

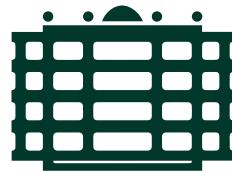


TECHNISCHE UNIVERSITÄT
CHEMNITZ

TEAMORIENTIERTES PRAKTIKUM

**Erzeugung interaktiver Umgebungen
für verkörperte digitale Technologien**

7. Februar 2024



TECHNISCHE UNIVERSITÄT CHEMNITZ

Aufgabenstellung

*Erzeugung interaktiver Umgebungen für verkörperte digitale
Technologien*

Fakultät
Informatik

Professur
Graphische Datenverarbeitung und Visualisierung

Aufgabentyp
Teamorientiertes Praktikum

Ausgabe der Aufgabenstellung
15. Mai 2023

Studenten

Leon Rollenhagen
Angelique Gräfe

Lisa Neuhaus
Carlo Kretzschmann

Sophie Neuhaus
Linus Thriemer

Prüfer und Betreuer

Prof. Dr. Guido Brunnett
Technische Universität Chemnitz

Tom Uhlmann M.Sc.
Technische Universität Chemnitz

Titel: *Erzeugung interaktiver Umgebungen für verkörperte digitale Technologien*

Vorbetrachtungen und Zielstellungen:

Im Sonderforschungsbereich Hybrid Societies wird die Interaktion von Menschen und verkörpernten digitalen Technologien (EDTs) untersucht. EDTs können autonome Fahrzeuge oder Roboter, Service- oder Info-Roboter, virtuelle Charaktere oder Menschen sein, die mit technischen Erweiterungen wie intelligenten Brillen, Prothesen oder Exoskeletten ausgestattet sind. Obwohl uns im täglichen Leben noch wenige dieser Dinge begegnen, ergeben sich viele technische, logistische und soziale Implikationen, welche sich mit dem Einzug solcher Technologien in unseren Alltag ergeben und bereits jetzt untersucht werden sollten. Diese Implikationen betreffen unter anderem die Wahrnehmung, Kommunikation, Interaktion, Umsetzung und den räumlichen Bedarf der jeweiligen Akteure. Die Erforschung dieser Aspekte ist schwierig, da die entsprechenden Geräte und Roboter entweder noch nicht existieren oder sehr teuer sind. Daher ist es praktisch, eine virtuelle Umgebung zu haben, in der diese Aspekte erforscht werden können.

In diesem teamorientierten Praktikum werden Möglichkeiten erforscht und umgesetzt, wie eine lebendige, interaktive Welt erschaffen werden kann, in der ein Nutzer mit EDTs interagieren kann oder die Interaktion von Agenten untereinander und in der Welt beobachten kann. Um dies zu erreichen, müssen die entsprechenden Objekte wie die Welt, die Agenten und die Charaktere erschaffen werden. Zusätzlich müssen die Objekte interagierbar gemacht werden, und die Agenten müssen sich selbstständig in der Welt bewegen und mit ihr interagieren können. Der Fokus sollte dabei auf Glaubwürdigkeit und Plausibilität liegen. Das bedeutet, dass Ansätze aus der wissenschaftlichen Literatur und bereits existierende EDTs verwendet werden sollten. Die virtuelle Welt sollte so angelegt sein, dass neue Agenten in die Welt eingefügt oder das Verhalten bestehender Agenten geändert werden kann, um die Auswirkungen zu untersuchen.

Inhaltsverzeichnis

1 Motivation	1
1.1 Projektziel Innenstadt	1
1.2 Minimum Viable Product	2
2 Konzept	3
2.1 Vorbedingungen	3
2.2 Architektur und benötigte Systeme	3
2.3 Entity Component System	4
2.4 Partikelsystem	4
2.5 Multiagentensystem	5
2.5.1 Sensoren	5
2.5.2 Entscheidungsfindung	5
2.5.3 Entscheidungsausführung	6
2.5.4 Bewegungssteuerung	6
2.6 Navigationssystem	12
2.7 Physiksystem	13
2.8 Szenen Editor	13
2.8.1 GLTF als Szenenbeschreibung	13
2.8.2 Eigenes Format zur Szenenbeschreibung	14
2.9 Dialogsystem	14
2.9.1 Imgui	14
2.9.2 JsonCpp	15
2.9.3 Dialoggraph	15

2.9.4 Dialogmap	15
2.10 Modelle und Szene	15
3 Implementierung	19
3.1 Entity Component System	19
3.2 Multiagenten System	20
3.2.1 Sate Machines	20
3.2.2 Behaviour Tree	20
3.2.3 Steering Behaviour	21
3.3 Partikelsystem	23
3.4 Navigationssystem	23
3.5 Physiksystem	23
3.6 Szenen Editor	24
3.7 Dialogsystem	24
3.7.1 ImGui	25
3.7.2 JsonCpp	25
3.7.3 Dialoggraph	25
3.7.4 Dialogmap	25
3.8 Modelle und Szene	26
4 Management	33
4.1 Aufgabenverteilung und Arbeitsweise	33
4.2 Analyse der Probleme	34
5 Zusammenfassung & Ausblick	37
A Nachsätze	39
A.1 Arbeitsaufteilung	39
Literaturverzeichnis	41

Kapitel 1

Motivation

1.1 Projektziel Innenstadt

Gemeinsam als Gruppe haben wir uns für das Szenario Innenstadt entschieden. Inspiriert von Solarpunk ist es das Ziel, ein modernes, menschenorientiertes, aber trotzdem platzeffizientes Gelände zu erschaffen. Der Baustil soll modern und luxuriös, aber trotzdem mit der Natur verbunden sein. Deswegen möchten wir die Umgebung mit geschwungenen organischen Formen gestalten und verschiedene moderne und klassische Materialien wie Granit, Sandstein, Holz und Glas miteinander verbinden. Die Gebäude sollen über Brücken miteinander verbunden werden, sodass man nicht ins Erdgeschoss zurück gehen muss, um von einem Haus zum nächsten zu gelangen. An allen Fassaden und in den öffentlichen Aufenthaltsbereichen wachsen Pflanzen, um für ein angenehmes Klima zu sorgen.

In dieser Stadt leben Menschen und werden von verschiedenen Robotern im Alltag unterstützt. Die Roboter übernehmen dabei die Arbeit, die nicht erfüllend, stupide oder gefährlich ist. Putzroboter halten Fußböden sauber, entleeren Mülleimer und heben Müll auf. Sie können auch komplexere Oberflächen wie Toiletten, Waschbecken oder Türgriffe reinigen. Essensverkäufer-Roboter arbeiten in Geschäften wie McDonalds, Nordsee oder Subway. Sie nehmen Bestellungen entgegen und bereiten diese zu. Gepäckroboter helfen den Menschen, ihre Einkäufe nach Hause zu tragen. Gärtnerroboter gießen Pflanzen, sägen Äste ab und sperren Bereiche ab. Sie arbeiten mit Menschen zusammen, welche Anweisungen geben, welche Äste abgeschnitten werden sollen.

Die Menschen sollen sich möglichst realistisch verhalten und haben ein eigenes Leben mit eigenen Zielen. Die Tagesabläufe von Kindern, Erwachsenen und Rentnern sollen sich je nach den individuellen Bedürfnissen unterscheiden. Diese Menschen interagieren mit den Robotern und arbeiten eng mit ihnen zusammen. Einige Kombinationen könnten zum Beispiel sein: Koch und Kellnerroboter, Gärtner und Gärtnerroboter oder Händler und Lagerroboter. Die Menschen interagieren

aber nicht nur mit Robotern, sondern auch miteinander. Sie gehen gemeinsam essen, verreisen miteinander und haben sinnvolle, natürliche Unterhaltungen.

Die Menschen haben auch tragbare Computergestützte Systeme, die sie im Alltag begleiten. So kann zum Beispiel eine AR-Brille die Navigation übernehmen und über ein HUD die Wegpunkte direkt in der Welt anzeigen. Zusätzlich können noch weitere Informationen wie Ankünfte, Wartezeiten oder die nächsten Termine dargestellt werden.

1.2 Minimum Viable Product

Das Projektziel ist sehr umfangreich, weswegen wir ein zuerst ein Minimum Viable Product (MVP) erstellen möchten, das inhaltlich stark vereinfacht ist, aber alle Grundideen umsetzt. Der Plan ist, zuerst das MVP zu entwickeln und dann aus dem MVP das Projektziel zu verwirklichen.

Die Szene soll aus zwei Platformen bestehen, die über eine Rampe miteinander verbunden sind. Auf jeder Platform befinden sich Pflanzen, die von fahrenden Robotern gegossen werden, wenn sie Wasser benötigen. Als Spieler kann man die Roboter fragen, um welche Pflanze es sich handelt.

Obwohl man das Minimum Viable Product in wenigen Sätzen beschreiben kann, sind trotzdem alle wichtigen Aspekte des Projektziels einbegriffen: die Szene ist nicht eben, sondern erstreckt sich über mehrere Plattformen. Als unabhängige Agenten hat man die Roboter, die selbständig Handlungen planen und ausführen. Man kann mit den Robotern Dialoge führen, die sich je nach Zustand der Welt anpassen.

Kapitel 2

Konzept

2.1 Vorbedingungen

Gemeinsam mit unserem Betreuer haben wir uns darauf geeinigt, CrossForge zu verwenden. Die Engine unterstützt Windows, Linux und WebAssembly und benutzt nur OpenGL als Grafik API. Sie unterstützt physikalisch basiertes Rendering und Materialien, Deferred und Forward Rendering Pipelines und ein Shadersystem. Aber besonders wichtig für uns sind der Szenengraph, die Skelettanimationen und Text Rendering.

2.2 Architektur und benötigte Systeme

Aus dem MVP-Szenario und der Engine Wahl ergeben sich mehrere Anforderungen, die über verschiedene Systeme gelöst werden können. Die Roboter sind selbständige Agenten, die Entscheidungen treffen und diese auch ausführen sollen. Dieses Problem löst das Multiagentensystem. Sie müssen ihren Weg zur nächsten durstigen Pflanze finden, dafür ist das Navigationssystem zuständig. Die Roboter müssen die Pflanze gießen, was durch ein Partikelssystem visualisiert werden soll. Der Spieler und die Roboter sollen sich auf verschiedenen Plattformen bewegen und von einer zur nächsten laufen können. Zusätzlich sollen die Pflanzen verschiebbar sein. Um diese beiden Sachen zu realisieren, wird ein Physiksystem benötigt. Der Spieler soll sich mit den Robotern unterhalten können, was durch das Dialogsystem abgedeckt wird. Da CrossForge keinen eigenen Szenen Editor hat und die MVP Szene schon zu komplex ist, um diese mit Programmcode zu beschreiben, ist ein Szenen Editor nötig.

Um alle Systeme möglichst flexibel und unabhängig voneinander zu gestalten, haben wir uns für das Entity Component Pattern entschieden.

2.3 Entity Component System

Die klassische Herangehensweise der objektorientierten Programmierung ist der Aufbau einer Vererbungshierarchie, bei der Objekte wie der Gießroboter von allgemeingültigen Klassen abgeleitet werden. Der Roboter erbt beispielsweise, weil er sich bewegen kann, von einer Klasse Moveable und, da er gezeichnet wird, von der Klasse Renderable. Um eine Mehrfachvererbung zu verhindern, wird in der Regel eine Hierarchie eingefügt, sodass Renderable von Movable abstammt. Doch genau hier entsteht das Problem. Wenn nun ein gezeichneter Roboter erstellt werden soll, welcher sich nicht bewegen kann, so ist dies nicht mehr möglich. Grund hierfür ist die gleichzeitige Vererbung von Renderable und der Klasse Moveable. Das bedeutet, dass die Vererbungshierarchie angepasst werden muss, was bei einer großen Anzahl von Objekten und Klassen schnell sehr komplex wird. Aus diesem Grund wurde das Prinzip Entity Component Systems (ECS) eingeführt und in dem Projekt genutzt. Das ECS besteht aus drei Bausteinen. Der erste ist die Komponente (Component), welche Informationen über Objekte hält. Im Fall des Gießroboters wären das beispielsweise Daten über dessen Position, Geschwindigkeit, Modell und noch viele mehr. Der zweite ist die Entität (Entity), welche ein Objekt in dem Projekt ist. Es besteht aus beliebig vielen Komponenten und einer ID, damit es eindeutig identifizierbar ist. Es speichert dabei keine Daten oder Programmlogik. Der letzte Teil des ECS ist das System. Es enthält die gesamte Logik, jedoch keine Daten. Ein Projekt besteht aus mehreren Systemen, die die Werte der Komponenten auslesen und anpassen. Beispielsweise liest ein SteeringSystem die Geschwindigkeit aus einer SteeringComponent und aktualisiert basierend darauf die Position in der PositionsKomponente. Der große Vorteil besteht darin, dass die Eigenschaften eines Objekts leicht angepasst werden können, indem die Werte seiner Komponenten ändert, oder sogar ganze Komponenten hinzufügt beziehungsweise entfernt werden.

