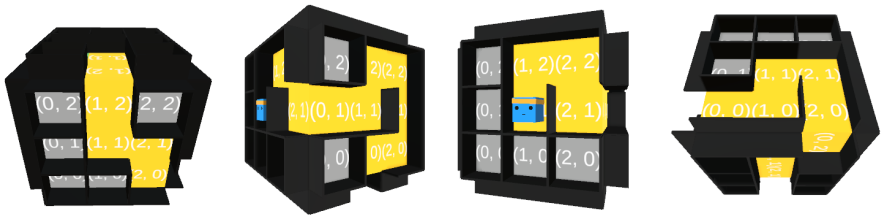

Independent Coursework 1

Hochschule für Technik und Wirtschaft Berlin

Hochschule für Technik und Wirtschaft Berlin
University of Applied Sciences

Generieren von 3D Labyrinthen mit Unitys ML Agents

Projektbericht

Vivienne Drongowski
Studiengang Internationale Medieninformatik
Matrikelnummer: 581491
Github Repository: https://github.com/fayebullwinkel/fifi-ml-agents

10. Oktober 2023

Supervision: Prof. Dr. Tobias Lenz

Inhaltsverzeichnis

1. Einleitung

In der Welt der künstlichen Intelligenz und des maschinellen Lernens gibt es ständig neue Herausforderungen, die Forscher:innen und Entwickler:innen auf der ganzen Welt inspirieren und faszinieren. Im Rahmen eines Independent Courseworks habe ich mich entschlossen, mich einer spannenden Fragestellung zu widmen: Wie können maschinelles Lernen und die Macht des Reinforcement Learning zur Schaffung intelligenter Agenten genutzt werden, die in der Lage sind, Labyrinthe auf der Oberfläche eines Würfels zu generieren?

In Zusammenarbeit mit Faye Bullwinkel haben wir uns in diesem Projekt intensiv mit beiden Aspekten der Labyrinthgenerierung beschäftigt. Während F. Bullwinkel sich auf die Entwicklung eines Agenten konzentrierte, der die Herausforderung annimmt, ein Labyrinth zu lösen[1], konzentriert sich dieser Bericht auf den Prozess der Labyrinthgenerierung. Wir werfen einen detaillierten Blick auf die Schritte, die unternommen wurden, um einen Agenten zu entwickeln, der in der Lage ist, komplexe Labyrinthe zu erzeugen.

Das Projekt verbindet die mathematische Herausforderung, komplexe dreidimensionale Umgebungen zu erzeugen, mit der spannenden Frage, wie Agenten in diesen Umgebungen navigieren und lernen können. Dieser Projektbericht dokumentiert meinen Weg durch dieses Forschungsprojekt.

1.1 Zielsetzung

Das Ziel dieses Projekts ist die Entwicklung eines intelligenten Agenten, der in der Lage ist, ein gültiges Labyrinth auf der Oberfläche eines Würfels zu erzeugen. Dabei stehen mehrere Teilziele im Vordergrund: die Konzeption und Implementierung einer geeigneten Datenstruktur zur Repräsentation des Labyrinths, die Definition klarer Bedingungen für die Gültigkeit eines generierten Labyrinths, die Definition des Verhaltens und der Bewertungen des Agenten sowie dessen Training durch Reinforcement Learning. Das übergeordnete Ziel besteht darin, einen Agenten zu schaffen, der in der Lage ist, autonom und effizient ein Labyrinth zu generieren, das einen Weg vom Start zum Ziel enthält und dabei möglichst viele Felder des Labyrinths nutzt.

Dieses Projekt trägt nicht nur zur Erforschung des maschinellen Lernens und der künstlichen Intelligenz bei, sondern eröffnet auch neue Möglichkeiten für die automatische Generierung komplexer dreidimensionaler Umgebungen.

[1] Bullwinkel, Faye (2023), Lösen von 3D-Labyrinthen mit Unitys ML Agents [Projektbericht]

1.2 Aufbau der Arbeit

Diese Arbeit gliedert sich in mehrere Abschnitte, um die Entwicklung und das Training eines intelligenten Agenten zur Generierung von Labyrinthen im dreidimensionalen Raum systematisch abzuhandeln. Nach der Erläuterung der Zielsetzung des Projektes wird im zweiten Abschnitt das verwendete Toolkit ML-Agents von Unity vorgestellt. Kapitel 3 befasst sich mit der Konzeption des Projekts, einschließlich der Definition der Bedingungen für ein gültiges Labyrinth. Kapitel 4 beschäftigt sich mit der Implementierung der einzelnen Komponenten. Dazu wird die Datenstruktur für das Labyrinth in zwei- und dreidimensionaler Form diskutiert und die Überprüfungsmechanismen für ein gültiges Labyrinth vorgestellt. Außerdem wird die Anwendung selbst, einschließlich der Kameraansicht und des Debug-Modus, betrachtet.

Der Agent, der für die Generierung der Labyrinthe verantwortlich ist, wird in Kapitel 5 behandelt, wobei der Start einer Episode, Beobachtungen, Aktionen und Auswertungen im Detail erklärt werden. Anschließend wird in Kapitel 6 das Training des Agenten mittels Reinforcement Learning und Imitation Learning diskutiert. Die beiden oben genannten Agenten *MazeGeneration* und *MazeSolving* werden in Kapitel 7 zusammengeführt, so dass der erste Agent ein Labyrinth generiert, das der zweite Agent dann lösen soll. Abschließend werden in Kapitel 8 die Ergebnisse des Projekts zusammengefasst und ein Fazit gezogen.

2. Unity ML Agents

Mit dem *ML-Agents Toolkit*[1] stellt Unity eine simple Integration einer Deep-Learning Technologie für die Entwicklung von intelligenten und lernfähigen virtuellen Charakteren, Robotern und KI-gesteuerten Gegnern in Videospielen und Simulationen bereit. Dies ermöglicht es Entwickler:innen, interaktive 3D-Welten mit realistischer und adaptiver KI zu erstellen, die auf Reinforcement Learning und neuronalen Netzwerken basieren. Dadurch wird die Entwicklung immersiver und intelligenter Spielerlebnisse unterstützt.

In diesem Kapitel wird die Einrichtung und Funktionsweise des Toolkits beschrieben. Dies bildet die Grundlage für die Entwicklung eines eigenen Agenten. Ich habe mich dafür entschieden, zuerst die Funktionsweise des Toolkits zu verstehen, um die weitere Konzeption und Implementierung der Datenstruktur bestmöglich darauf abstimmen zu können.

2.1 Einrichtung

Bei der Installation von ML-Agents habe ich mich grob an die Anleitung von Unity [2] gehalten. Da der eigentliche Prozess jedoch mehr Schritte umfasst, werden diese im Folgenden beschrieben.

Voraussetzungen

1. Unity Version 2021.3.12
2. Python 3.8.10
3. Anaconda
4. ML-Agents Release 20 [3]

Python Umgebung einrichten
In der Anaconda Prompt:

1. Erstellen und Aktivieren einer neuen Umgebung:

```
        $ conda create -n mlagents_r20 python=3.8
        $ conda activate mlagents_r20

```

[1] https://unity.com/de/products/machine-learning-agents
[2] https://github.com/Unity-Technologies/ml-agents/blob/develop/docs/Installation.md
[3] https://github.com/Unity-Technologies/ml-agents/releases

2. Installieren von ML-Agents in der Anaconda Umgebung:

```
        $ pip3 install torch~=1.7.1 -f https://download.
    pytorch.org/whl/torch_stable.html
        $ pip3 install -e ./ml-agents-envs
        $ pip3 install -e ./ml-agents

```

3. Überprüfen der Installation:

```
        $ mlagents-learn --help

```

4. Weitere Installationen:

```
        $ pip install protobuf==3.20.3
        $ pip install six
        $ pip install onnx

```

Einrichten von ML-Agents in Unity

1. in Window/PackageManager/Add Package from Disk:
   $/ml-agents-release\_20/com.unity.ml-agents/package$
2. in Window/PackageManager/Add Package from Disk:
   $/ml-agents-release\_20/com.unity.ml-agents.extensions/package$
3. ML-Agents Ordner in Assets integrieren oder als Subrepository einhängen:
   $ml-agents-release\_20/Project/Assets/ML-Agents$

2.2 Erste Schritte

Um in die Welt der ML-Agents einzusteigen und ein grundlegendes Verständnis für die Verwendung des Toolkits zu erlangen, habe ich ein Tutorial absolviert, das auf der offiziellen ML-Agents-Github Seite[4] verfügbar ist. Das Beispiel führte mich Schritt für Schritt durch den Prozess der Erstellung einer neuen Lernumgebung für einen Agenten, der sich von einem zufälligen Startpunkt zu einem zufällig platzierten Zielobjekt bewegen sollte.

In Unity wurden dafür verschiedene Objekte erstellt, darunter eine Bodenfläche, ein Zielwürfel und ein Agent in Form einer Kugel. Diese Elemente wurden in einer Trainingsumgebung organisiert, um das spätere Training in mehreren Gruppen zu ermöglichen.

Für den Agenten wurde ein eigenes Skript namens *MoveToGoalAgent* erstellt, das von der Agentenklasse erbt. Dieses Skript definiert drei wichtige Funktionen, die das Verhalten des Agenten beeinflussen.

[4]https://github.com/Unity-Technologies/ml-agents/blob/develop/docs/Learning-Environment-Create New.md

OnEpisodeBegin() Der Agent wird in Episoden trainiert, wobei jede Episode solange läuft, bis der Agent die Aufgabe gelöst oder verloren hat. Zu Beginn jeder Episode wird diese Funktion aufgerufen. Hier wird der Agent neu positioniert, wenn er von der Plattform gefallen ist, und in jedem Fall wird das Ziel neu positioniert.

CollectObservations(VectorSensor sensor) Um Entscheidungen treffen zu können, sammelt der Agent Informationen über seine Umgebung. In diesem Fall sind das seine eigene Position, die Position des Ziels und seine Geschwindigkeit.

OnActionReceived(ActionBuffers actionBuffers) Der Agent liefert Eingaben, die eine Aktion auslösen, die dann wiederrum bewertet werden. In diesem Fall gibt der Agent jeweils eine Eingabe für die Kraft, die in x- und in z-Richtung auf die Kugel wirken. Für die Auswertung wird der Abstand zum Ziel betrachtet. Ist der Agent sehr nah am Ziel, wird der Reward auf 1 gesetzt und die Episode beendet. Wenn der Agent von der Plattform gefallen ist, wird die Episode ebenfalls beendet, jedoch ohne Belohnung.

