1 Danksagungen

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

2 Einleitung

2.1 Motivation und Problemstellung

2.2 Zielsetzung und Vorgehensweise

2.3 Übersicht

3 Grundlagen

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

4 Verwandte Arbeiten

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

5 Fachliches Vorgehen

Keine technischen Details (wie z.B. Implementierung)

5.1 Projektorganisation

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

5.1.1 Creature Animator

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

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

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

Tabelle 5.1: Die zwei Untergruppen und ihre Mitglieder

5.2 Creature Generation

5.2.1 Automatisches Rigging

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

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

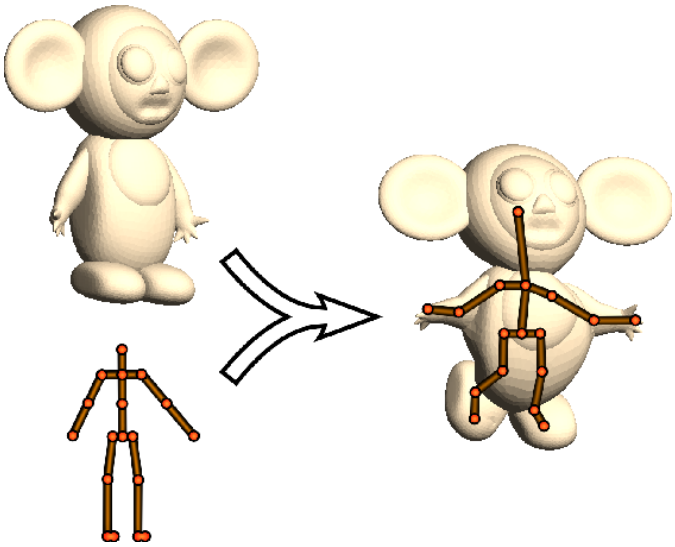

Abbildung 5.1: Beispiel für ein korrekt eingebettetes Skelett in einem Mesh [1].

Während der Entwickelung wurde zunächst eine triviale Methode als Zwischenlösung verwendet. Dabei wurden alle Vertices nur an den Bone mit der geringsten Distanz an-

gehängt. Diese Methode ist jedoch visuell unbrauchbar, da bei Bewegung des Skeletts an den Gelenken zwischen den Bones Löcher und verschiedene andere Artefakte entstehen. Es wird also eine Lösung benötigt, die es ermöglicht Vertices an mehrere Bones anzuhängen und die Gewichte so bestimmt, dass das Mesh an den Übergängen zwischen den Bones möglichst natürlich deformiert wird.

Ein bereits bekannter Algorithmus zur automatisch Gewichtsberechnung und Verknüpfung des Meshes ist die Bone-Heat Methode [1]. Dieser berücksichtig 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 Bones sollte ungefähr proportional zu der Distanz des Gelenks zur Oberfläche des Meshes sein. Ein Algorithmus, welcher die Gewichte alleine aus der Distanz der Bones zu den Vertices berechnet, kann oft schlechte Ergebnisse liefern, da er die Geometrie des Modells ignoriert. Zum Beispiel können Teile des Torsos mit einem Arm verbunden werden. Die Bone-Heat Methode behandelt stattdessen das innere Volumen des Modells als einen wärmeleitenden Körper. Es werden für jeden Vertex die Gleichgewichts-Temperatur berechnet und diese als Gewichte für die Bones verwendet. Wie in Abbildung 5.2 zu sehen ist, wird ein Bone auf 1° gesetzt und der andere auf 0°. Bei Deformation der beiden Bones mit den Gewichten aus dem Temperatur-Gleichgewichts entsteht an dem Gelenk eine natürlich aussehende Verformung der Oberfläche.