2.4 Partikelsystem

Partikelsysteme werden verwendet, um natürliche Phänomene, wie Wasser, Rauch oder Feuer darzustellen. Dabei gibt es stets einen sogenannten Emitter, welcher Partikel ausstößt. Diese besitzen einige Eigenschaften, wie beispielsweise Lebensdauer, Position und Richtung. Damit sie sichtbar werden, müssen ihnen grafische Merkmale, wie ein Modell, Textur und Material, zugeordnet werden. Durch die grafische Beschaffenheit, die Anzahl der Partikel und ihr Verhalten untereinander können verschiedene natürliche Phänomene simuliert und dargestellt werden. In dem Projekt wird das Partikelsystem verwendet, um dem Nutzer ein visuelles Feedback zu geben, wenn ein Roboter eine Pflanze gießt. In diesem Fall wird ein Wasserstrahl, wie er von einer Gießkanne erzeugt wird, simuliert.

2.5 Multiagentensystem

Das Multiagentensystem ist das Herz unseres Projektes, da das Verhalten von allen belebten und unbelebten Agenten von diesem System gesteuert werden soll. Man unterscheidet zwischen Zentraler KI und Agentenbasierter KI. Bei Zentraler KI werden die Entitäten von einem externen, globalen und allwissenden System gesteuert. Individuen haben deswegen keine Kontrolle über ihre eigenen Handlungen. Zentrale KI wird sehr oft eingesetzt, weil sich damit Gruppendynamiken und taktische Vorgehensweisen einfacher implementieren lassen. Bei Agentenbasierter KI treffen die Agenten unabhängige und individuelle Entscheidungen basierend auf den Informationen, die ihnen bereit stehen. Es gibt zwar trotzdem globale Informationen, die aber nicht dafür missbraucht werden dürfen, die Handlungen eines Agenten zu diktieren. Wir haben uns für Agentenbasierte KI entschieden, weil sie für unser Szenario leichter umzusetzen ist. Die Hoffnung ist, dass die Akteure dadurch natürliche Verhaltensweisen und Interaktionen zur Schau stellen.

In verschiedenen GDC [3][4][5] Talks wurde empfohlen, ein KI-System in mehreren Schichten aufzubauen:

- Sensoren
- Entscheidungsfindung
- Entscheidungsausführung
- Bewegungssteuerung

2.5.1 Sensoren

Sensoren sind ein Querschnittskonzept und tauchen deshalb in allen Ebenen auf. Sie filtern Informationen aus der Umgebung und stellen diese der Schicht bereit, in der sie eingesetzt werden. Zusätzlich können Informationen über Blackboards mit anderen Agenten geteilt werden.

2.5.2 Entscheidungsfindung

Für die Entscheidungsfindung standen die folgenden Algorithmen in der näheren Auswahl: Beliefs, Desires, Intentions (BDI), Goal Oriented Action Planning (GOAP) und Finite State Machines (FSM). Am Ende haben wir uns für Finite State Machines entschieden, weil das das einfachste Verfahren war und vorerst für die Roboter ausreicht.

Eine State Machine ist ein gerichteter Graph mit limitierter Anzahl an Stati und Aktionen. Der Agent wechselt von einem Status in den nächsten, falls eine Bedingung

erfüllt ist. In unserer Simulation sollen FSMs für die Roboter eingesetzt werden, um die Stati *Pflanzen gießen*, *Dialog mit Spieler*, *Spieler folgen*, etc. abdecken zu können.

2.5.3 Entscheidungsausführung

Wenn der Agent eine Entscheidung getroffen hat, dann lässt sich diese Entscheidung meistens in weitere Teilprozesse zerlegen. Wenn ein Roboter zum Beispiel entschieden hat, dass er jetzt Pflanzen gießt, dann muss er:

- eine durstige Pflanze finden
- zur Pflanze hinfahren
- die Gießkanne zur Pflanze ausrichten
- und schließlich die Pflanze gießen

Diese Sequenz von Handlungen lässt sich nur sehr schlecht mit FSMs darstellen, weswegen dieser Nachteil durch Behaviour Trees ausgeglichen werden soll. Behaviour Trees sind sehr gut darin, solche Sequenzen darzustellen oder sogar nebenläufige Handlungen zu beschreiben. Ihr Nachteil ist jedoch, dass sie nur sehr schwierig auf Übergänge von einen Status in den nächsten reagieren können. Ein Beispiel hierfür ist, dass der Spieler den Roboter anspricht und ihn über Pflanzen fragt. Um das mit Behaviour Trees festzustellen, müssen überall Monitore eingebaut werden, die dieses Ereignis erkennen und behandeln. Das macht den Baum unübersichtlich und nicht wartbar. Behaviour Trees und State Machines ergänzen sich also sehr gut.

In Behaviour Trees werden Handlungen durch Knoten beschrieben. Die Knoten können die Stati *Laufend*, *Fehler* oder *Abgeschlossen* haben. Knoten können dabei über die Bearbeitung ihrer Kindknoten entscheiden. Bei einem Sequenzknoten werden die Kinder von links nach rechts abgearbeitet. Nur wenn der Knoten den Status *Abgeschlossen* hat, wird der rechte Geschwisterknoten aufgerufen. Wenn der Status *Laufend* ist, dann wird der Knoten solange bearbeitet, bis der Status *Abgeschlossen* oder *Fehler* erreicht ist. Bei dem Status *Fehler* wird die Abarbeitung abgebrochen und dieser nach oben propagiert. Der Elternknoten kann dann entscheiden, wie dieser Fehlerstatus behandelt wird. Der Fallbackknoten behandelt den Fehler zum Beispiel, indem er den ersten Kindknoten findet, der nicht fehlschlägt und somit erfolgreich ausführt. Nur wenn alle Kindknoten fehlschlagen, ist der Status des Fallbackknotens *Fehler*.

2.5.4 Bewegungssteuerung

Die Ebene der Bewegungssteuerung ist dafür verantwortlich, die Entscheidungen in Beschleunigung, Geschwindigkeit und Rotation umzuwandeln. Dafür haben wir uns Steering Behaviour [6] näher angesehen. Es gibt verschiedene Bewegungsmuster:

- Seek
- Wander
- Collision Avoidance
- Queue
- ...

Für uns ist vor allem Collision Avoidance in Verbindung mit Seek interessant.

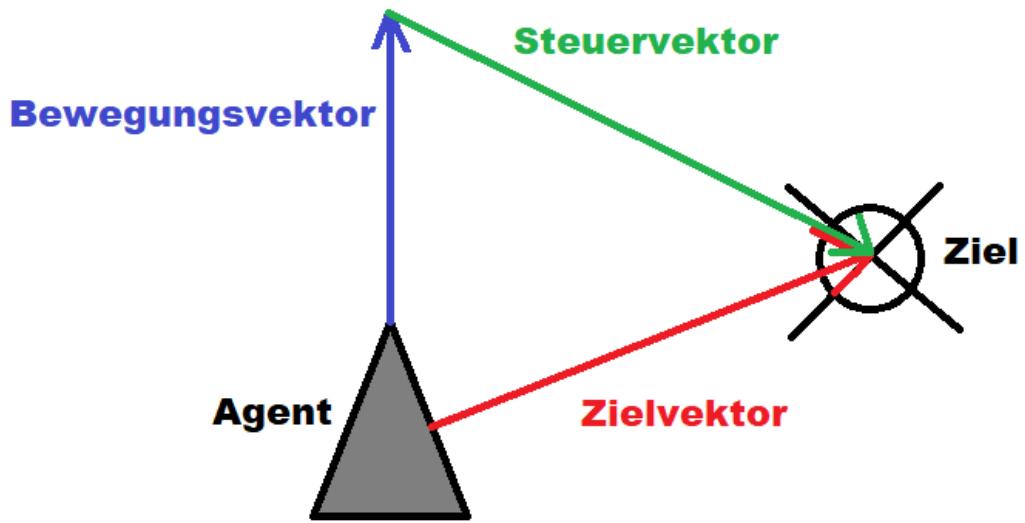


Abbildung 2.1: Suchverhalten des Roboters

Das Suchverhalten (Seek), bedeutet, dass der Agent sich auf einen Zielpunkt zubewegt und sich dabei in die richtige Richtung dreht. Dies geschieht mit Hilfe von Vektoren. Zum einen wird ein Zielvektor genutzt, dieser zeigt von der Position des Roboters in Richtung des Ziels. Des Weiteren gibt es einen Bewegungsvektor, welcher in die Richtung zeigt, in die der Agent sich aktuell bewegt und schaut. Die Länge des Vektors entspricht dabei seiner aktuellen Geschwindigkeit. Um nun dessen Fahrtrichtung anzupassen, wird ein Steuervektor benötigt, der aus der Subtraktion des normierten Zielvektors und des Bewegungsvektors entsteht. Jetzt kann der Bewegungsvektor angepasst werden, sodass er in die richtige Richtung zeigt. Dazu werden der Steuervektor und Bewegungsvektor addiert. Das Problem hierbei ist, dass bei dieser Umsetzung der Roboter sofort in Richtung des Ziels schaut und sich auf dieses zubewegt. Dieses Verhalten ist unrealistisch. Aus diesem Grund erhält der Roboter zusätzliche Eigenschaften, wie Masse, maximale Geschwindigkeit und Lenkkraft. Beim Addieren der beiden Vektoren, wird der Lenkvektor zuerst durch die Masse geteilt, dadurch wird die Länge des Vektors reduziert und der Effekt verringert.

Zudem muss sichergestellt werden, dass der Betrag des Vektors kleiner gleich der maximalen Lenkkraft ist. Ebenso darf der Bewegungsvektor nicht größer sein als die Höchstgeschwindigkeit. Wenn diese Anforderungen berücksichtigt werden, dreht sich der Roboter pro Schleifendurchlauf nur um einen kleinen Anteil und fährt ein Stück in Richtung Ziel. Dieses Verhalten wirkt für einen Betrachter realistischer.

