es besteht die Möglichkeit PyTorch zu bauen um die Unterstützung zu erhalten. Dies musste für unsere Arbeitsumgebung nicht gemacht werden, da die PG-Betreuer ein Paket mit einer für LiDO funktionierenden PyTorch-Version von einer vorherigen PG zur Verfügung stellen konnten. Wie in 2 dargestellt müssen zuerst die Abhängigkeiten von PyTorch, dann ML-Agents und zuletzt die spezielle PyTorch Version installiert werden, da sonst die Abhängigkeiten Probleme bereiten.

NeroRL-Anpassungen

Im späteren Verlauf der PG wurde die Umstellung auf NeroRL durchgeführt, welche nur geringe Anpassungen an den Skripten bedeutete. Eine Anleitung für die Installation steht im Github-Wiki⁵, des für diese Projektgruppe genutzten Forks, zur Verfügung.

⁴<https://www.anaconda.com/>

⁵<https://github.com/PG649-3D-RPG/neroRL/wiki>

5.3 World Generation

5.3.1 Übersicht

Die Spielwelt basiert auf einem Terrain mit quadratischer Grundfläche. Mit Space Partitioning Algorithmen werden innerhalb des Terrains Levels erstellt. Diese werden durch Korridoren miteinander verbunden, sodass eine zusammenhängende Spielwelt entsteht. Zu dem Terrain werden Masken erstellt, die Levels, Korridore und weitere Objekte und Bereiche markieren. Mithilfe der Masken und Bildfiltern, sowie Perlin-Noise werden Bereiche des Terrains modifiziert, um eine nicht planare Spielwelt zu erhalten. Innerhalb der Levels wird durch L-Systeme generierte Vegetation an zufälligen Positionen in der Welt platziert. Dabei wird ein Mindestabstand zwischen platzierten Objekten eingehalten, so dass innerhalb der Levels alle Bereiche für den Spieler erreichbar bleiben. Weiterhin erhält das Terrain anhand der Masken Texturen. Zu dem fertigen Terrain mit platzierten Objekten wird ein NavMesh erstellt. Da alle Objekte mit einem dem AgentRadius des NavMesh angepassten Mindestabstand platziert werden, entsteht ein zusammenhängendes Navmesh.

5.3.2 Generierung von Levels mit Space Partitioning

Zunächst wird die Seitenlänge der quadratischen Spielwelt festgelegt. Mithilfe von Space Partitioning Algorithmen wird die Spielwelt in Bereiche unterteilt, in denen Levels platziert werden können. Dabei kann die Partitionierung mit 2-d-Bäumen oder Quadtrees erfolgen. Die Ausgangsfläche wird so lange unterteilt, bis eine weitere Unterteilung eine Mindestgröße unterscheidet. Die Ausgabe des Algorithmus besteht aus Rechtecken, in denen kleinere, rechteckige Levels platziert werden. Dabei kann über Parameter für alle vier Seiten der Abstand zu den Begrenzungen des Rechtecks durch ein Intervall aus Minimum und Maximum angegeben werden. Die Abstände werden in den jeweiligen Intervallen zufällig gewählt. So sind die Level durch Zwischenräume voneinander getrennt. Die Größe der Spielwelt, sowie die Parameter zur Begrenzung der Größe und Platzierung der Levels sind ganzzahlig. Damit lässt sich die Spielwelt in Einheitsquadrate mit jeweils einem Quadratmeter Flächeninhalt einteilen, was die Platzierung von Objekten in Levels, das Setzen von freien Pfaden, sowie die Generierung des Terrains vereinfacht.

5.3.3 Levels

Levels in der Spielwelt haben eine rechteckige Grundfläche und enthalten Spawnpunkte für Kreaturen, Vegetation und freie Pfade, die ein Durchqueren der Level unabhängig von den zufällig platzierten Objekten ermöglichen. Zu jedem Level wird ein zweidimensionales Array `free` mit Einträgen vom Typ `bool` gespeichert. Das Array hat die Dimensionen der Grundfläche des Levels, sodass zu jeder ganzzahligen, zum Level relativen Koordinate auf der Grundfläche des Levels ein Eintrag im Array existiert. Ist der Eintrag im

5 Umsetzung

Array für eine Koordinate `true`, so ist das Einheitsquadrat an dieser Koordinate in der Spielwelt frei, andernfalls befindet sich hier ein platziertes Objekt oder es soll an dieser Position kein weiteres Objekt platziert werden.

Freie Pfade in Levels

Da die Levels rechteckige Grundfläche haben, hat jedes Level eine linke, rechte, vordere und hintere Seite. Zu jedem Level werden Punkte auf der Grundfläche gespeichert, an denen anliegende Korridore beginnen oder enden. Die Punkte befinden sich auf einer der vier Seiten des Levels. Damit keine Bereiche der Spielwelt durch platzierte Objekte abgeschnitten werden, muss von jedem anliegenden Korridor jeder andere anliegende Korridor durch einen durch das Level führenden Pfad erreichbar sein. Die Pfade werden vor der Platzierung von Objekten definiert. Dazu werden alle Paare von Punkten anliegender Korridoren aufgezählt und mit einem naiven Algorithmus wird ein Pfad gefunden, der die Punkte jedes Paares miteinander verbindet. Die Lage der beiden Punkte auf den Seiten des Levels kann durch drei Fälle unterschieden werden. Im ersten Fall liegen beide Punkte auf der gleichen Seite. Da dieser Fall nach dem oben beschriebenen Verfahren zur Verbindung von Levels über die von einem Space Partitioning Algorithmus berechnete Baumstruktur nicht eintritt, sondern jede Seite des Levels maximal einen anliegenden Korridor haben kann, wird er nicht behandelt. Falls die Punkte auf gegenüberliegenden Seiten liegen, wird ein Pfad aus drei geraden Segmenten gebildet. Dazu wird ein zufälliger Punkt zwischen den Seiten gewählt auf dem das mittlere, zu beiden Seiten parallele Segment des Pfades liegt. Das mittlere Segment wird über gerade Segmente mit den beiden an den Seiten liegenden Punkten verbunden. Im letzten Fall liegen die Punkte auf miteinander verbunden Seiten, wie z.B. auf der hinteren und linken Seite und können durch einen eindeutigen Pfad, der aus zwei geraden Segmenten besteht verbunden werden. Die letzten beiden Fälle sind in Abbildung 5.12 dargestellt. Die Breite der Pfade ist durch einen minimalen und einen maximalen Wert begrenzt und wird zufällig gewählt. Einheitsquadrate im Level, durch die Pfade verlaufen werden durch einen `false`-Eintrag im `free` Array des Levels als belegt markiert.

Platzierung von Objekten und Spawnpunkten

Objekte, wie zum Beispiel Pflanzen werden an zufällig gewählten Positionen im Level platziert. Dazu wird das zu jedem Level gespeicherte `free` Array genutzt. Zu einem Objekt das platziert werden soll muss die Breite w und Höhe h der Fläche, die das Objekt benötigt vorher bekannt sein. Dabei wird gefordert, dass beide Werte ungerade ganze Zahlen sind. Es wird angenommen, dass sich das Objekt zentriert innerhalb dieser Fläche befindet, wobei sich das tatsächliche Objekt nicht notwendigerweise bis zu den durch w und h gegebenen Begrenzungen ausdehnen muss. Bei der Platzierung des Objektes müssen vorher platzierte Objekte und sich im Level befindende freie Pfade berücksichtigt werden. Es wird eine Position im Level gesucht, an der das Objekt mittig platziert werden

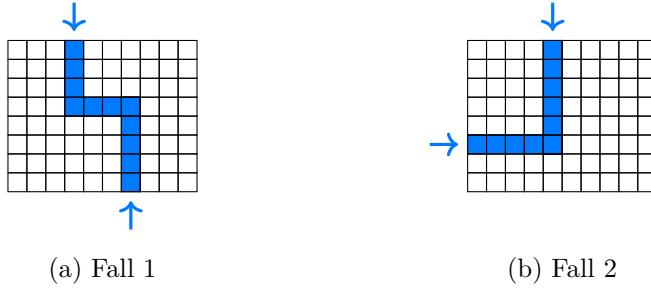


Abbildung 5.12: Verbinden von anliegenden Korridoren in Levels. 5.12a: Punkte befinden sich auf gegenüberliegenden Seiten des Levels. 5.12b: Punkte liegen auf benachbarten Seiten des Levels. Der generierte Pfad ist blau markiert.

kann und dabei nicht mit anderen Objekten oder freien Pfaden kollidiert. Falls $w = h = 1$ ist, kann eine beliebige Position im Level gewählt werden, an der `free` den Eintrag `true` enthält. Andernfalls wird ein zusätzliches Array `a` mit den gleichen Dimensionen wie `free` vom Typ `float` verwendet. Zunächst wird das Array `free` in das Array `a` mit der Zuordnung `false` \mapsto 1.0, `true` \mapsto 0.0 abgebildet. Das Array `a` wird nun so modifiziert, dass an Positionen mit dem Wert 0.0 das Objekt kollisionsfrei platziert werden kann. Um Kollisionen mit anderen Objekten oder Pfaden zu verhindern, werden zwei Faltungsfilter verwendet. Es wird eine Matrix F_w für die horizontale und eine Matrix F_h für die vertikale Ausdehnung des Objektes wie folgt definiert.

$$F_w = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \quad F_h = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

Mithilfe der Faltungsfilter werden Bereiche um bereits platzierte Objekte erkannt, an denen die Platzierung eines neuen Objektes zu einer Kollision führen würde. Angenommen das Level, in dem das Objekt platziert werden soll, hat die Breite L_w und Höhe L_h . Falls $w \geq 3$ ist, wird zunächst \mathbf{a} mit F_w gefaltet, d.h. für alle i, j mit $1 \leq i \leq L_w - 2, 0 \leq j \leq L_H - 1$ wird

$$\mathbf{a}[i, j] = a[i - 1, j] + a[i, j] + a[i + 1, j]$$

gesetzt. Anschließend werden die Einträge in \mathbf{a} , die unmittelbar an der linken und rechten Seite des Levels anliegen auf 1.0 gesetzt, um eine Ausdehnung des Objektes über die Begrenzungen des Levels zu verhindern. Analog wird, falls $h \geq 3$ ist, \mathbf{a} mit F_h gefaltet, d.h. für $0 \leq i \leq L_w - 1$ und $1 \leq j \leq L_h - 2$ wird

$$a[i, j] = a[i, j - 1] + a[i, j] + a[i, j + 1]$$

gesetzt. Die Einträge unmittelbar an der vorderen und hinteren Seite des Levels werden auf 1.0 gesetzt. Zuletzt wird die Faltungsoperation mit $F_w \lfloor w/2 \rfloor - 1$ -mal wiederholt und

5 Umsetzung

die Faltungsoperation mit F_h wird $\lfloor h/2 \rfloor - 1$ -mal wiederholt. Bei den Wiederholungen müssen keine Einträge nahe der Seiten des Levels gesetzt werden, da die in der ersten Iteration der Operationen mit F_w und F_h gesetzten Einträge durch die Wiederholungen erweitert werden. An allen Positionen im Level, an denen das Array `a` den Wert 0.0 enthält, kann das Objekt kollisionsfrei platziert werden. Eine solche Position wird zufällig ausgewählt und in `free` auf `false` gesetzt. Zusätzlich wird die umliegende, durch w und h beschränkte Fläche, die das Objekt in der Welt einnimmt auf `false` gesetzt.

Abbildung 5.13 veranschaulicht die Platzierung eines Objektes der Größe 3×3 in einem Level der Größe 14×12 . In Abbildung 5.13a sind `false`-Einträge in `free` blau markiert. Diese Felder entsprechen 1.0-Einträgen in `a`. Das Ergebnis der Faltung von `a` mit den Filtern F_w und F_h und dem Setzen der Einträge für die Seiten des Levels ist in Abbildung 5.13b dargestellt. Rot markiert sind die Felder, die durch diese Operationen einen größeren Wert als 0.0 erhalten. Abbildung 5.13c zeigt alle Positionen, an denen das Objekt kollisionsfrei platziert werden kann. Das Array `free` nach der Platzierung des Objektes an einer zufälligen grün markierten Position ist in Abbildung 5.13d gegeben.

In der Spielwelt sollen während des Spielverlaufs Kreaturen platziert werden. Dazu wird in jedem Level eine vorher definierte Anzahl an Spawnpunkten platziert. Bei den Spawnpunkten handelt es sich um quadratische Flächen mit ungerader Seitenlänge, die als Paar von Koordinate im Level und Radius r gespeichert werden. Die Spawnpunkte werden ähnlich wie oben beschrieben zufällig in der Spielwelt platziert. Da die Spawnpunkte quadratisch sind, wird hier der Faltungsfilter

$$F_{wh} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

verwendet, der r -fach angewandt wird. Die r -fache Anwendung von F_{wh} entspricht der r -fachen Anwendung von F_w gefolgt von der r -fachen Anwendung von F_h und optimiert die Berechnung des Arrays `a`. Die von Spawnpunkten belegten Positionen werden im Array `free` auf `false` gesetzt.

5.3.4 Generierung des Terrains

Die Spielwelt ist ein Unity Terrain quadratischer Größe. Dem Terrain ist eine Heightmap zugeordnet, über die die Höhe eines Punktes auf dem Terrain gesetzt werden kann. Die Heightmap hat den Wertebereich $[0, 1]$ und wird um einen festgelegten Wert skaliert. Abhängig von der Ausgabe des zuvor beschriebenen Space Partitioning Algorithmus werden Masken in der Größe des Terrains generiert. Die Masken wählen Bereiche in der Spielwelt aus, die mit verschiedenen Operationen modifiziert werden, um eine Abgrenzung zwischen Levels, Korridoren und Zwischenräumen zu erhalten.

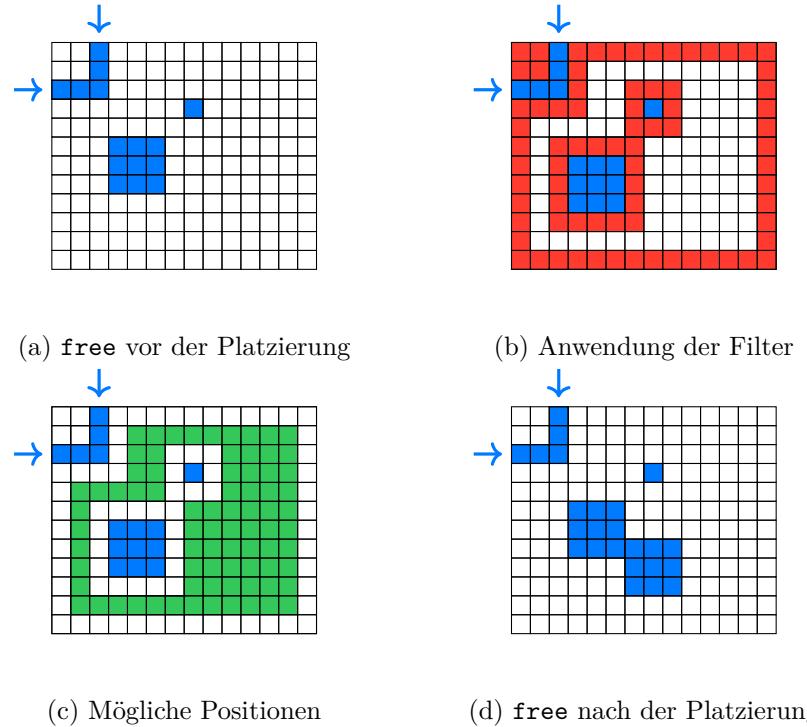


Abbildung 5.13: Beispiel: Platzierung eines Objektes der Größe 3×3 in einem Level, in dem sich oben links ein freier Pfad und zwei bereits platzierte Objekte befinden. Blau markierte Quadrate entsprechen einem `false`-Eintrag im `free` Array.

