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 Parametrische Kreatur

Als Grundlage für die parametrische Generierung der Kreaturen dient das von Jon Hudson in seiner Thesis [3] beschriebene Modell, welches in Abschnitt **??** näher beschrieben wird.
Die Kreatur besteht aus mehreren Körperteilen, die separat generiert werden und jeweils ihre eigenen Parameter besitzen. Diese Körperteile sind Torso, Beine, Arme, Hals, Füße und Kopf, die jeweils aus einem oder mehreren Knochen bestehen. Alle in diesem Abschnitt erwähnten Zufallsvariablen entstammen einer uniformen Verteilung.

Knochen Koordinaten

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

Generierung

Unsere Methode setzt voraus, dass zunächst gewisse Vorgaben zur generellen Struktur der Kreatur gemacht werden. In unserem konkreten Fall bedeutet dies, dass wir uns zunächst auf Zwei- und Vierbeiner beschränken. Der Zweibeiner orientiert sich an der menschlichen Anatomie und besitzt deshalb in jedem Bein jeweils zwei Knochen und einen einzelnen Fußknochen sowie zwei Arme. Die Beine eines Vierbeiners besitzen, angelehnt an die Skelette echter vierbeiniger Säugetiere, jeweils vier Knochen. Arme werden

hier nicht generiert. Um Kreaturen zu erzeugen, die nicht einer dieser Strukturen entsprechen, wie beispielsweise Spinnentiere oder Echsen, müsste man weitere Skelette definieren.

Zuerst wird der **Torso** Generiert. Im Falle des Zweibeiners ist dieser nach oben gerichten, beim Vierbeiner parallel zum Boden. Als Parameter werden jeweils Minima und Maxima für die Länge und den Umfang des Torsos übergeben. Daraus wird zunächst ein zufälliger Wert für die Länge bestimmt. Da wir uns vorerst für einen Torso bestehend aus drei Knochen entschieden haben, wird diese Länge zufällig auf drei kleinere Längen aufgeteilt. Jeder dieser Knochen erhält dann einen zufälligen Umfang. Das unterste Torsosegment dient als Elternknochen der Kreatur, die anderen beiden Knochen werden dann nacheinander am distalen Punkt des vorangegangenen Knochens befestigt.
Anschließend werden die **Beine** paarweise generiert, wodurch nur symmetrische Kreaturen erstellt werden können. Auch hier wird zunächst die Länge zufällig bestimmt und dann auf zwei beziehungsweise vier Knochen aufgetelt. Auch Die Umfänge werden zufällig bestimmt, jedoch zusätzlich sortiert, sodass das das dickste Segment der Körpermitte am nächsten gelegen ist. In beiden Fällen werden die Beine gerade nach unten gereriert. Bei einem Zweibeiner wird am distalen Punkt des unteren Beinknochen in Richtung 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 vorderen Hüftknochens. Dabei ist die Rotation des ersten Knochens ein zufälliger Wert zwischen der der proximalen und der der negativen ventralen Achse des Elternknochens. Jedes weitere Halssegment erhält eine zufällige Rotation um $\pm 20°$.
Der **Kopf** ist ein einzelner Knochen mit zufälliger Länge und Umfang, der als Verlängerung des letzten Halsknochens betrachtet werden kann.

Die glaubwürdigkeit und Variation der Kreaturen ist dadurch also auch stark von den gewählten Parametern Abhängig. Diese werden von Hand gesetzt, wobei größere Wertebereiche auch für größere Unterschiede zwischen den Kreaturen sorgen, aber gleichzeitig die Kontrolle über das Ergebnis einschränken. Die Beschränkungen der Bewegungsradii der Gelenke werden momentan händisch mit erfahrungsgemäß guten Werten gesetzt. Ziel

ist es allerdings auch diese in Zukunft soweit möglich prozedural mit Hilfe von Parametern zu erzeugen, um verschiedene Bewegungsmuster zu ermöglichen.

5.2.2 L-System Creature

5.2.3 Metaballs

Zur Generierung der Geometrie der Kreatur verwenden wir, angelehnt an den Ansatz von Madis Janno [2] (siehe **??**), eine modifizierte Form von Metaballs. 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.