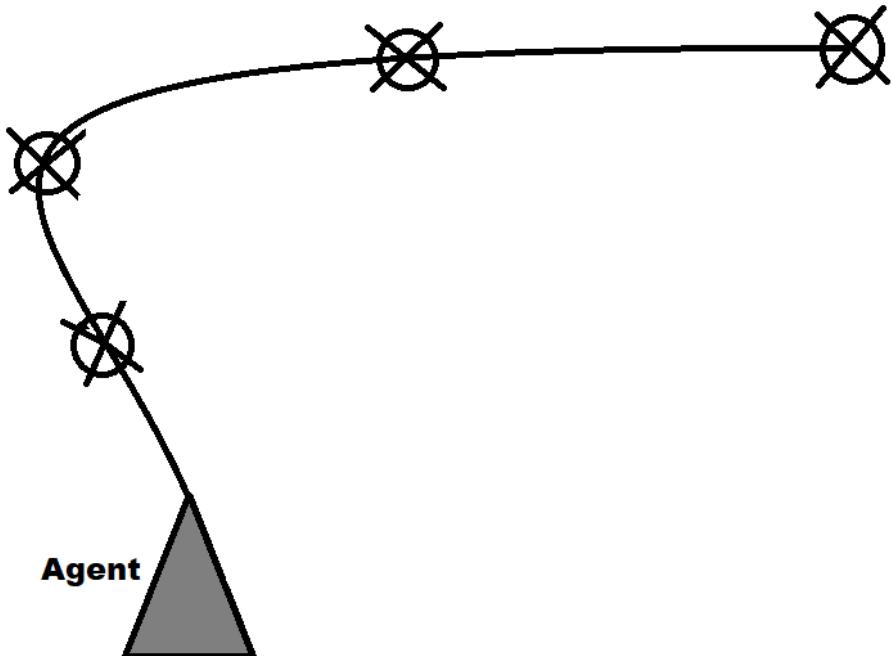


Abbildung 2.2: Pfadverfolgen durch angepasstes Suchverhalten

Das zweite Lenkverhalten, das in dem Projekt genutzt wurde, ist das Verfolgen eines Pfades. Dazu wird das Suchverhalten so angepasst, dass der Roboter zahlreiche Zielpunkte entlang des Pfades sucht. Er ist zu dem Zeitpunkt jedoch noch nicht in der Lage, zu wissen, wann er an einem Ziel ankommt. Dazu wird eine weitere Funktion und der Vektor zwischen Agenten und Zielposition benötigt. Der Roboter ist angekommen, wenn die Länge des Zielvektors kürzer ist als der Radius des Agenten plus einen Sicherheitsabstand. Währenddessen muss sichergestellt werden, dass der Roboter nicht mit Hindernissen, Robotern oder Spielern kollidiert.

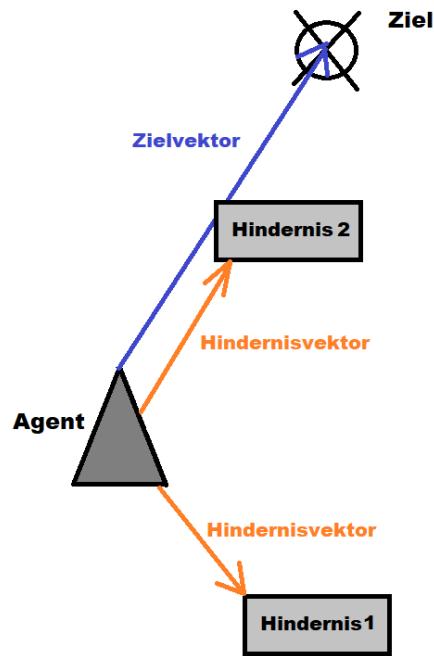


Abbildung 2.3: Testen der Hindernisse auf ihre Position in Relation zum Zielpunkt und der Position des Roboters

Dazu wird eine Liste aller möglichen Hindernisse auf ihre Distanz zum Agenten überprüft und geordnet. Nun wird getestet, ob sich das Hindernis zwischen Roboter und Ziel befindet, oder hinter dem Agenten. Hierfür wird das Kreuzprodukt aus dem Zielvektor und dem Vektor, der vom Roboter ausgehend in Richtung Hindernis zeigt, gebildet. Nur wenn dieses kleiner als Null ist, befindet sich das Hindernis vor dem Agenten und muss weiter getestet werden.

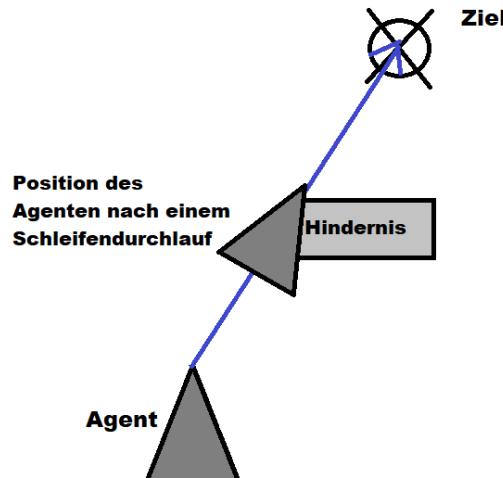


Abbildung 2.4: Testen auf mögliche Kolission nach einem Schleifendurchlauf

Im nächsten Schritt wird die Position bestimmt, an der sich der Roboter in dem nächsten Schleifendurchlauf befinden wird. An dieser Stelle wird der Abstand zwischen ihm und dem Hindernis betrachtet. Solange dieser nicht kleiner ist als der Roboterradius plus einen Sicherheitsabstand plus den Hindernisradius kann davon ausgegangen werden, dass der Roboter mit diesem nicht kollidieren wird. Ist der Abstand jedoch kleiner, so muss der aktuelle Wegpunkt so angepasst werden, dass der Roboter dem Hindernis ausweicht.

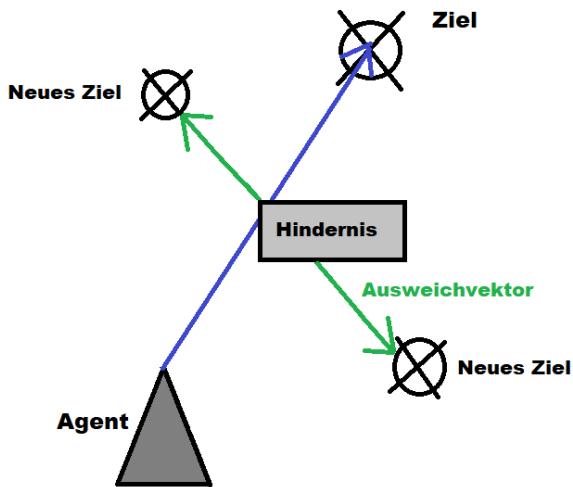


Abbildung 2.5: Verschieben des Zielpunktes

Dies geschieht, indem ein Vektor erzeugt wird, der von dem Hindernis weg zeigt. Es werden nun zwei potenzielle neue Wegpunkte erzeugt, die sich rechts beziehungsweise links von dem Hindernis befinden. Nun wird überprüft, bei welchem der beiden die Abweichung vom aktuellen Pfad kleiner ist. Dieser wird als der neue Wegpunkt gesetzt und der alte wird entfernt. Der Roboter bewegt sich nun auf diesen zu und weicht dem Hindernis aus.

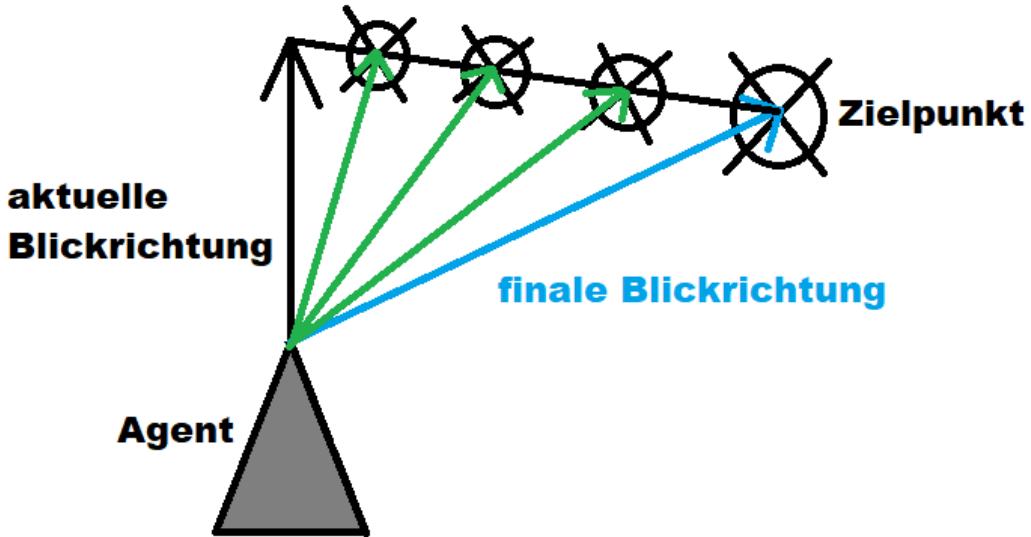


Abbildung 2.6: Drehen des Roboters

Das letzte Lenkverhalten ist das auf der Stelle Drehen. Dieses wird genutzt, wenn der Spieler mit dem Roboter redet oder er eine Pflanze gießen will. Um die Verhaltensweise umzusetzen, werden die Zielrichtungen benötigt, in die der Agent final schauen soll und seine aktuelle Blickrichtung. Er soll sich dabei wieder nur langsam drehen, um ein realistisches Verhalten zu simulieren. Mit Hilfe von linearer Interpolation werden zusätzliche Punkte zwischen der Ausgangssituation und der Zielrichtung gebildet. Der Roboter kann sich nun nach und nach in Richtung dieser ausrichten. Das hat zur Folge, dass er sich nur langsam dreht. Durch das Berechnen der einzelnen Zwischenpunkte ist diese Umsetzung zusätzlich unabhängig von der Framerate.

2.6 Navigationssystem

In der Mitte der MVP-Szene befindet sich die Wendeltreppe, weswegen die Roboter nicht immer den direkten Weg zur Pflanze fahren können, weil sie dieses Hindernis beachten müssen. Das Navigationssystem übernimmt die Aufgabe, einen Pfad von einem Startpunkt zum Zielpunkt zu finden. Um Zeit zu sparen und uns auf die Hauptaufgabe konzentrieren zu können, haben wir uns entschieden, die Bibliothek Recast und Detour einzusetzen. Recast berechnet das Navigationsmesh aus der statischen Geometrie, und Detour findet zur Laufzeit einen Pfad auf diesem Navigationsmesh. Da das Navigationsmesh nicht dynamisch angepasst werden muss, reicht es aus, wenn man es einmal vor der Kompilierung berechnet, speichert und dann in CrossForge lädt. Die Bibliothek stellt hierfür ein Beispielprogramm bereit, was wir nutzen.

2.7 Physiksystem

Die Entities und der Spieler sollen mit der statischen Szenengeometrie und sich selber kollidieren. Um diese Kollisionen zu erkennen und aufzulösen, ist das Physiksystem nötig. Dafür möchten wir die Bullet Bibliothek einsetzen, da Linus schon einmal den Separating Axis Theorem Algorihmus in 2D implementiert hat und das sich als sehr zeitaufwändig und fehleranfällig herausgestellt hat.

2.8 Szenen Editor

Wir benötigen einen Szenen Editor, weil schon bei einer geringen Anzahl an platzierten Entitäten der Code unübersichtlich und schwer zu warten ist. Als Alternative könnten wir die Agenten und Pflanzen prozedural platzieren, was aber ein zu komplexes Gebiet wäre und damit den Rahmen sprengen würde. Selber einen Szenen Editor von Grund auf neu zu schreiben ist keine Option, weil auch das eine riesige Aufgabe ist. Deswegen haben wir uns dazu entschieden, die Open Source Software Blender zu verwenden. Blender ist ein mächtiger 3D-Editor, den man über Addons erweitern kann. Somit haben wir zwei Möglichkeiten, unsere Szenen zu erstellen und in CrossForge zu laden. Wir können die Szene als GLTF-Datei exportieren und dann die vorhandenen Funktionen in CrossForge erweitern, um aus der GLTF-Datei die einzelnen Transformationen für die Entitäten zu extrahieren. Oder wir schreiben ein Add-On, welches die Szene nach unseren Anforderungen exportiert.

2.8.1 GLTF als Szenenbeschreibung

Khronos Group veröffentlichte am 19.10.2015 das offene GLTF Format, um dreidimensionale Szenen und Modelle darzustellen. In dem Format können Szenen mit ihren Knoten, Kamerainformationen, Animationen, Texturen, Materialien und natürlich auch Modellinformationen gespeichert werden. CrossForge unterstützt über die Assimp Bibliothek verschiedene Dateiformate, unter anderem auch GLTF.