5 Umsetzung

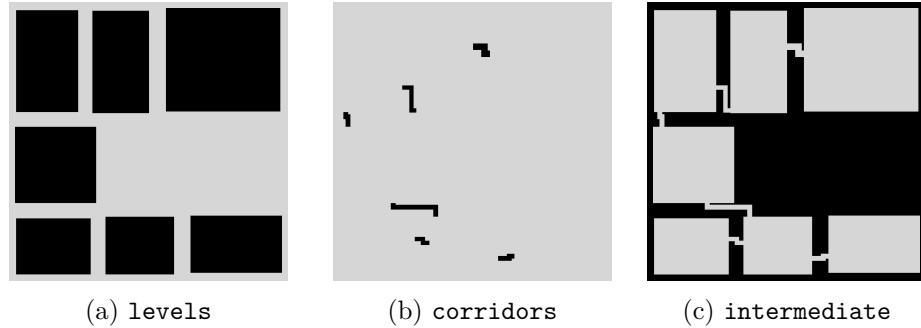


Abbildung 5.14: Masken

Masken

Die Spielwelt besteht aus Levels, Korridoren und Zwischenräumen. Das Spiel findet in den Levels und Korridoren statt. Die Zwischenräume werden mit Bergen gefüllt, wobei nicht vorgesehen ist, dass der Spieler die Zwischenräume betritt. Die Oberfläche des Terrains muss entsprechend modifiziert werden. Dazu werden Masken generiert, die Levels, Korridore und Zwischenräume auf dem Terrain maskieren. Die Masken sind zweidimensionale `bool` Arrays, die die gleiche Größe wie das Terrain haben. Damit ist jedem ganzzahligen Punkt auf der Oberfläche des Terrains ein Wert vom Typ `bool` zugeordnet. Grundlegend werden drei Masken generiert.

- Die Maske `levels` wird aus der unmittelbaren Ausgabe des Space Partitioning Algorithmus generiert und maskiert alle Positionen auf dem Terrain, die in einem Level liegen.
- Die Maske `corridors` maskiert alle Positionen durch die ein Korridor führt.
- Die Maske `intermediate` maskiert die Zwischenräume und wird für die Generierung von Bergen genutzt. Die Maske wird durch Invertierung der Addition der Masken `levels` und `corridors` gebildet.

Weitere Masken können durch Invertierung, Addition, Subtraktion und weitere Operationen erstellt werden. In Abbildung 5.14 sind die Masken `levels`, `corridors` und `intermediate` einer Spielwelt der Größe 512×512 dargestellt. Dabei sind Positionen, an denen die Maske den Wert `true` enthält schwarz markiert und nicht maskierte Positionen sind grau markiert.

Heightmap Operationen

Zur Modifizierung der Heightmap stehen die Operationen `SetByMask`, `AverageFilter`, `PerlinNoise` und `Power` zur Verfügung. Die Operationen modifizieren die Höhen der

Heightmap an den Positionen, die von einer als Eingabe geforderten Maske ausgewählt sind.

- **SetByMask** wird mit einem Parameter vom Typ `float` aufgerufen. Die Operation ersetzt an maskierten Positionen die Höhe der Heightmap durch den gegebenen Parameter. Alle nicht maskierten Positionen bleiben erhalten, bzw. können durch einen optionalen Parameter vom Typ `float` auf einen Wert gesetzt werden.
- **AverageFilter** faltet die Heightmap mit einem Faltungsfilter der Größe 3×3 . Der Filter ist durch

$$F_{\text{Avg}} = \begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix}$$

gegeben und berechnet an jedem Punkt der Heightmap den Mittelwert in einem Radius von 1.

- **PerlinNoise** addiert an maskierten Positionen den Wert einer Perlin-Noise Funktion auf die in der Heightmap gespeicherten Höhe. Dabei kann über Parameter die Skalierung des Perlin-Noise reguliert werden.
- **Power** erhält einen Parameter z vom Typ `float`. Jeder maskierte Eintrag v der Heightmap wird durch v^z ersetzt.

Abbildung 5.15 zeigt die Operationen am Beispiel eines Terrains der Größe 64×64 mit einer Maske, die eine mittig liegende quadratische Fläche auswählt.

5.3.5 Terrain-Transformationen

Nachdem das Layout der Welt mittels Space-Partitioning 2.2 berechnet wurde und dieses anschließend in entsprechende Terrain Masken 5.3.4 unterteilt wurde, muss die Welt nun in Unity generiert werden. Die Basis dafür bildet das Unity Terrain Objekt. Um das Terrain aus dem Welt-Layout prozedural zu generieren, werden sukzessive die verschiedenen, in 5.3.4 beschriebenen, Operationen auf den Teilen der Heightmap ausgeführt, die durch die Terrain-Masken vorgegeben werden. Die folgenden Transformationen werden in dieser Reihenfolge auf Teilen der Heightmap ausgeführt:

1. **SetByMask**: Die Räume und Korridore des Space-Partitioning werden in der Heightmap umgesetzt. Dies geschieht einmalig am Anfang und bildet die Grundlage für das Welt-Layout.
2. **AverageFilter**: Ausgehend von Schritt 1 ist die Trennung vom begehbareren Bereich (Räume und Korridore) und den Bergen im Terrain eine scharfe Kante. Die Heightmap springt an den Begrenzungen von 0 auf 1, was im Spiel unnatürlich wirkt. Um diese Übergänge zu glätten wird der **AverageFilter** 600-mal auf den nicht-begehbareren Bereich angewandt. Dies sorgt dafür, dass eine glatte Kante zwischen den Räumen und Korridoren und den Bergen entsteht.

5 Umsetzung

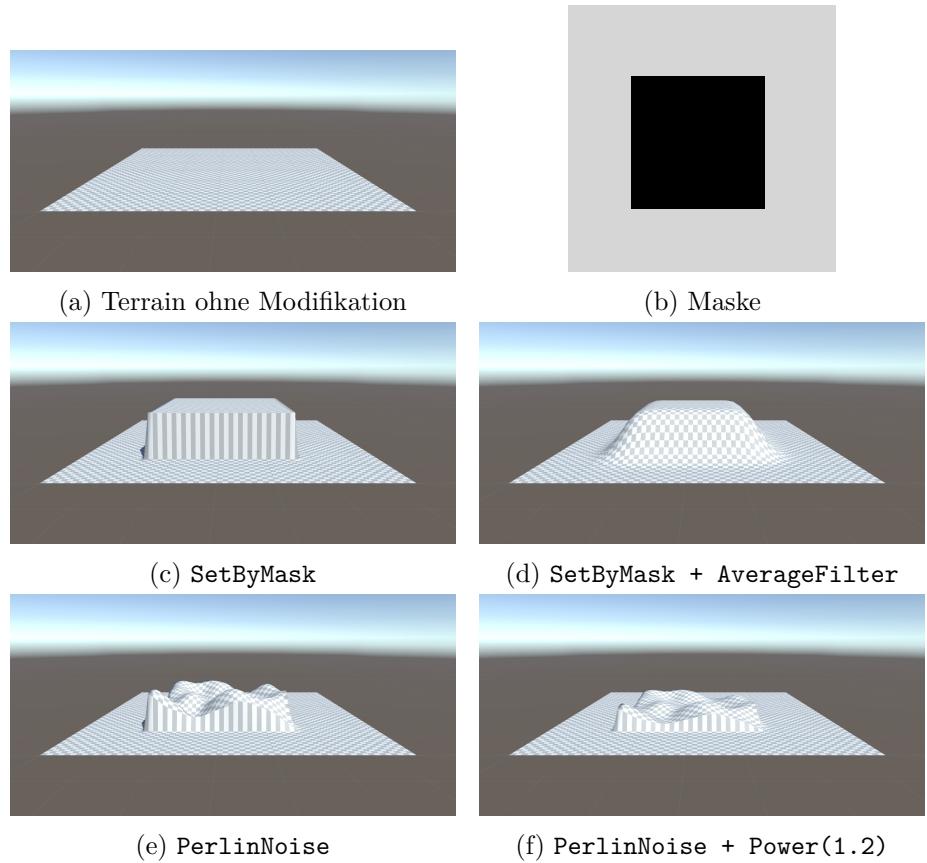


Abbildung 5.15: Operationen zur Modifikation der Heightmap des Terrains

3. **PerlinNoise:** Damit die Berge eine natürliche Struktur erhalten, wird auf diesen Bereichen 3-mal ein Perlin Noise angewandt. Diese Anwendung des Perlin Noise lässt die hohen Berge entstehen.
4. **Power:** Um die Höhenunterschiede in den Bergen natürlicher wirken zu lassen, wird an jedem Punkt der Wert der Heightmap einmal mit 3 potenziert.
5. **PerlinNoise:** Ausgehend davon, wird nochmals auf den Bergen 2-mal ein Perlin Noise angewandt, welches Berge entstehen lässt, die etwas kleiner sind.
6. **AverageFilter:** Um die Korridore und Räume herum, wird ein 6 Einheiten breiter Bereich geglättet. Um dies zu erreichen, wird der **AverageFilter** 15-mal dort angewandt.
7. **SetByMask:** Zum Schluss wird um die Korridore und Räume ein 1 Einheiten breiter Bereich leicht erhöht. Dies trennt die Berge besser von der begehbarer Spielwelt.

Eine Herausforderung bei der Umsetzung dieser Idee ist, dass diese Operationen auf allen, von der jeweiligen Maske vorgegebenen, Punkten ausgeführt werden müssen. Besonders bei größeren Heightmaps ist dies problematisch, da beispielsweise für eine Heightmap mit einer Seitenlänge von 1024, Operationen auf bis zu 1.048.576 Punkten ausgeführt werden müssen. Rechenintensive Operationen, wie der **AverageFilter** werden zwar nur auf einer Teilmenge der Punkte ausgeführt, jedoch besteht dabei auch das Problem, dass der Filter sehr oft angewandt werden muss. Damit die Welt auch zur Laufzeit generiert werden kann, ohne dass dies zu sehr langen Ladezeiten führt, ist es wichtig, dass die einzelnen Operationen effizient implementiert werden.

Implementation der Terrain-Transformationen

Eine zentrale Beobachtung bei der Implementation der Transformationen ist, dass die Operationen auf den einzelnen Punkten der Heightmap, alle unabhängig voneinander ausgeführt werden können. Die einzige Abhängigkeit besteht zwischen den einzelnen Anwendungen einer Operation. Dies führt dazu, dass sich die Operationen effizient parallelisieren lassen. Insgesamt wurden fünf Versionen der Terrain-Transformationen implementiert:

Um die Ideen der verschiedenen Transformations-Operationen auszuprobieren, wurden zunächst Funktionen implementiert, die nahezu beliebige Operationen auf Punkten der Heightmap ausführen, die von einer Maske vorgegeben werden. Diese Funktionen iterieren über alle Punkte der Heightmap und prüfen dabei, ob der aktuelle Punkt in der Maske liegt. Ist dies der Fall, wird die Operation auf dem Punkt ausgeführt. Eine Besonderheit ist, dass in dieser Version die Implementation des **AverageFilter**, nicht auf den 3x3 **AverageFilter** beschränkt ist, sondern dass die Funktion, Faltungsfilter beliebiger Größe und Form auf die Heightmap anwenden kann. Diese Implementation wurde genutzt um herauszufinden, welche Transformationen sinnvoll sind. Die Ausführung dieser Version der Operationen erfolgt stets sequenziell und ausschließlich auf der CPU.

5 Umsetzung

```

1      static const float avg_3x3 = 1/(float)9;
2      [numthreads(256,1,1)]
3      void AverageFilterAdd3x3 (uint3 id : SV_DispatchThreadID) {
4          uint item = _work_items[id.x];
5          if (id.x >= _work_item_count) return;
6          uint row = item / _size_input;
7          uint col = item % _size_input;
8          float sum = 0;
9          sum += _input[(row - 1) * _size_input + (col - 1)] * avg_3x3;
10         sum += _input[(row - 1) * _size_input + (col)] * avg_3x3;
11         sum += _input[(row - 1) * _size_input + (col + 1)] * avg_3x3;
12         sum += _input[(row)*_size_input + (col - 1)] * avg_3x3;
13         sum += _input[(row)*_size_input + (col)] * avg_3x3;
14         sum += _input[(row)*_size_input + (col + 1)] * avg_3x3;
15         sum += _input[(row + 1) * _size_input + (col - 1)] * avg_3x3;
16         sum += _input[(row + 1) * _size_input + (col)] * avg_3x3;
17         sum += _input[(row + 1) * _size_input + (col + 1)] * avg_3x3;
18         _output[row * _size_input + col] = sum;
19     }

```

Listing 4: ComputeShader für den **AverageFilter**.

Nachdem die gewünschten Transformationen identifiziert und implementiert wurden, musste der **AverageFilter** effizienter werden, da dieser die meiste Rechenzeit in Anspruch nimmt. Da diese Operation sehr häufig (bis zu 600-mal) wiederholt wird und vollständig parallelisiert werden kann, ist es naheliegend für den **AverageFilter** einen ComputeShader für die Ausführung auf einer Grafikkarte zu schreiben. Ein ComputeShader ist eine Funktion, die beispielsweise auf einer Grafikkarte ausgeführt werden kann. Hierbei wird pro Funktionsaufruf jeweils ein Datenpunkt übergeben und auf einer Recheneinheit der Grafikkarte ausgeführt. Es werden der Grafikkarte stets Gruppen von Datenpunkten übergeben, die gleichzeitig ausgeführt werden. Zusätzlich zur Implementation des **AverageFilter** in einem ComputeShader, müssen die Daten, auf die die Grafikkarte zugreifen soll, erst noch so vorbereitet werden, dass diese in den Speicher der Grafikkarte geladen werden können. Listing 4 zeigt den Programmcode des ComputeShaders für den **AverageFilter**. Da häufige Fallunterscheidungen auf einer Grafikkarte ineffizient sein können, wird hier die Terrain-Maske nicht übergeben und für jeden Datenpunkt überprüft, sondern es werden direkt nur die Punkte der Heightmap bearbeitet, die auch in der Maske enthalten sind. Zudem wird die zweidimensionale Heightmap in ein eindimensionales Array umgewandelt und dementsprechend anders adressiert. Diese Umwandlungen und das Laden der linearisierten Heightmap auf die Grafikkarte müssen einmalig zum Start der **AverageFilter** Operation ausgeführt werden. Da der Filter häufiger nacheinander ausgeführt wird und nicht jedes Mal die Daten konvertiert werden sollen, kann der **AverageFilter** Funktion, direkt die Anzahl an Wiederholungen übergeben werden. Damit das Ergebnis-Array nicht nach jeder Ausführung von der Grafikkarte

zurück in den Hauptspeicher kopiert werden muss, werden im Speicher der Grafikkarte Platz für ein Eingabe-Array und ein Ausgabe-Array alloziert. In jeder Iteration werden die Ein- und Ausgabe-Arrays vertauscht, sodass die Daten der Heightmap zwischen den Iterationen nie den Speicher der Grafikkarte verlassen müssen. Dies würde die Laufzeit erheblich verlängern.

Die dritte Optimierung der Terrain-Transformationen umfasst parallelisierte Versionen der anderen Operationen. Diese werden jedoch auf der CPU ausgeführt, da der Overhead für die Vorbereitung der Daten für die Grafikkarte zu hoch ist. Der Programmcode für die einzelnen Operationen verändert sich hier nicht stark, da die `for`-Schleifen einfach durch `Parallel.For` Konstrukte ersetzt werden können. Hierdurch werden die zu verarbeitenden Daten in Gruppen unterteilt und auf der CPU parallel verarbeitet.

Ein Problem, bei der Nutzung eines ComputeShaders ist, dass dieser eine geeignete Grafikkarte voraussetzt. Im Laufe des Projekts ist es aufgefallen, dass beispielsweise beim Trainieren der Kreaturen auf dem LiDO3 die Grafikkarte nicht zum Generieren der Welt verwendet werden konnte. Wenn Unity ohne Bildschirm gestartet wurde (headless mode), trat dieses Problem auch auf. Aus diesem Grund musste auch eine performante Version des `AverageFilter` implementiert werden, der nur auf der CPU ausgeführt wird. Dazu wurde zunächst der Programmcode der ursprünglichen Filter-Operation mittels `Parallel.For` Konstrukten auf der CPU parallelisiert. Diese Funktion dient als Ersatz für den ComputeShader, falls keine Grafikkarte verfügbar ist, oder diese den ComputeShader nicht unterstützt. Die Laufzeit dieser Implementation war zwar für das Training akzeptabel, aber schlechter geeignet, falls im echten Spiel der ComputeShader nicht unterstützt würde. Die aktuelle Implementation des `AverageFilter` verwendet den Unity *Burst*-Compiler, um hochoptimierten Programmcode für den Filter zu erzeugen. Der *Burst*-Compiler verwendet das Unity Job System, indem einzelne Funktionen als Jobs definiert und als Gruppen parallel auf der CPU ausgeführt werden können. Das Vorgehen ist hier ähnlich wie bei den ComputeShadern. Der große Vorteil des *Burst*-Compilers ist, dass dieser aus einem Job hochoptimierten Maschinencode generieren kann, der zudem automatische Vektorisierung auf der CPU unterstützt. Das heißt, der Compiler erzeugt automatisch Maschinencode, welcher Vektorinstruktionen der CPU verwendet, um mehrere Datenpunkte gleichzeitig auf einem Kern zu verarbeiten. Dies erfolgt zusätzlich zur Parallelisierung auf mehreren Kernen der CPU.