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

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

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

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

Unser Ansatz um dieses Problem zu umgehen ist es, die Anzahl der einzelnen Metabälle drastisch zu reduzieren. Anstatt einer beliebig großen Zahl an Bällen entlang jedes Segments, erzeugen wir jeweils nur einen einzigen. Dazu ersetzen wir die Bälle durch Kapseln, also Zylinder mit jeweils durch eine Halbkugel abgerundeten Enden. Möglich macht uns dies eine Modifikation der Falloff-Funktion, beziehungsweise der darin verwendeten Distanz. Wir berechnen hierbei nicht den Abstand zum Zentrum einer Kugel, sondern zu der Verbindungslinie zwischen Start- und Endpunkt.

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 5.1). Die Distanz lässt sich dann folgendermaßen bestimmen:

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

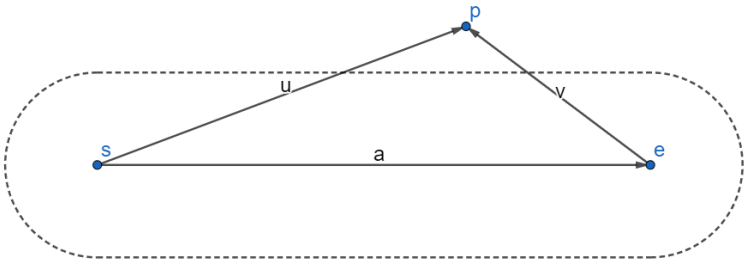

Abbildung 5.1: Beispiel Metakapsel; Die gestrichelte Linie enthält alle Punkte mit $r = R$

Es werden drei Fälle unterschieden. Liegt der Punkt p im Falle von Abbildung 5.1 links von s, beziehungsweise rechts von e, ist Die Distanz von p zum Segment einfach der euklidische Abstand zum jeweiligen Punkt. Ob dies der Fall ist, lässt sich mit Hilfe der Skalarprodukte $a \cdot u$ beziehungsweise $a \cdot v$ überprüfen. Ansonsten berechnet man die Distanz von Punkt p zur Geraden, die durch s und e verläuft.

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

5.2.4 Marching Cubes / Mesh Generation

5.2.5 Automatic Rigging

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

[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

- Nero?

5.3.3 Terraingeneration

5.3.4 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* [4].

Space Partitioning

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

Bei der Partitionierung eines zweidimensionalen Raumes mit einem k-d-Baum wird in jedem Schritt eine Achse zufällig ausgewählt, an der die Eingabefläche in zwei Teile partitioniert wird. Anschließend wird ein Punkt ausgewählt, der die Grenze zwischen den beiden Partitionen bestimmt. Dabei stellt die Eingabefläche einen Knoten im Baum dar, an den zwei Kindknoten für die Partitionen angehangen werden. So entsteht ein Binärbaum, in dem die Wurzel die Ausgangsfläche enthält und ein innerer Knoten oder Blattknoten eine der zwei Partitionen des Elternknotens enthält. Der Algorithmus wird so lange rekursiv angewandt, bis ein Abbruchkriterium erfüllt ist. Als Abbruchkriterium kann z.B. eine minimale Größe einer Partition gefordert werden. Alternativ kann die Höhe des Baumes durch einen maximalen Wert beschränkt werden. Die Blätter des Baumes enthalten die Partitionen der Ausgangsfläche. Abbildung 5.2 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.

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

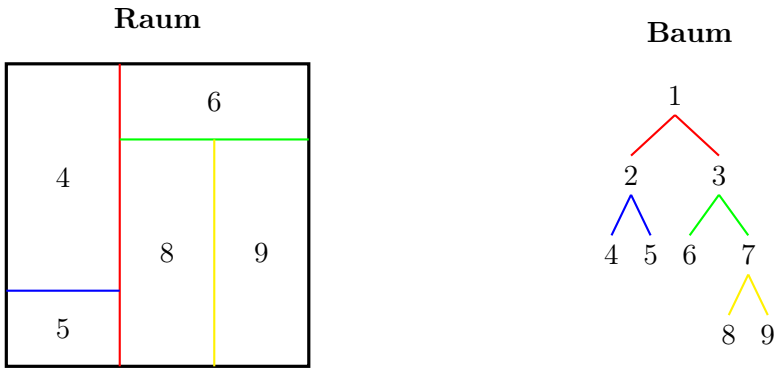

Abbildung 5.2: Space Partitioning mit k-d-Baum

Partitionen werden in diesem Fall direkt zu Blättern im Baum. Als Abbruchkriterien können die gleichen, wie bei k-d-Bäumen verwendet werden. Abbildung 5.3 zeigt die Partitionierung einer quadratischen Fläche mit einem Quadtree, wobei in jedem Schritt in vier gleich große Partitionen unterteilt wird.