Die Skripte wurden dem Agentenobjekt in Unity zugewiesen, um sein Verhalten zu steuern. Zusätzlich wurden weitere Komponenten wie der *Decision Requester* und das *Behavior Parameters* Skript benötigt, um die Steuerung und die Parameter des Agenten anzupassen.
Um die Funktionalität zu testen, habe ich in der **Heuristic()** Methode eine Übersetzung von manuellen Eingaben in die Aktionen implementiert, die es erlaubt, den Agenten manuell zu steuern.
Nach einem erfolgreichen manuellen Test der Umgebung habe ich eine YAML-Datei in $ml-agents-release\_20/config$ erstellt, um die Hyperparameter für das Training zu definieren, einschließlich des Trainingstyps, der Lernrate und der maximalen Anzahl von Schritten.

Das Training wurde mit folgendem Befehl zuerst in der Python-Umgebung von der Anaconda Prompt aus und dann in Unity gestartet.

```
1    $ mlagents-learn config/moveToGoal_config.yaml --run-id=
   MoveToGoal_01
```

Während des Trainingsprozesses konnte der Fortschritt mit Hilfe von *TensorBoard* überwacht werden, um sicherzustellen, dass der Agent seine Fähigkeiten zur Navigation und zur Lösung der gestellten Aufgabe kontinuierlich verbessert. Um das TensorBoard zu starten muss folgender Befehl in die Anaconda Prompt eingegeben werden.

```
1    $ tensorboard --logdir results
```

Daraufhin kann der Ablauf des Trainings im Browser unter *http://localhost:6006/* verfolgt werden.
Sobald der Agent zufriedenstellend arbeitet und die Aufgabe fehlerfrei löst, kann das Training beendet werden. Die Ergebnisse befinden sich unter $ml-agents-release\_20/results/MoveToGoal\_01$. Um das soeben trainierte Neuronale Netzwerk verwenden zu können, muss die Datei *MoveToGoal.onnx* in den Unity Assets Ordner importiert werden. In den *Behavior Parameters* des Agenten kann diese Datei dann als *Model* hinzugefügt werden. Anschließend muss noch der *Behavior*

Type auf *Interference Only* gesetzt werden, womit der Agent in der Lage ist, die Aufgabe mit eigenständigen Entscheidungen zu lösen.

Diese ersten Schritte lieferten wertvolle Einblicke in die Grundlagen von ML-Agenten und bildeten die Basis für die Entwicklung eines intelligenten Agenten zur Generierung von Labyrinthen auf der Oberfläche eines Würfels.

3. Konzeption

Im folgenden Kapitel zur Konzeption werden zwei grundlegende Ansätze zur Labyrintherzeugung auf der Oberfläche eines Würfels im Detail betrachtet. Dafür wird der *"Weg ausschneiden"*-Ansatz mit dem *"Weg bilden"*-Ansatz verglichen. Darüber hinaus wird definiert, was ein gültiges Labyrinth ist und die damit verbundenen Bedingungen und Kriterien eingeführt. Diese Konzeption bildet die Grundlage für die spätere Umsetzung und Implementierung der Labyrinthgenerierung sowie für die Bewertung der generierten Labyrinthe hinsichtlich Qualität und Lösbarkeit.

3.1 Ansatz für die Labyrinthgenerierung

Bei der Konzeption der Labyrinthgenerierung wurden im Rahmen dieses Projekts zwei verschiedene Ansätze in Betracht gezogen: der Ansatz *"Weg ausschneiden"* und der Ansatz *"Weg bilden"*. Im Folgenden werden beide Ansätze kurz erläutert und warum letztendlich der Ansatz *"Weg ausschneiden"* bevorzugt wurde.

3.1.1 Weg ausschneiden

Bei diesem Ansatz wird das Spielfeld zunächst mit einem *Wandgitter* gefüllt. Ein Agent wird zufällig auf einem Feld platziert, und dieses wird als Startfeld markiert. Der Agent kann sich in vier Richtungen bewegen und eine Wand wird durch Berührung gelöscht. Auf diese Weise schneidet er in dem Gitter einen Weg zum Ziel aus.
Wenn der Agent das Ziel platziert hat, wird das Labyrinth auf seine Lösbarkeit geprüft. Gibt es keinen zusammenhängenden Weg vom Startfeld zum Zielfeld erhält der Agent eine negative Belohnung von -1.0. Das Labyrinth ist lösbar und wird belohnt, wenn der Agent einen Pfad vom Start zum Ziel erstellt hat. Bei diesem Ansatz werden verschiedene Parameter wie die Anzahl der möglichen Wege, die Anzahl der Sackgassen, die Tiefe der Sackgassen, die Anzahl der Kreuzungen und die Positionen der Start- und Zielfelder berücksichtigt.
Der Ansatz ist einfacher zu implementieren und erfordert keine komplexen Algorithmen zur Platzierung der Wände. Zu Beginn werden alle Zellen mit ihren Wänden in eine Datenstruktur eingepflegt und die zerstörten Wände werden lediglich entfernt. Außerdem können unterschiedliche Schwierigkeitsgrade durch Anpassung der Parameter erreicht werden.
Das Verhalten des Agenten kann ebenfalls weniger komplex implementiert werden, denn es stehen zu jedem Zeitpunkt nur fünf Entscheidungen zur Verfügung. Das Verhältnis zwischen Wänden und Wegen ist schwer zu kontrollieren und es besteht die Gefahr, dass zu viele Sackgassen entstehen oder der Agent zu viele Wände

an einer Stelle entfernt. Die Gefahr eines leeren Labyrinths muss also an anderer Stelle abgefangen werden.

3.1.2 Weg bilden

Im Gegensatz dazu konzentrierte sich der *"Weg bilden"*-Ansatz auf die Idee, dass sich der Agent auf einem leeren Gitter bewegt und in der Lage ist, Wände zu platzieren, um den Weg zu bilden. Dieser Ansatz ermöglicht eine größere Flexibilität und Kontrolle über die Generierung des Labyrinths, da der Agent aktiv Entscheidungen treffen kann, um den Weg zum Ziel zu formen. Ein großer Nachteil dieses Ansatzes ist die sehr aufwendige und komplexe Datenstruktur, die benötigt wird. Auch die Eingabe und die daraus resultierenden Aktionen des Agenten wären wesentlich schwieriger zu definieren. Wenn der Agent z.B. die Aktion auslöst, eine vertikale Wand zu platzieren, müsste das interne Feld gefunden werden, auf dem sich der Agent gerade befindet. Nach einer Überprüfung, ob das Feld an dieser Stelle noch keine Wand hat, müsste eine Wand in die Datenstruktur eingefügt werden.

3.2 Bedingungen für ein gültiges Labyrinth

Die Bedingungen für ein gültiges Labyrinth wurden zu Beginn klar definiert. Sie spielen eine zentrale Rolle bei der Bewertung der erzeugten Labyrinthe. Ein gültiges Labyrinth muss lösbar sein, d.h. es muss einen durchgehenden Weg vom Start- zum Zielfeld geben. Die Bewertung eines Labyrinths hängt von der Länge des Weges ab, wobei längere Wege, die mehr Felder umfassen, höher bewertet werden. Ein längerer Weg stellt eine größere Herausforderung dar und führt daher zu einer besseren Bewertung des erzeugten Labyrinths.

Um die Gültigkeit zu sichern und das Risiko eines leeren Labyrinths zu minimieren, wurden spezifische Regeln für ein ungültiges Labyrinth festgelegt. Ein Labyrinth wird als ungültig betrachtet, wenn zu viele Wände zerstört wurden. Die Bedingung ist, dass jede Ecke eines Feldes mindestens eine ausgehende Wand haben muss. Zu Beginn hat jede Ecke vier ausgehende Wände, und der Agent bewegt sich über die Oberfläche, um Wände durch Berührung zu zerstören. Wenn eine Ecke keine ausgehende Wand mehr hat, wird das Labyrinth als ungültig markiert und die Generierung abgebrochen. Diese klaren Bedingungen garantieren die Qualität der erzeugten Labyrinthe und stellen sicher, dass sie sowohl lösbar als auch herausfordernd sind.

4. Implementierung

In diesem Kapitel tauchen wir in die eigentliche Implementierung des Labyrinth-Generierungssystems ein. Dabei werden die verschiedenen Aspekte der Implementierung im Detail betrachtet und die Datenstruktur der Labyrinthe sowie die Überprüfung ihrer Gültigkeit und der Einhaltung der definierten Bedingungen behandelt. Schließlich betrachten wir die Anwendungsebene, einschließlich der Kamera Ansicht und des Debug-Modus, um die Generierung und Überprüfung von Labyrinthen in einer realen Umgebung zu erleben. Dieses Kapitel bietet einen detaillierten Einblick in die technische Umsetzung des Projekts und zeigt, wie die theoretischen Konzepte in die Praxis umgesetzt wurden.

4.1 Datenstruktur

Das Kapitel zur Datenstruktur wurde in zweidimensionale und dreidimensionale Aspekte unterteilt. Es gibt zwei grobe Ansätze, wie die Daten des Labyrinths gehalten werden können. Der erste Ansatz bestand darin, eine dreidimensionale Matrix zu verwenden, in der jedes Element entweder eine Wand oder eine begehbare Zelle repräsentiert. Das Innere des Würfels wäre ebenfalls durch nicht begehbare Felder dargestellt. Da diese Felder jedoch niemals vom Agenten besucht werden können, da sich die Generierung nur auf der Oberfläche abspielt, würden diese Daten umsonst gespeichert werden, was die Datenstruktur ineffizient macht, vor allem bei größeren Würfeln.

Der zweite Ansatz hingegen bestand darin, die Daten in Form von sechs zweidimensionalen Matrizen zu organisieren, wobei jedes 2D-Gitter (*Grid*) eine der sechs Seiten des Würfels darstellt. Trotz der erhöhten Komplexität habe ich mich für den zweiten Ansatz entschieden, da er eine übersichtlichere Struktur bot und es ermöglichte, die Zusammenhänge zwischen den Zellen und Wänden in einer natürlicheren Weise darzustellen. Dieser Ansatz erforderte zwar eine aufwendigere Implementierung, weil für jede Wand bestimmt werden musst, welche beiden Zellen sie verbindet und in welche Richtung sie zeigt, und für jede Ecke festgelegt werden musste, welche vier Wände von ihr ausgehen. Doch letztendlich ermöglichte diese Herangehensweise eine effizientere und intuitivere Anwendung der Datenstruktur. Der Würfel, auf dem das Labyrinth generiert wird, wurde in sechs Seiten unterteilt, wobei jedes Gitter die Zellen, Wände und Ecken für diese Seite beinhaltet.