Laufzeitmessungen

Um die einzelnen Implementierungen zu vergleichen und zu evaluieren, ob die Optimierungen auch wirklich zu einer verbesserten Laufzeit führen, wurden Messungen für alle Implementierungen durchgeführt. Die folgenden Messungen wurden auf einem PC durchgeführt, der über 32 GB RAM verfügt und einen Intel Core i7-5820k (6 Kerne/12 Threads) sowie eine NVIDIA GTX 780 Ti enthält. Die Größe der Heightmap der generierten Welt ist 1024x1024, was einer Fläche von ungefähr 1 km² entspricht. Dies ist auch die Größe der Welt im Spiel. Die Messungen sind in Abbildung 5.16 dargestellt.

5 Umsetzung

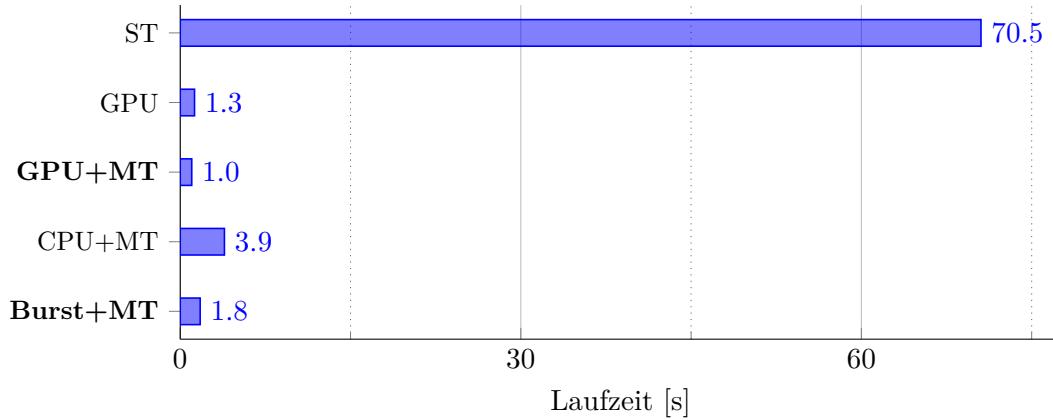


Abbildung 5.16: Messungen der Laufzeiten für die Terrain-Transformationen.

Zunächst lässt sich erkennen, dass die initiale, sequenzielle, Implementation (ST) mit einer Laufzeit von über einer Minute sehr viel Zeit beansprucht. Diese Version ist ungeeignet um die Welt beim Start des Spiels in einem Ladebildschirm zu generieren. Sobald der `AverageFilter` auf der Grafikkarte ausgeführt wird (GPU), ist die Laufzeit mit 1,3 s allerdings sehr gut. Die weitere Parallelisierung der anderen Operatoren (GPU+MT) sorgt für die kürzeste Generierungsdauer von nur 1 s. Die Ausführungszeit für den naiv parallelisierten `AverageFilter` auf der CPU (CPU+MT), ist mit 3,9 s akzeptabel, aber erheblich langsamer als die Version die ComputeShader verwendet. Mit einer Laufzeit von 1,8 s ist die Implementation, die den *Burst*-Compiler verwendet (Burst+MT), schnell genug, um als Ersatz für die ComputeShader zu dienen, falls diese nicht unterstützt werden. Die Messungen der Laufzeiten zeigen, dass die Optimierungen der Terrain-Transformationen es erlauben, die Welt stets prozedural beim Start des Spiels zu generieren.

5.3.6 L-System Vegetation

Das in 2.1 beschriebene L-System wird verwendet um die Welt mit prozedural generierten Pflanzen zu befüllen. Diese dienen als Hindernisse für den Spieler und die Kreaturen. Die Pflanzen werden zur Laufzeit aus vordefinierten L-Systemen erzeugt und in der Welt platziert. Für jedes Objekt wird zunächst das L-System ausgewertet, sodass eine Liste von Tupeln mit Start- und Endpunkten der Segmente entsteht. Ähnlich wie in Abschnitt 5.1.1 beschrieben, wird nun für jedes Segment ein primitives Mesh erzeugt, welche zusammen die Pflanze darstellen. Eine Auswahl an Bäumen die durch L-Systeme erzeugt werden, ist in Abbildung 5.17 zu sehen.

Auswirkungen auf die Leistung Die Bäume fügen dem Level einiges an komplexer Geometrie hinzu, sodass darauf geachtet werden muss, dass die Leistung nicht zu stark

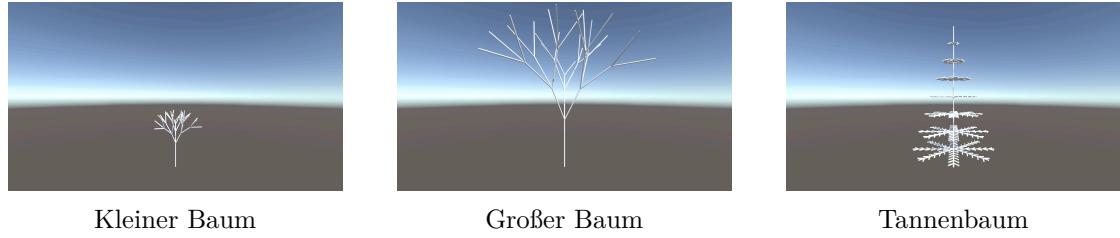


Abbildung 5.17: Bäume die durch L-Systeme erzeugt werden.

einbricht und das Spiel spielbar bleibt. Um dem entgegenzuwirken wurden verschiedene Optimierungen vorgenommen und die Auswirkungen auf die Leistung gemessen. Als Grundlage für die Messungen dient die in Abbildung 5.18 dargestellte Szene.

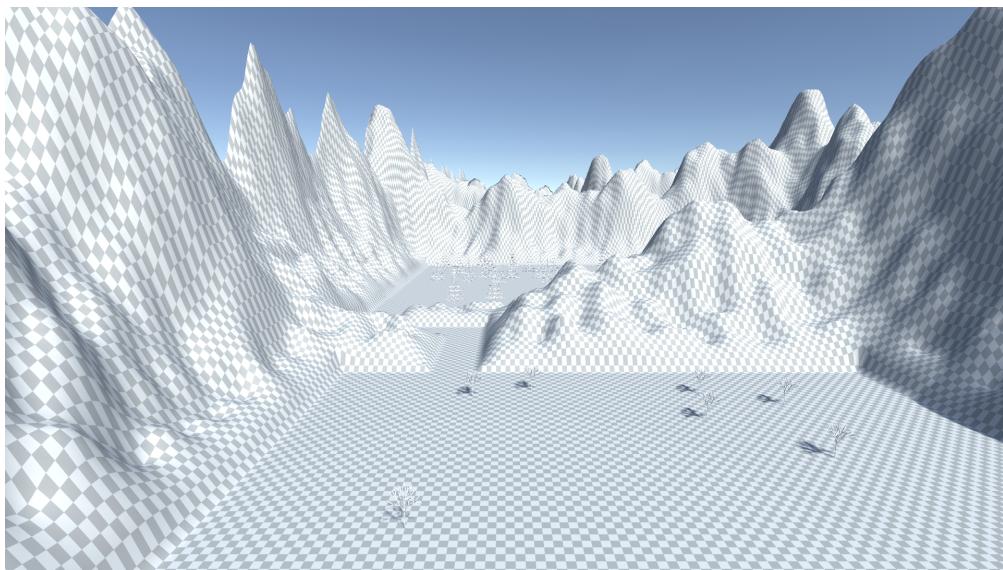


Abbildung 5.18: Testszene für die Vegetationsmessungen

Im Folgenden werden die verschiedenen Optimierungen beschrieben und anhand von Messungen evaluiert. Für jede Version wurden dabei die resultierenden Bilder pro Sekunde (FPS) und die Gesamtdauer für die Generierung erfasst. Zusätzlich zur Gesamtdauer wurde auch die Zeit erfasst, die benötigt wird, um die GameObjects zu erzeugen und um das NavMesh zu berechnen. Die Messungen sind in Abbildung 5.19 dargestellt.

Die zunächst implementierte Version (capsules) folgt dem Vorgehen aus Abschnitt 5.1.1. Hierbei wird für jedes Segment ein neues primitives Capsule-GameObject erzeugt, welche als Kinder dem Pflanzen-Objekt angehangen werden. Für den großen Baum resultiert dies in jeweils 121 Kind-Objekte und für den Tannenbaum in 696 Kind-Objekte. Jedes dieser Objekte wird von Unity einzeln verarbeitet, was zu unspielbaren 8 Bildern pro Sekunde führt. Zudem dauert es mit 19,6s inakzeptabel lange die Welt zu generieren.

5 Umsetzung

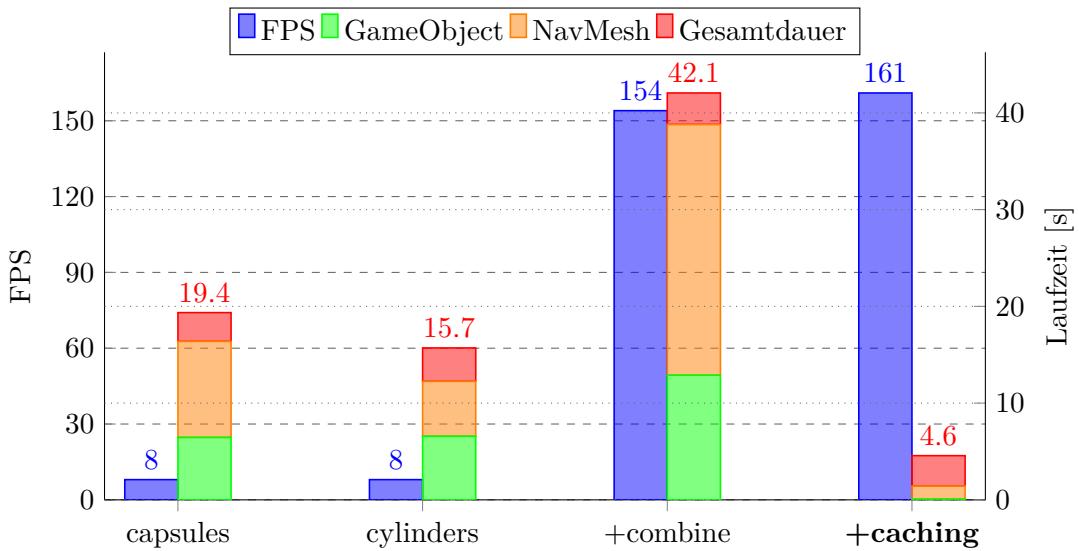


Abbildung 5.19: Messungen über die resultierenden Bilder pro Sekunde (FPS) und Dauer zum Generieren des Levels mit L-System Vegetation.

An den Laufzeit-Anteilen ist gut erkennbar, dass die Berechnung des NavMeshes mehr als die Hälfte der Zeit in Anspruch nimmt. Auch das Erzeugen der GameObjects hat mit ca. 7 s einen großen Einfluss auf die Gesamtdauer.

Die erste Optimierungsidee (cylinders) war, anstatt Kapseln, Zylinder als GameObjects für die Segmente zu verwenden. Ein **CapsuleMesh** besteht aus 832 Dreiecken, während ein **CylinderMesh** nur aus 80 Dreiecken besteht. Dies führt zu einer erheblich verringerten Komplexität der Pflanzen, ohne dass sich das Aussehen deutlich verändert. In der Testszene werden nun nur noch 13,1Mio. Dreiecke gerendert, anstatt der 126,5Mio. Dreiecke, wenn das **CapsuleMesh** verwendet wird. Dies führt zwar zu einer Reduktion der Gesamtdauer auf 15,7 s, aber keiner Steigerung der Bilder pro Sekunde. Gut zu beobachten ist hier, dass die reduzierte Komplexität der einzelnen Pflanzen auch die Berechnung des NavMeshes beschleunigt.

Damit pro Pflanze nicht mehr hunderte Kind-Objekte erstellt werden müssen, werden nun die **CylinderMeshes** der einzelnen Segmente zu einem großen Mesh kombiniert. Das zusammengesetzte Mesh des großen Baums besteht nun aus 9680 Dreiecken und das Mesh des Tannenbaums aus 55680 Dreiecken. Die einzelnen Meshes der Pflanzen sind also erheblich komplexer geworden, jedoch muss Unity nun nicht mehr jedes einzelne Segment als eigenes GameObject verarbeiten. Diese Optimierung (+combine) löst das Bilder-pro-Sekunde-Problem, da nun 154 Bilder pro Sekunde erreicht werden. Es ergibt sich nun jedoch ein neues Problem, da die Gesamtdauer der Generierung nun auf inakzeptable 42,1 s steigt. Zu erkennen ist, dass die Berechnung des NavMeshes nun ungefähr 25 s benötigt und auch das Erzeugen der GameObjects nun länger dauert. Die deutlich komplexeren Meshes der Pflanzen sind also ein Problem, wenn jede Pflanze ein

unabhängiges GameObject ist, obwohl diese gleich aufgebaut sind.

Die letzte Optimierung (+caching) löst dieses Problem. Es wird nun für jede Pflanze nur einmal das L-System ausgewertet und das Mesh gebildet. Jede weitere Instanz der Pflanze wird als Klon des ursprünglichen GameObjects instanziert. Dies führt zu einer weiteren Steigerung auf 161 Bilder pro Sekunde und einer stark verringerten Generierungszeit auf 4,6 s. Durch das Klonen der Pflanzen dauert das Erzeugen der GameObjects nur noch insignifikant lang und die Berechnung des NavMeshes reduziert sich auch auf ungefähr 1 s. Insgesamt führen die Optimierungen dazu, dass die Pflanzen zur Laufzeit prozedural generiert und platziert werden können, ohne dass sehr lange Wartezeiten entstehen.

5.4 Spiel

Die folgenden Abschnitte werden zuerst das Design des Spiels darlegen und einige der getroffenen Design-Entscheidungen erläutern und dann einen Überblick über die Implementation des Spiels liefern.

5.4.1 Design

Setting

Diese Wahl des Settings birgt einige Vorteile. Untote als Gegner zu verwenden erklärt sowohl die zwar Humanoiden, aber nicht unbedingt korrekt proportionierten, Ergebnisse des Creature Generators, als auch die zwar physikalisch basierten, aber nicht unbedingt natürlichen, Ergebnisse des Trainingsprozesses.

Kein Mitglied der Projektgruppe ist ein Erfahrener Künstler. Für Texturen und Modelle wurden deshalb frei verfügbare Quellen verwendet. Dieser Ansatz kann nicht mit der künstlerischen Kohärenz eines Teams von 3D-Modellierern, Grafikern, und Künstlern mithalten. Die Dunkelheit des Settings hilft allerdings stark dabei dies zu kaschieren.

Einen Wald als Schauplatz zu verwenden erlaubt es das Wissen über L-Systeme, dass sich in der Kreaturengenerierung als Sackgasse erwiesen hatte, weiter zu verwenden um prozedural Bäume zu generieren.

Die Wahl des Settings komplementiert somit die Ergebnisse der Projektgruppe und gleicht die Schwächen ihrer Mitglieder aus.

5 Umsetzung

Gameplay

Wie zuvor erwähnt ist es die Aufgabe des Spielers sich vor den Untoten Kreaturen zu schützen. Der Spieler verliert, wenn die Kreaturen ihn insgesamt dreimal berühren. Der Spieler gewinnt, wenn er es schafft 10 Minuten lang nicht dreimal getroffen zu werden.

Der Spieler bekommt dafür zwei Werkzeuge

- eine Taschenlampe, mit der er Kreaturen aus größerer Entfernung sehen kann,
- ein Gewehr, dass Kugeln verschießt, die sich auf etwa einen Meter Durchmesser aufblasen und an Oberflächen kleben bleiben.