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

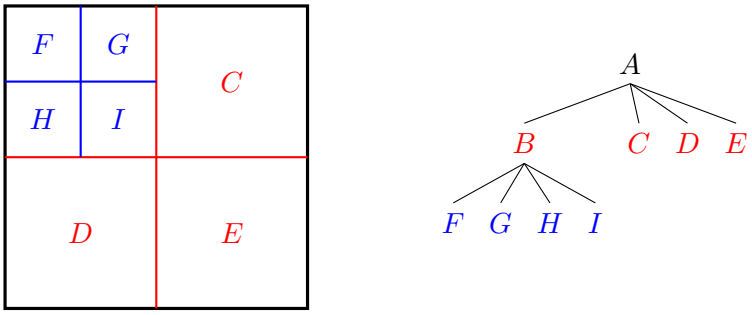

Abbildung 5.3: Space Partitioning mit Quadtrees

Platzierung der Räume und Korridore

Ausgehend von einer in einer mittels Space Partitioning erstellten Baumstruktur vorliegenden Partitionierung einer rechteckigen Fläche können nun Räume in den Partitionen platziert werden. Da die Partitionen zunächst direkt aneinander anliegen, werden die Räume mit einem durch Parameter beschränkten, zufällig gewählten Abstand zu der linken, rechten, oberen und unteren Begrenzung platziert. Nach der Platzierung der Räume wird der Baum traversiert, um Räume durch Korridore miteinander zu verbinden. Dazu wird in jedem Knoten ein Korridor generiert, der zwei Räume in den Partitionen der Teilbäume der Kinder miteinander verbindet. Der Korridor verläuft dabei durch eine in

dem Knoten gespeicherte Unterteilungsebene, wodurch sichergestellt wird, dass Korridore sich nicht überschneiden. Ein Beispiel für einen prozedural generierten Dungeon mit zugehörigem k-d-Baum ist in Abbildung 5.4 gegeben. Der blau markierte Korridor wurde beispielsweise im Knoten 0 platziert.