Die Idee ist, dass alle Entities einer bestimmten Namenskonvention folgen. Die Szene wird als Ganzes in das GLTF-Format exportiert und mit Assimp geladen. Assimp hat einen eigenen Szenengraph, den man traversieren kann, und wenn man an einem Knoten mit bestimmter Namenskonvention angelangt ist, weiß man, dass es sich um ein Entity handelt. Die Transformation des Knotens wird dann zur Transformation des Entities. Alle Kindknoten können dann zu einem Modell zusammengefasst werden und für die Geometriekomponente des Entities benutzt werden.

Dieser Ansatz hat den Vorteil, dass er editoragnostisch ist, und wir können vorhandenes Wissen über Assimp nutzen. Leider hat diese Variante aber auch große Nachteile: man muss die Änderungen ziemlich tief in CrossForge vornehmen, was sehr viele Code-Änderungen nach sich zieht. Der Szenenersteller muss die Knoten

im Editor korrekt benennen, weil man sie in CrossForge sonst nicht mehr erkennen kann.

2.8.2 Eigenes Format zur Szenenbeschreibung

Wir können ein Blender Plugin schreiben, das die Transformationen der einzelnen Entities in eine JSON-Datei exportiert. Das Problem ist, dass Blender erkennen muss, was statische Geometrie ist und was Entities sind. Es gibt aber eine Funktion, um externe .blend-Dateien in die aktuelle Szene zu linken. Jetzt kann man die Regel aufstellen, dass alles, was gelinkt ist, ein eigenes Entity ist. Die Vorteile sind, dass keine Eingriffe ins Engine-Innere nötig sind und auch die Implementierung auf CrossForge-Seite sehr simpel ist, da man das Dateiformat selber bestimmen kann. Der Nachteil ist, dass noch niemand von uns ein Blender Add-On geschrieben hat und wir deswegen noch keine Erfahrung in diesem Bereich haben.

2.9 Dialogsystem

Wir haben uns dazu entschieden ein Dialogsystem zu implementieren, damit man als Nutzer mit den Robotern kommunizieren kann. Der Nutzer soll dazu ein Dialogfenster öffnen können, wenn er sich nah genug an einem Roboter befindet. Im Dialogfenster werden dann die Aussagen des Roboters und die Antwortoptionen des Nutzers angezeigt. Durch Klicken auf die Antworten wird der Dialog fortgesetzt. Wenn der Dialog beendet ist wird das Dialogfenster wieder geschlossen. Ziel des Dialogsystems für das Minimum Viable Product ist es, dass man den Roboter nach dem Name einer Pflanze fragen kann.

2.9.1 ImGui

Unsere erste Idee für die Umsetzung des Dialogfensters war die Features zu nutzen, die in CrossForge schon implementiert waren. Dafür wollten wir ursprünglich Texturen verwenden. Da das allerdings wahrscheinlich sehr aufwendig geworden wäre, hat Linus vorgeschlagen ImGui zu nutzen. ImGui ist eine C++-Bibliothek für grafische Benutzeroberflächen. Normalerweise ist ImGui zwar eher für Debug-Tools gedacht, da es grafisch relativ simpel ist und nur sehr eingeschränkte Möglichkeiten zur grafischen Individualisierung bietet. Für unsere Zwecke ist es aber trotzdem gut geeignet, weil es uns, insbesondere für das Minimum Viable Product, wichtiger ist schnell ein Dialogfenster angezeigt zu bekommen als, dass es besonders hübsch aussieht.

2.9.2 JsonCpp

Die Dialoge selbst wollen wir in Json-Dateien speichern. Json-Dateien ermöglichen es uns zum einen die Dialoge vom Code zu trennen und somit zum anderen leicht neue Dialoge hinzuzufügen und andere Dialoge auszuwählen. Damit wir die Daten aus diesen Dateien auslesen können nutzen wir JsonCpp, eine C++-Bibliothek, die es ermöglicht Json-Dateien mit C++ ein- und auszulesen.

2.9.3 Dialoggraph

Als Datenstruktur für den Dialog nutzen wir eine Baumstruktur. Dabei beinhaltet jeder Knoten einen Text, einen Boolean, der angibt ob der Text vom Spieler gesprochen wird, und eine Liste mit möglichen Antworten, die jeweils durch weitere Knoten repräsentiert werden. Dieser Baum wird erstellt, indem die entsprechenden Informationen aus einer Json-Datei ausgelesen werden.

2.9.4 Dialogmap

Um den Dialog zum Erfragen eines Pflanzenamens umsetzen zu können, benötigt man im Dialog in der Json-Datei einen Platzhalter für den Name der Pflanze. Dieser Platzhalter wird dann durch den entsprechenden Name ersetzt. Dafür wollen wir eine Map nutzen, die den Platzhalter als Key entgegen nimmt und einen Funktionspointer zurückgibt. Mithilfe der dadurch aufgerufenen Funktion wird dann der Name der Pflanze in Abhängigkeit von der Position des Roboters und der Position des Spielers ermittelt, sodass die Map letztendlich indirekt den Pflanzenamen zurückgibt.

Die Nutzung einer Map ermöglicht es schnell und einfach neue Platzhalter/Keys hinzuzufügen, durch die auch andere Ersetzungen mithilfe entsprechender Funktionen möglich sind. Die genutzten Funktionen können aber auch andere Funktionalitäten erfüllen als nur einfache Textersetzung; eine Funktion kann auch beispielsweise ein Event auslösen. Ein solches Event könnte zum Beispiel sein, dass man den Roboter anweist dem Spieler zu folgen oder aus dem Weg zu gehen.

2.10 Modelle und Szene

Die Szene und die zugehörigen Modelle sind von wesentlicher Bedeutung, um alle anderen Funktionen überhaupt darstellen zu können. Natürlich könnte man dafür auch jedes beliebige Modell verwenden, sogar einen simplen Würfel. Dennoch hatten wir ein klares Bild davon, was wir darstellen möchten. Wie bereits im Abschnitt Motivation erläutert, besteht unser MVP aus zwei Plattformen, jeweils einem Roboter und mehreren Pflanzen.

Der erste und entscheidendste Schritt war, zu überlegen, wie unsere Roboter

aussehen sollten, da sie die Hauptakteure in unserem Projekt sind und direkt mit dem Spieler interagieren. Ursprünglich war geplant, den Robotern ein eher menschenähnliches Design zu verleihen. Allerdings entschieden wir uns zunächst für eine simplere Gestaltung. Daher habe ich mich für ein eher "simples und knuffiges" Design entschieden und verschiedene Konzepte skizziert. Zusätzlich habe ich mir Gedanken über potenzielle Funktionen gemacht und diese als Randnotizen zum Design festgehalten.

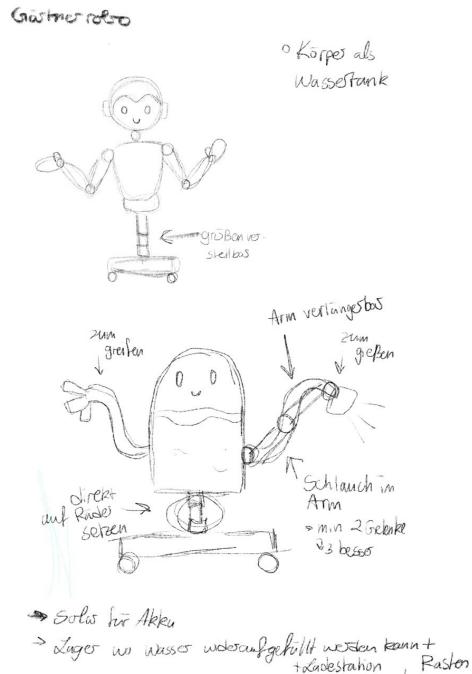


Abbildung 2.7: Konzeptskizze für den Roboter

Eine weitere Überlegung war, den Roboter schweben zu lassen anstatt ihn zu fahren, um zusätzliche Programmierungsaufwände zu vermeiden. Jedoch wurde diese Idee zunächst als zu futuristisch angesehen und daher vorerst abgelehnt. Bei der abschließenden Skizzierung habe ich das Radsystem erneut überdacht und im Nachhinein noch einige Änderungen vorgenommen.

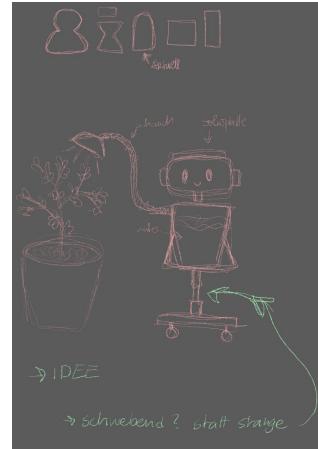
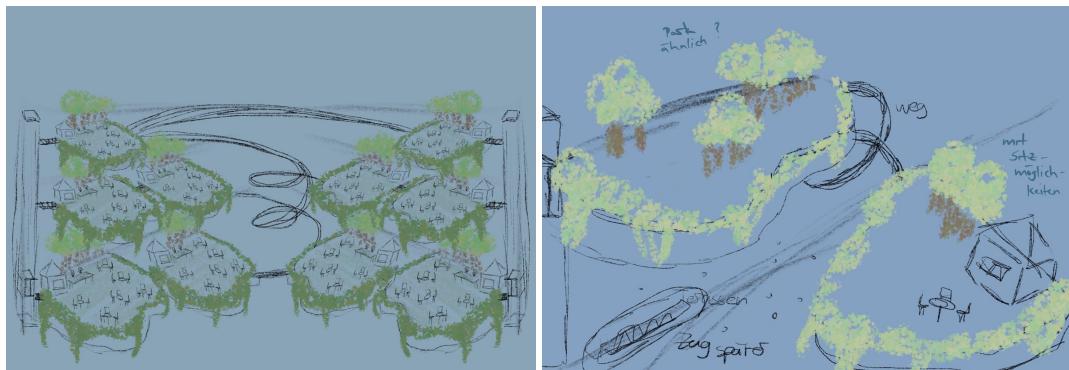


Abbildung 2.8: Finaler Entwurf für den Gärtnerroboter

Die nächsten Überlegungen galten den Pflanzen. Dabei habe ich eher spontan gearbeitet, ohne vorherige Skizzen. Es war mein Ziel, eine einfache und eine eher exotische Pflanze zu gestalten, um etwas Variation zu erreichen.

Die Gestaltung der Plattformen und der Szene im Allgemeinen erforderte ähnlich viel Aufwand wie bei den Robotern im Konzept. Auch hier habe ich zahlreiche Ideen skizziert und im Anschluss diskutiert. Anfangs habe ich weitreichend gedacht und komplexere Szenen entworfen.



(a) Ursprüngliche Idee

(b) Szene für das MVP

Abbildung 2.9: Konzeptskizzen für die Szene

Speziell auf das MVP ausgerichtet, habe ich die Skizzen vereinfacht und eine gänzlich neue Form in Betracht gezogen, die nach einigen Anpassungen schließlich die finale Gestalt annahm. Im späteren Verlauf war es natürlich auch erforderlich, Texturen für die Modelle zu erstellen. Auf diesen Aspekt werde ich im Implementierungsteil

noch einmal eingehen.