Der Spieler muss das Gewehr nutzen um die ihn verfolgenden Kreaturen zu Fall zu bringen, zum Beispiel indem er genügend Kugeln auf die Kreatur schießt, dass diese unter dem Gewicht der Kugeln zusammenbricht.

Gefallene Kreaturen die sich einige Zeit lang nicht mehr nennenswerte Bewegen können sterben und werden aus dem Spiel entfernt. Genauso entfernt das Spiel Kreaturen, die zu weit vom Spieler entfernt sind um noch Einfluss auf das Spiel zu haben. Dafür werden an bestimmten, vom Terrain Generator festgelegten, Punkten regelmäßig neue Kreaturen erstellt, wobei das Spiel die Kreaturen möglichst nah am Spieler platziert, so dass der Spieler immer in Gefahr ist.

Das Gewehr nutzt elegant das Alleinstellungsmerkmal des Spiels aus, dass die Kreaturen durch ein neuronales Netz animiert werden. Eine Kugel am Bein einer Kreatur zu platzieren ändert sichtlich ihr Bewegungsverhalten, da die Kreatur weiterhin versucht sich normal zu bewegen, was aber durch die zusätzliche Masse am Bein nicht gelingt. Eine normale Animation würde dieses Verhalten nicht aufweisen. Der Spieler kann also experimentieren, wie er sein Gewehr am effizientesten verwenden kann.

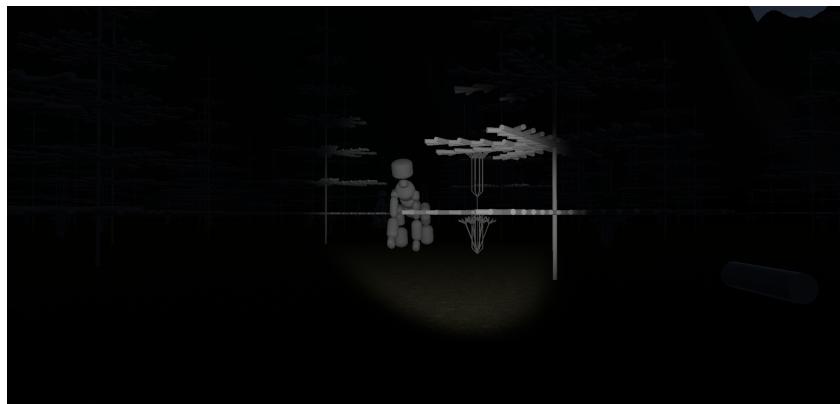
Steuerung

Das Spiel nutzt die konventionelle Steuerung für FPS-Spiele. Der Spieler bewegt sich mit den WASD-Tasten und kontrolliert die Kamera mit der Maus. Tabelle 5.2 zeigt die vollständige Tastenbelegung.

5.4.2 Implementation

Das Spiel besteht aus 5 Unity-Scenes. Die `EntryPoint` Szene dient als Einstiegspunkt in das Spiel. Die `Level` Szene enthält das eigentliche Spiel. Die `LoadingScreen`, `GameOverScreen`, und `WinScreen` Szenen enthalten jeweils lediglich User Interfaces.

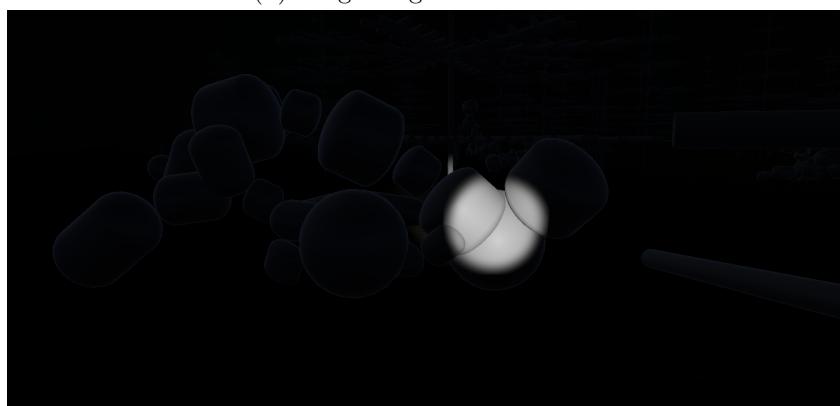
Die folgenden Abschnitte erläutern zunächst die Szenen genauer und gehen anschließend genauer auf den Spielercharakter, die Kreaturen, und die Werkzeuge ein.



(a) Kreatur im dunklen Wald



(b) Berge umgeben den Wald



(c) Zu Fall gebrachte Kreatur

5 Umsetzung

Taste	Funktion
W	Vorwärts
A	Links
S	Rückwärts
D	Rechts
Maus	Umschauen
Links-Klick	Taschenlampe an/aus schalten, Schießen
1	Taschenlampe ausrüsten
2	Gewehr ausrüsten
F12	Screenshot

Tabelle 5.2: Steuerung des Spiels

EntryPoint

Die EntryPoint Szene enthält drei `GameObjects`.

Das `Camera` Objekt enthält die eigentliche Kamera Komponente und das `CinemachineBrain`. Dies ist die einzige Kamera im Spiel. Alle weiteren Szenen werden Additiv zur EntryPoint Szene geladen und enthalten jeweils nur virtuelle Kameras. So ist Cinemachine's Invariante, dass nur eine nicht-virtuelle Kamera existieren darf, gesichert.

Das `Debug` Objekt enthält eine simple `DebugActions` Komponente, die das Erstellen von Screenshots mittels der F12 Taste ermöglicht.

Das `Init` Objekt enthält eine `Init` Komponente, die in ihrer Start-Methode den Szenenübergang zur `Level` Szene auslöst. Der Übergang wird von der `SceneTransition` Klasse geleitet, die ebenfalls an dem `Init` Objekt anhängt. Für den Szenenübergang werden zunächst additiv die `LoadingScreen` und `Level` Szenen geladen und die `Level` Szene wird als aktive Szene gesetzt. Danach werden in einer Koroutine das Terrain generiert, Kreaturen generiert, und das Spielerobjekt erzeugt, wobei die Koroutine nach jedem Arbeitsschritt die Kontrolle abgibt, um das Spiel nicht einzufrieren.

LoadingScreen, GameOverScreen, WinScreen

Die drei UI Szenen sind identisch aufgebaut. Sie enthalten jeweils ihre eigene virtuelle Kamera, ein `Input` Objekt mit einem `EventSystem` und dem `Input System UI Input Module`, und ein `UIDocument`.

Level

Die `Level` Szene enthält erneut eine virtuelle Kamera und ein direktionales Licht für die Szene.

Die **CreatureFactory** verwaltet die vom **CreatureGenerator** erzeugten Kreaturen. Der **CreatureGenerator** produziert Bäume von **GameObjects**. Da **GameObjects** immer Teil einer Szene sind, muss eine Prototyp Kreatur irgendwo in der Szene existieren, von der neue Kreaturen geklont werden können. Die **CreatureFactory** verwaltet diese Prototypen und bietet Methoden zum erstellen neuer Kreaturen. An dem **CreatureFactory** **GameObject** hängen außerdem Konfigurationsskripte, die ML-Agents Einstellungen beinhalten.

Das **SpawnManager** Skript ist für das Erzeugen und Entfernen von Kreaturen wie in Abschnitt 5.4.1 beschrieben zuständig. Dazu verwaltet es eine Liste von momentan aktiven Kreaturen. In jedem Frame kontrolliert es zunächst ob Kreaturen sich selbst entfernt haben, dies kommt vor, wenn eine Kreatur bewegungsunfähig geworden ist. Danach überprüft es für die verbliebenen Kreaturen, ob sie entfernt werden müssen, da sie zu weit vom Spieler entfernt sind, und erzeugt schließlich neue Kreaturen an den dem Spieler an nächsten liegenden Spawnpunkten.

Das **GameManager** Skript verwaltet die seit Spielstart vergangene Zeit und beendet das Spiel nachdem 10 Minuten vergangen sind.

Spieler

Der Spielercharakter ist als Prefab gespeichert und wird während des Ladevorgangs im Level platziert. Es besteht aus einem Player **GameObject**, dass die für den Spielercharakter nötige Logik enthält, einer Kapsel als **Collider**, und zwei weiteren **GameObjects** die als Markierungen für die Positionen von Kamera und Werkzeugen dienen.

Das **Player** Objekt enthält die von Unity gestellten Komponenten **Character Controller** und **Player Input**, das Skript **Player Input State**, dass die von **Player Input** generierten Events erhält und so einen momentanen Eingabezustand verwaltet, und das Skript **FPSController**, was eine Weiterentwicklung des FPS Controllers aus Unitys FPS Template ist.

Der **FPSController** wurde um die Verwaltung des Lebenspunkte des Spielers und der Werkzeuge erweitert. Um die Lebenspunkte zu verwalten wird ein Kollisionscallback im **FPSController** aufgerufen, wenn der Spieler mit den Knochen einer Kreatur kollidiert. Das Verwalten der Werkzeuge erfolgt mittels eines **IEquipment** Interfaces, dass in Abschnitt 5.4.2 genauer erläutert wird.

Kreaturen

Herz der Kreaturen Implementation ist das **BasicCreatureController** Skript. Es ist dafür verantwortlich Pfade auf dem Navmesh zu finden und den jeweils nächsten Wegpunkt an die KI zu liefern. Außerdem überwacht es die Geschwindigkeit der Kreatur an der es angebracht ist und entfernt sie aus dem Spiel, wenn sie sich zu lange nicht

5 Umsetzung

bewegt. So wird verhindert, dass das Budget des **SpawnManagers** mit Bewegungsunfähigen Kreaturen verbraucht wird. Das **BasicCreatureController** Skript wird in der **CreatureFactory** automatisch an neu erstellte Kreaturen gehangen.

Werkzeuge

Die Werkzeuge sind als Prefabs gespeichert. Das oberste Objekt des Prefabs muss jeweils ein Skript besitzen, dass das **IEquipment** Interface implementiert. Das Interface besitzt die 4 Methoden **OnPrimary**, **OnSecondary**, **OnEquip**, und **OnUnequip**, die vom **FPSController** aufgerufen werden. Mit diesen Methoden können die Werkzeuge darauf reagieren, dass die linke, bzw. rechte, Maustaste gedrückt wurde, dass sie ausgerüstet wurden, und dass ein anderes Werkzeug ausgerüstet wurde.

Die Taschenlampe enthält ein **Spotlight**, dass beim Klicken der linken Maustaste an- und ausgeschaltet wird.

Das Gewehr enthält ebenfalls ein **Spotlight** für den Lichtkegel des Gewehrs. Beim Klicken der Linken Maustaste führt das Gewehr einen Raycast von der momentanen Kamera aus und platziert ein **BallGunProjectile** an der so gefundenen Stelle. Das **BallGunProjectile** spielt dann selbstständig seine Animationen ab und entfernt sich nach einiger Zeit selbst wieder aus der Szene.

Beide Werkzeuge reagieren auf **OnEquip**, **OnUnequip** damit sich selbst An- und Abzuschalten.

6 Evaluation

In diesem Kapitel wird die Trainingsqualität der generierten Kreaturen evaluiert.

6.1 Einschränkungen der Evaluation

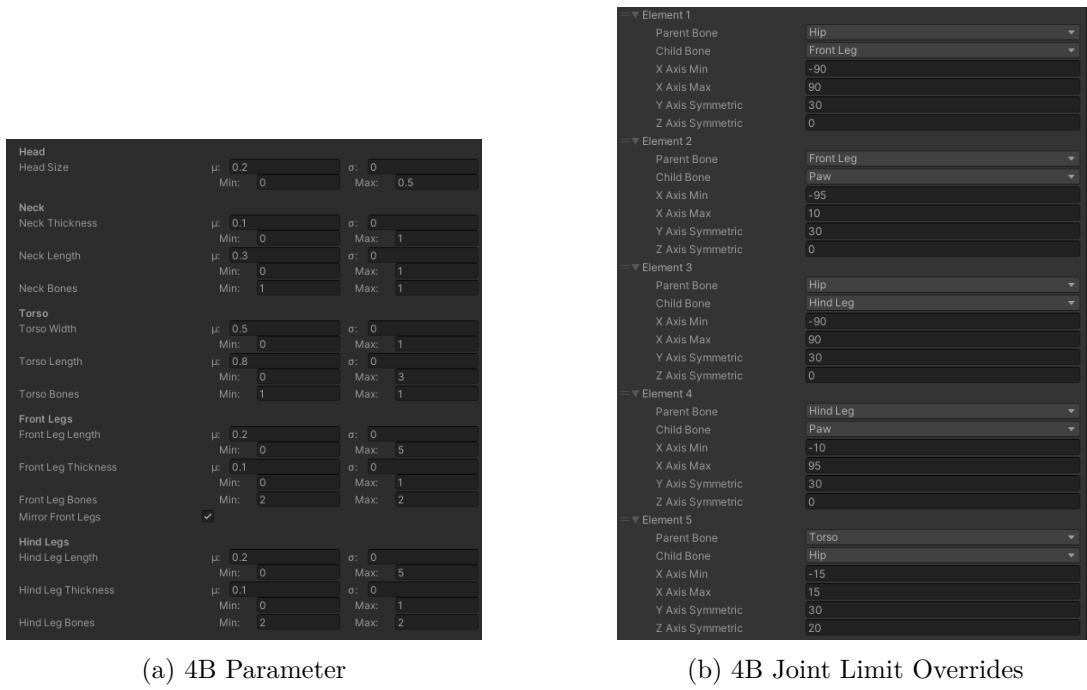
6.1.1 Auswahl der Kreaturen

Die Evaluation wird aufgrund der limitierten Rechenressourcen auf jeweils eine Vierbeiner Kreatur und eine Zweibeiner Kreatur beschränkt. Abbildung 6.1a zeigt die Parameter zur Generierung der Vierbeiner Kreatur und Abbildung 6.1b zeigt die Einstellungen zum Überschreiben der Joint Limits der generierten Vierbeiner Kreatur. Abbildung 6.2 zeigt die Parameter zur Generierung der Zweibeiner Kreatur.

Vierbeiner-Einstellungen Bei der Wahl der Parameter für das Generieren des Vierbeiners war es wichtig, dass die generierte Kreatur sowohl eine gewisse Stabilität beim Laufen als auch eine ungefähre Größe von 2m hat. Um geeignete Parameter zu finden, wurden viele Vortests durchgeführt. Bei den vielen Vortesten hat sich gezeigt, dass Vierbeiner, deren Beine nur aus zwei Elementen bestehen, stabiler laufen als Vierbeiner, deren Beine aus drei Elementen bestehen. Außerdem werden die standard Joint Limits des Generators für die Beine der Kreaturen überschrieben, da in den Vortests die Kreaturen mit den standard Joint Limits weder stabil laufen noch aufstehen konnten. Aus diesen Vortesten ergab sich eine Basisvierbeinerkreatur, deren Beine aus zwei Elementen bestehen. In weiteren Vortests wurden Variationen dieser Basisvierbeinerkreatur getestet. Bei den Variationen wurden zum Beispiel die Anzahl der Torsoknochen oder die Breite und Länge von Beinknochen verändert. Die Variationen konnten zwar alle recht stabil laufen, hatten aber Schwierigkeiten beim Aufstehen. Aufgrund der Schwierigkeiten der Variationen wurde schließlich auch die Basisvierbeinerkreatur für die finale Abgabe verwendet.