Zuerst wird der Aufbau eines solchen Gitters im Abschnitt 4.1.1 beschrieben, und anschließend erfolgt die Zusammensetzung zu einer dreidimensionalen Struktur, die alle Seiten des Würfels umfasst, wie in Abschnitt 4.1.2 erläutert. Dies spiegelt den eigentlichen Ablauf der Implementierung wider, da ich ebenfalls zu-

erst ein zweidimensionales Gitter erzeugt habe und die Struktur mit einem manuellen Agenten getestet habe. Anschließend habe ich diese Datenstruktur in eine dreidimensionale Struktur überführt, um sie für ein Labyrinth auf einer Würfel Oberfläche nutzen zu können.

Bevor ich mit der Implementierung angefangen habe, habe ich die Datenstruktur zunächst modelliert. Dabei habe ich besonderen Fokus auf die späteren Anwendungsfälle gelegt. Mir war es sehr wichtig, dass Funktionen, wie das *Wege finden* oder das *Prüfen auf Gültigkeit* besonders effizient aufgerufen werden können.

Zu Beginn Modellierung stand die Entscheidung, ob ein Array verwendet werden sollte, das nur die begehbaren Zellen und die nicht begehbaren Zellen (Wände) enthält, oder eins, das nur die begehbaren Zellen mit den Referenzen zu ihren angrenzenden Wänden und Ecken beinhaltet. Letztendlich habe ich mich für die zweite Option entschieden, da sie ermöglicht, eine klarere und effizientere Datenstruktur zu erstellen. In dieser Struktur sind Ecken, Wände und Zellen jeweils als separate Entitäten definiert.

Die Nachbarschaftsverhältnisse zwischen den Zellen sind von entscheidender Bedeutung, insbesondere für die Implementierung der Breitensuche in der Funktion *Wege finden*. Die Erreichbarkeit einer Nachbarzelle wird definiert durch das nicht-Vorhandensein einer Wand in der jeweiligen Richtung.

Für das Validieren eines Labyrinths sind Informationen über den zusammenhängenden Pfad und die besuchten Zellen von großer Bedeutung. Eine wichtige Kennzahl ist die Länge des Pfades, die den Fortschritt der Labyrinthgenerierung verfolgt.

4.1.1 Zweidimensionales Gitter

Das *Grid* besteht aus einem 2D Array, welches wiederum aus Zellen besteht und die Grundlage für das gesamte Labyrinth darstellt. Dieses Array hat eine Größe von *Size x Size*, wobei *Size* die Anzahl der Zellen pro Zeile und Spalte im Labyrinth darstellt. Die Grafik 4.1 unten zeigt die grundlegende Struktur eines *Grids*.

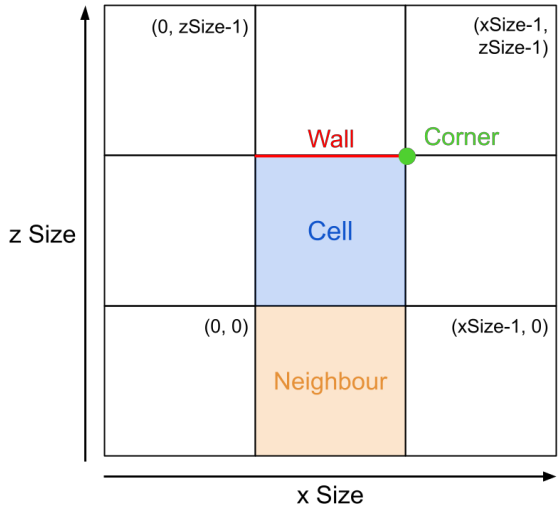

Abbildung 4.1: Die Datenstruktur eines Grids mit *Size x Size* Zellen.

MazeCell Jede Zelle innerhalb dieses Arrays ist eindeutig durch ihre x- und z-Koordinaten positioniert und gehört zu einem bestimmten *Grid*, das eine Seite des Würfels repräsentiert. Jede Zelle hat eine Liste von Nachbarzellen, die oben, unten, links und rechts an sie angrenzen können. Je nach Position im Labyrinth können Zellen 2 bis 4 Nachbarn haben (in einem 2D-Labyrinth) oder immer 4 (in einem 3D-Labyrinth).

Jede Zelle speichert eine Liste von Wänden, die oben, unten, links und rechts von ihr stehen können. Hierbei können Zellen zwischen 0 und 4 Wände haben, je nachdem, wie viele ihrer Seiten von Wänden begrenzt sind. Ebenso verfügt jede Zelle über eine Liste von Ecken. Hierbei sind immer 4 Ecken möglich. Zusätzlich zu diesen Informationen hat jede Zelle einen Wert, der anzeigt, ob die Zelle bereits vom Agenten betreten wurde.

MazeWall Die Wände im Labyrinth werden durch den Wandtyp definiert, um ihre Ausrichtung (horizontal oder vertikal) anzugeben. Jede Wand hat immer zwei angrenzende Zellen.

MazeCorner Die Ecken im Labyrinth sind Orte, an denen vier Wände zusammenstoßen. Jede Ecke hat eine Liste von Zellen, die an sie angrenzen, und eine Liste von Wänden, die von ihr ausgehen. Hierbei sind immer 4 Zellen und 0 bis 4 Wände möglich.

Damit die Zellen zusammenhängend sind, müssen für jede Zelle ihre Nachbarn bestimmt werden. Dabei wird für jede Zelle ein Nachbar pro Kante hinzugefügt, es sei denn, die Zelle befindet sich an der Kante des Labyrinths. In Abschnitt 4.1.2 wird erläutert wie diese Fälle auf dem Würfel behandelt werden. Zum Beispiel wird für eine Zelle, die sich nicht an der linken Kante befindet (d. h. der x-Wert ist größer als 0), die Zelle links davon als Nachbar hinzugefügt. Dieser Vorgang wird für jede Zelle und in alle Richtungen durchgeführt, um die Nachbarschaftsbeziehungen korrekt zu definieren.

```
1     if (x > 0)
2     {
3         cell.Neighbours.Add(Cells[x - 1, z]);
4     }
```

Die Initialisierung der Wände erfolgt, indem für jede Zelle Wände platziert werden, wo keine Außenwand ist und noch keine Wand existiert. Das Wand-Prefab wird instantiiert und entsprechend seiner Ausrichtung (links, rechts, oben, unten) positioniert. Basierend auf der Zellengröße und dem Wandtyp (horizontal oder vertikal) wird die Wand anschließend skaliert. Für beide angrenzenden Zellen werden die Wände hinzugefügt, und beide dieser Zellen werden der Wand hinzugefügt. Schließlich wird die Wand der Liste im *Grid* hinzugefügt.

Die Initialisierung der Ecken erfolgt aufgrund der Anordnung der Zellen im Labyrinth. Jeder Ecke werden alle ihrer anliegenden Zellen hinzugefügt.

```
1     var cells = new List<MazeCell>
2     {
3         Cells[x, z],
4         Cells[x + 1, z],
```

```
            Cells[x, z + 1],
            Cells[x + 1, z + 1]
        };
```

Dann werden alle Wände hinzugefügt, die zwischen zwei der zuvor definierten Zellen liegen. Dies führt zur korrekten Zuordnung von Zellen und Wänden zu den Ecken.

```
        walls.Add(Walls.Find(x => x.Cells.Contains(cells[0]) &&
                                  x.Cells.Contains(cells[1])));
        walls.Add(Walls.Find(x => x.Cells.Contains(cells[0]) &&
                                  x.Cells.Contains(cells[2])));
        walls.Add(Walls.Find(x => x.Cells.Contains(cells[1]) &&
                                  x.Cells.Contains(cells[3])));
        walls.Add(Walls.Find(x => x.Cells.Contains(cells[2]) &&
                                  x.Cells.Contains(cells[3])));
```

Diese zweidimensionale Datenstruktur ermöglicht eine effiziente Verwaltung der Labyrinthinformationen. Sie ermöglicht es, die Zusammenhänge zwischen den Zellen, Wänden und Ecken auf eine intuitive Weise darzustellen.

Durch das Platzieren von Wänden wird eine physische Struktur des Labyrinths erzeugt. Das Gitter mit den Zellen, die jeweils von vier Wänden umschlossen werden, ist in Abbildung 4.2 zu sehen.

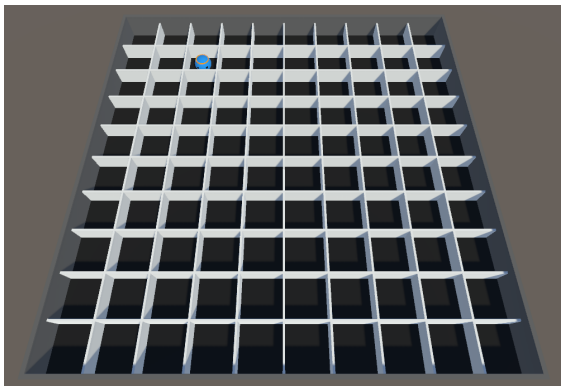

Abbildung 4.2: Ein neu initialisiertes *Grid*.

4.1.2 Dreidimensionaler Würfel

Die Implementierung der dreidimensionalen Datenstruktur für das Labyrinth erfolgte nachdem der ursprüngliche 2D-Ansatz erfolgreich umgesetzt wurde. Für die 3D-Version wurde die gleiche Datenstruktur *Grid* verwendet, jedoch mit sechs *Grids*, eines für jede Seite des Würfels. Die *Grids* habe ich in einem Dictionary gespeichert. Auf diese Weise kann das entsprechende *Grid* mit der Würfelseite als *Key* aufgerufen werden.

```
    public Dictionary<ECubeFace, Grid> Grids { get; private set; }
```

Eine der herausfordernden Aufgaben bestand darin, die korrekte Positionierung und Rotation der Grids sicherzustellen. Im ursprünglichen 2D-Konzept war es einfach, da die Zellen im *Grid* bei (0,0) beginnen und sich von dort aus nach rechts und hinten erstrecken. In der 3D-Umgebung gestaltete sich dies schwieriger, da das Grid-Objekt zunächst um seinen Pivotpunkt gedreht und positioniert wurde,

anstatt um den tatsächlichen Mittelpunkt. Dies führte zu einer komplexeren Berechnung der Positionen. Um dieses Problem zu lösen, habe ich die Visualisierung der Zellen (definiert durch die vier Wand-Objekte im Raum) angepasst, sodass sich die Zellen in alle vier Richtungen jeweils um die Hälfte der Größe des *Grids* erstrecken.