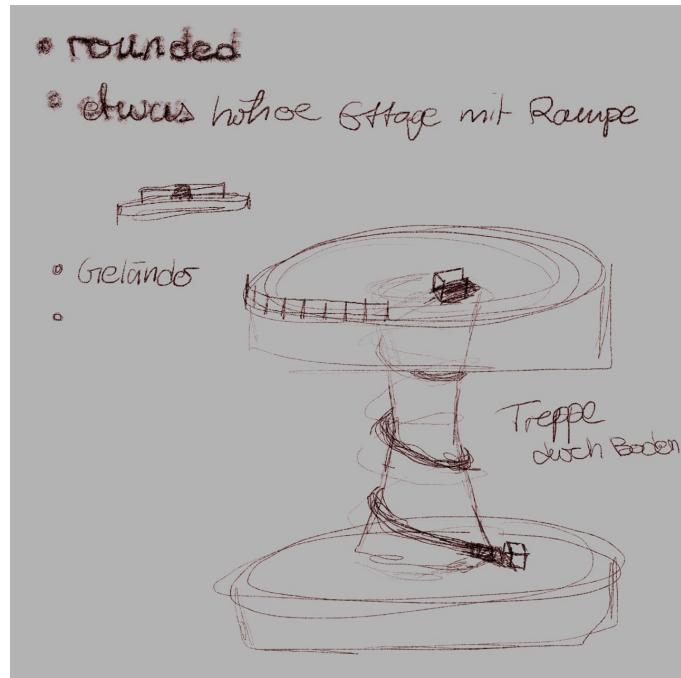


Abbildung 2.10: Finaler Entwurf die Szene für das MVP

Kapitel 3

Implementierung

3.1 Entity Component System

Zu Beginn der Implementierung wurde über verschiedene Entity Component Systeme (ECS) recherchiert. Die folgenden Kriterien wurden bei der Auswahl eines passenden ECS berücksichtigt.

- Aktive Entwicklung
- Datum des letzten Updates
- Unterstützte Sprachen
- Qualität der Dokumentation
- Performance
- Unterstützung des Open-Source-Paketmanager vcpkg

Bei der Recherche wurden, die folgenden zwei ECS, Flecs und EnTT, in Betracht gezogen. Letztendlich fiel die Wahl auf Flecs, da dieses besser dokumentiert ist und das letzte Update aktueller war. Im Anschluss an die Recherche wurde Flecs mit Hilfe von vcpkg eingebunden. Im Laufe des Projekts wurden die Entities Spieler, Pflanzen und Roboter angelegt und erhielten Komponenten, wie beispielsweise Fahr-, Spieler- und Pflanzenkomponente. So enthält zum Beispiel die Positionskomponente Informationen über die Translation, Rotation, Skalierung, Translationsdelta, und Rotationsdelta. Die Pflanzenkomponente speichert, wie viel Wasser sich in der Pflanze befindet und wie viel maximal in den Topf passt. Die Spielerkomponente hat Informationen über den Gamestate, die Kamera, Tastatur und Maus. Des Weiteren wurde die Steuerung der Kamera angepasst. Es wurde die Maus in dem Fenster gefangen, wodurch es nun möglich ist die Kamera zu bewegen, ohne vorher eine Maustaste drücken zu müssen. Zudem wurde die Blickrichtung eingeschränkt, sodass es nicht mehr möglich

ist so lange die Maus nach oben zu bewegen, bis das Kamerabild auf dem Kopf steht. Außerdem wurde die Bewegungsgeschwindigkeit des Spielers in Bezug auf das Gehen und Rennen angepasst. Zudem wurden Systeme, wie das Fahrssystem, Pflanzensystem, Gießsystem und noch viele mehr umgesetzt. Das Pflanzensystem, reduziert bei jedem Aufruf den Wasserstand aller Pflanzen, welches alle Objekte sind, die eine Pflanzenkomponente besitzen. Der Stand kann dabei jedoch nicht negativ werden. Ein weiteres Beispiel ist das „FindePflanzensystem“, welches aus allen Pflanzen diejenige aussucht, welche am wenigsten Wasser hat und diese als neues Ziel festlegt. Da bei der Implementierung einige Komponenten und Systeme erzeugt werden, ist es entscheidend einen Überblick über diese zu bewahren. Aus diesem Grund wurden im Laufe des Projekts Aufräumarbeiten durchgeführt. Das bedeutet, es wurden Komponenten entfernt, die nicht mehr benötigt wurden. Der Rest wurde in einer Datei zusammengefasst. Das erleichtert zum einen das Einbinden von diesen in anderen Dateien, zum anderen findet man sie leichter. Das Gleiche wurde mit kleinen Systemen gemacht, wie beispielsweise dem AIssystem, dem Pflanzensystem und den beiden Gießsystemen.

3.2 Multiagenten System

3.2.1 State Machines

Wir haben noch keine State Machines für den Roboter implementiert, weil der Roboter sich nur in den zwei Zuständen Pflanzen gießen und Mit Spieler reden befinden kann. Diese Funktionalität kann über ein einfaches if-else abgedeckt werden und deswegen sind State Machines noch nicht nötig.

3.2.2 Behaviour Tree

Um den Behaviour Tree für den Handlungsablauf des Roboters effizient umzusetzen, wurde die Bibliothek BehaviorTree.CPP genutzt. Diese wurde ausgesucht, da sie vcpkg unterstützt, für C++ entwickelt wurde und alle nötigen Funktionalitäten bietet, ohne zu komplex zu sein. Im Falle des Gießroboters gibt es fünf Schritte, die er ausführen muss, um eine Pflanze zu gießen. Zuerst muss eine Pflanze ausgewählt werden, die den geringsten Wasserstand hat. Im Anschluss wird der Weg von der aktuellen Position zu dieser gesucht. Daraufhin fährt der Roboter den Weg ab. Dort angekommen muss er sich noch so drehen, dass sich sein Gießarm über dem Blumentopf befindet, damit am Ende die Pflanze gegossen werden kann und sich ihr Wasserstand erhöht. Alle diese Schritte wurden in dem Behaviour Tree als Blattknoten implementiert. Darüber sitzt ein sogenannter Sequenzknoten, dieser merkt sich welches Blatt als letztes ausgeführt wurde und ob dieses beendet wurde. Wenn es noch nicht fertig ist, wird es im nächsten Schritt erneut aufgerufen. Dies geschieht so lang, bis das Blatt fertig ist. Der Sequenzknoten erhält als Rückmeldung „SUCCES“, oder, wenn

das Blatt noch nicht fertig ist, „RUNNING“. Wenn ein Blatt fertig ist, wird das nächste ausgeführt. Wurde jedes Blatt einmal ausgeführt und hat jeweils „SUCSES“ zurückgegeben, so beginnt der Sequenzknoten erneut das erste Blatt aufzurufen. Die Baumstruktur wird in einem XML-Dokument gespeichert. Die Funktionalität der einzelnen Knoten befindet sich in einer C++ Datei und die Knoten und Blätter werden in einer sogenannten Treefactory erstellt, welche ebenfalls in C++ geschrieben ist.

3.2.3 Steering Behaviour

Auch die Bewegungssteuerung wurde auf Basis des ECS implementiert. Dazu wurden ein Fahrssystem und Komponenten, wie Fahr-, Hindernis-, Pfad- und Positionskomponente, erstellt. Das Fahrssystem untersucht dabei, welcher Bewegungsmodus aktuell ausgewählt ist und passt darauf basierend das Bewegungsverhalten an. Es wird zwischen dem Pfadverfolgen und Drehen unterschieden. Ist Pfadverfolgen aktiviert, so folgt der Roboter einem Pfad, welcher sich in der Pfadkomponente befindet, die dem Entity Roboter zugeordnet wird. Der Pfad ist dabei eine Liste von Wegpunkten, die der Agent abfahren soll. Dazu wird das, wie im Konzeptteil beschriebene Suchverhalten genutzt. Zudem wird, während der Roboter die Punkte abfährt, darauf geachtet, dass er mit keinen Hindernissen kollidiert. Diese sind alle Objekte in der Szene, welche eine Hinderniskomponente haben. In dieser ist der Radius des Gegenstandes oder Spielers gespeichert. Damit das Fahrverhalten des Roboters sichtbar ist, muss die Position in der Positions komponente nach jedem Durchlauf aktualisiert werden. Das Ausweichverhalten wurde, wie oben erwähnt, implementiert. Das bedeutet, es wird überprüft, ob das Objekt, welches am nächsten zu dem Roboter ist, sich vor dem Agenten befindet. Ist dies der Fall, so wird getestet, ob der Roboter nach dem nächsten Schleifendurchlauf mit diesem kollidiert, wenn ja wird der aktuelle Wegpunkt nach rechts oder links verschoben. Es wird dabei berücksichtigt, bei welcher Verschiebung die Abweichung zum Pfad am geringsten ist. Je nach dem wird der Pfad angepasst. Da der Punkt nun eine neue Position hat, weicht der Roboter dem Hindernis aus. Soll der Roboter sich drehen, um beispielsweise mit dem Spieler zu reden, oder um eine Pflanze zu gießen, muss in der Fahrkomponente der aktuelle Modus auf Drehen gesetzt werden. In ihr sind zudem Informationen über die Masse, die Höchstgeschwindigkeit, die maximale Lenkkraft und die Größe des Sicherheitsabstandes gespeichert. Bei der Implementierung des Drehverhaltens werden jedoch keine weiteren Punkte bestimmt, die der Agent anschauen soll, sondern der Drehwinkel wird interpoliert. Das bedeutet, dass der Wert für diesen reduziert wird, sodass der Roboter sich unabhängig von der Framerate langsam in die richtige Richtung dreht und zum Schluss korrekt ausgerichtet ist. Alle Objekte, die sich in dem Projekt bewegen sollen, erhalten eine Fahrkomponente, aktuell sind dies die zwei Gießroboter. Die Bewegungssteuerung hat noch zahlreiche Probleme. Da die Steuerung des Roboters auf einfachen mathematischen Formeln basiert, verfügt der Roboter nur über ein stark eingeschränktes Wissen über die Umgebung. Aus diesem Grund kann er zwar einem

Hindernis ausweichen, es wird jedoch dabei immer nur ein Objekt berücksichtigt. Das bedeutet, dass er beispielsweise rechts an einem Hindernis vorbeifährt und damit auf einen Kollisionskurs mit einem anderen Gegenstand gerät, was nicht passiert wäre, wenn er dem ersten Hindernis auf der linken Seite ausgewichen wäre. Hätte der Roboter ein besseres Verständnis über die aktuelle Situation, so könnte er sich besser in der Welt bewegen. Zudem wird bei dem Ausweichverhalten nicht zwischen Objekten unterschieden, die stillstehen, oder sich bewegen. Es könnte passieren, dass er einen Bogen nach rechts fährt, um einem Objekt auszuweichen, dieses sich jedoch so bewegt, dass es trotz Ausweichmanöver immer noch im Weg befindet. Im schlimmsten Fall könnte dieses Hindernis sich aufgrund seiner Bewegung dauerhaft zwischen Roboter und Ziel befinden. In dieser Situation wäre es für ihn mit dem aktuell implementierten Bewegungssystem unmöglich die Pflanze zu erreichen. Des Weiteren kann auf Grund der Simplizität nicht garantiert werden, dass der Roboter, während er auf eine Pflanze zufährt, nicht auf eine Kreisbahn um diese gelangt. Dies würde geschehen, wenn das Ziel sich links beziehungsweise rechts von dem Roboter befindet und er eine Kurve fahren muss, um es zu erreichen. Wenn die Krümmung der Kurve stärker ist, als der Roboter mit seinem maximalen Lenkwinkel fahren kann, ist es für ihn unmöglich diesen Pfad zu fahren. Im schlimmsten Fall befindet sich das Ziel im Mittelpunkt der Kurve, die der Roboter aktuell fährt und dem maximalen Lenkwinkel entspricht. Dies hätte zur Folge, dass sich der Abstand zur Pflanze nicht verändert und er sie nicht erreichen kann. Das aktuell implementierte Bewegungssystem kann nicht erkennen, ob er auf einer Kreisbahn ist. Daher können auch keine Korrekturzüge ausgeführt werden. Außerdem ist die Bewegungssteuerung nicht intuitiv, da Eigenschaften wie die Masse des Roboters oder seine maximale Krafteinwirkung diese beeinflussen. Es ist dementsprechend bei der Erstellung neuer Roboter schwierig passende Werte für die Merkmale zu finden, um das gewünschte Fahrverhalten zu erhalten.