Zweibeiner-Einstellungen Da die Zweibeinerkreatuen am meisten Probleme bei der Stabilität haben, gab es keine genauen Erfahrungen zum Aufbau der Kreatur. Durch den symmetrischen Generierungsprozess und Vorüberlegungen der Kreaturen-Erststeller sollten möglichst alle generierbaren Zweibeiner im Gleichgewicht sein. Der Einstellungsprozess für die Beispielfigur reduziert sich daher auf das Reduzieren der Größe. Hierfür wurden Menschen im Raum als Vorlage genutzt, um Möglichst die Proportionen in dem

6 Evaluation



Generator nachzuahmen. Die daraus resultierende Figur bestätigte sich in mehreren Vortest und wurde deshalb auch für die finale Abgabe verwendet

6.1.2 Konfiguration der Trainingsdurchläufe

Die Erfahrung während der Entwicklungsphase der Projektgruppe hat gezeigt, dass die verwendete PPO Implementierung weitestgehend robust gegenüber der verwendeten Hyperparameter ist, solange eine ausreichend große Batchgröße verwendet wird. Aufgrund der limitierten Rechenressourcen wurde deswegen auf eine Hyperparameteroptimierung verzichtet und es werden die Hyperparameter aus der Konfigurationsdatei des Walkers von ml-Agents verwendet. Abbildung 6.3 zeigt die für die Evaluation verwendete neroRL Konfigurationsdatei. Für die Trainingsdurchläufe werden Unity Builds mit 16 Agenten verwendet, die auf 8 parallelen Workern ausgeführt werden. Dementsprechend werden die Daten parallel von 128 Agenten gesammelt. Jeder Agent führt bei jeder fünften Aktualisierung der Physikengine eine Aktion aus. Für das Training wird eine Batchgröße von 102400 verwendet, somit entsprechen 10 Updates etwa 1 Mio Schritten in der Umgebung. Die in Tabelle 5.1 beschriebenen Reward-Funktionen für das Aufstehen und Laufen werden jeweils zwei Mal für die Vierbeinige und zwei Mal für die Zweibeinige Kreatur trainiert. Dabei werden alle 100 Updates die Policies zwischengespeichert und zum Sammeln der Evaluationsdaten verwendet. Für die Evaluation werden mit den gespeicherten Policies jeweils 128 zufällige Episoden ausgeführt und dabei Reward und Episodenlänge gesammelt.

6.2 Ergebnisse

Head	
Head Size	$\mu: 0.15$ $\sigma: 0$ Min: 0.15 Max: 0.2
Neck	
Neck Thickness	$\mu: 0.1$ $\sigma: 0$ Min: 0.75 Max: 0.1
Neck Length	$\mu: 0.2$ $\sigma: 0$ Min: 0.2 Max: 0.3
Neck Bones	$\mu: 1$ $\sigma: 0$ Min: 1 Max: 1
Torso	
Torso Width	$\mu: 0.5$ $\sigma: 0$ Min: 0 Max: 0.5
Torso Length	$\mu: 0.15$ $\sigma: 0$ Min: 0.1 Max: 0.4
Torso Bones	$\mu: 2$ $\sigma: 0$ Min: 2 Max: 2
Arms	
Arm Thickness	$\mu: 0.3$ $\sigma: 0$ Min: 0.05 Max: 0.1
Arm Length	$\mu: 0.3$ $\sigma: 0$ Min: 0.25 Max: 0.35
Arm Bones	$\mu: 2$ $\sigma: 0$ Min: 2 Max: 2
Legs	
Leg Length	$\mu: 0.35$ $\sigma: 0$ Min: 0.3 Max: 0.5
Leg Thickness	$\mu: 0.2$ $\sigma: 0$ Min: 0.05 Max: 0.1
Leg Bones	$\mu: 2$ $\sigma: 0$ Min: 2 Max: 2
Feet	
Feet Width	$\mu: 0.2$ $\sigma: 0$ Min: 0.1 Max: 0.2
Feet Length	$\mu: 0.2$ $\sigma: 0$ Min: 0.4 Max: 0.4
Hands	
Hand Radius	$\mu: 0.1$ $\sigma: 0$ Min: 0.05 Max: 0.1

Abbildung 6.2: 2B Parameter

6.2 Ergebnisse

In diesem Abschnitt werden die Ergebnisse der Trainingsdurchläufe für die Vierbeiner und Zweibeiner Kreaturen beschrieben. Zunächst werden Daten bezüglich des Rewards und der Länge der Trainingsepisoden analysiert. Anschließend wird die Qualität der Trainingsergebnisse bei der visuellen Evaluation in Unity beschrieben.

6.2.1 Vierbeiner Kreatur

Laufen

Die Entwicklung des Rewards beim Trainieren des Laufens der Vierbeiner Kreatur ist in Abbildung 6.4 abgebildet. Der Reward steigt in den ersten 1000 Updates deutlich an und erreicht ein Maximum von 14000. Danach fällt der Reward langsam ab auf ca. 7000-1000, ein Policy Collapse tritt nicht ein.

Die Episodenlänge der Vierbeiner beim Laufen ist unbeschränkt, eine Episode wird beendet, wenn die Kreatur mit einem Körperteil außer den Füßen den Boden berührt. Da alle 5 Physik-Aktualisierungen der Unity Umgebung eine Aktion des Agenten angefordert wird und die Unity Umgebung mit 120 Aktualisierungen pro Sekunde ausgeführt

6 Evaluation

```

### TRAINER CONFIG ####
trainer:
    algorithm: "PPO"
    resume_at: 0
    gamma: 0.995
    lambda: 0.95
    updates: 10000
    epochs: 3
    refresh_buffer_epoch: -1
    n_mini_batches: 16
    value_coefficient: 0.5
    max_grad_norm: 0.5
    share_parameters: False
    normalize_advantage: "none" # options: "none", "minibatch", "batch"
learning_rate_schedule:
    initial: 3.0e-4
    final: 3.0e-6
    power: 1.0
    max_decay_steps: 10000
beta_schedule:
    initial: 0.005
    final: 0.005
    power: 1.0
    max_decay_steps: 10000
clip_range_schedule:
    initial: 0.2
    final: 0.2
    power: 1.0
    max_decay_steps: 10000

### EVALUATION CONFIG ####
evaluations:
    # Evaluation parameters
    sampler:
        type: "PyTorchSampler"
        n_workers: 8
        worker_steps: 1024
        batch_size: 1024 # Buffer size should be greater than batchsize*(n_workers*n_agents)
        batch_size: 102400

```

Abbildung 6.3: neroRL Konfigurationsdatei für die Evaluation

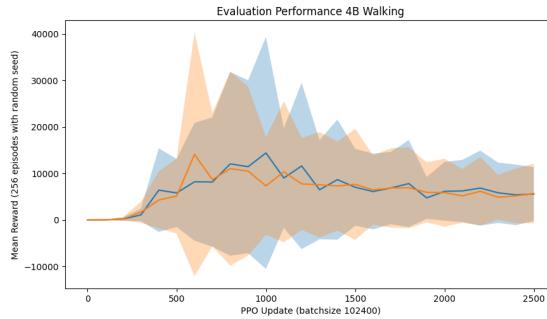


Abbildung 6.4: Walking 4B Reward

wird, entspricht eine Länge von 1000 ca. 41.5 Sekunden. Abbildung 6.5 zeigt die Entwicklung der Episodenlänge während der Trainingsdurchläufe. Analog zum Reward steigt die durchschnittliche Episodenlänge in den ersten 1000 Updates deutlich an und stagniert danach bzw. fällt langsam ab. Das Maximum wird mit einer Episodenlänge von ca. 6000, umgerechnet ca. 4 Minuten und 9 Sekunden, erreicht.

Die Visuelle Evaluation in Unity zeigt, dass die Vierbeiner Kreatur mit beliebigen Policies ab ca. 500 Updates weitestgehend stabil läuft. Instabilität tritt insbesondere dann auf, wenn die Kreatur mit hoher Geschwindigkeit einen Wegpunkt erreicht und eine große Drehung in Richtung des nächsten Wegpunkts durchführen muss.

In Abbildung 6.6 werden die 3 Schritte des Laufzyklus eines Vierbeiners dargestellt. Am Anfang des Zyklus springt der Vierbeiner ab (siehe 6.6a) und streckt während des Sprungs seine Vorderbeine nach vorne und breitet seine Hinterbeine aus (siehe 6.6b). Abschließend landet der Vierbeiner wieder auf seinen Beinen (siehe 6.6c) und bereitet sich auf den nächsten Absprung vor. Das Springen des Vierbeiners beim Laufen trägt zu der oben genannten Instabilität bei.

6.2 Ergebnisse

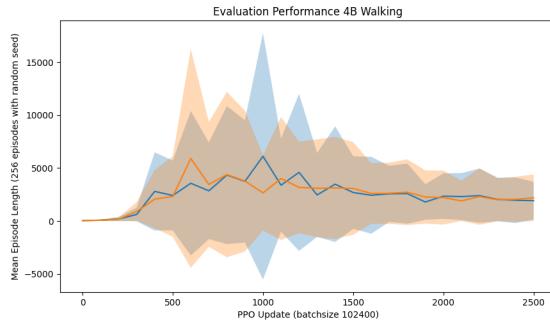
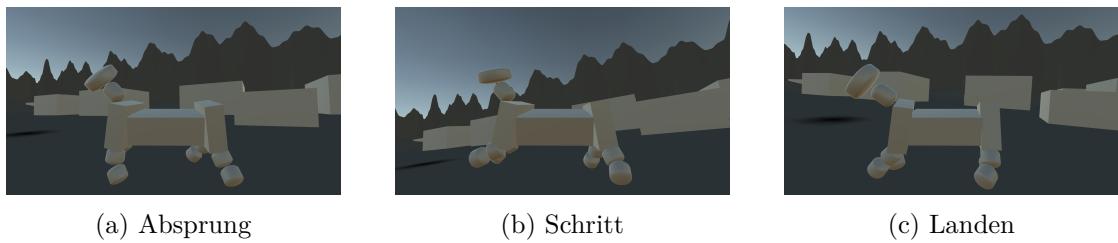


Abbildung 6.5: Walking 4B Length



(a) Absprung

(b) Schritt

(c) Landen

Abbildung 6.6: 3 Schritte im Laufzyklus eines Vierbeiners.

Aufstehen

Abbildung 6.7 zeigt die Entwicklung des Rewards während der Updates des Trainingsdurchlaufs. Der Reward steigt in den ersten 250 Updates deutlich von 0 auf ca. 1700. Das Training stagniert für mehrere hundert Updates bei ca. 1700, bevor ein Performance Collapse eintritt, von dem sich die Policy nicht erneut erholt.

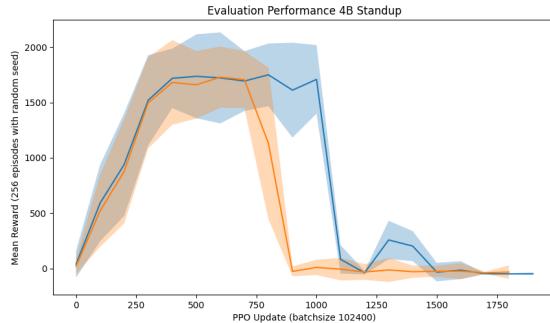


Abbildung 6.7: Standup 4B Reward

Die Episodenlänge für das Aufstehen der Vierbeiner ist auf 1000 Schritte beschränkt und es gibt keine Bedingung, die den Start einer neuen Episode vor Ablauf der Schritte bedingt. Da alle 5 Schritte eine Aktion des Agenten angefordert wird, liegen die in Abbildung 6.8 dargestellten Episodenlängen konstant bei 201 Schritten.

6 Evaluation

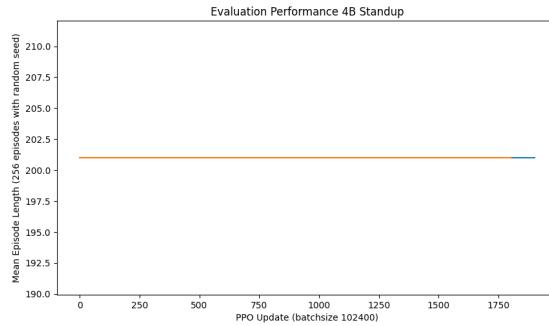


Abbildung 6.8: Standup 4B Length



Abbildung 6.9: 3 Schritte im Aufstehzyklus eines Vierbeiners.

Die Visuelle Evaluation in Unity zeigt, dass beliebige Policies aus den Update-Bereichen, in denen der Reward bei ca. 1700 stagniert, in der Lage sind den Vierbeiner erfolgreich aus einer liegenden Startposition aufzustehen zu lassen. Die Policies nach dem Performance Collapse führen nur minimal wahrnehmbare Aktionen aus und die Kreatur bleibt unbewegt liegen.

Die Abbildung 6.9 zeigt drei Schritte des Aufstehzyklus eines Vierbeiners. In der Abbildung 6.9a befindet sich die Vierbeinerkreatur in der liegenden Startposition auf dem Boden. Die Kreatur richtet sich über die Seite auf (siehe 6.9b) und bleibt abschließend relativ stabil auf den Beinen stehen (siehe 6.9c).

6.2.2 Zweibeiner Kreatur

Laufen

Abbildung 6.10 zeigt die Entwicklung des Rewards beim Trainieren des Laufens der Zweibeiner Kreatur. Der Reward steigt über die ersten ca. 1500 Updates von 0 auf ca. 3000-4000 deutlich an. Danach ist der Reward langsam weiter und schwankt dabei stark. In den Trainingsdurchläufen der Evaluation wurde ein maximaler Reward von ca. 5300 erreicht.

Die Episodenlänge der Zweibeiner beim Laufen ist nicht beschränkt. Eine Episode wird beendet, wenn die Kreatur mit einem Körperteil außer den Füßen den Boden berührt.

6.2 Ergebnisse

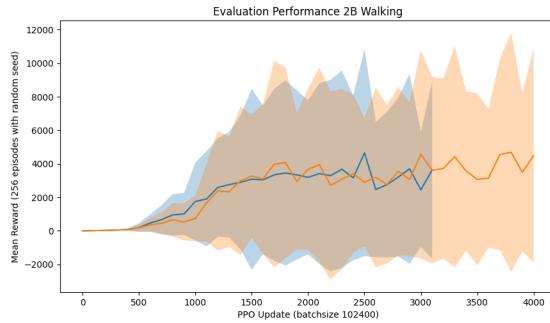


Abbildung 6.10: Walking 2B Reward

Die in Abbildung 6.11 dargestellte Episodenlänge steigt parallel zum Reward in den ersten 1500 Updates deutlich von 0 auf ca. 1000 und stagniert danach mit Schwankung. Da alle 5 Schritte eine Aktion des Agenten angefordert wird, bedeutet dies ca. 5000 Schritte. Die maximale durchschnittliche Episodenlänge beträgt ca. 2000.

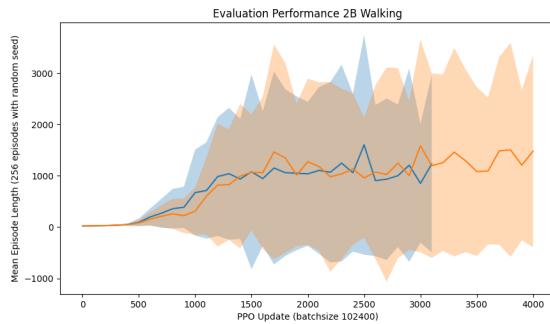


Abbildung 6.11: Walking 2B Length

Die Visuelle Evaluation in Unity zeigt, dass die Policies mit höchstem Reward den Zweibeiner laufen lassen. In Abbildung 6.12 wird ein Laufschritt auf gerade Ebene mit einer Geschwindigkeit von 5 dargestellt.

Hier ist zu erkennen, dass es sich nicht um ein Laufen wie bei einem Menschen, welcher ein Fuß auf den Boden stehen hat, den anderen nach vorne zieht und später den zweiten hinterherzieht, handelt. Sonder es eher einem Rennen gleicht, bei dem beide Beine sich in der Luft befinden. Die Belohnung für das Erreichen der Geschwindigkeit wird hierbei nicht vollständig erreicht und schwankt im Bereich zwischen 0.5 und 0.6¹. Ein weniger trainiert Netzwerk²

¹Hier wurden nur die Werte aus dem Log der Belohnungsfunktion im Editor abgelesen. Es kann sein, dass die Stichprobe eine Ausnahme abbildet, obwohl diese über mehrere Resets gleich geblieben ist. Eine genauere Untersuchung wäre als Folgearbeit angeraten.

²Zum Sichttest wurden die Netzwerke **Generated2B-2000** und **Generated2B-3000** benutzt. Das erstere ist hierbei das weniger trainierte Netzwerk.