Ein weiteres Problem bestand darin, die Nachbarschaftsbeziehungen über die Kanten des Würfels hinaus korrekt zu bestimmen. In 2D hatte eine Zelle an der Kante beispielsweise nur drei Nachbarn, während sie an einer Ecke nur zwei hatte. In der 3D-Version musste jedoch berücksichtigt werden, dass eine Zelle an der Kante einen Nachbarn in einem benachbarten Grid auf der gegenüberliegenden Seite der Kante haben soll. Dies erforderte eine sorgfältige Koordinationsarbeit, die durch die Erstellung eines physischen 3D-Modells des Würfels aus Papier unterstützt wurde. Folgender Cod-Auschnitt zeigt den Fall, wenn eine Zelle an der linken Ecke des *Grids* liegt.

```
1     else if (x == 0)
2     {
3         var neighbourCell = FindNeighbourCellFromNeighbourGrid(
4             cell, EDirection.Left);
5         cell.AddNeighbour(neighbourCell);
6     }
```

Die Funktionen *GetPositionFromCell()* und *GetCellFromPosition()* stellten ebenfalls Herausforderungen dar. In der 3D-Version musste eine zusätzliche Achse berücksichtigt werden, abhängig von dem Grid, in dem sich der Agent befindet. Die Lösung bestand darin, dass der Agent das Grid, in dem er sich gerade befand, speicherte. Dadurch konnte die lokale Position mithilfe einer einfachen Berechnung ermittelt werden.

Einige Anpassungen erwiesen sich wiederum als unkompliziert, da sie auf der Eigenschaft basieren, dass das *Grid* mit seinen Zellen zusammenhängend ist. Die Funktionen *Wege finden* funktionierten dank des Breitensuche-basierten Ansatzes auch in der 3D-Version reibungslos, sobald die Nachbarn korrekt initialisiert werden.

Die Funktion zur Prüfung der Gültigkeit und die zur Überprüfung der Bedingungen waren ebenfalls einfach zu übertragen, da sie verschiedene Punkte in der Datenstruktur überprüften, die in beiden Versionen verankert waren.

4.2 Labyrinth generieren

Im Folgenden wird eine erste Umsetzung der Labyrinthgenerierung nach dem zuvor besprochenen Konzept vorgestellt. Es handelt sich um eine Fläche mit einem Wandgitter, das später eine Seite des Würfels bilden wird. Zunächst habe ich den Agenten als Kugel dargestellt. Ähnlich wie in dem *"MoveToGoal"*-Tutorial aus Kapitel 2.2 werden die horizontalen und vertikalen Eingaben in Kräfte in x- und z-Richtung übersetzt. Später wurde der Agent als Würfel dargestellt. Dieser Würfel hat keine physikalischen Eigenschaften und wird anders bewegt (siehe Kapitel 5.1). Dies macht einen Unterschied für die Bewegung auf der Oberfläche eines dreidimensionalen Körpers - dem Würfel-Labyrinth.

Platzieren des Agenten Der Agent wird zufällig auf eine der Zellen des Würfels gesetzt, die als Startfeld markiert wird.

Wände zerstören Wenn der Agent eine Wand berührt, wird diese Wand zerstört und die beiden angrenzenden Zellen werden als *besucht* markiert, wodurch ein Weg entsteht. Auf diese Weise arbeitet sich der Agent vom Start- zum Zielfeld vor. Die folgende Abbildung 4.3 zeigt, wie sich der Kugel-Agent über die Oberfläche bewegt und durch Berührung der Wände einen Weg erstellt.

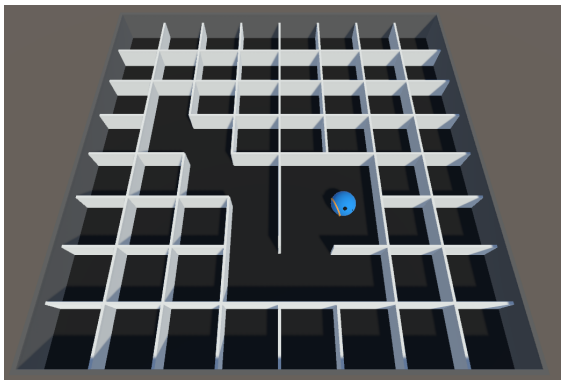

Abbildung 4.3: Ein Agent bewegt sich in einem 2D-Wandgitter.

Platzieren von Start und Ziel Das Startfeld befindet sich in der Zelle, in der der Agent zu Beginn platziert wird. Das Zielfeld kann vom Agenten zu einem beliebigen Zeitpunkt platziert werden, unter der Einschränkung, dass nur ein Ziel platziert werden kann und dass dieses nicht mit dem Startfeld übereinstimmen darf. Sobald der Agent das Ziel platziert hat, wird das Labyrinth auf die Einhaltung der festgelegten Bedingungen überprüft, wie es im Kapitel 4.3.2 beschrieben wird. In Abbildung 4.4 ist ein Agent zu sehen, der von dem grünen Startpunkt aus einen Weg gebildet hat und dann das rote Zielfeld platziert hat.

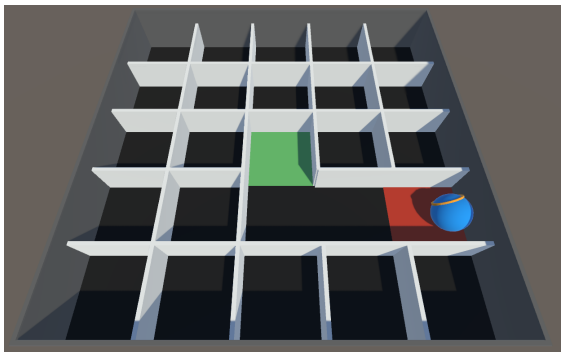

Abbildung 4.4: Ein Wandgitter, in dem Start- und Zielfeld vom Agenten platziert wurden.

Pfad anzeigen Als Debug-Option kann der generierte Pfad mit den zugehörigen Feldern angezeigt werden. Die Felder, die den kürzesten Weg vom Start zum Ziel darstellen, werden gelb markiert und ermöglichen so eine visuelle Darstellung des generierten Labyrinths. Dies ist in Abbildung 4.5 visualisiert. Dieser Schritt dient

nicht nur der Überprüfung der Pfad-Generierung, sondern gibt auch Aufschluss über die Qualität des generierten Labyrinths.

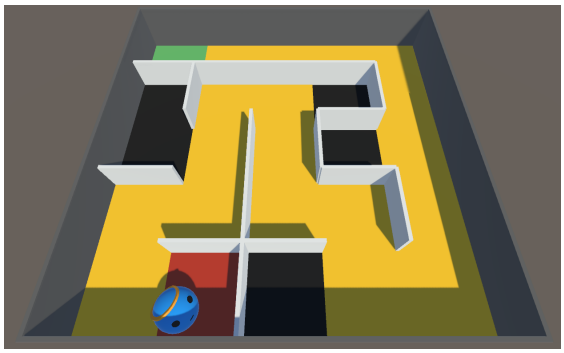

Abbildung 4.5: Der Weg vom Start zum Ziel, bestehend aus den einzelnen Feldern in gelb.

4.3 Labyrinth prüfen

Dieses Unterkapitel konzentriert sich auf die entscheidende Phase der Überprüfung des vom Agenten erzeugten Labyrinths. Der Agent hat nicht nur die Aufgabe, ein Labyrinth zu erstellen, sondern auch sicherzustellen, dass es bestimmte Kriterien erfüllt. In dieser Phase wird das Labyrinth Schritt für Schritt daraufhin überprüft, ob es die vorgegebenen Bedingungen und Anforderungen erfüllt. Ziel des Agenten ist es, ein gültiges Labyrinth zu erzeugen, das einen eindeutigen Weg vom Startpunkt zum Zielpunkt aufweist. Nur wenn dieses Ziel erreicht wird, erhält der Agent die begehrte Belohnung. Andernfalls muss er seine Strategie überdenken und das Labyrinth neu erstellen. Dieses Kapitel beleuchtet die Mechanismen und Bewertungskriterien, die verwendet werden, um die Qualität des generierten Labyrinths sicherzustellen.

4.3.1 Gültigkeit prüfen

Das Labyrinth ist abgeschlossen, sobald es einen Start- und einen Zielpunkt hat. Sobald der Agent den Zielpunkt gesetzt hat, wird das fertige Labyrinth auf Gültigkeit geprüft. Optional kann in den Einstellungen festgelegt werden, dass ein Labyrinth nur dann gültig ist, wenn alle seine Felder besucht wurden (Labyrinth ist verbunden). Dazu wird überprüft, ob jede Zelle jedes Gitters besucht wurde.

```
1     if (requireAllCellsToBeVisited)
2     {
3         // Check if all cells are visited -> maze is connected
4         if (Grids.Values.Any(grid =>
5             grid.GetVisitedCells() != grid.Cells.Length))
6         {
7             return false;
8         }
9     }
```

In der Trainingsphase hat sich diese Einstellung jedoch als ineffizient erwiesen, da der Agent nicht herausfindet, was seine eigentliche Aufgabe ist.

Für ein lösbares Labyrinth ist ein zusammenhängender Weg vom Start zum Ziel erforderlich. Um diesen Weg zu finden, habe ich die *Breitensuche* (breadth-first search) implementiert.

Wege finden Mit dieser Methode kann der kürzeste Weg vom Startfeld zum Ziel ermittelt werden. Der Algorithmus verwendet eine Warteschlange, um die noch zu überprüfenden Zellen zu verfolgen, und ein Set, um die bereits besuchten Zellen zu markieren. Während des Durchlaufs werden die Beziehungen zwischen den Zellen in einer Eltern-Kind-Beziehung gespeichert, so dass am Ende der Suche der Pfad rekonstruiert werden kann. Dabei wird Schritt für Schritt überprüft, ob es zwischen den Zellen Wände gibt, die den Pfad blockieren. Die Methode liefert eine Liste mit allen Zellen, die zum kürzesten Pfad gehören. Wenn diese Liste mehr als 0 Elemente enthält und sowohl das Start- als auch das Zielfeld enthält, handelt es sich um ein gültiges Labyrinth.

4.3.2 Bedingungen prüfen

Die Überprüfung der Bedingungen ist ein entscheidender Schritt in der Generierung des Labyrinths und wird bei jedem Schritt des Agenten durchgeführt, um sicherzustellen, dass das erstellte Labyrinth gültig bleibt. Die Bedingung bezieht sich auf die Anzahl der Wände, die von den Ecken jeder Zelle ausgehen. Um sicherzustellen, dass der Agent nicht alle Wände des Labyrinths zerstört und damit ein leeres oder unlösbares Labyrinth erzeugt, muss jede Ecke mindestens eine ausgehende Wand haben.