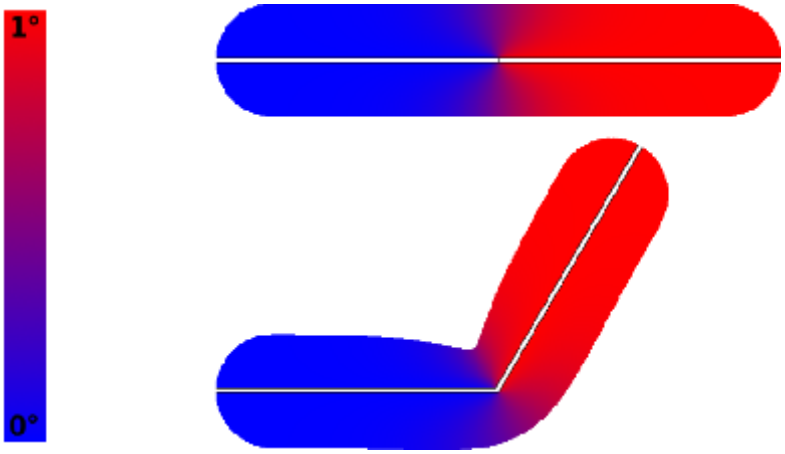

Abbildung 5.2: Temperatur-Gleichgewicht für zwei Bones [1].

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

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

Dabei ist Δ der diskrete Laplace-Beltrami Operator auf der Oberfläche des Meshes, welcher mit der Kotangens-Methode approximiert wird [2]. $\mathbf{p}^i$ ist ein Vektor mit $p^i_j = 1$ wenn der nächste Bone zum Vertex j der Bone i ist. Sonst ist $p^i_j = 0$. $\mathbf{H}$ ist eine Diagonalmatrix wobei H_{jj} die Hitze des nächsten Bones von Vertex j ist. Sei $d(j)$ die Distanz zum nächsten Bone von Vertex j, dann wird $H_{jj} = c/d(j)^2$ gesetzt. Allerdings nur wenn das Geradensegment von dem Vertex zu dem Bone vollständig in dem Volumen des Modells enthalten ist. Wenn der Bone von dem Vertex aus also nicht sichtbar ist, wird $H_{jj} = 0$ gesetzt. Dies verhindert, dass beispielsweise Vertices am Arm and den Torso angehängt werden. Wenn mehrere Bones nahezu die gleiche Distanz zu dem Vertex haben und sichtbar sind, werden ihre Anteile an der Temperaturverteilung gleich berücksichtigt. p_j wird dann $1/k$ und $H_{jj} = kc/d(j)^2$

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

Für die effiziente Lösung des Matrix-Systems 5.1 ist es wichtig die Eigenschaften des diskreten Laplace-Beltrami Operators Δ näher zu betrachten. Der Operator wird durch die Kotangens-Methode approximiert, indem von jedem Vertex x_i für jede ausgehende Kante ein Gewicht $v(x_i, x_j)$ aus den gegenüberliegenden Winkel α_{ij} und α_{ji} berechnet wird (siehe Abbildung 5.3).

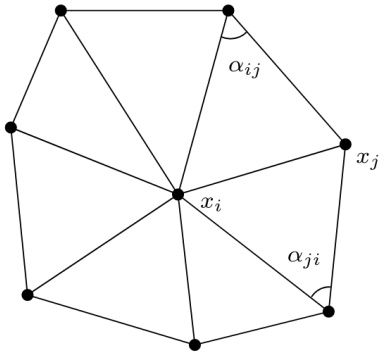

Abbildung 5.3: Berechnung der α-Winkel für eine innere Kante [2].

$$v(x_i, x_j) = \begin{cases} \cot \alpha_{ij} + \cot \alpha_{ji} & \text{for interior edges} \\ \cot \alpha_{ij} & \text{for boundary edges} \end{cases} \tag{5.2}$$

Aus den Kotangens-Gewichten aus Gleichung 5.2 kann nun in Formel 5.3 der Laplace-Beltrami Operator in seiner Matrix-Form berechnet werden [3, 2]. Dabei bescheibt $N(x_i)$ 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 [3] wurden hier weggelassen, da sie für diese Anwendung sowieso wegfallen.

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

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 in der Matrix pro Reihe [3]. Ein solches System lässt sich, wie in [3] gezeigt, sehr effizient lösen, wenn es sich um eine SPD-Matrix (symetrisch positiv definit) handelt. Die Matrix ist durch die Konstruktion symetrisch. (TODO Delaunay Bedingung erklären und Verbindung zu Definitheit erklären)

5.3 Creature Animation

5.3.1 Trainingsumgebung

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

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

Trotz den vielen Änderungen an der Kreatur war es nicht möglich, die Kreatur aus dem L-System zum Laufen zu bringen. Aus diesem Grund ersetzte eine andere Methode, welche von Jona und Markus umgesetzt wurde, das L-System. Mit der neuen Methode wurde auch der Creature-Generator direkt in die Trainingsumgebung eingebunden. Dies

ermöglichte es, schnell verschiedene Kreaturen zu testen. Die Bugreports und Featurerequests zum Generator werden direkt als Issue in das entsprechende GitHub Repository geschrieben und werden gegebenenfalls im Jour Fixe oder über Discord besprochen.