Während der Programmierung traten zahlreiche Herausforderungen auf, insbesondere aufgrund der begrenzten Erfahrung mit C++. Zum Beispiel wurde in der Funktion, die das Gießen der Pflanzen steuert, zu Beginn ein Pointer auf die ID des gießenden Roboters genutzt. Allerdings wurde dieser ungültig, sobald der Funktionsaufruf beendet wurde. Dies führte dazu, dass keine Unterscheidung zwischen den beiden Robotern mehr möglich war. Die ID musste aus diesem Grund als Variable vom Typ `flecs::id_t` gespeichert werden. Ein weiteres Problem entstand bei der Berechnung der Steuervektoren. Dieser zeigte beispielsweise in Richtung minus unendlich, wenn durch Zufall das Ziel die Koordinaten (0,0,0) hatte. Zudem wurden verschiedene IDEs verwendet. Das hatte zur Folge, dass Veränderungen, die in einer IDE gemacht wurden zu Fehlern in einer anderen führen konnten. Ein Beispiel hierfür ist die Deaktivierung von Backface Culling in CLion. Wenn nun das Projekt in Visual Studio gestartet wurde, kam es zu einem fehlerhaften Speicheraufruf. Durch sorgfältiges Analysieren und Zusammenarbeiten konnten jedoch viele Probleme im Verlauf des Projekts bewältigt werden.

3.3 Partikelsystem

Um das Partikelsystem in dem Projekt umzusetzen, wurden zwei neue Komponenten mit zusätzlich zwei weiteren Systemen implementiert. Die Komponenten waren dabei eine Emitter- und eine Partikelkomponente. Die Emitterkomponente enthält Informationen über die Anzahl der Partikel, die pro Iteration erschaffen werden und die relative Position, wo dies geschehen soll. In dem Projekt muss berücksichtigt werden in welche Richtung der Roboter schaut, damit die Wassertropfen an der Stelle erscheinen, wo sich der Arm des Roboters befindet. In der Partikelkomponente wird die sogenannte Lebensdauer gespeichert, die es erlaubt die Tropfen nach einer definierten Zeit verschwinden zu lassen beziehungsweise diese zu entfernen. Das ist von großer Bedeutung, da sonst immer mehr Entities erschaffen werden, was die Performance stark reduzieren würde und im schlimmsten Fall das Programm zum Absturz bringen kann. Das Partikelsystem erschafft Objekte mit einer Partikelkomponente, Geometriekomponente und einer Positions komponente. Die Geometriekomponente ist dabei wichtig, um den einzelnen Tropfen ein Modell zuzuordnen, anderenfalls wäre der Benutzer nicht in der Lage das Wasser zu sehen. Bei der Erstellung der Entities wird zudem darauf geachtet, dass die Position, an der sie erschaffen werden, leicht variiert. Dadurch entsteht ein breiterer Strahl aus Tropfen, was das Gießen realistischer darstellt. Das zweite System ist das PartikelRemovalSystem. Dieses reduziert bei jeder Iteration den Wert der Lebensdauer aller Tropfen, bis dieser kleiner gleich Null ist. In diesem Fall wird das entsprechende Objekt entfernt.

3.4 Navigationssystem

Um einen Pfad für ein Entity zu finden, muss man dem Entity eine PathRequest-Component mit Start und Ziel hinzufügen. Das Navigationssystem liest die Daten aus dem Request aus, entfernt diesen und fügt stattdessen eine PathComponent mit dem gefundenen Pfad hinzu.

Um einen Pfad von einem Startpunkt zu einem Zielpunkt zu finden, sind drei Schritte nötig. Zuerst muss man die Navigationsnetzpolygone finden, auf denen Start und Ziel liegen. Danach sucht man die Polygone, die Start- und Zielpolygon verbinden, und zum Schluss kann man diese Liste an Polygonen zu einem Pfad aus Punkten umwandeln.

3.5 Physiksystem

Die ECS-Bibliothek Flecs unterstützt Entity Event Listener, sogenannte Observer. Man kann eine Callbackfunktion definieren die gerufen wird, wenn eine Komponente eines Entities hinzugefügt, modifiziert oder entfernt wird. Das Physiksystem nutzt zwei Observer: ein Observer überwacht, wenn eine Physikkomponente zu einem Entity

hinzugefügt wird und fügt den enthaltenen Rigidbody auch in die Physikwelt ein. Der zweite Observer wird gerufen, wenn ein Entity aus der Welt entfernt wird, welches eine Physikkomponente besitzt. In dem Fall wird auch der dazugehörige Rigidbody aus der Physikwelt entfernt.

Theoretisch benötigen wir nur die Kollisionserkennung und -auflösung von Bullet. Aber es war deutlich schwerer, diese beiden Funktionen getrennt von der Physiksimulation zu rufen, als die Physikwelt und die ECS-Welt zu synchronisieren. Deswegen funktioniert die Kollisionserkennung und -auflösung jetzt in drei Schritten. Zuerst werden alle Entities mit Physikkomponente und Transformationskomponente gesucht und die Geschwindigkeit und Position der Transformationskomponente auf den Rigidbody der Physikkomponente übertragen. Dann wird die Simulation der Physikwelt mit dem jetzigen Zeitschritt gerufen. Zum Schluss werden die Geschwindigkeit und Position des Rigidbodies wieder in die Transformationskomponente geschrieben.

Die Plattformen, Zäune und Wendeltreppe sind statische Geometrie und zusätzlich noch konkav. Deswegen sind diese mit einem Deiecks-Collider repräsentiert. Dabei wird einfach das 3D-Modell der statischen Geometrie als Collider benutzt.

Alle Entities haben Kapsel-Collider anstatt Zylinder-Collider, da diese an den Dreieckskanten der statischen Geometrie hängen geblieben sind.

3.6 Szenen Editor

Die Assimp-Variante hat nicht funktioniert, weil die Eingriffe zu tief in der Engine vorgenommen werden mussten. Außerdem war das Abstraktionslevel zu gering: wenn man nur Knoten hat, ist es schwer, Entities zu erkennen.

Das Blender Plugin wurde in Python programmiert, und man hat Zugriff auf alle Optionen, auf die man auch im Editor Zugriff hat. Um jetzt die Entities zu exportieren, wird über alle Objekte der Szene iteriert und überprüft, ob diese gelinkt sind. Ist dies der Fall, dann wird die Transformation, der Name und der Name der Modeldatei in eine JSON-Datei geschrieben. Um jetzt auch noch die statische Geometrie zu exportieren, werden alle gelinkten Objekte unsichtbar gemacht, und nur die sichtbaren Modelle werden exportiert. Alle unsichtbaren Objekte werden anschließend wieder sichtbar gemacht, um den Ausgangszustand wieder herzustellen.

In CrossForge wird dann die JSON-Datei geladen und für jeden Eintrag ein entsprechendes Entity zur Welt hinzugefügt.

3.7 Dialogsystem

Die gesamte Implementierung des Dialogsystems haben wir (Lisa und Sophie) im Pair-Coding übernommen.

3.7.1 Imgui

Die Bibliothek für Imgui konnte über vcpkg recht schnell und einfach in das Projekt eingebunden werden. Zur Erzeugung von Dialogfenstern erstellen wir mit Imgui Frames, in denen oben der aktuelle Text des Dialogs angezeigt wird und unten die möglichen Antworten als Buttons. Wir nutzen außerdem einige Style-Anpassungen, um den Dialog etwas anschaulicher für den Nutzer zu gestalten, sodass z.B. das Fenster immer im Viewport zentriert ist.

3.7.2 JsonCpp

Ähnlich wie Imgui konnten wir auch JsonCpp über vcpkg einbinden. Mithilfe dieser Bibliothek können wir dann den Dialoggraphen initialisieren. Dafür sollten die Json-Dateien die gleiche Baumstruktur aufweisen, wie der Dialoggraph selbst. Über JsonCpp werden die Informationen aus der Datei gelesen und gespeichert, sodass sie dann weiter genutzt werden können, um den Dialoggraph zu füllen.

3.7.3 Dialoggraph

Den Dialoggraph haben wir so wie im Konzeptteil beschrieben implementiert. In einer dafür erstellten Klasse werden die zuvor genannten Daten gespeichert: der Dialogtext, ein Boolean für die Unterscheidung zwischen Roboter und Spieler und die möglichen Antworten. Zusätzlich gibt es noch eine Funktion, die die Initialisierung des Baumes umsetzt und eine Funktion für das Ersetzen der Strings mithilfe der Dialogmap.

3.7.4 Dialogmap

Mit der implementierten Dialogmap kann eine Funktion aufgerufen werden, die den Name einer Pflanze zurückgibt. Der Rückgabewert dieser Funktion ist allerdings statisch festgelegt, da zum Zeitpunkt der Abgabe das Dialogsystem noch nicht in das ECS eingebunden ist. Diese Integration wäre aber notwendig um die Positionen von Roboter und Spieler zu ermitteln, die gebraucht werden um zu bestimmen welcher Pflanzennname zurückgegeben werden soll. Auch die Pflanzennamen selbst sind im ECS gespeichert, weshalb ein Zugriff darauf vom Dialogsystem aus aktuell nicht möglich ist. Weitere Funktionalitäten der Dialogmap, wie Eventhandling sind ebenfalls noch nicht implementiert.

3.8 Modelle und Szene

Wie im Konzept bereits erläutert, wurde ein beträchtlicher Teil der Anfangsphase für die Konzeptentwicklung aufgewendet. Darüber hinaus erforderte die Einarbeitung in Blender eine gewisse Zeit. Die Modellierung aller Objekte musste aufgrund verschiedener Herausforderungen mehrfach überarbeitet werden, darunter Unklarheiten im Design, zu viele Polygone aufgrund detaillierter Elemente oder inkorrekte Vorgehensweisen, usw.

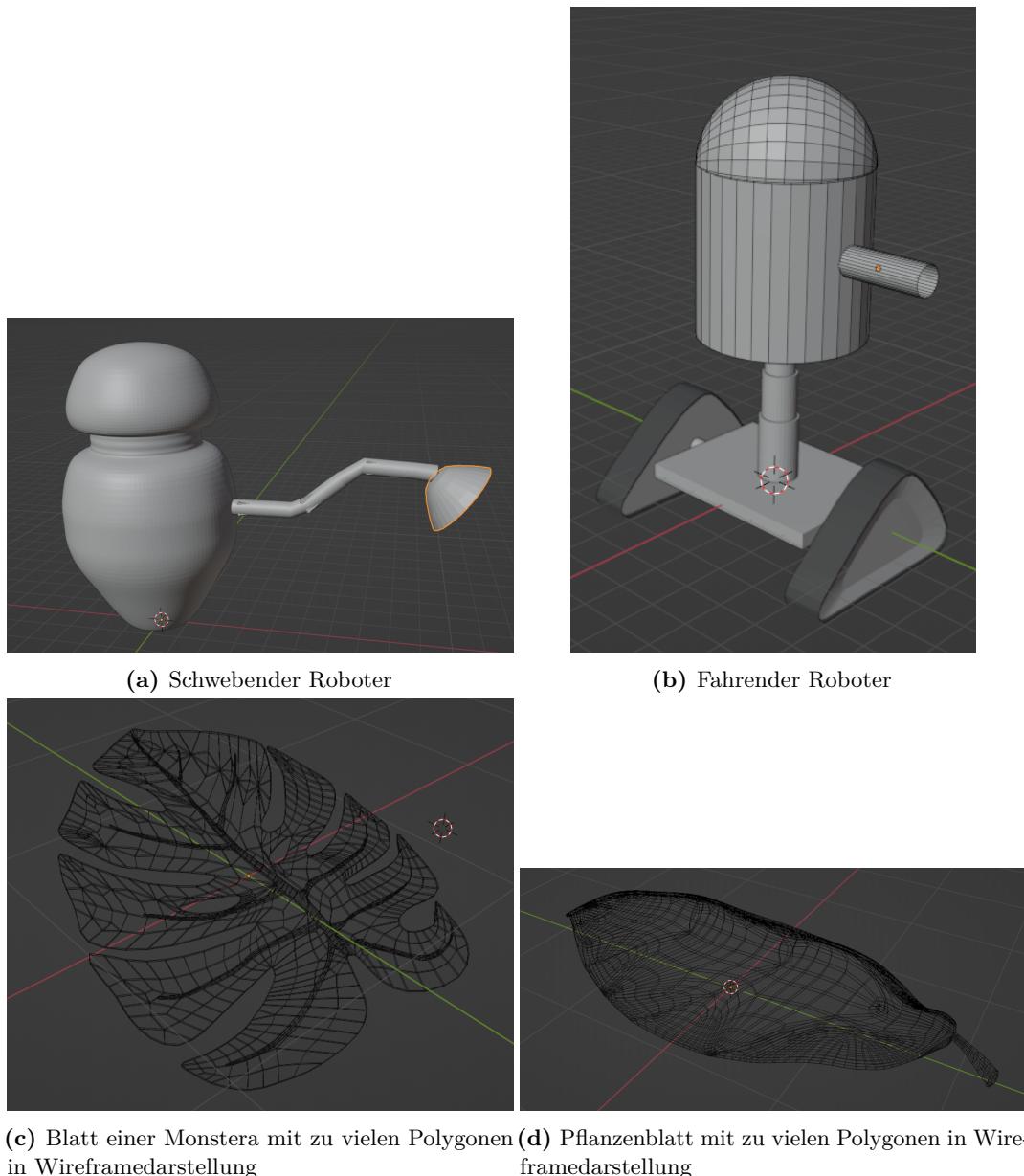


Abbildung 3.1: Erste Versuche für die Modelle

Die Modelle wurden mithilfe von Blender erstellt, während die zugehörigen Texturen eigenständig auf dem iPad mit ProCreate gezeichnet wurden.