Die Datenstruktur des Labyrinths erleichtert diese Überprüfung erheblich. Jede Ecke in der Datenstruktur speichert, wie viele Wände von ihr ausgehen. Wenn der Agent eine Wand zerstört und damit eine Verbindung zwischen zwei Zellen herstellt, werden die entsprechenden Ecken und Wände aktualisiert.

```
1     // Check if all corners have at least one wall -> maze is not
      empty
2     if (Grids.Values.Any(grid =>
3         grid.Corners.Any(corner => corner.Walls.Count == 0)))
4     {
5         return false;
6     }
```

Diese Bedingung ist sehr wichtig, da sie sicherstellt, dass das Labyrinth eine gewisse Struktur und Herausforderung beibehält. Der Agent wird aufgefordert, ein gültiges Labyrinth zu erzeugen, in dem es immer noch Hindernisse und Wege gibt, die zu überwinden sind. Die ständige Überprüfung der Bedingungen stellt sicher, dass der Agent sein Ziel erreicht, ohne dass das Labyrinth unspielbar wird.

4.4 Die Anwendung

In diesem Abschnitt wird auf die praktische Umsetzung der Labyrinthgenerierung in der Unity-Anwendung eingegangen. Der Schwerpunkt liegt hierbei auf der Visualisierung des Labyrinths und der Bereitstellung von wichtigen Informationen

für Entwickler:innen und Nutzer:innen. Es werden die verschiedenen Kamera Ansichten vorgestellt, die es ermöglichen, das generierte Labyrinth aus verschiedenen Blickwinkeln zu betrachten.

Des Weiteren werden die Einstellungsmöglichkeiten erläutert, die vor der Generierung des Labyrinths vorgenommen werden können, um die Anpassbarkeit der Anwendung zu verdeutlichen. Außerdem wird der Debug-Modus erläutert, der zusätzliche Einblicke in die Funktionsweise des Algorithmus bietet und wertvolle Informationen zur Fehlerdiagnose und -behebung liefert.

4.4.1 Kamera Ansicht

Das Konzept der Kamera Ansicht in der Anwendung sieht vor, sechs Kameras zu verwenden, von denen jede auf eine Seite des Würfels gerichtet ist. Jede dieser Kameras erzeugt eine *RenderTexture* als Ausgabe, die dann wiederum auf sechs *RawImages* projiziert wird, um die jeweilige Ansicht auf dem Bildschirm darzustellen. Zusätzlich werden UI-Texte verwendet, um anzuzeigen, welche Seite des Würfels gerade betrachtet wird. Außerdem gibt es für jede Seite einen UI-Text, der angibt, wie viele Zellen insgesamt auf dieser Seite bereits besucht wurden, wie in folgender Abbildung 4.6 zu sehen ist.

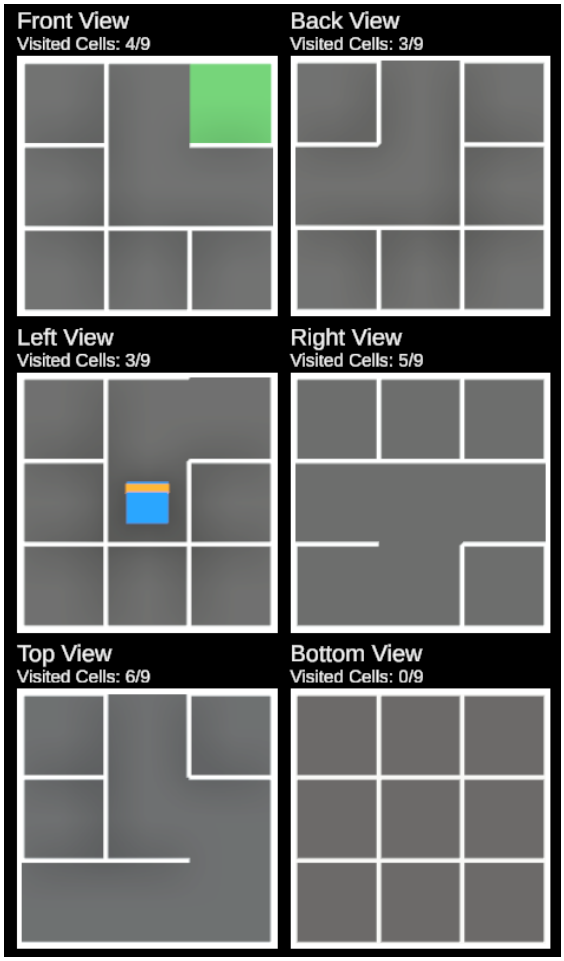

Abbildung 4.6: Die Ansicht der sechs Würfelseiten.

Um die Anwendung noch benutzerfreundlicher zu gestalten, habe ich eine

Hauptkamera implementiert, die um den Würfel rotiert und eine Gesamtansicht des Labyrinths zeigt. Dies ermöglicht eine bessere Kontrolle und Übersicht über den Prozess der Labyrinthgenerierung und die Handlungen des Agents. Die Abbildung 4.7 zeigt die Anwendung mit der Ansicht der Würfelseiten auf der rechten Seite und der rotierende Würfel auf der linken Seite.

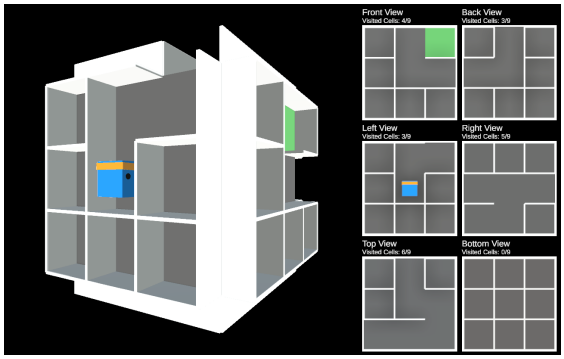

Abbildung 4.7: Die Anwendung für die Labyrinthgenerierung.

4.4.2 Labyrinth-Einstellungen

In den Einstellungen für das Labyrinth gibt es die Möglichkeit, verschiedene Parameter anzupassen, um die Visualisierung und den Prozess der Generierung sowie die Agentenaktivitäten anzupassen. Abbildung 4.8 zeigt, wo die Einstellungen in Unity zu finden sind und die einzelnen Parameter sind im Folgenden näher erläutert.

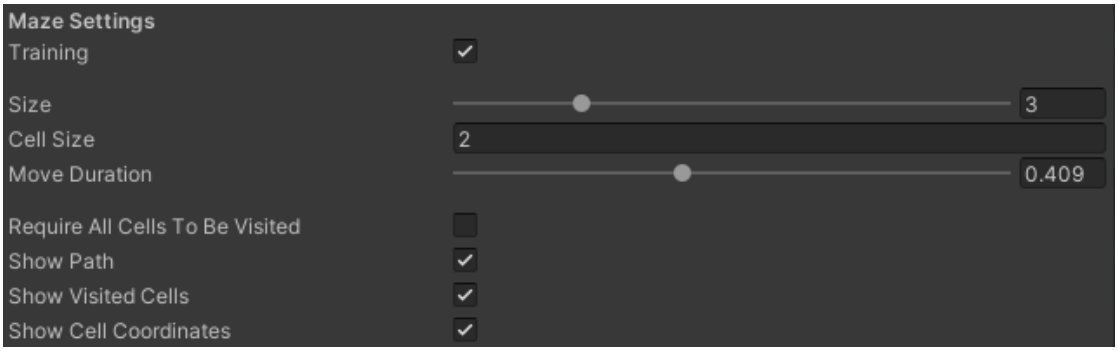

Abbildung 4.8: Die Einstellungen für ein Labyrinth.

Training Wenn diese Option aktiviert ist, wird der Agent bei Beginn jeder neuen Episode zurückgesetzt, und das Labyrinth wird erneut generiert. Wenn diese Option deaktiviert ist, wird die Szene für das Lösen mit dem zuvor generierten Labyrinth geladen.

Size Diese Einstellung bestimmt die Anzahl der Zellen in vertikaler und horizontaler Richtung auf jeder Seite des Würfels. Sie kann einen Wert zwischen 1 und 10 annehmen.

Cell Size Dieser Wert gibt die Größe jeder Zelle im Labyrinth an.

Move Duration Diese Einstellung bestimmt die Zeit, die der Agent benötigt, um von einer Zelle zur nächsten zu gelangen. Der Wert kann zwischen 0 und 1 angepasst werden.

Require All Cells To Be Visited Wenn diese Option aktiviert ist, wird das Labyrinth nur als gültig angesehen, wenn alle Zellen im Labyrinth mindestens einmal besucht wurden.

Show Path Durch Aktivieren dieser Option wird der Pfad vom Startpunkt zum Ziel im Labyrinth visualisiert.

Show Visited Cells Wenn diese Option aktiviert ist, werden die Zellen im Labyrinth gelb markiert, die der Agent bereits besucht hat. Diese Einstellung ist besonders sinnvoll für das Beobachten des Trainingsprozesses.

Show Cell Coordinates Diese Option zeigt die Koordinaten jeder Zelle auf dem Würfel an.

In Abbildung 4.9 ist zu sehen, wie die Anwendung aussieht, wenn die Koordinaten der Zellen angezeigt werden und die besuchten Zellen gelb markiert werden. Das Startfeld befindet sich hier in grün auf der Vorderseite des Würfels in der linken unteren Ecke. Der Agent bewegt sich auf der Rückseite des Würfels in Feld (3, 3).

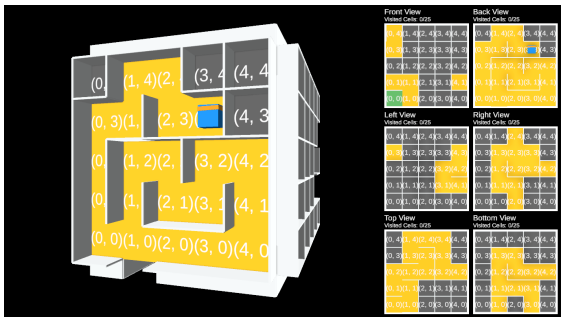

Abbildung 4.9: Die Anwendung mit einigen Debug Optionen.