LiDO3

Das RL-Training benötigt viele Rechenressourcen. Die ersten Trainingstests mit der ML-Agents-Walker-Umgebung haben gezeigt, dass eine Trainingsdauer von über einen Tag auf aktueller Hardware zu erwarten ist. Deswegen muss das Training auf einen Server laufen. Als besondere Anforderungen benötigen die Server eine Nvidia Grafikkarte, um mit CUDA[1] pytorch[2] zu beschleunigen.

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

Konfiguration

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

5.3.2 Training

Generalisierung

- PPO
- ML-Agents
- Nero?

5.4 Terraingeneration

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

6 Technische Umsetzung

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

6.1 Creature Animation

6.1.1 Trainingsumgebung

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

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

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

Dynamic Enviorment Generator

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

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

3. Generiere die Kreatur

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

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

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

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

TerrainGenerator

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

Konfigurationsobjekte

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

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

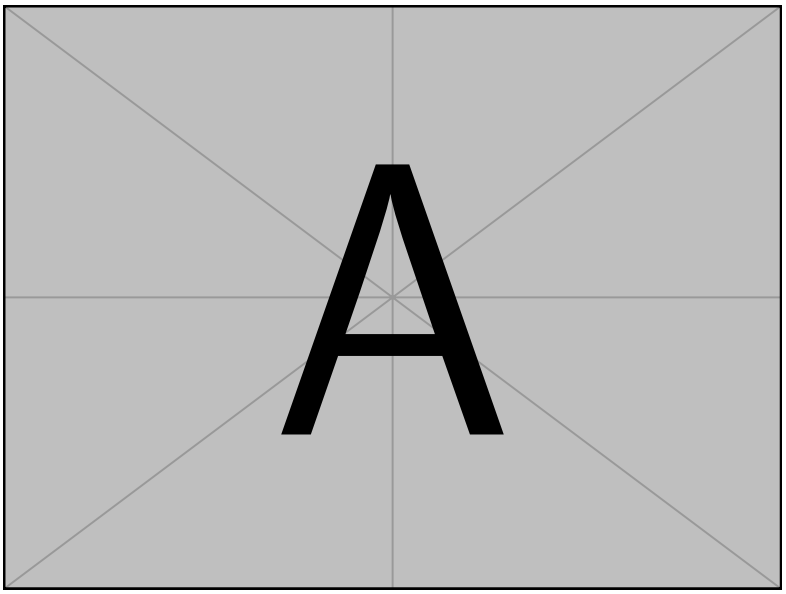

Abbildung 6.1: Konfigurationsmöglichkeiten des `DynamicEnviormentGenerator` im Unity-Editor.

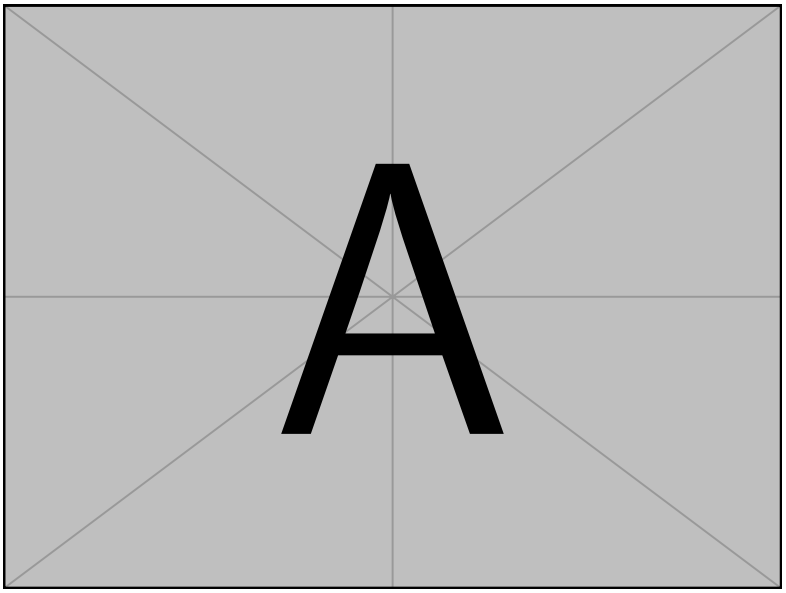

Abbildung 6.2: Ein Beispiel der generierten Trainingsumgebung mit mehreren Arenen.