6 Evaluation

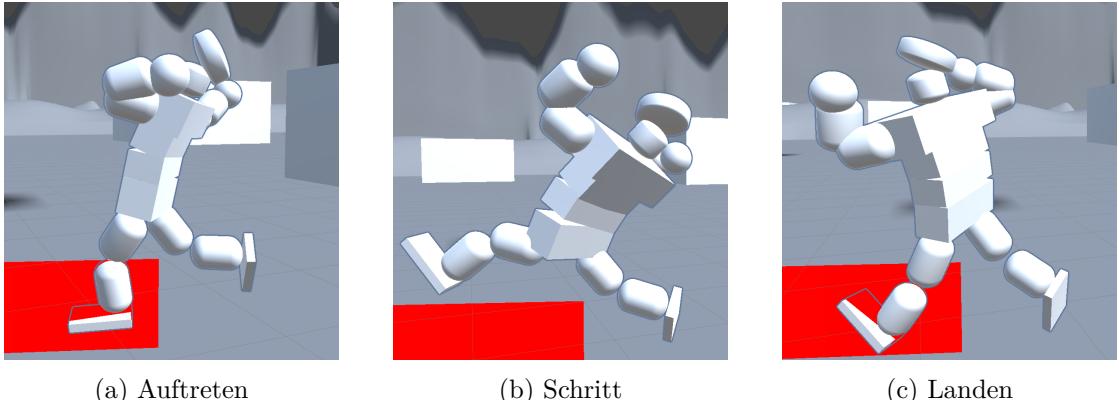


Abbildung 6.12: 3 Schritte im Laufzyklus eines Zweibeiners.

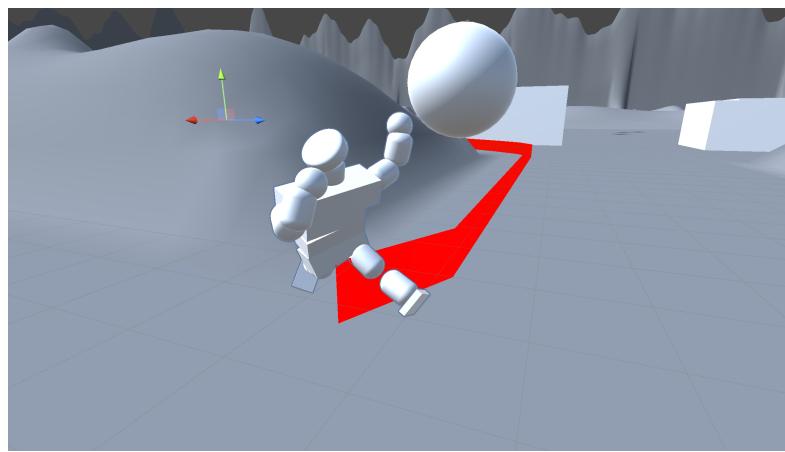


Abbildung 6.13: Die Stabilitätsprobleme eines Zweibeiners in einer Kurve. Hierbei ist die Kugel das aktuell anvisierte Ziel und die rote Linie die Kurve der weiteren Ziele.

Das Laufen ist insbesondere in Kurven instabil. In Abbildung 6.13 ist eine Kreatur in einer Kurve abgebildet. Hierbei ist die rote Linie der Pfad bestehend aus Wegpunkten zum Ziel. Der jeweils nächste Punkt ist die Kugel und bildet das Ziel des Agenten ab. Da die Figur mit einer relativ hohen Geschwindigkeit in die Kurve geht und die Stabilität des Laufens dafür nicht ausreicht, springt dieses Ziel häufig in Kurven und trägt so zu der erhöhten Instabilität in Kurven bei.

Aufstehen

Abbildung 6.14 zeigt die Entwicklung des Rewards während der Updates des Trainingsdurchlaufs. Der Reward steigt in den ersten 1500 bis 2000 Updates kontinuierlich von

6.2 Ergebnisse

0 auf ca. 1200. Danach tritt ein Performance Collapse ein, von dem sich die Policy innerhalb der weiteren berechneten 2000 Updates nur langsam erholt.

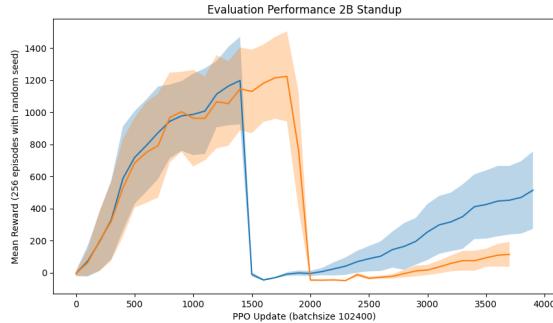


Abbildung 6.14: Standup 2B Reward

Die Episodenlänge für das Aufstehen der Zweibeiner ist auf 1000 Schritte beschränkt und es gibt keine Bedingung, die den Start einer neuen Episode vor Ablauf der Schritte fordert. Da alle 5 Schritte eine Aktion des Agenten angefordert wird, liegen die in Abbildung 6.15 dargestellten Episodenlängen konstant bei 201 Schritten.

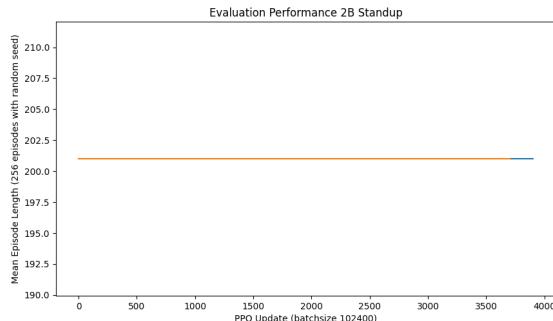


Abbildung 6.15: Standup 2B Length

Die Visuelle Evaluation in Unity zeigt, dass selbst die besten Policies (nach 1400 und 1800 Updates) nur in der Lage sind die Kreatur in einer schwungvollen Bewegung aufzustehen zu lassen. Die Policies schaffen es allerdings nicht den benötigten Schwung richtig einzuschätzen und daher tritt bei jedem Versuch einer der folgenden Fälle ein.

- Sie nimmt zu wenig Schwung und die Kreatur fällt zurück bevor sie aufrecht steht
- Sie nimmt zu viel Schwung und schafft es nicht zu stoppen, wodurch die Kreatur wieder umfällt

Selbst diese Policies erfüllen also nicht das Ziel eines stabilen Aufstehens der Kreatur. Die Policies nach dem Performance Collapse führen nur minimal wahrnehmbare Aktionen aus und die Kreatur bleibt unbewegt liegen.

6 Evaluation

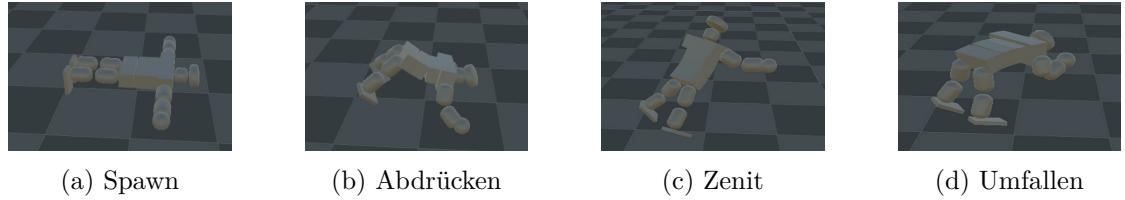


Abbildung 6.16: Aufstehen des Zweibeiners: Fall 1

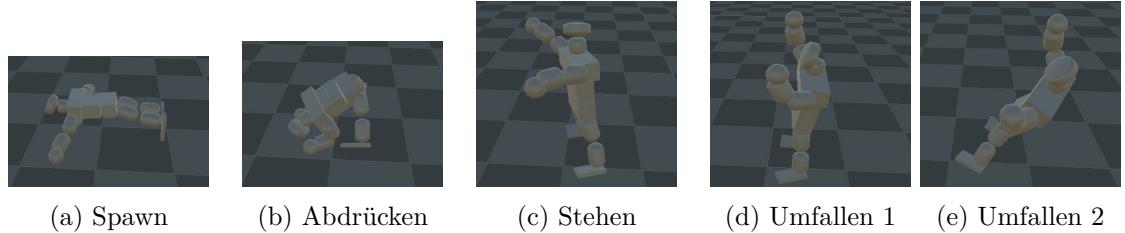


Abbildung 6.17: Aufstehen des Zweibeiners: Fall 2

6.3 Probleme

Bei der Bearbeitung der gegebenen Aufgabenstellung der Projektgruppe haben einige Probleme zu Verzögerungen geführt, welche sich auf die endgültige Funktionalität des Endprodukts ausgewirkt haben.

6.3.1 Stabilität des Skeletts

Das Hauptproblem für die Animatoren-Teilgruppe, welche Hauptsächlich mithilfe von maschinellem Training eine beliebige Kreatur zum laufen bringt, liegt in der Kreaturenstabilität. Zum Anfang der Arbeitsphase wurde die bestehende Walker-Umgebung von Unity analysiert und auf Basis der in dieser Umgebung vorhandenen Kreatur die ersten Tests erstellt. Hierbei konnte verifiziert werden, dass die neu entwickelte Trainingsumgebung funktioniert und die grundlegenden Einstellung zu funktionierenden Ergebnissen führen. Als danach die ersten prozedural generierten Kreaturen eingesetzt wurden, kam es zu einer Vielzahl von Problemen, welche durch andere Defizite in verschiedene Bereichen verstärkt wurden. In Abbildung 6.18 ist eine Kreatur aus der Anfangsphase der PG dargestellt. Diese Figur konnte nicht trainiert werden, da ihre einzelnen Teile durch die im Folgenden beschriebenen Probleme instabil waren.

6.3.2 Dokumentation

Die meisten Projektgruppenmitglieder hatten vor der PG wenig Erfahrungen mit Unity. Deshalb ist die Dokumentation von Unity eine der wichtigsten Quellen für die Um-

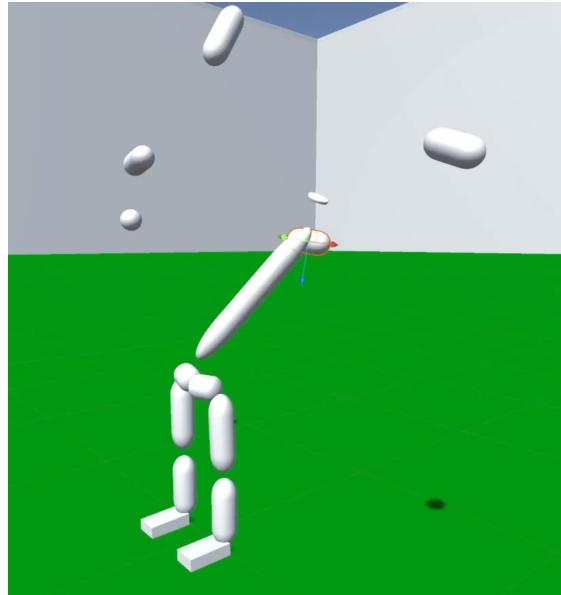


Abbildung 6.18: Beispiel einer durch fehlerhaften Einstellungen zerstörte Kreatur.

setzung der einzelnen Teilprojekte. Die Unity-Dokumentation³ ist online frei einsehbar für die verschiedenen Versionen der Grafik-Engine. Dabei ist problematisch, dass insbesondere in dem Teil zur Physik-Engine oder neueren Pakete, welche noch nicht den Vorschau-Status verlassen haben, deutliche Formulierungen fehlen. Ein Beispiel dafür ist die Hilfestellung zur Ragdoll-Stabilität. In einen Nebensatz⁴ wird erwähnt, dass ein zu großer Massenunterschied zwischen zwei direkt verbundenen Elementen zu unruhigen Ragdolls führen kann. In der Praxis bedeutet dies, dass die generierten Kreaturen bei jegliche Krafteinwirkung explodieren. Eine Fehler-findung und -behebung dieses Problems hat mehrere Wochen gedauert, da die Auswirkungen nur in sehr abgeschwächter Form beschrieben wurden.

Ein weiteres Beispiel ist der *Solver Type*⁵, welcher auf *Projected Gauss Seidel* oder *Temporal Gauss Seidel* gesetzt werden kann. Für das Training wurde in der Anfangsphase versucht jeweils die besten Einstellungen von Unity zu verwenden, was nach Dokumentation die zweitere Option sein sollte. In Gegensatz zu der Dokumentation warnen mehrere Internetquellen⁶ vor dieser Einstellung. Ein nicht repräsentativer Test hat dies für unsere Kreaturen bestätigt, weshalb im weiteren Verlauf *Projected Gauss Seidel* genutzt wurde. Ein Hinweis, dass *Temporal Gauss Seidel* problematische Ergebnisse produzieren kann, fehlt zum Zeitpunkt des Projektgruppenberichts weiterhin in der Dokumentation.

³<https://docs.unity3d.com/Manual/index.html>

⁴Zu finden auf dieser Unterseite der Dokumentation <https://docs.unity3d.com/Manual/RagdollStability.html>

⁵Die Komponente der Physikumgebung, welche die Berechnungen für die Kollisionserkennung durchführt.

⁶Siehe beispielsweise das zum PG-Zeitpunkt erste Google-Ergebnis

6 Evaluation

Insgesamt gab es weitere Beispiele, wie zum Beispiel bei dem `com.unity.ai.navigation`-Paket⁷ bei welchem die unterstützten Unity-Versionen unklar ist, welche zusammen zu viel Recherchearbeit geführt haben und deshalb die Bearbeitung der Kernaufgaben verzögert haben.

6.3.3 Organisatorische Probleme

Ein weiterer Teilbereich, der zu Verzögerung in dem Arbeitsablauf der Animatoren-Teilgruppe geführt hat, ist organisatorischen Problemen zuzuschreiben. Einige der größten Hindernisse sind die starke Abhängigkeit von den Animatoren und Generatoren, veraltete Rechenhardware und fehlende Vorkenntnisse.

Das erste Problem kann wie folgt beschrieben werden. Immer wenn die Änderung an der Kreatur nötig waren, musste das für die Generierung verantwortliche Paket angepasst werden. Inklusive der Kommunikation und der dafür benötigten Arbeitszeit dauerte dies ungefähr eine Woche. In diesen Phasen konnte das Training häufig nicht fortgesetzt werden, da die Kreaturen zu große Fehler hatten. In die andere Richtung konnten die Generatoren nicht weiterarbeiten, da diese auf Feedback von den Trainingsversuchen gewartet haben.

Verstärkt wurde dies durch das zweite Problem. Da LIDO von der ganzen Universität genutzt wird, kann es einige Stunden bis Tage dauern, bis eine Aufgabe abgearbeitet wird. Inklusive der Berechnungszeit des Auftrags konnten so 2 aufeinanderfolgende Experimente gestartet werden je Woche. Da insbesondere in der Mitte der Projektgruppe die Fehler nicht bekannt waren, dauerte das Finden dieser dadurch besonders lange. Zusätzlich zu der Wartezeit ist die Rechendauer eines Auftrags auf LIDO für Studenten eingeschränkt. Ein Knoten mit Grafikkarte kann 2 Tage lang reserviert werden. Bei den finalen Training auf einen Knoten des Lehrstuhls stellt sich heraus, dass diese Zeit nicht ausreichend ist, um das (lokale) Maximum der Belohnungsfunktion zu erreichen. Theoretisch wäre ein Fortsetzen der Trainingsaufgabe möglich, führt aber zu einer weiteren Wartezeit auf einen Knoten und Verfälschungen der Trainingsergebnisse durch nicht perfekt zurückgesetzten Parameter des Trainings. Des Weiteren stellte sich heraus, dass das Training auf den alten Knoten signifikant länger dauert, als auf den aktuell ausgestatteten Rechenknoten des Lehrstuhls. Ein Trainingsschritt auf den öffentlichen Knoten dauert um die 160sec und auf den Lehrstuhlknoten 60sec. Dies führt zu einer 3-Fachen Wartedauer während der meisten Experimente.

Zuletzt fehlten insbesondere bei dem maschinellen Lernen viele Vorkenntnisse. Das zu Beginn der PG gehaltene Seminar beschäftigte sich mit den eigentlich genutzten Algorithmen, hat aber keine Übersicht über die Forschung im Bereich des physikalischen Laufens gegeben. Hier wurde später [10] genutzt, welches aufgrund des Erscheinungsjahrs kein Überblick für Netzwerkbasierte-Lernmethoden gibt. Eine Einschätzung von weiteren Papieren war dadurch erschwert. Weiterhin beziehen sich viele Arbeiten auf andere

⁷Eine Dokumentation ist hier zu finden

6.4 Diskussion

Physikumgebungen, nutzen Imitation zum Lernen oder nutzen explizite Designcharakteristiken der Kreaturen aus[26].