Der Roboter wurde primär aus simplen Zylindern modelliert und entsprechend angepasst. Bei der Gestaltung lag der Fokus darauf, dass er ein nicht allzu klassisches Aussehen erhält, da er für Kommunikationszwecke genutzt wird. Die Integration der Solarplatte auf seinem Kopf erfolgte mit der Absicht, dass er sich mittels Son-

nenenergie aufladen kann. Der Arm wurde als Schlauch konzipiert, um damit die Pflanzen zu bewässern. Alle einzelnen Objekte wurden bis auf den Schlauch zum Abschluss zu einem einzigen Objekt vereint, während der Schlauch separat blieb, da er später animiert werden soll. In Bezug auf die Texturen wurden bis auf die Solarplatte alle direkt in Blender mithilfe von Geometry Nodes eingestellt. Dadurch erhielt der Körper ein metallisches Aussehen und die Räder einen gummiartigen Look. Die Solarplatte wurde in ProCreate handgezeichnet und anschließend im UV-Editor eingefügt und angepasst. Nach Rücksprache mit dem Team entstand dann unser aktueller Roboter, wie er im MVP zu sehen ist.

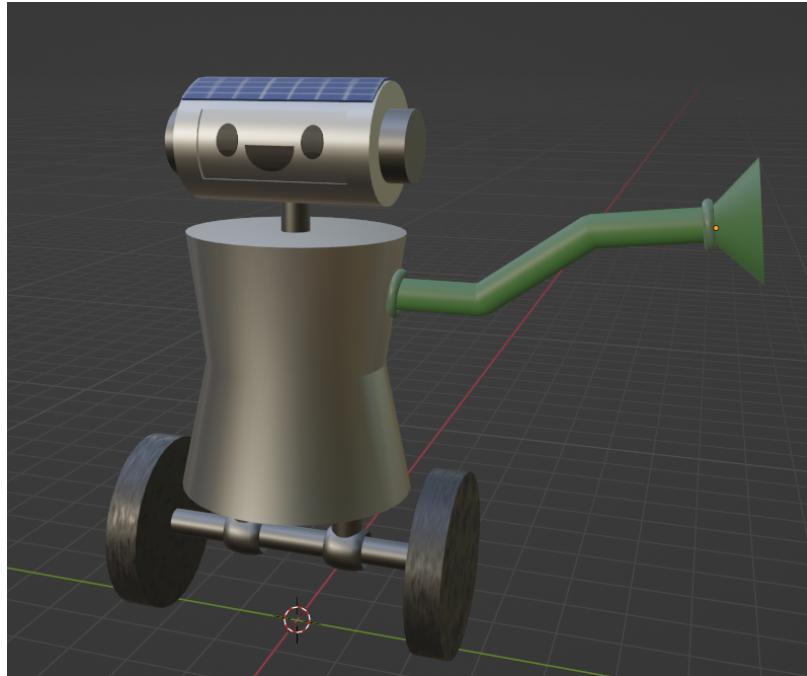


Abbildung 3.2: Finales Modell für den Roboter

Die beiden Pflanzen wurden größtenteils aus spontanen Ideen heraus gestaltet. Die Hauptüberlegung bestand darin, zwei verschiedene Pflanzen zu kreieren. Die Blumentöpfe für beide Pflanzen wurden aus simplen Zylindern modelliert. Ursprünglich wurde der Stiel der kleineren Pflanze mit verschiedenen Einstellungen der Geometry Nodes als komplexes Mesh erstellt. Diese Herangehensweise wurde jedoch verworfen und stattdessen vereinfacht. Die Blätter, die anfangs zu detailliert waren, wurden letztendlich auf zwei einfache 2D-Texturen reduziert. Sowohl die Texturen für die Blätter als auch für die Töpfe mit Erde wurden eigenständig in ProCreate erstellt. Zunächst wurde versucht, die Modelle direkt in ProCreate zu importieren und darauf zu zeichnen. Diese Methode sollte es ermöglichen, die Texturen einfach in Blender zu importieren. Leider funktionierte dieser Ansatz nicht wie erwartet, daher wurden die Texturen schließlich als 2D-Bilder gezeichnet und dann mittels UV-Mapping in

Blender angepasst. Ähnlich wie beim Roboter wurden die Pflanzen nach Absprache mit dem Team genehmigt und in das MVP integriert.



Abbildung 3.3: Finale Modelle der Pflanzen

Die größte Herausforderung lag in der Gestaltung und Implementierung der Plattformen. Wie bereits im Konzept beschrieben, wurden zunächst verschiedene Ansätze dafür entwickelt. Letztendlich wurden beide Plattformen frei nach Gefühl gestaltet, zusammen mit einer etwas unkonventionellen Verbindung zwischen ihnen. Um sicherzustellen, dass die Roboter im Notfall zwischen den Plattformen navigieren können, wurde eine Rampenstruktur um die Säule herum entwickelt. Zusätzlich erhielten sowohl die Rampe als auch die Plattformen Zäune und Geländer, um ihre Sicherheit zu erhöhen. Auch hier wurden alle Texturen eigenhändig mit ProCreate erstellt. Das Modell wurde ebenfalls mit dem Team abgestimmt, bevor es finalisiert wurde.

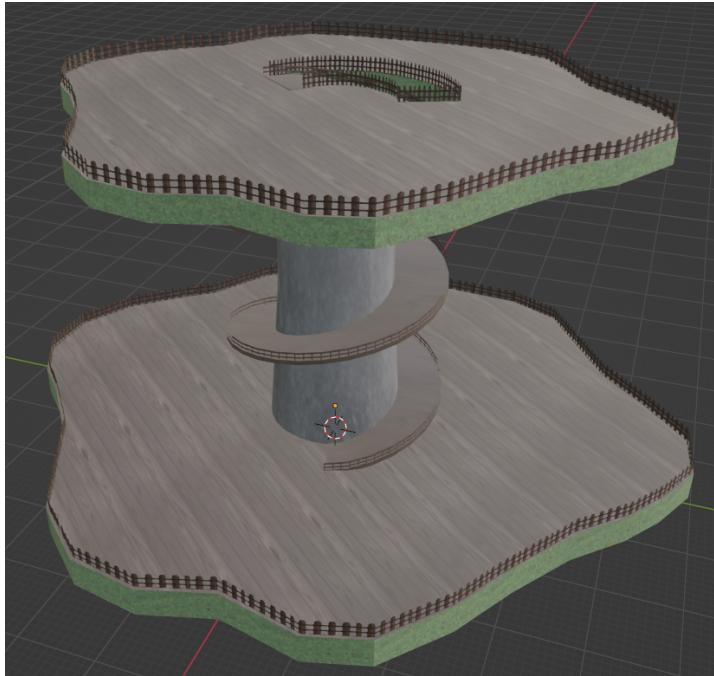


Abbildung 3.4: Finales Modell der Szene

Zum Schluss wurde leider bemerkt, dass die Rampe breiter sein sollte, um Platz für beide Roboter zu bieten, wenn sie gleichzeitig die Ebenen wechseln und sich auf der Rampe treffen. Derzeit gibt es dort keinen Raum zum Ausweichen, was dazu führt, dass einer der Roboter abdriftet. Nachdem alle Modelle fertiggestellt waren, wurden sie in der Szene zusammengesetzt und weitgehend angepasst. Das gesamte Setup wurde dann mithilfe eines Dropper-Addons in das Projekt integriert. Zunächst gab es einige Probleme und Unklarheiten, die jedoch unter der Leitung des Teamleiters erfolgreich gelöst wurden. Um die Zusammenarbeit zwischen den Modellen und anderen programmatischen Aufgaben wie dem Navigationssystem zu verbessern, wurden die Modelle in "linked" (Pflanzen und Roboter) und "static" (Plattformen & Co.) unterteilt. Nachdem die Szene weitgehend fertiggestellt und in das Projekt integriert war, mussten nur noch einige allgemeine Probleme hinsichtlich Texturen und Ordnerstrukturen gelöst werden. Die nächste Aufgabe bestand darin, die MVP-Szene weiter auszubauen und zu verschönern. Dazu wurden mit Hilfe eines entsprechenden Addons zwei Bäume und ein Haus erstellt. Auch hier wurden alle Texturen für die Bäume und das Haus mit ProCreate erstellt, wobei das Haus noch nicht vollständig fertiggestellt wurde.

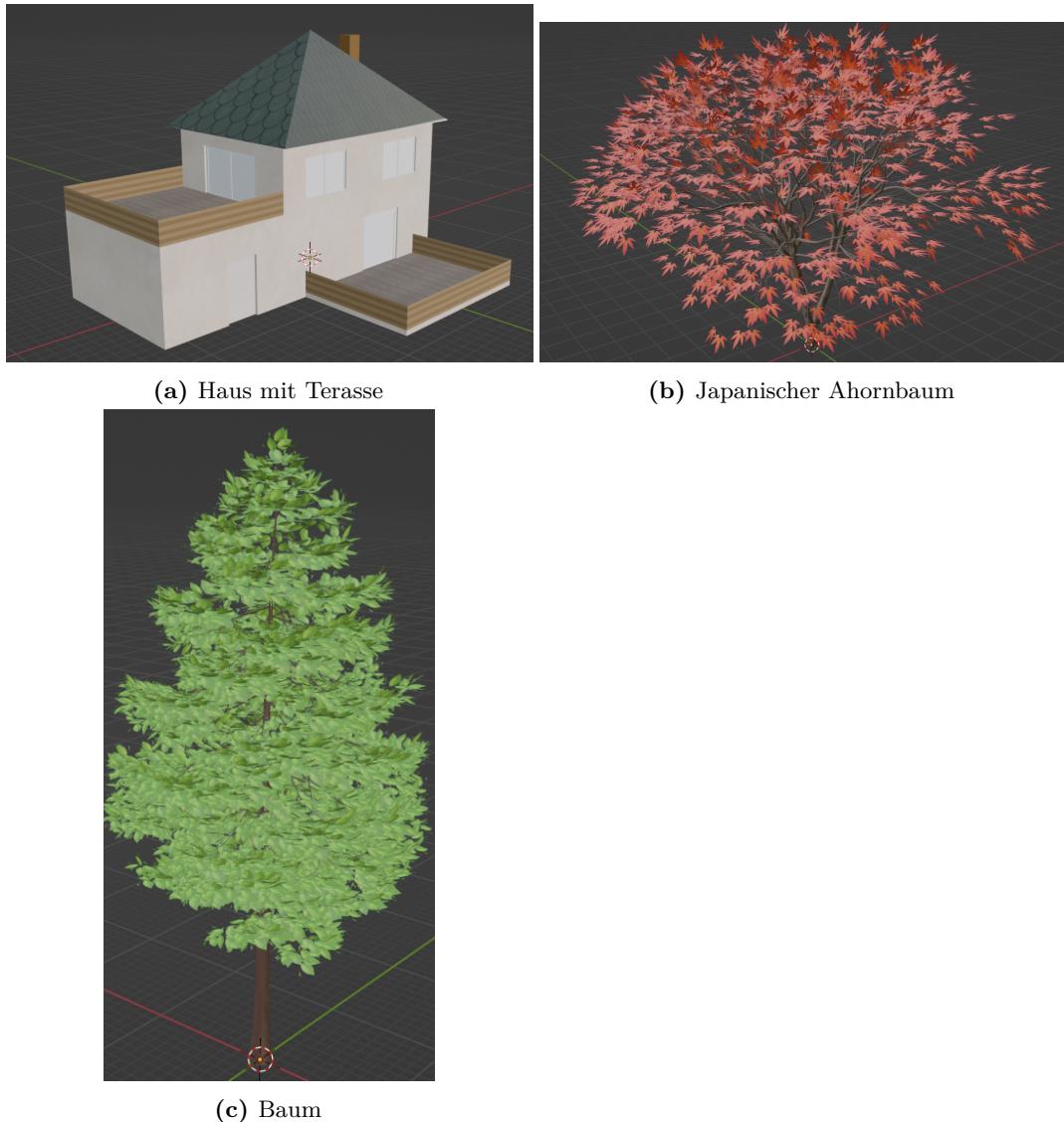


Abbildung 3.5: Modelle für erweiterte Szene

Der Ausbau der Szene wurde jedoch vorzeitig beendet, um die verbleibende Zeit dem Bericht zu widmen.