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

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

texttt im Titel?

6.1.2 Erweiterung der Agent-Klasse

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

GenericAgent

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

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

JointDriveController und BodyPart

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

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

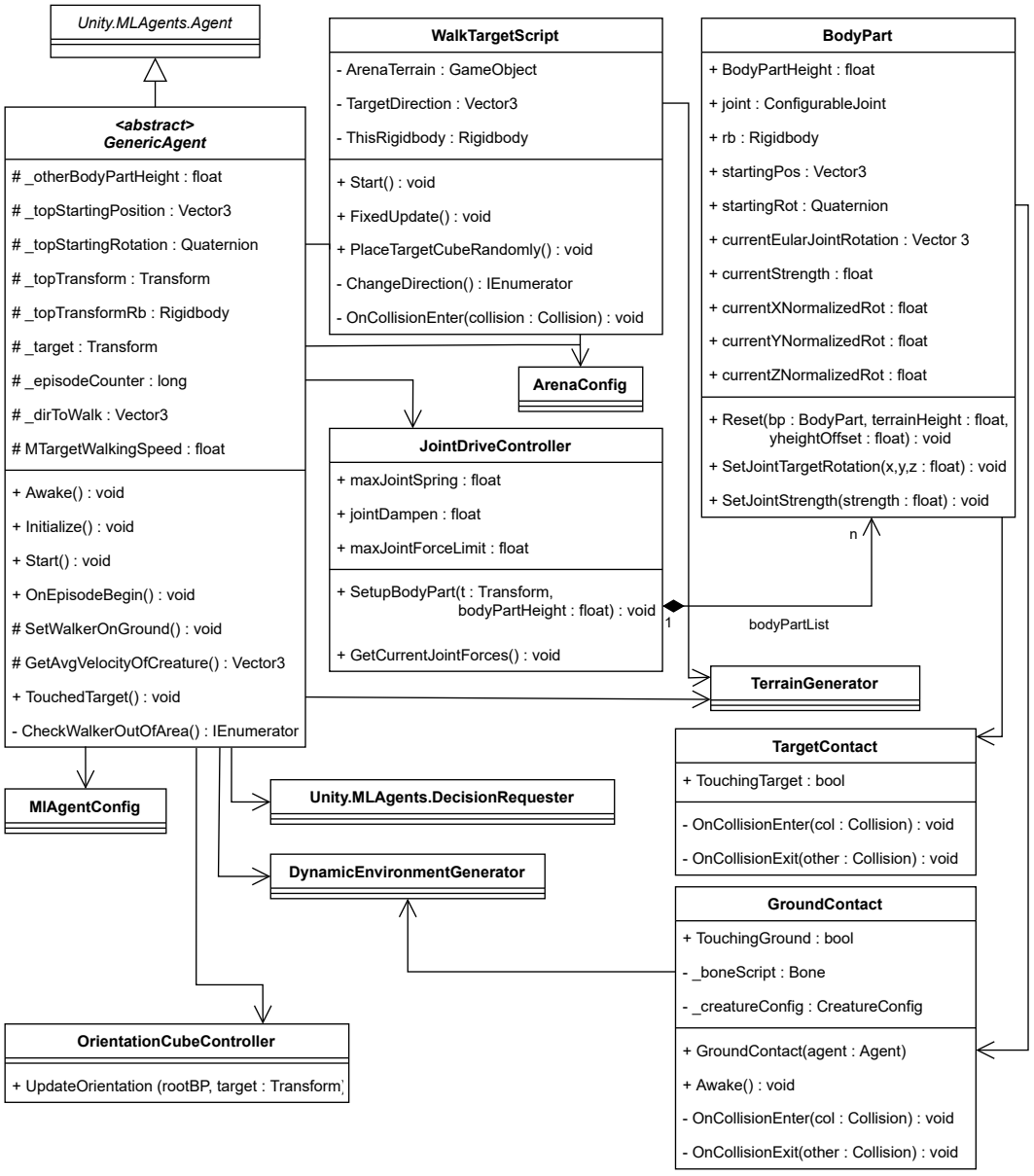

Abbildung 6.3: UML-Diagram der GenericAgent-Klasse und weitere relevante Klassen

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

OrientationCubeController

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

WalkTargetScript

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

6.1.3 LiDO3

Wie bereits erwähnt, werden die Berechnungen jeweils auf den HPC der TU Dortmund ausgeführt. Um auf LiDO3 zu arbeiten wird mit Hilfe eines Gatewayservers auf das Cluster zugegriffen. Der Zugriff ist ausschließlich über das TU Dortmund Netzwerk möglich. Über den Gatewayserver kann ein Zugriff auf die Rechenressourcen direkt über die Shell oder über Skripte angefordert werden. Da die Shell-Methode einen dauerhaften Login erfordern würde, wird mit Skripten gearbeitet. Diese bestehen aus Konfigurationen für LiDO3 und den eigentlich Programmteil, welcher ausgeführt werden soll. LiDO3 nutzt als Jobmanager Slurm, weshalb die Skripte die Slurm-Syntax nutzen. Eine ausführliche Beschreibung die LiDO3 Konfiguration findet sich im Benutzerhandbuch[5];

```
#!/bin/bash -l
#SBATCH -C cgpu01
#SBATCH -c 20
#SBATCH --mem=40G
#SBATCH --gres=gpu:2
#SBATCH --partition=long
```

```
#SBATCH --time=48:00:00
#SBATCH --job-name=pg_k40
#SBATCH --output=/work/USER/log/log_%A.log
#SBATCH --signal=B:SIGQUIT@120
#SBATCH --mail-user=OUR_MAIL@tu-dortmund.de
#SBATCH --mail-type=ALL
#-----------------------------------

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


module purge
module load nvidia/cuda/11.1.1

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

chmod -R 771 $GAME_PATH
cd $GAME_PATH

srun mlagents-learn /work/smnidunk/games/config/Walker.yaml --run-id=$GAME_NAME --env=t.x86_64
```