5. Der Agent

In diesem Kapitel wird der Agent, der die Hauptrolle bei der Generierung von Labyrinthen spielt, im Detail betrachtet. Der Agent ist dafür verantwortlich, Labyrinthe zu erzeugen, die bestimmte Bedingungen erfüllen, und seine Aktionen werden ständig überwacht, um sicherzustellen, dass die erzeugten Labyrinthe gültig und lösbar sind. Die verschiedenen Aspekte des Agenten, seine Interaktionen mit dem Labyrinth und seine Bewertungsmechanismen werden untersucht, um ein umfassendes Verständnis seiner Funktionsweise zu entwickeln. Darüber hinaus werden wir die Trainingsmethoden und die Möglichkeiten, den Agenten zu steuern, im Detail betrachten.

5.1 Bewegung

Die Bewegung des Agenten im Labyrinth erwies sich als schwierige Aufgabe, die verschiedene Ansätze und Anpassungen erforderte, um eine optimale Kontrolle und Präzision zu erreichen. In den frühen Stadien der Entwicklung wurde zunächst versucht, den Agenten in der 3D-Umgebung ähnlich wie in der 2D-Version zu bewegen. Dabei wurden kontinuierliche Aktionen für die Kraft in x- und z-Richtung verwendet. Dies ist auf dem Würfel naturgemäß nicht möglich, da der Agent aufgrund der Schwerkraft herunterfallen würde. Ich habe die Bewegung so angepasst, dass sich der Agent in x- und z-Richtung bewegt, je nachdem, wie groß der eingegebene Wert ist. Die Steuerung des Agenten erforderte jedoch extreme Feinfühligkeit. Sowohl bei der manuellen Steuerung als auch beim Agenten gab es Schwierigkeiten und es wurden schnell zu viele Wände zerstört.

Um dieses Problem zu lösen, wurde ein zweiter Ansatz mit diskreten Aktionen eingeführt. Dabei wurden fünf Zweige verwendet, von denen jeder den Wert 0 (keine Aktion) oder 1 (Bewegung in eine bestimmte Richtung oder Platzierung eines Ziels) annehmen konnte. Dieser Ansatz ermöglichte es dem Agenten, seine Bewegung besser zu kontrollieren. Eine manuelle Steuerung über die Tasten W, A, S und D wurde als Heuristik implementiert, um Bewegungen in die entsprechenden Richtungen auszulösen.

Es stellte sich jedoch heraus, dass der Agent immer noch Schwierigkeiten hatte, da er mehrere Aktionen gleichzeitig auslösen konnte. Zum einen ist dies schwieriger zu verarbeiten, da eine Entscheidung getroffen werden muss, welche der Aktionen ausgeführt wird bzw. in welcher Reihenfolge sie ausgeführt werden. Zum anderen ist es für den Agenten schwieriger, seine Eingaben mit den Aktionen zu verknüpfen.

Schließlich wurde der endgültige Ansatz implementiert, der einen einzigen Zweig

für diskrete Aktionen verwendet, der fünf verschiedene Werte annehmen kann: nach links, nach rechts, nach oben, nach unten bewegen und Ziel platzieren. Dieser Ansatz ermöglicht eine präzise Steuerung des Agenten, da er immer nur eine Aktion ausführen kann und somit eine klare Verbindung zwischen seinen Eingaben und seinen Aktionen hergestellt werden kann. Dieser Ansatz hat sich als effektiv erwiesen, um die Bewegung des Agenten in 3D-Labyrinthen zu steuern und gleichzeitig die Anforderungen an die Labyrinthgenerierung zu erfüllen.

Die Bewegung des Agenten wird mit der Methode *MoveAgent(EDirection direction)* gesteuert. Diese Methode erlaubt es dem Agenten, sich in eine bestimmte Richtung zu bewegen. Sie prüft, ob die gewünschte Bewegung gültig ist, indem sie die nächste Zelle und das nächste *Grid* ermittelt. Wenn die Bewegung gültig ist, wird der Agent an die neue Position bewegt.

Um die Bewegung flüssiger zu gestalten, wird eine *Coroutine* aufgerufen, die den Agenten schrittweise zur Zielposition bewegt. Während der Bewegung wird überprüft, ob sich der Agent in der Mitte der Bewegung befindet, um gegebenenfalls Wände zwischen der aktuellen und der nächsten Zelle zu entfernen. Nach Beendigung der Bewegung wird sichergestellt, dass sich der Agent exakt an der Zielposition befindet und die Variable *AgentIsMoving* auf *false* gesetzt. gesetzt. Diese Variable stellt sicher, dass der Agent während der Bewegung keine weiteren Eingaben vornehmen kann. Wenn die Bewegung abgeschlossen ist, wird die *currentCell* des Agenten auf die neue Zelle gesetzt und wenn der Agent während seiner Bewegung die Würfelseite gewechselt hat, wird die neue Seite als *currentGrid* gesetzt.

Die Implementierung der Agentenbewegung begann zunächst mit der direkten Steuerung innerhalb der Agentenklasse. Dabei wurde die Unity-Funktion *GetKey* verwendet, um festzustellen, ob eine bestimmte Taste gedrückt ist. Es stellte sich jedoch heraus, dass diese Methode die Taste so lange als gedrückt erkannte, wie sie gehalten wurde. Dies führte zu unerwünschten Effekten bei der manuellen Steuerung, da sich der Agent mehrere Felder weit bewegte, selbst wenn die Taste nur kurz gedrückt wurde.

Dieses Problem wurde zunächst durch die Einführung der Variablen *AgentIsMoving* gelöst, die Eingaben während der Bewegung abfing. Diese Lösung funktionierte jedoch nicht, wenn die Dauer der Bewegung auf Null gesetzt wurde. Um dieses Problem zu lösen, wurde die Unity-Funktion *GetKeyDown* verwendet, die *true* zurückliefert, wenn die Taste im ersten Frame gedrückt wird, und danach auf *false* zurückgesetzt wird. Die Verwendung dieser Eingabe erwies sich jedoch als ungenau, da die Funktion *Heuristik* nicht in jedem Frame aufgerufen wird, sondern intern von den ML-Agenten gesteuert wird. Um dieses Problem zu lösen, wurde die manuelle Steuerung in eine eigene Klasse ausgelagert. In der Update-Funktion dieser Klasse werden die entsprechenden Eingaben definiert und in der Agentenklasse lediglich in diskrete Aktionen übersetzt. Dies führte zu einer präziseren und kontrollierbareren manuellen Steuerung des Agenten im Labyrinth.

5.2 Beginn einer Episode

Die Funktion *OnEpisodeBegin* markiert den Startpunkt einer neuen Episode und wird zu Beginn jeder Episode aufgerufen. Ihr Hauptzweck besteht darin, die Umgebung und den Agenten zurückzusetzen, um sicherzustellen, dass der Agent die Aufgabe in einem sauberen Anfangszustand beginnen kann. Im Laufe der Entwicklung haben sich die Parameter, die bei diesem Aufruf zurückgesetzt werden, geändert. Folgend wird aufgeführt, welche Parameter zurückgesetzt werden.

Der Parameter *ManualInput* wird auf *EManualInput.None* zurückgesetzt, um sicherzustellen, dass der Agent zu Beginn einer Episode keinen Input hat.

Außerdem wird die Umgebung vorbereitet, indem das alte Labyrinth gelöscht und ein neues erzeugt wird (siehe Abschnitt 4.1). Dadurch wird sichergestellt, dass der Agent mit einem frischen Labyrinth startet. Auch der Agent selbst wird entsprechend platziert, um die Aufgabe in einer sauberen Umgebung zu starten.

Der Agent startet jede Episode an einer neuen zufälligen Position im Labyrinth. Dazu wird zunächst ein zufälliges *Grid* aus den verfügbaren *Grids* ausgewählt. Dieses legt fest, auf welcher Seite des Würfels der Agent platziert wird.

```
1     // get random grid from Grids
2     var gridIndex = Random.Range(1, Enum.GetValues(typeof(
      ECubeFace)).Length);
3     var cubeFace = (ECubeFace)gridIndex;
4     var grid = Grids[cubeFace];
```

Nachdem die Seite ausgewählt wurde, wird eine zufällige Position innerhalb dieses *Grids* bestimmt. Dies geschieht durch die Auswahl zufälliger X- und Z-Koordinaten innerhalb der *Grid*-Größe. Anschließend wird die Zufallszelle anhand der Koordinaten im ausgewählten *Grid* bestimmt. Die Position des Agenten wird dann auf die Position dieser Zelle gesetzt, wobei die Y-Koordinate auf 0,5 gesetzt wird, um sicherzustellen, dass der Agent auf einer geeigneten Höhe platziert wird.

```
1     var randomX = Random.Range(0, size);
2     var randomZ = Random.Range(0, size);
3     var randomCell = grid.Cells[randomX, randomZ];
4     var position = grid.GetPositionFromCell(randomCell);
5     position.y = 0.5f;
```

Der Agent wird dem ausgewählten *Grid* als Kindobjekt zugewiesen, um sicherzustellen, dass er sich in diesem *Grid* befindet. Die Position des Agenten wird innerhalb des ausgewählten *Grids* gesetzt und die aktuellen Gitter- und Zelleninformationen werden im Agentenobjekt aktualisiert, so dass der Agent weiß, in welchem *Grid* und in welcher Zelle er sich befindet.

```
1     agent.transform.SetParent(grid.Parent.transform);
2     agent.transform.localPosition = position;
3     var mazeAgent = agent.GetComponent<MazeGenerationAgent>();
4     mazeAgent.CurrentGrid = grid;
5     mazeAgent.CurrentCell = randomCell;
```

Schließlich wird die Startzelle entsprechend markiert, um den Beginn des Lösungswegs festzulegen. Außerdem wird die Zufallszelle als *besucht* markiert.

5.3 Beobachtungen

Bei der Implementierung der Beobachtungen für den Agenten wurden verschiedene Ansätze ausprobiert, um sicherzustellen, dass der Agent genügend Informationen erhält, um die Aufgabe zu lösen. Unity selbst sagt, dass eine gute Faustregel für die Entscheidung, welche Informationen gesammelt werden sollten, darin besteht, sich zu überlegen, welche Informationen benötigt würden, um eine analytische Lösung des Problems zu berechnen. [1] Nachfolgend sind die Schritte und Erkenntnisse aufgeführt, die während dieses Prozesses gesammelt wurden:

Zunächst wurden nur Informationen über die Zelle, auf der sich der Agent gerade befindet, sowie deren Nachbarn, Wände und Ecken, jeweils mit der Anzahl der Wände, als Beobachtungen berücksichtigt. Dies führte jedoch dazu, dass der Agent während des Trainings nicht besser, sondern sogar schlechter in der Bewältigung der Aufgabe wurde.