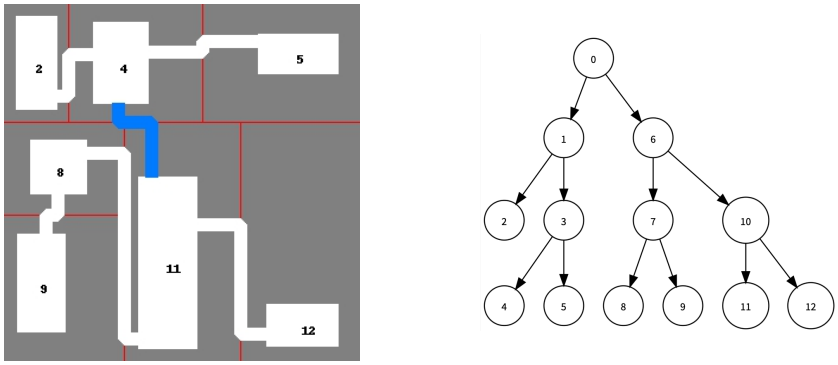

Abbildung 5.4: Prozedural generierter Dungeon mit k-d-Baum

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

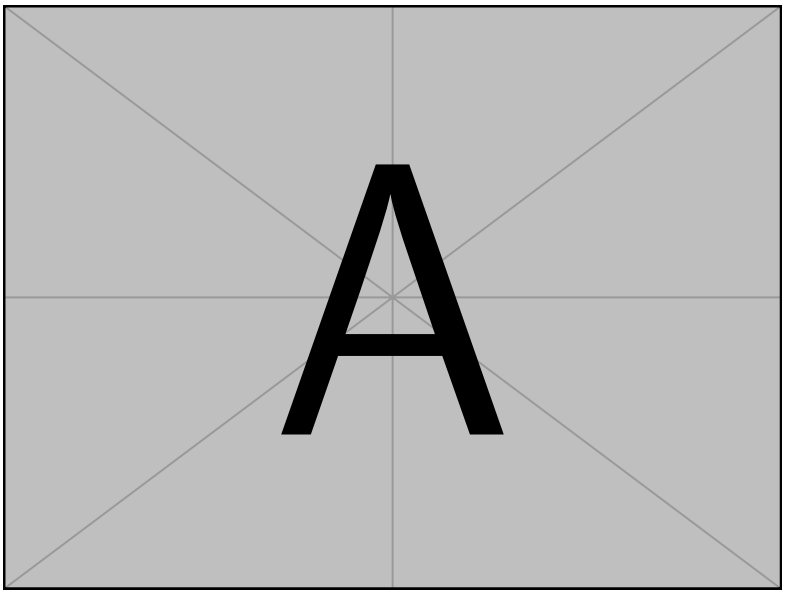

Abbildung 6.1: Konfigurationsmöglichkeiten des `DynamicEnviormentGenerator` im Unity-Editor.

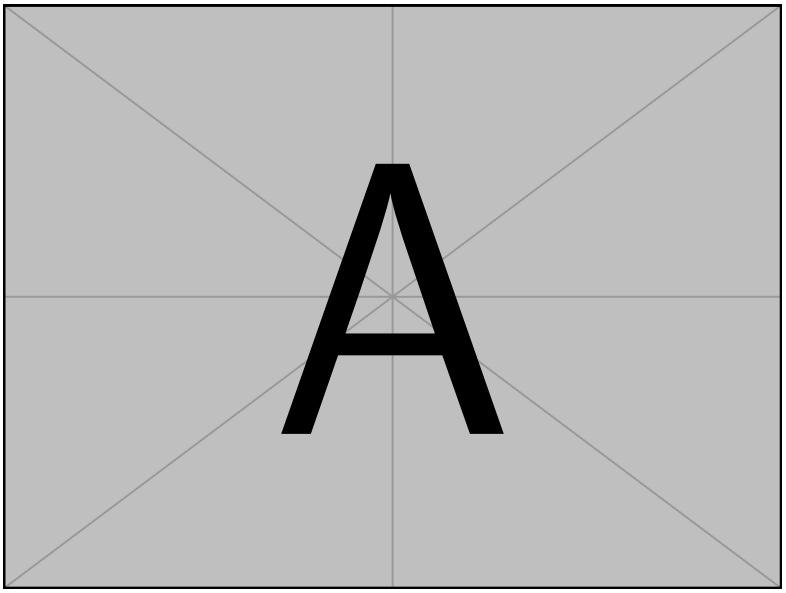

Abbildung 6.2: Ein Beispiel der generierten Trainingsumgebung mit mehreren Arenen.