In dem Beispielskript 6.1.3 sind Anweisungen an die LiDO-Umgebung jeweils mit einem Kommentarzeichen gefolgt von *SBATCH* gekennzeichnet. Die Konfiguration wird so gewählt, dass eine maximale Laufzeit mit exklusiven Ressourcenrechten auf den Rechenknoten besteht. Zusätzlich muss sichergestellt werden, dass eine Grafikkarte zur Verfügung steht. Diese stehen auf den *cgpu01*-Rechenknoten mit jeweils 20 CPU-Kernen und 48 Gigabyte RAM zur Verfügung. Die maximale Laufzeit des Prozesses ist bei den GPU-Knoten auf *long* begrenzt, was 48 Stunden entspricht. Es wird jeweils ein Log mitgeschrieben, aus dem der Trainingsfortschritt gelesen werden kann und bei besonderen Ereignissen eine Mail geschickt, um sofort benachrichtigt zu werden, falls der Job fertig ist oder fehlschlägt.

Kompatibilitätsprobleme

Um das beschriebene Skript auszuführen, muss auf LiDO3 eine ML-Agents-Umgebung installiert werden. Dabei handelt es sich um ein Python Umgebung, mit PyTorch und CUDA. In dem Slurm-Skript 6.1.3 ist die Einrichtung einer funktionierenden Umgebung dargestellt.

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

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

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

Problematisch ist die Installation von PyTorch, da ab Version 1.5 die Installationsbinärdateien keine Unterstützung für die von LiDO3 genutzten NVIDIA Tesla K40 Grafikarten bietet. Es besteht die Möglichkeit PyTorch zu bauen um die Unterstürzung zu erhalten. Dies musste für unsere Arbeitsumgebung nicht gemacht werden, da die PG-Betreuer ein Paket mit einer für LiDO funktionierenden PyTorch-Version von einer vorherigen PG zur Verfügung stellen konnten. Wie in 6.1.3 dargestellt müssen zuerst die Abhängigkeiten von PyTorch, dann ML-Agents und zuletzt die spezielle PyTorch Version installiert werden, da sonst die Abhängigkeiten Probleme bereiten.

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

7 Vorläufige Ergebnisse

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

7.1 Diskussion

Diskussion der Ergebnisse in Bezug auf die initiale Zielsetzung.

8 Ausblick

Literatur

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

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

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

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

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