In einem weiteren Schritt wurden auch die Nachbarzellen der Agentenzelle mit all ihren Informationen als Beobachtungen einbezogen. Dies führte jedoch nicht zu einer Verbesserung im Trainingsverlauf.

Eine entscheidende Erkenntnis war, dass der Agent Informationen über die gesamte Datenstruktur des Labyrinths benötigt. Dies wurde besonders deutlich, als die Größe des Labyrinths auf 1 reduziert wurde, so dass nur noch ein Feld pro Würfelseite zur Verfügung stand. In diesem Fall konnte der Agent die Aufgabe nahezu perfekt lösen, da er aufgrund der Nachbarschaftsbeziehungen wusste, wie die gesamte Datenstruktur aussah.

Daraus ergaben sich zwei wichtige Schlussfolgerungen:

1. Der Agent benötigt Informationen über die gesamte Datenstruktur des Labyrinths, um die Aufgabe erfolgreich zu lösen.

2. Die Aufgabe wird für den Agenten deutlich komplexer und schwieriger, wenn das Labyrinth größer wird, da er dann mehr Informationen verarbeiten und eine bessere Strategie entwickeln muss.

Um diesen Anforderungen gerecht zu werden, werden schließlich alle Zellen des Labyrinths mit all ihren Informationen als Beobachtungen für den Agenten betrachtet.

Der Agent führt eine beträchtliche Anzahl von Beobachtungen durch. Jede Zelle des Labyrinths wird betrachtet und für jede dieser Zellen werden Informationen über die Würfelseite, die Zellenposition, den Besuchsstatus und die Anzahl der Wände an den Ecken der Nachbarzellen berechnet. Zusätzlich werden Informationen über die aktuelle Zelle des Agenten, die Startzelle, die aktuelle Pfadlänge im Labyrinth vom Start bis zur aktuellen Zelle, der prozentuale Pfadfortschritt im Verhältnis zur Gesamtzahl der Zellen im Labyrinth und die besuchten Zellen im Labyrinth gesammelt. Dies stellt sicher, dass der Agent genügend Informationen erhält, um die Aufgabe effizient zu lösen und das Labyrinth entsprechend den Anforderungen zu generieren.

[1] https://unity-technologies.github.io/ml-agents/Learning-Environment-Create-New/

5.4 Aktionen und Bewertung

In der Funktion *OnActionReceived* wird eine Aktion des Agenten auf der Grundlage der Entscheidung des neuronalen Netzes ausgeführt und anschließend die Umgebung bewertet.

Aktionen Der Agent hat die Möglichkeit, eine von fünf Aktionen auszuführen, die in den diskreten Aktionen des Entscheidungsvektors kodiert sind. Für die manuelle Steuerung gibt es eine sechste Option, nichts zu tun. Die sechs möglichen Aktionen sind folgende:

0: Bewegung nach links (*EDirection.Left*)

1: Bewegung nach rechts (*EDirection.Right*)

2: Bewegung nach oben (*EDirection.Top*)

3: Bewegung nach unten (*EDirection.Bottom*)

4: Ziel in der aktuellen Zelle platzieren (*Maze.PlaceGoal(CurrentCell)*)

5: Keine Aktion (nur bei manueller Steuerung)

Bewertung Die Belohnungen werden auf Basis der durchgeführten Aktionen und der aktuellen Umgebungsbedingungen vergeben. Im Folgenden werden die Belohnungsaspekte erläutert.

Erfolgreiches Labyrinth: Wenn der Agent ein gültiges Labyrinth erstellt, wird die Belohnung anhand des Prozentsatzes der Länge des Weges im Verhältnis zur Gesamtzahl der Zellen berechnet. Je mehr Zellen der kürzeste Weg vom Start zum Ziel enthält, desto höher ist die Belohnung. Ein Pfad, der alle Zellen enthält, wird mit 10.0 belohnt, während ein ungültiges Labyrinth mit -1.0 bestraft wird.

Nichterfüllung der Anforderungen: Wenn der Agent die Anforderungen für ein gültiges Labyrinth nicht erfüllt, wird er bestraft und das Training wird beendet. Dies soll sicherstellen, dass der Agent nicht einfach alle Wände entfernt und ein leeres Labyrinth erstellt.

Existenzielle Strafe: Um sicherzustellen, dass der Agent nicht in einer Endlosschleife stecken bleibt, wird für jede Aktion ohne sichtbaren Fortschritt eine kleine Strafe (-0,00025) vergeben. Wenn die kumulative Belohnung unter -1.0 fällt, wird die Episode beendet, um den Agenten aus einer möglichen Blockade zu befreien.

Beim Training des Agenten wurden verschiedene Bewertungsstrategien ausprobiert. Insgesamt wurde für die komplexe Aufgabe der Labyrinthgenerierung festgestellt, dass es effektiver ist, den Agenten auf das Hauptziel der Aufgabenerfüllung zu fokussieren. Daher wurde der Bewertungsansatz auf einfache und klare Belohnungen umgestellt, was zu besseren Trainingsergebnissen und einem effizienteren Lernen des Agenten führte.

6. Training

Kapitel 6 bietet einen Einblick in den Trainingsprozess des Agenten und dessen Optimierung. Hier werden die Trainingsschleifen, Hyperparameter-Abstimmungen und Erfahrungen bei der Verbesserung des Agenten vorgestellt. Es zeigt, wie maschinelles Lernen und neuronale Netzwerke eingesetzt werden, um den Agenten zu befähigen, komplexe Aufgaben wie das Generieren von Labyrinthen erfolgreich zu bewältigen.

6.1 Reinforcement Learning

Im Zusammenhang mit der Entwicklung des Labyrinthgenerierenden Agenten ist das Kapitel über *Reinforcement Learning (RL)* von zentraler Bedeutung. RL ist ein Ansatz des maschinellen Lernens, bei dem ein Agent zufällige Aktionen ausführt und auf Belohnungen oder Bestrafungen in seiner Umgebung reagiert, um seine Aktionen mit der Zeit zu optimieren.

Während der Entwicklung des Agenten wurden verschiedene Parameter seines Verhaltens intensiv angepasst und ausgiebig getestet, um das bestmögliche Trainingsergebnis zu erzielen. Ein Schlüsselaspekt war die Gestaltung der Beobachtungen, die dem Agenten zur Verfügung stehen. Um erfolgreich zu sein, benötigt der Agent ausreichend aussagekräftige Informationen über seine Umgebung und seine Position darin.

Ein weiterer entscheidender Faktor war die Gestaltung der Bewertungen, die der Agent während seiner Handlungen erhält. Angesichts der Komplexität der Aufgabe war es entscheidend, klare und relevante Bewertungskriterien zu schaffen.

In den ersten Versuchen war der Agent jedoch nicht sehr erfolgreich. Er startete mit einer negativen Durchschnittsbewertung und verschlechterte sich im Laufe des Trainings weiter.

Die Trainingsverläufe wurden detailliert im *Tensorboard* visualisiert. Eine der zentralen Statistiken war der kumulative Belohnungswert (*Cumulative Reward*), der die durchschnittliche kumulative Episodenbelohnung aller Agenten darstellt. Dieser sollte während einer erfolgreichen Trainingssitzung kontinuierlich ansteigen. Dies war jedoch nicht der Fall, wie die folgenden Grafiken 6.1 und 6.2 zeigen.

Abbildung 6.1: Statistik der kumulativen Belohnung.

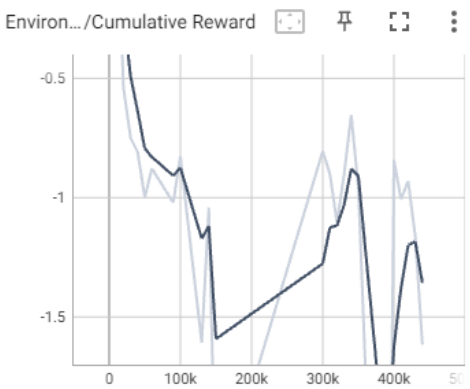

Abbildung 6.2: Statistik der kumulativen Belohnung.

Um die Trainingsergebnisse zu verbessern, wurden nicht nur die Parameter innerhalb der Agentenklasse angepasst, sondern auch die Hyperparameter in der YAML-Datei optimiert. Eine wichtige Änderung war die Reduzierung der Lernrate (*learning_rate*), um die Stabilität des Trainings zu erhöhen. Die Lernrate beschreibt die Stärke bzw. Größe der Aktualisierungsschritte, die bei der Anpassung der Modellparameter durchgeführt werden. Die Lernrate bestimmt, wie stark die Parameter nach jeder Berechnung des Gradienten der Verlustfunktion angepasst werden. Ein hoher Wert der Lernrate führt zu großen Schritten und schnellen Änderungen der Parameter, während ein niedriger Wert der Lernrate zu kleinen Schritten und langsamen Änderungen führt. [1]

Ein weiterer Schritt zur Verbesserung des Trainings war die Erhöhung des *Beta-Wertes*. Beta steuert die Stärke der Entropie-Regularisierung, die die Strategie des Agenten ßufälliger"macht und sicherstellt, dass der Aktionsraum während des Trainings angemessen erkundet wird. Eine Erhöhung des Beta-Wertes führt zu mehr zufälligen Aktionen. [2]

Durch diese umfangreichen Iterationen und Anpassungen zeigte der Agent schließlich eine deutliche Verbesserung seiner Leistung. Wie die folgende Statistik 6.3 zeigt, verbesserte sich die durchschnittliche Bewertung des Agenten während des Trainings kontinuierlich und erreichte nach 1.240.000 Schritten den beeindruckenden Wert von 3.146.

[1] https://github.com/yosider/ml-agents-1/blob/master/docs/Training-PPO.md
[2] https://github.com/yosider/ml-agents-1/blob/master/docs/Training-PPO.md

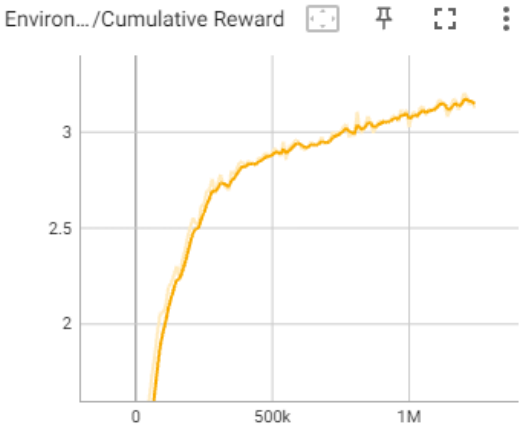

Abbildung 6.3: Statistik der kumulativen Belohnung.