Insgesamt führten die Probleme häufig zu Arbeitsphasen in den die Animatorengruppe oder Generatoren keine neuen Ergebnisse produzieren konnten. In dieser Zeit wurde versucht zukünftige Aufgaben wie beispielsweise eine Generalisierung der Belohnungsfunktionen, eine stärkere anpassbare Trainingsumgebung oder Zusatzfunktionen wie ein unebener Boden zu implementieren. Durch die grundlegenden Probleme bei der Stabilität konnte am Ende keine dieser Erweiterungen fertiggestellt werden.

6.4 Diskussion

Dieser Abschnitt diskutiert die zuvor beschriebenen Ergebnisse der Projektgruppe im Bezug zur initialen Zielsetzung 1.2.

Die Generierung von Spiellevels Die Ziele bezüglich der Generierung von Spiellevels wurden erfolgreich umgesetzt. Das Layout des Spiellevels wird prozedural mittels Space-Partitioning generiert und anschließend durch Terrain Transformationen in eine natürlich wirkende Welt verwandelt. Die Welt besteht aus Räumen und Korridoren, die durch Gebirge voneinander getrennt werden. Anders als bei einer typischen Generierung von Dungeons mittels Space-Partitioning, wird hierbei auf die Decke des Dungeons verzichtet, sodass die Welt aus offenen Arenen besteht, die durch Korridore miteinander verbunden sind. In der Welt werden Spawn-Punkte für den Spieler, Kreaturen und Hindernisse platziert. Die Hindernisse bestehen aus Pflanzen, die zur Laufzeit aus L-Systemen erzeugt werden. Allgemein ist es möglich die Größe der Welt, die Anzahl an Räumen und die Anzahl an Spawn-Punkten für Hindernisse und Kreaturen einzustellen. Der Schwierigkeitsgrad lässt sich durch kleinere Räume mit mehr Spawn-Punkten erhöhen.

Generierung der Monster Von Anfang an wurden bei der Generierung der Monster viele methodische Freiheiten gelassen. Damit konnten Beschränkungen bezüglich der Körpераusprägungen vermieden werden. Während der Ausarbeitung sind dann die Zwei- und Vierbeiner in den Vordergrund gerückt, da diese auch für das Training bekannte und einfach erweiterbare Ansätze ermöglicht haben. Gleichzeitig ist schnell ersichtlich gewesen, dass die Auswahl der Körperteile eingeschränkt werden sollte, sodass die Komplexität der Kreaturen auf ein Minimum beschränkt wird und möglichst wenig Freiheitsgrade für das Training und Animieren der Kreaturen notwendig sind. Trotzdem ist es möglich, die entstandene parametrische Methode von Jona Heinrichs so zu erweitern, dass mit möglichst wenig Aufwand, Kreaturen mit einem noch größer variierenden Körperbau (z.B. Flügel, noch mehr Gliedmaßen, Acht-Füßler, etc.) entstehen können. Bei der parametrischen Methode ist es weiterhin wichtig, dass die Wertebereiche der Parameter

6 Evaluation

passend zu der erwartenden Anatomie eines Zwei- und/oder Vierbeiners gewählt werden und dieser sich physikalisch korrekt animieren lässt.

Somit sind die Ziele, dass die Kreaturen-Erstellung Algorithmen-basiert ist und die innerhalb von Unity bereitgestellten Frameworks (z.B. Joints) verwendet werden sollten, erreicht worden. Für das Erreichen dieser Ziele, wurde sich ebenfalls so wie Anfangs angeschnitten, erfolgreich an dem Unity-ML-Agents-Walker orientiert.

Die folgenden Darstellungen in 6.19 & 6.20 repräsentieren das fertige Ergebnis der Creature-Generator.

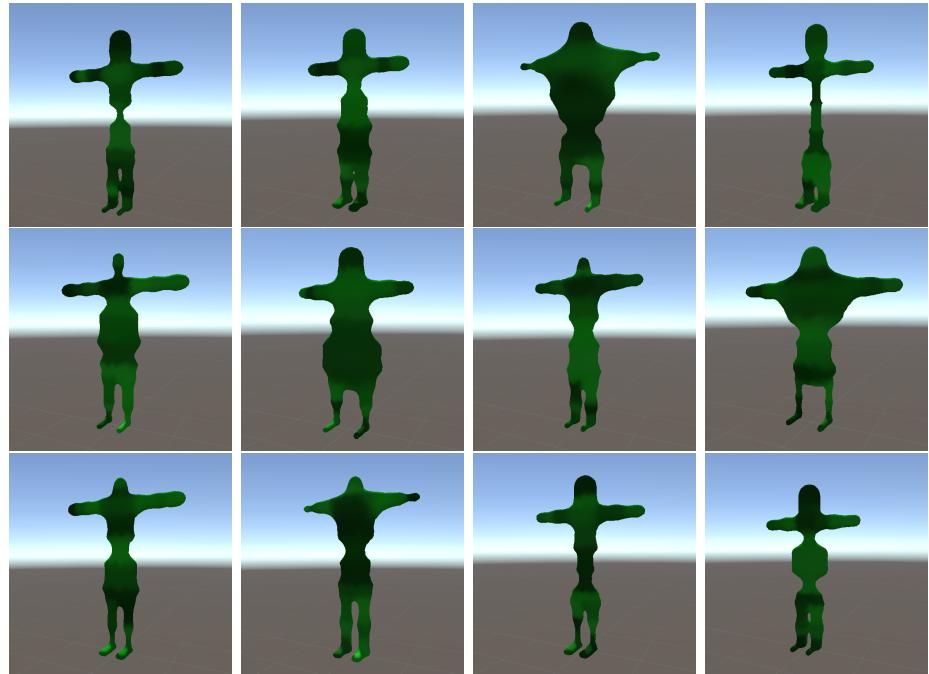


Abbildung 6.19: Der fertige Creature-Generator Stand: Beispiele für Zweibeiner

Fortbewegung der Monster

Das festgelegte Minimalziel für die Fortbewegung der Monster war, dass die Animationen nicht manuell erstellt werden sollen. Stattdessen sollte mit Deep Reinforcement Learning ein Agent trainiert werden, der lernt die Monster zu bewegen, indem er Kräfte auf deren Joints ausübt. Bei dem Versuch die ursprünglichen Vorstellung der Fortbewegung der Kreaturen umzusetzen traten allerdings einige Probleme auf.

Zum einen war das Trainieren der Fortbewegung von vollständig zufällig erstellten Kreaturen nicht möglich. In den ersten Trainingsdurchläufen wurde schnell klar, dass einige

6.4 Diskussion

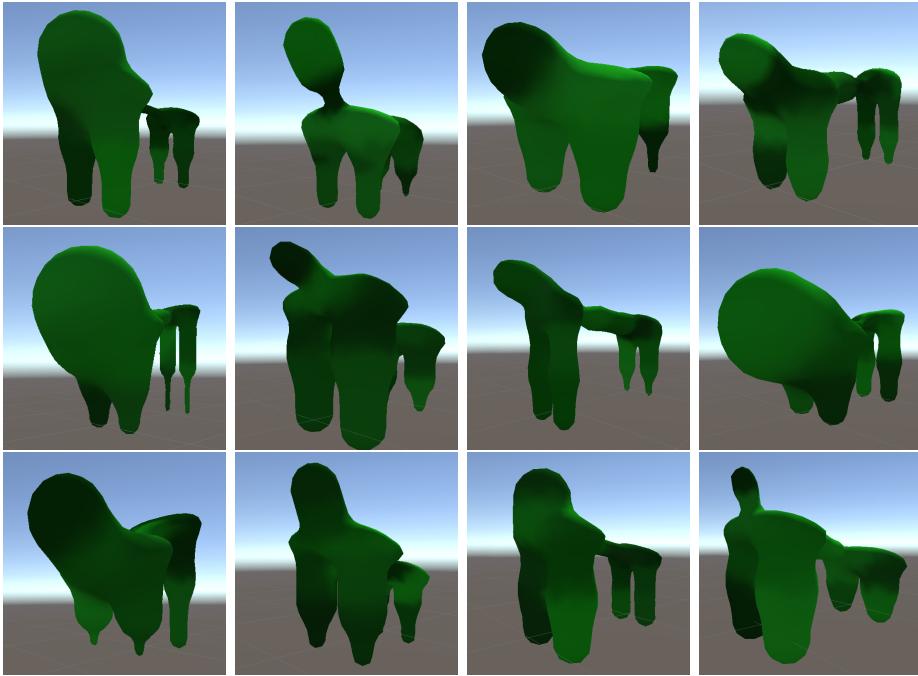


Abbildung 6.20: Der fertige Creature-Generator Stand: Beispiele für Vierbeiner

Kreaturen durch die Struktur ihres Körpers nicht dazu in der Lage waren sich stabil zu bewegen oder aufzustehen. Ein Grund dafür kann zum Beispiel sein, dass die Bewegung einiger Joints zu weit eingeschränkt sind und die Agenten deswegen die mit diesen Joints verbundenen Körperteile nicht auf eine Art bewegen können, welche eine stabile Fortbewegung ermöglicht. Das selbe Problem trat auf, wenn eine Kreatur am Boden lag und im Zweifel ihre Arme und Beine nicht genug Freiheit hatten, um aus dieser Position herauszukommen. Das entgegengesetzte Problem konnte auch auftreten, wenn die Freiheitsgrade zu hoch waren, da die Bewegungen dann nichts mehr mit der Fortbewegung von realen Tieren gemein hatten. Es war also notwendig die Parameter zur Generierung der Kreaturen einzuschränken, damit Fortbewegung möglich war.

Ein anderes Problem war die Stabilität der Fortbewegung. Die vierbeinige Kreatur, deren Trainingsergebnisse in dem Abschnitt 6.2 vorgestellt werden, ist nach dem Training in der Lage stabil zu laufen und auch wieder aufzustehen und kann deswegen in dem Spiel verwendet werden. Dies war allerdings nicht für alle generierten Vierbeiner der Fall, selbst wenn die zuvor erwähnten Einschränkungen an den Generator übergeben werden. Nach dem Training konnten zwar fast alle Vierbeiner laufen, aber in der Stabilität gab es große Variationen und einige der Kreaturen konnten nicht lernen aufzustehen. Aus diesem Grund wurde nur das eine vorgestellte Vierbeiner Modell dem Spiel hinzugefügt. Die zweibeinige Kreatur ist nach dem absolvierten Training zwar in der Lage zu Laufen, aber dies ist insbesondere in Kurven sehr instabil. Dieses kann allerdings auch der

6 Evaluation

gewählten Rewardfunktion geschuldet sein, da diese für das gesamte Training eine Geschwindigkeit vorgibt, welche der Agent einhalten sollte. Insbesondere in engen Kurven ist es für einen Zweibeiner aber aus Stabilitätsgründen effizient seine Geschwindigkeit zu verringern, wofür die Funktion den Agenten bestrafen würde. Das Problem mit der Stabilität ließe sich also eventuell entweder durch eine andere Rewardfunktion, die in Kurven langsamere Geschwindigkeiten erlaubt, oder eine andere Nav-Mesh Implementierung, die keine engen Kurven macht, lösen. Im Rahmen der Projektgruppe war allerdings keine Zeit mehr diese Vermutungen zu überprüfen.

Die Zweibeiner hatten außerdem Probleme mit dem aufstehen. Nach dem Training konnte die Kreatur zwar aus dem liegen wieder in den Stand kommen, aber sie blieb danach nicht stehen, sondern fiel wieder zu Boden. Diese Kombination aus instabilem Laufen mit häufigen umfallen in Kurven und der Unfähigkeit aufzustehen macht die Zweibeiner ungeeignet für eine Verwendung in dem Spiel. Daher wurde entschieden keine zweibeinigen Monster in das Spiel zu integrieren.

Abschließend lässt sich also sagen, dass das zu Beginn festgelegte Minimalziel erreicht wurde und für die exemplarischen Kreaturen Modelle trainiert wurden, dass diese zum laufen und aufstehen verwenden können. Außerdem wurde ein einfacher Mechanismus implementiert, der Anhand einer übergebenen Bedingung entscheidet, ob die Kreatur im nächsten Schritt das Modell zum laufen und das zum aufstehen verwenden soll.

Die Umsetzung der weiterführenden Ideen ist allerdings zum größten Teil gescheitert. Insbesondere die Ansätze zur Diversifikation der Generierung, ein Agent der allen Kreaturen beibringen kann zu laufen, und zur Generalisierung des Trainings, ein Modell welches von allen Kreaturen mit ähnlichen Skeletten zum laufen verwendet wird, konnten nicht umgesetzt werden.

7 Fazit & Ausblick

Innerhalb der Projektgruppe wurden anfangs Pläne und Ziele angeführt, welche während der Arbeit allgegenwärtig waren und auf welchen die Umsetzung und Entwicklung auf- und ausgebaut wurde. Zwei Semester, in welchen über die Projektgruppe hinaus viel Zeit der Studierenden für andere Aspekte des Studiums hingegeben wurde, reichen nicht aus um einen fertigen Prototypen und erst recht nicht ein fertiges Spiel zu implementieren. Es mussten wie in der Evaluation diskutiert wurde daher mehrere Inhalte gekürzt werden. Zuletzt waren es die vierbeinigen Kreaturen welche erfolgreich die Skills des Aufstehens und des Laufens gelernt haben, wobei die Zweibeiner Inkonsistenzen aufgewiesen haben. Bestrebungen sollten von Anfang an realistisch gehalten werden, um zeitlich und inhaltlich nicht über die tatsächlichen Gegebenheiten hinaus zu laufen und damit die anfangs angeführten Pläne mitten in der Entwicklungsphase nochmal überarbeiten zu müssen. An dieser Stelle ergibt sich, dass die ursprünglichen Ziele in dieser Arbeit zu größten Teilen eingehalten wurden und vor allem relevant für den Entwicklungsprozess der Projektgruppe waren.

Insgesamt könnten weitere Maßnahmen ergriffen werden, um über die Entwicklung eines Prototypen hinaus zu einem ausgefeiltem Spiel zu gehen.

In der Einleitung wurde angemerkt, dass eine Generalisierung der erzeugten Kreaturen mit mehreren verschiedenen Körpераusprägungen wünschenswert wäre. Während der Ausarbeitung dieses Forschungsprojektes, wurde jedoch aufgrund des Zeitmangels und der Limitationen der damit verbundenen Animationen und des Frameworks die Generierung der Kreaturen auf simple 2- und 4-Beiner beschränkt. Eine, wie eigentlich geplante Generalisierung der Erzeugung und des Trainings der Kreaturen würde weitreichende Vorteile mit sich bringen; zu Anfang müsste ein längeres Training auf großen Netzwerken ausgeführt werden und es somit ermöglichen ein Metalevel bzw. einen Standard zu setzen. Dieser Standard könnte einheitlich für eine große Menge an Kreaturen mit einer großen Variation an Körpermerkmalen genutzt werden, indem jede Kreatur, die innerhalb des durch das Training vorgegebenen Frameworks erzeugt werden würde, direkt durch dieses zur Laufzeit animiert werden könnte. Somit könnte also die Generation von Kreaturen mit vielfältigen Körpermerkmalen zur Laufzeit behandelt werden. Um diese jedoch erst gewährleisten zu können, müsste die in dieser Arbeit angeführte parametrische Erzeugung der Kreaturen ebenfalls weiter ausgebaut werden, um eine größere Palette von Körpераusprägungen für Kreaturen zu ermöglichen.

Bei der Animation könnten weiterhin über die in dieser Arbeit vorgestellten Skills des Aufstehens und Laufens weitere Fähigkeiten von Kreaturen berücksichtigt und erlernt

7 Fazit & Ausblick

werden. Dabei ist jedoch nicht nur das Training, sondern darüber hinaus auch die Kombination der verschiedenen Skills von Kreaturen eine relevante Herausforderung, welche sich dann wiederum durch verschiedene Körperausprägungen (wie z.B. Flügel, Anzahl der Gliedmaßen, etc.) von Kreatur zu Kreatur unterscheiden könnten. Hier könnte vor allem in das Gebiet des Reward-Function-Engineering tiefer eingegangen werden.

Des Weiteren könnte um über den Umfang der Projektgruppe hinauszugehen, graphische Aspekte deutlicher berücksichtigt werden. Dabei könnten beispielsweise Ray-Tracing zur Laufzeit oder eine Echtzeit-Graphik-Pipeline angewendet werden, um es auf ein heutiges Videospiel-Niveau zu heben. Somit würde ebenfalls das Thema des Texture-Mappings und des Parametrisierens von Texturen auf Körper betrachtet werden.