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

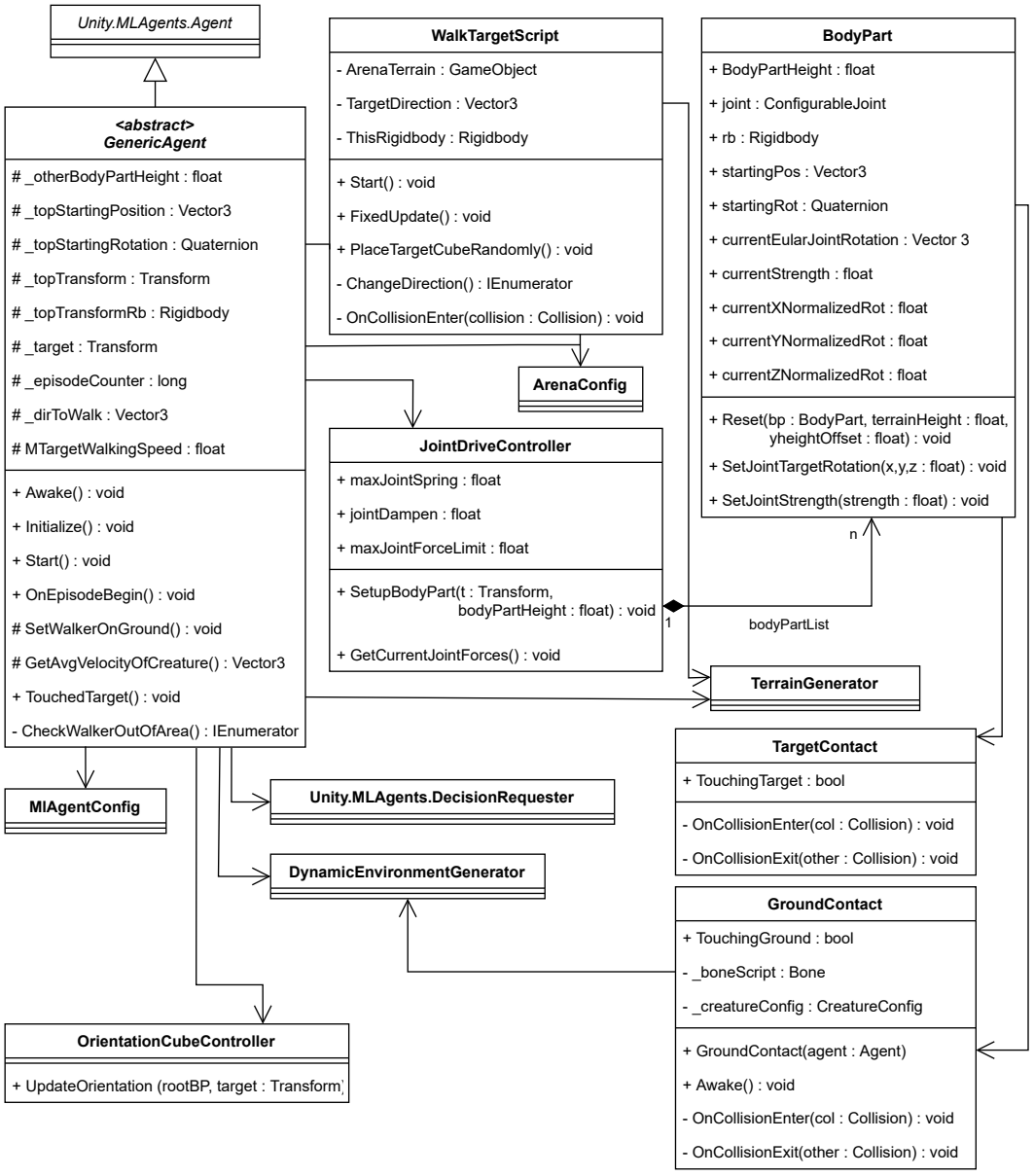

Abbildung 6.3: UML-Diagram der GenericAgent-Klasse und weitere relevante Klassen

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[1];

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

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

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


module purge
module load nvidia/cuda/11.1.1

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

chmod -R 771 $GAME_PATH
cd $GAME_PATH

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

6.2 Creature Generation

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

6.2.1 Unity Package

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

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

6.2.2 Konfiguration

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

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

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

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

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

6.2.3 Bone Definition

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

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

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

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

6.2.4 Parametrische Generatoren

Parametrische Generatoren haben die Aufgabe anhand der `ParametricCreatureSettings` `BoneDefinition`-Bäume zu generieren. Das Package enthält momentan Generatoren für

zwei verschiedene Typen von Kreaturen: `BipedGenerator` und `QuadrupedGenerator`. Einstiegspunkte in die Generatoren sind jeweils die `BuildCreature` Methoden.

Beide Generatoren erzeugen zunächst aus den Intervallen in den `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`.

Explain tree structure if not done in chapter 5

6.2.5 Skeleton Definition

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

Die `SkeletonDefinition` dient dann im nächsten Schritt als Eingabe für den `SkeletonAssembler`.

6.2.6 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 Knochen ab.
- ein `Bone`, der Metadaten, wie z.B. Länge oder Kategorie des Knochens, enthält

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

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

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

Configurable Joints

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

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

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

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

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

6.2.7 Mesh Generator

6.2.8 Creature Generator

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

7 Vorläufige Ergebnisse

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

7.1 Diskussion

Diskussion der Ergebnisse in Bezug auf die initiale Zielsetzung.

8 Ausblick