Dies unterstreicht die Bedeutung einer sorgfältigen Anpassung von Beobachtungen, Bewertungen und Hyperparametern bei komplexen RL-Aufgaben.

6.2 Imitation Learning

Imitation Learning, zu Deutsch "Lernen durch Nachahmung", stellt einen interessanten Ansatz im Bereich der Labyrinthgenerierung dar. Bei dieser Methode versucht der Agent, das Verhalten eines Demonstrators zu kopieren. Die Generierung eines Labyrinths ist eine sehr anspruchsvolle Aufgabe, bei der viele Faktoren berücksichtigt werden müssen. Jedes Feld kann vier mögliche Wege haben, und die Wahl eines jeden Weges kann zu einem vorzeitigen Abbruch führen, wenn die beiden Eckpunkte der Wände, die durchbrochen werden, nur noch jeweils eine Wand haben. Darüber hinaus muss der Agent den optimalen Ort für das Ziel finden, wobei er verstehen muss, dass der Weg zum Ziel so lang wie möglich sein sollte, um eine gute Bewertung zu erhalten. Dies erfordert wiederum, dass viele Zellen besucht werden, aber nicht zu viele Wände zerstört werden.

Die Wahrscheinlichkeit, dass der Agent von sich aus ein "perfektes"Labyrinth generiert (wobei "perfekt"bedeutet, dass der generierte Weg zum Ziel so lang wie möglich ist), ist aufgrund der Komplexität der Aufgabe und der Tatsache, dass die Ausgangsbedingungen in jeder Episode unterschiedlich sind, äußerst gering. Daher wurde der Ansatz des *Imitation Learning* in Betracht gezogen. Hierbei kann dem Agenten gezeigt werden, wie die Aufgabe richtig gelöst wird. Dieses Verfahren erfordert jedoch, dass der Agent die Demonstration sehr oft beobachtet, was sehr zeitaufwendig ist. In der YAML-Datei kann festgelegt werden, wie stark die Demonstration das Verhalten des Agenten beeinflussen soll. Der Agent orientiert sich an der Demonstration, versucht aber gleichzeitig, die Aufgabe selbstständig zu lösen und sogar zu perfektionieren.

Nach umfangreichen Tests stellte sich jedoch heraus, dass der Agent mit *Imitation Learning* nicht die gewünschten Ergebnisse erzielt, wie in der folgenden Abbildung 6.4 visualisiert.

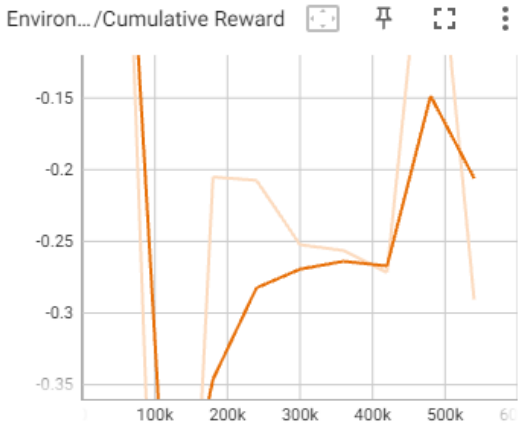

Abbildung 6.4: Statistik der kumulativen Belohnung.

Es ist zu vermuten, dass die Aufgabe aufgrund ihrer Komplexität und der variablen Ausgangsbedingungen in jeder Episode zu schwierig ist. Der Agent kann möglicherweise nicht genau bestimmen, welche Aktionen zum Erfolg führen. Aus diesem Grund wurde der Schwerpunkt auf das autonome Training des Agenten in seiner Umgebung verlagert, um herauszufinden, wie seine Aktionen die Umgebung beeinflussen und welche Veränderungen sie bewirken können.

7. Zusammenführen der Agenten

Die Zusammenführung der beiden Agenten, *MazeGeneration* (in diesem Bericht behandelt) und *MazeSolving*. [1], stellte eine spannende Herausforderung dar, da ihre internen Datenstrukturen sehr unterschiedlich sind. Während *MazeSolving* ein 3D-Array namens "Maze"verwendet, das aus Würfeln besteht, die Wände darstellen können oder nicht, verwendet *MazeGeneration* sechs 2D-Arrays, eines für jede Seite des Würfels. Jedes dieser Gitter, auch *Grids* genannt, ist mit Zellen gefüllt, die jeweils 0 bis 4 Wände und 4 Ecken haben. Um diese beiden unterschiedlichen Darstellungen zu vereinen, wurde eine Datenstruktur namens *SharedMaze* entwickelt.

Füllen des 3D-Würfel-Arrays Jede Zelle in meinem *MazeGeneration*-Labyrinth wird durch neun Würfel in der *MazeSolving*-Szene repräsentiert. Diese neun Würfel repräsentieren verschiedene Eigenschaften der Zelle. Sie werden wie folgt gefüllt:

Für jede Zelle wird ein Würfel mit der entsprechenden Position platziert. Diese Position variiert je nach Seite und Ausrichtung des Cubes. Dieser Würfel stellt dar, dass die Zelle keine Wand ist. Für jede Zelle werden vier Wand-Cubes gesetzt, zwei oben/unten und zwei links/rechts von der Zelle. Diese Cubes sind mit *isWall* gekennzeichnet, wenn in der jeweiligen Richtung eine Wand vorhanden ist. Ebenso werden für jede Zelle vier Eckwürfel hinzugefügt. Für diese Eckwürfel wird *isWall* immer auf *true* gesetzt. Zusätzlich wird geprüft, ob es sich bei einer Zelle um den Anfangs- oder Endpunkt des Labyrinths handelt und entsprechend im 3D-Cube-Array markiert. Bei der Platzierung der Würfel wird geprüft, ob an der betreffenden Position im Würfelarray bereits ein Würfel vorhanden ist. Dies ist notwendig, da sich zwei Zellen immer eine Wand und vier Zellen immer eine Ecke teilen. Dadurch wird sichergestellt, dass Wände und Ecken nur einmal hinzugefügt werden. Die Überführung von der *MazeGeneration* Datenstruktur in die *SharedMaze* Datenstruktur ist in der Abbildung 7.1 dargestellt.

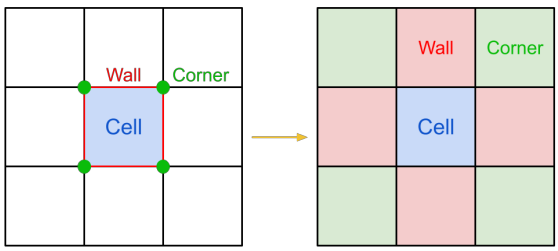

Abbildung 7.1: Umwandeln der *MazeGeneration* Datenstruktur in ein *SharedMaze*.

[1] Bullwinkel, Faye (2023), Lösen von 3D-Labyrinthen mit Unitys ML Agents [Projektbericht]

Schwierigkeiten und Anpassungen Eine der Herausforderungen bestand darin, die richtige Position im 3D-Array zu finden, da jede Seite des Würfels unterschiedlich rotiert ist. Dies führte zunächst zu einigen Spiegelverkehrungen um bestimmte Achsen. Um dieses Problem zu lösen, wurden die Werte der Zellen und Würfel entsprechend angepasst. Beispielsweise wurde die Position eines Würfels im Array durch den Ausdruck $Cube.x = (Size - 1) - (Cell.x * 2 + 1)$ korrigiert.

Übergang zwischen den Szenen Der Übergang zwischen den beiden Szenen erfolgt wie folgt: In der Szene *MazeGeneration* wird ein gültiges Maze generiert. Ist dies der Fall, wird ein *SharedMaze* mit den entsprechenden Daten gefüllt. Danach wird die *MazeSolving*-Szene geladen. In dieser Szene wird das Labyrinth mit den Daten aus dem *SharedMaze* generiert und der Agent von *MazeSolving* auf das Startfeld gesetzt. Damit ist die Grundlage für die Zusammenarbeit der beiden Agenten geschaffen, wobei *MazeGeneration* das Labyrinth generiert und *MazeSolving* die Aufgabe löst.

8. Fazit und Ausblick

In diesem Projekt sind wir tief in die Welt der KI-basierten Agenten eingetaucht und haben uns mit dem Thema Labyrinthgenerierung beschäftigt. Dabei wurden wichtige Erkenntnisse im Bereich des Reinforcement Learning gewonnen und Herausforderungen gemeistert.

Eine wichtige Erkenntnis ist, dass die Generierung eines Labyrinths eine sehr komplexe Aufgabe ist. Verschiedene Ansätze zur Definition des Agentenverhaltens wurden durch Beobachtung und Auswertung getestet. Es zeigte sich, dass die Aufgabe besser gelöst werden konnte, wenn der Agent genügend Informationen über die Umgebung erhielt und sich die Bewertungen auf die Hauptaufgabe konzentrierten. Dies führte zu einem besseren Trainingsergebnis.

Die Welt der KI-Agenten bietet viele Möglichkeiten für zukünftige Entwicklungen und Forschung. Ein interessanter Ansatz könnte darin bestehen, Agenten zu implementieren, die in Echtzeit auf Veränderungen in ihrer Umgebung reagieren können, anstatt im Voraus trainiert zu werden. Dies würde den Einsatz von Agenten in komplexen und sich ständig verändernden Umgebungen ermöglichen.

Darüber hinaus könnte die Zusammenarbeit der beiden diskutierten Agenten, *MazeGeneration* und *MazeSolving*, weiter erforscht werden. Sie könnten voneinander lernen und sich gegenseitig verbessern. Ein spannendes Ziel wäre es, den *MazeGeneration*-Agenten so zu trainieren, dass er perfekte Labyrinthe erzeugt, während der *MazeSolving*-Agent darauf abzielt, diese Labyrinthe möglichst schnell und in möglichst wenigen Schritten zu lösen.

Weitere Untersuchungen und Experimente könnten durchgeführt werden, um die Leistung der Agenten in komplexeren Umgebungen und Szenarien zu testen. Die Anpassung von Hyperparametern, die Erweiterung von Beobachtungen und Bewertungen sowie die Integration moderner KI-Techniken könnten weitere Fortschritte bei der Entwicklung dieser Agenten ermöglichen.

Insgesamt eröffnet uns die Forschung im Bereich der KI-Agenten spannende Perspektiven und die Möglichkeit, Lösungen für komplexe Probleme in verschiedenen Anwendungsbereichen zu finden.