Themenallokation

Zusammenfassung: Thomas

1. Einleitung: Thomas
2. Grundlagen
 - 2.1. L-System: Kay
 - 2.2. Dungeon Generierung mittels Space Partitioning: Tom
 - 2.3. Metaball:
 - 2.4. Reinforcement Learning: Jannik
 - 2.5. Cellular Automata: Markus
 - 2.6. Unity Features:
3. Verwandte Arbeiten:
4. Projektorganisation: Carsten, Thomas, Jannik, Leonard
 - 4.1. Creature Generator
 - 4.2. Creature Animator
5. Umsetzung
 - 5.1. Creature Generation: Markus
 - 5.1.1. Fachliche Umsetzung
 - 5.1.1.1. Parametrische Kreatur:
 - 5.1.1.2. L-System Kreatur: Tom, Kay
 - 5.1.1.3. Mesh Generation: Leonard
 - 5.1.1.4. Automatisches Rigging: Leonard
 - 5.1.1.5. Skinning: Leonard
 - 5.1.1.6. Metaballs: Jona
 - 5.1.1.7. Jonas Creature Generation Method: Jona, Markus
 - 5.1.2. Technische Umsetzung

7 Fazit & Ausblick

- 5.1.2.1. Unity-Package: Markus
 - 5.1.2.2. Konfiguration: Markus
 - 5.1.2.3. Bone-Definition: Markus
 - 5.1.2.4. Parametrische Generatoren: Markus
 - 5.1.2.5. Skeleton-Definition: Markus
 - 5.1.2.6. Skeleton-Assembler: Markus
 - 5.1.2.7. Mesh-Generator:
 - 5.1.2.8. Creature Generator: Markus
 - 5.2. Creature Animation: Nils, Carsten, Jan
 - 5.2.1. Fachliche Umsetzung
 - 5.2.1.1. Trainingsumgebung: Carsten
 - 5.2.2. Technische Umsetzung
 - 5.2.2.1. Trainingsumgebung: Nils
 - 5.2.2.2. Erweiterung der Agent Klasse: Jan
 - 5.2.2.3. Konkrete Implementation des Agents:
 - 5.2.2.4. RL-Framework: Niklas
 - 5.2.2.5. LiDO3: Nils
 - 5.3. World Generation: Kay, Tom
 - 5.3.1. Generierung von Levels mit Space Partitioning: Tom
 - 5.3.2. Levels: Tom
 - 5.3.3. Generierung des Terrains: Tom
 - 5.3.4. Terrain-Transformationen: Kay
 - 5.3.5. L-System Vegetation: Kay
 - 5.4. Spiel: Markus
- 6. Evaluation
 - 6.1. Einschränkungen: Jannik, Nils, Niklas
 - 6.2. Ergebnisse: Nils, Jannik, Carsten, Niklas
 - 6.3. Diskussion: Niklas, Kay, Leonard, Jonas, Markus, Thomas
 - 6.4. Probleme: Nils

7. Fazit & Ausblick: Thomas

Abbildungsverzeichnis

2.1	Blumenkohl	5
2.2	L-System 2D Rotation. Der rote Pfeil zeigt in die Richtung der Turtle.	6
2.3	Darstellung der ersten drei Ableitungsschritte eines Bracketed L-Systems	7
2.4	Drei Resultate desselben stochastischen L-Systems	8
2.5	L-System 3D Baum	9
2.6	Space Partitioning mit k -d-Baum	11
2.7	Space Partitioning mit Quadtrees	12
2.8	Prozedural generierter Dungeon mit k -d-Baum	12
2.9	2D Beispiel eines Metaballs bestehend aus drei Kugeln	13
2.10	Reinforcement Learning Kontrollfluss [12]	14
2.11	Prozedural generierte Umgebung mit NavMesh (in blau)	20
2.12	Auszug aus den Einstellungen für die NavMeshSurface-Komponente.	21
4.1	Stand des (fertigen) Status-Quo am 26.02.2023, Teil der Creature-Generator	30
4.2	Stand des (fertigen) Status-Quo am 26.02.2023, Teil der Creature-Animator	30
4.3	Stand des (fertigen) Status-Quo am 26.02.2023, Teil der World-Generator	30
5.1	Mittels L-System generiertes Skelett	36
5.2	Beispiel Metakapsel; Die gestrichelte Linie enthält alle Punkte mit $r = R$	38
5.3	Beispiel für ein korrekt eingebettetes Skelett in einem Mesh [3].	41
5.4	Temperatur-Gleichgewicht für zwei Knochen [3].	42
5.5	Berechnung der α -Winkel für eine innere Kante [6].	43
5.6	Vergleich von Linear Blend Skinning (links) und Dual Quaternion Skinning (rechts) [21].	45
5.7	Beispiel der generierten Trainingsumgebung	51
5.8	Konfigurationsmöglichkeiten des <code>DynamicEnviormentGenerator</code>	53
5.9	Konfigurationsmöglichkeiten des <code>AdvancedEnviormentGenerator</code>	54
5.10	UML-Diagramm der <code>GenericAgent</code> -Klasse und weitere relevante Klassen	55
5.11	Benötigte Sekunden zum Sammeln eines Batches der Größe 65536, LiDo3 cgpu01 Knoten, 20 Kerne, 48GB Ram, 2 GPUs, Übersicht nach 5-6 Updates	63
5.12	Verbinden von anliegenden Korridoren in Levels.	69
5.13	Platzierung eines Objektes	71
5.14	Masken	72
5.15	Operationen zur Modifikation der Heightmap des Terrains	74
5.16	Laufzeit Terrain-Transformationen	78
5.17	Bäume die durch L-Systeme erzeugt werden.	79

Abbildungsverzeichnis

5.18	Testszene für die Vegetationsmessungen	79
5.19	Messung von Vegetationsoptimierungen	80
6.2	2B Parameter	89
6.3	neroRL Konfigurationsdatei für die Evaluation	90
6.4	Walking 4B Reward	90
6.5	Walking 4B Length	91
6.6	3 Schritte im Laufzyklus eines Vierbeiners.	91
6.7	Standup 4B Reward	91
6.8	Standup 4B Length	92
6.9	3 Schritte im Aufstehzyklus eines Vierbeiners.	92
6.10	Walking 2B Reward	93
6.11	Walking 2B Length	93
6.12	3 Schritte im Laufzyklus eines Zweibeiners.	94
6.13	Zweibeiner in einer Kurve	94
6.14	Standup 2B Reward	95
6.15	Standup 2B Length	95
6.16	Aufstehen des Zweibeiners: Fall 1	96
6.17	Aufstehen des Zweibeiners: Fall 2	96
6.18	Beispiel explodierende Kreatur	97
6.19	Der fertige Creature-Generator Stand: Beispiele für Zweibeiner	100
6.20	Der fertige Creature-Generator Stand: Beispiele für Vierbeiner	101

Tabellenverzeichnis

4.1	Primäre Gruppenaufteilung der Projektgruppe	28
4.2	Die zwei Untergruppen und ihre Mitglieder	32
5.1	Varianten des AgentNavMesh mit Reward-Funktionen	58
5.2	Steuerung des Spiels	84

Algorithmenverzeichnis

1	L-System Auswertung	10
2	PPO Pseudocode Algorithmus [12]	19

Literaturverzeichnis

- [1] ML-Agents. mlAgents: Beispiel Umgebungen. 2022. URL: <https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Learning-Environment-Examples.md> (besucht am 15.03.2023).
- [2] ML-Agents. mlAgents: Documentation. 2022. URL: <https://github.com/Unity-Technologies/ml-agents/tree/main/docs> (besucht am 15.03.2023).
- [3] I. Baran und J. Popović. Automatic Rigging and Animation of 3D Characters. In *ACM Transactions on Graphics*, Band 26 der Reihe Nummer 3, Seiten 72–80. Association for Computing Machinery, 2007.
- [4] Blender Online Community. *Blender - a 3D modelling and rendering package*. Blender Foundation. Stichting Blender Foundation, Amsterdam, 2018. URL: <http://www.blender.org> (besucht am 15.03.2023).
- [5] J. F. Blinn. A Generalization of Algebraic Surface Drawing. *ACM Trans. Graph.*, 1(3):235–256, Juli 1982. ISSN: 0730-0301. DOI: 10.1145/357306.357310. URL: <https://doi.org/10.1145/357306.357310>.
- [6] A. Bobenko und B. Springborn. A Discrete Laplace-Beltrami Operator for Simplicial Surfaces. *Discrete & Computational Geometry*, 38:740–756, 2007.
- [7] D. Boris. Sur la sphere vid. *Izvestia Akademii Nauk SSSR*:793–800, 1934.
- [8] M. Botsch, D. Bommes und L. Kobbelt. Efficient Linear System Solvers for Mesh Processing. In *Proceedings of the 11th IMA International Conference on Mathematics of Surfaces*, Seiten 62–83, Berlin, Heidelberg. Springer-Verlag, 2005.
- [9] H. Dong, Z. Ding, S. Zhang, H. Yuan, H. Zhang, J. Zhang, Y. Huang, T. Yu, H. Zhang und R. Huang. *Deep Reinforcement Learning: Fundamentals, Research, and Applications*. S. Z. Hao Dong Zihan Ding, Herausgeber. Springer Nature, 2020. <http://www.deeprinforcementlearningbook.org>.
- [10] T. Geijtenbeek und N. Pronost. Interactive Character Animation Using Simulated Physics: A State-of-the-Art Review. *Computer Graphics Forum*, 31(8):2492–2515, 2012. DOI: <https://doi.org/10.1111/j.1467-8659.2012.03189.x>. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2012.03189.x>. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2012.03189.x>.

Literaturverzeichnis

- [11] R. Goldman, S. Schaefer und T. Ju. Turtle geometry in computer graphics and computer-aided design. *Computer-Aided Design*, 36(14):1471–1482, 2004. ISSN: 0010-4485. DOI: <https://doi.org/10.1016/j.cad.2003.10.005>. URL: <https://www.sciencedirect.com/science/article/pii/S0010448504000521>. CAD Education.
- [12] L. Graesser und W. Keng. *Foundations of Deep Reinforcement Learning: Theory and Practice in Python*. Addison-Wesley Data & Analytics Series. Pearson Education, 2019. ISBN: 9780135172483.
- [13] G. Guennebaud, B. Jacob u. a. Eigen v3, 2010. URL: <http://eigen.tuxfamily.org> (besucht am 15.03.2023).
- [14] J. Ho und S. Ermon. Generative Adversarial Imitation Learning. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon und R. Garnett, Herausgeber, *Advances in Neural Information Processing Systems*, Band 29. Curran Associates, Inc., 2016. URL: <https://proceedings.neurips.cc/paper/2016/file/cc7e2b878868cbae992d1fb743995d8f-Paper.pdf>.
- [15] L. Holmqvist und E. Ahlström. Comparing Traditional Key Frame Animation Approach and Hybrid Animation Approach of Humanoid Characters, 2017.
- [16] S. Huang, R. F. J. Dossa, A. Raffin, A. Kanervisto und W. Wang. The 37 Implementation Details of Proximal Policy Optimization. In *ICLR Blog Track*, 2022. URL: <https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/>. <https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/>.
- [17] J. A. Hudson. Creature Generation using Genetic Algorithms and Auto-Rigging. 2013. URL: <https://nccastaff.bournemouth.ac.uk/jmacey/OldWeb/MastersProjects/MSc13/06/Jon%20Hudson%20Thesis.pdf> (besucht am 15.03.2023).
- [18] IT & Medien Centrum. LiDO3 First Contact. 8. Nov. 2022. URL: https://www.lido.tu-dortmund.de/cms/de/LiD03/LiD03_first_contact_handout.pdf (besucht am 07.03.2023).
- [19] M. Janno. Procedural Generation of 2D Creatures. In 2018.
- [20] A. Juliani, V.-P. Berges, E. Teng, A. Cohen, J. Harper, C. Elion, C. Goy, Y. Gao, H. Henry, M. Mattar und D. Lange. Unity: A general platform for intelligent agents. *arXiv preprint arXiv:1809.02627*, 2020.
- [21] L. Kavan, S. Collins, J. Zara und C. O’Sullivan. Geometric Skinning with Approximate Dual Quaternion Blending. *ACM Trans. Graph.*, 27(4):105, 2008.
- [22] A. Kwiatkowski, E. Alvarado, V. Kalogeiton, K. Liu, J. Pettre, M. Panne und M.-P. Cani. A Survey on Reinforcement Learning Methods in Character Animation. *Computer Graphics Forum*, 41:613–639, Mai 2022. DOI: 10.1111/cgf.14504.

- [23] W. E. Lorensen und H. E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, Aug. 1987. ISSN: 0097-8930. DOI: 10.1145/37402.37422. URL: <https://doi.org/10.1145/37402.37422>.
- [24] M. Morales. Grokking Deep Reinforcement Learning, 2020. M. P. Co., Herausgeber.
- [25] L. Mourot, L. Hoyet, F. Le Clerc, F. Schnitzler und P. Hellier. A Survey on Deep Learning for Skeleton-Based Human Animation. *Computer Graphics Forum*, 41(1):122–157, 2022. DOI: <https://doi.org/10.1111/cgf.14426>. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14426>. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14426>.
- [26] L. Mourot, L. Hoyet, F. Le Clerc, F. Schnitzler und P. Hellier. A Survey on Deep Learning for Skeleton-Based Human Animation. *Computer Graphics Forum*, 41(1):122–157, 2022. DOI: <https://doi.org/10.1111/cgf.14426>. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14426>. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14426>.
- [27] T. Park, S.-H. Lee, J.-H. Kim und C.-H. Kim. CUDA-based Signed Distance Field Calculation for Adaptive Grids. In *2010 10th IEEE International Conference on Computer and Information Technology*, Seiten 1202–1206, 2010. DOI: 10.1109/CIT.2010.217.
- [28] X. B. Peng, Y. Guo, L. Halper, S. Levine und S. Fidler. ASE. *ACM Transactions on Graphics*, 41(4):1–17, Juli 2022. DOI: 10.1145/3528223.3530110. URL: <https://doi.org/10.1145%2F3528223.3530110>.
- [29] M. Pleines. neroRL: Documentation. <https://github.com/MarcoMeter/neroRL/tree/master/docs>. (Besucht am 15.03.2023).
- [30] P. Prusinkiewicz und A. Lindenmayer. *The Algorithmic Beauty of Plants*. Springer New York, 1990. ISBN: 978-0-387-94676-4. DOI: 10.1007/978-1-4613-8476-2.
- [31] K. Rudenko. DQ-Skinning for Unity. 2023. URL: <https://github.com/KosRud/DQ-skinning-for-Unity> (besucht am 15.03.2023).
- [32] J. Schulman, S. Levine, P. Moritz, M. Jordan und P. Abbeel. Trust Region Policy Optimization. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML’15, Seiten 1889–1897, Lille, France. JMLR.org, 2015.
- [33] J. Schulman, P. Moritz, S. Levine, M. Jordan und P. Abbeel. High-Dimensional Continuous Control Using Generalized Advantage Estimation. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.
- [34] J. Schulman, F. Wolski, P. Dhariwal, A. Radford und O. Klimov. Proximal Policy Optimization Algorithms. *CoRR*, abs/1707.06347, 2017. arXiv: 1707.06347. URL: <http://arxiv.org/abs/1707.06347>.
- [35] Scawk. Marching Cubes. 7. Mai 2022. URL: <https://github.com/Scawk/Marching-Cubes> (besucht am 15.03.2023).

Literaturverzeichnis

- [36] N. Shaker, J. Togelius und M. J. Nelson. *Procedural Content Generation in Games*. Springer International Publishing, 2016. ISBN: 978-3-319-42716-4. DOI: 10.1007/978-3-319-42716-4.
- [37] D. Sieger und M. Botsch. The Polygon Mesh Processing Library, 2019. URL: <http://www.pmp-library.org> (besucht am 15.03.2023).
- [38] I. Wegener. *Kontextfreie Grammatiken und Sprachen*. In *Theoretische Informatik: — eine algorithmenorientierte Einführung*. Vieweg+Teubner Verlag, Wiesbaden, 2005, Seiten 150–175. ISBN: 978-3-322-82204-8. DOI: 10.1007/978-3-322-82204-8_6.