Kapitel 4

Management

Unser Praktikumsteam bestand aus sechs Personen mit unterschiedlichen Wissenständen, weswegen ich die Aufgaben generell nach Schwierigkeit und Fähigkeiten der Bearbeiter verteilt habe. Angelique bricht ihr Informatikstudium ab und bekommt deshalb die kreative Aufgabe, die Modelle und die Szene zu erstellen. Lisa und Sophie haben noch sehr wenig Erfahrung und sollten deshalb die einfachsten Themen angehen. Auch Leon verfügt noch über wenig Erfahrung, aber dafür über eine gewisse Leidensfähigkeit. Mittelschwere bis schwere Aufgaben hat er ohne Zögern in Angriff genommen. Carlo und ich haben die meiste Erfahrung und arbeiten auch schon mehrere Stunden in der Woche als Werkstudenten. Deswegen haben wir uns auf die schwersten Probleme fokussiert.

Als Team haben uns darauf geeinigt, etwa sechs Stunden in der Woche dem Teamorientierten Praktikum zu widmen, da mehr neben Studium und Arbeit nicht möglich war. Deswegen haben wir uns auch gegen einen Scrum Prozess entschieden, weil die Retros, Plannings und Sprintwechsel zu viel Zeit in Anspruch nehmen. Um uns zu organisieren, wollten wir Jira einsetzen, jedoch hat die Software niemand wirklich benutzt. Da der Fortschritt langsam war, konnte man auch ohne Hilfsmittel sehr gut den Überblick darüber behalten, wer woran gearbeitet hat. Wir haben uns einmal in der Woche entweder über Discord oder in Präsenz getroffen, um uns zum Projektfortschritt zu synchronisieren.

4.1 Aufgabenverteilung und Arbeitsweise

Initial habe ich die Aufgaben wie folgt verteilt: Lisa implementiert das Dialogsystem, Sophie den Szeneneditor und ich das Multiagentensystem. Leon sucht eine ECS-Bibliothek und integriert diese, und Carlo beschäftigt sich mit dem Navigationssystem.

Da die meisten mit ihrer Aufgabe überfordert waren und nicht wussten, wo sie anfangen sollten, habe ich mit Lisa, Sophie und Leon jeweils eine Stunde in der Woche

eine Pair-Coding-Session gehalten. Das hatte den großen Vorteil, dass ich dann genau wusste, wer welchen Fortschritt hat. Dem gegenüber standen zwei große Nachteile. Erstens konnte ich somit keine Features implementieren, weil meine sechs Stunden voll mit Pair Coding und Management waren, und zweitens ist es sehr verlockend, bei jeder kleinen Hürde aufzugeben und auf das nächste Pair Coding zu warten, weil das jeweilige Problem dort ja sowieso gelöst wird. Zu diesem Zeitpunkt war Leon mit der Integration von Flecs und ich mit dem Konzept des Multiagentensystems fertig, so dass ich entschied, dass Leon die Implementierung von letzterem übernehmen kann.

In den Pair Coding Sessions haben wir gute Fortschritte mit dem Dialog- und Multiagentensystem gemacht. Die Aufgabe mit dem Szeneneditor bereitete Sophie keine Freude, und sie hat auch nicht verstanden, welche Schritte nötig sind und warum. Deswegen habe ich eigentlich nur diktiert, was sie schreiben soll. Das hat uns beiden jedoch keinen Spaß bereitet, weswegen wir zu dritt entschieden haben, dass es besser ist, wenn ich den Szeneneditor alleine fertigstelle. Sophie hat als nächstes Lisa mit beim Dialogsystem unterstützt.

In den Sommerferien haben wir die Pair Coding Sessions aufgrund von Prüfungen und Urlauben pausiert. Nach den Sommerferien habe ich eine Retrospektive vorbereitet und mit dem gesamten Team durchgeführt. Dabei haben sich verschiedene bevorzugte Arbeitsweisen heraus kristallisiert. Während Leon die Pair Coding Sessions bevorzugte und weiter führen wollte, sagten Lisa und Sophie, dass sie ihnen wenig bringen und sie eigenständig weiter arbeiten möchten. Auf diese Art und Weise haben wir dann auch bis zum Ende weiter gearbeitet.

4.2 Analyse der Probleme

Das Dialogsystem, an dem Lisa und Sophie gearbeitet haben, wurde nicht rechtzeitig fertig, weil es noch in das ECS integriert werden musste und das eine sehr zeitaufwendige Aufgabe ist. Generell habe ich darauf geachtet, dass sich die einzelnen Feature Branches nicht zu weit voneinander entfernen. Deswegen habe ich am 09.10.2023 alle Branches in den Master gemerged und danach den Master wieder in alle Featurebranches. Damit war der komplette Projektstand synchronisiert, und die beiden hätten anfangen können, das Dialogsystem in das Entity Component System zu integrieren. Wir haben aber gemeinsam entschieden, dass wir noch Zeit haben und das Dialogsystem zuerst fertig gestellt werden soll. Die beiden haben dann am 05.12.2023 begonnen, das Dialogsystem in das Entity Component System zu integrieren, sind jedoch nicht fertig geworden. Rückblickend wäre es sinnvoller gewesen, zuerst die Integration zu machen und danach das Dialogsystem fertig zu stellen.

Carlo hat das Praktikum, aus persönlichen Gründen, vor der Bewertung abgebrochen. Er besaß von uns allen die meiste C++ Erfahrung und arbeitet 18 Stunden

in der Woche als Werkstudent, weswegen er die Aufgabe mit der größten Unsicherheit hinsichtlich Komplexität lösen sollte. Seine Aufgabe bestand darin, das Pathfinding System in CrossForge zu integrieren. Aufgrund von Studium, Arbeit, Umzug, Problemen mit seiner Entwicklungsumgebung und mäßiger Dokumentation von Detour hat er nur sehr langsam Fortschritte gemacht. Bei der Retrospektive haben wir über diese Probleme gesprochen, und ich habe ihm vorgeschlagen eine andere, leichtere Aufgabe zu übernehmen. Er wollte nochmal einen Anlauf wagen, hat aber dann nach zwei Wochen mein Angebot angenommen. Ich habe seine Aufgabe übernommen, und er sollte als nächstes das Partikelsystem implementieren. Durch seine kontinuierlichen IDE Probleme hat er aber die Lust am Programmieren verloren. Das Partikelsystem hat dann Leon implementiert. Carlo sollte ab 12.12.2023 Angelique bei den Modellen und einer hübscheren Szene helfen. Er hat in der Woche vor Weihnachten noch die bestehende Szene ein wenig verbessert. Dann waren Weihnachtsferien und im Januar haben wir uns intensiv dem Bericht-Schreiben gewidmet.

Projektmanagement ist als Student schwer, weil man auf die intrinsische Motivation der Teammitglieder hoffen muss. Manager in der Industrie haben richtige Druckmittel (Mitarbeiter feuern) oder extrinsische Motivation (Lohn). Beides kann ich als studentischer Leiter nicht einsetzen. Zudem ist die Arbeitsverteilung, wie man in der Tabelle 4.1 gut erkennen kann, nicht gleichmäßig ausgefallen. Das ist auf die Tatsache zurückzuführen, dass die Teammitglieder unterschiedlich gut mit den ihnen zugewiesenen Aufgaben zurechtgekommen sind. Mitglieder, die ihre Aufgaben schnell fertig gestellt haben, haben weitere übernommen wodurch sich das Ungleichgewicht weiter verstärkt hat.

Metric	Leon	Sophie	Carlo	Lisa	Linus	Angelique
Insertions	1.877 (7%)	341 (1%)	296 (1%)	140 (0%)	23.092 (82%)	2.305 (8%)
Deletions	1.010 (20%)	143 (3%)	6 (0%)	43 (1%)	3.775 (76%)	3 (0%)
Files	141 (35%)	16 (4%)	2 (0%)	6 (1%)	230 (57%)	7 (2%)
Commits	39 (33%)	6 (5%)	3 (3%)	2 (2%)	64 (55%)	3 (3%)
Lines changed	2.887 (9%)	484 (1%)	302 (1%)	183 (1%)	26.869 (81%)	2.308 (7%)

Abbildung 4.1: Arbeitsverteilung anhand Git Historie im Zeitraum von 01.06.2023 bis 22.01.2024. Github Nutzernamen durch Klarnamen ersetzt

Kapitel 5

Zusammenfassung & Ausblick

In diesem Teamorientierten Praktikum haben wir grundlegende Systeme geschaffen, um Multiagentensimulationen durchzuführen.

Als Grundlage haben wir CrossForge genutzt und in diese Engine ein Pathfinding System, einen flexiblen KI-Stack und eine Physikengine integriert. Zusätzlich haben wir einen Szeneneditor für CrossForge geschrieben und die Grundsteine für ein Dialogsystem gelegt.

Das Zusammenspiel der Komponenten haben wir an einer Beispielszene gezeigt, in der Roboter sich durch die Welt bewegen und Pflanzen gießen.

Als nächstes würden wir das Dialogsystem vervollständigen und Animationen zu den Akteuren hinzufügen. Danach würden wir verschiedene Agenten implementieren und die Szene erweitern.

Anhang A

Nachsätze

A.1 Arbeitsaufteilung

- Angelique: Modelle
- Lisa und Sophie: Dialogsystem
- Leon:
 - Anbindung von Entity Component System: Flecs
 - Konzept Steering Behaviour
 - Implementierung vom Multiagentensystem
 - Konzept und Implementierung Partikelsystem
- Linus:
 - Projekt Management
 - Architektur
 - Konzept Multiagentensystem
 - Konzept und Implementierung von Navigationssystem, Physiksystem, Szenen Editor

Literatur

- [] *Blender 2.8x Curved/Spiral Road modeling (easy way)*. URL: <https://www.youtube.com/watch?v=dCcbWCbs2aA>.
- [] *Blender 3.0 Beginner Geometry Nodes Tutorial*. URL: https://www.youtube.com/watch?v=uslTaqiv_7k.
- [19] *AI Arborist: Proper Cultivation and Care for Your Behavior Trees*. Mai 2019. URL: https://www.youtube.com/watch?v=Qq_xX1JCreI.
- [22] *Game AI Basics*. Apr. 2022. URL: https://www.youtube.com/live/G5A0-_4dFLg?feature=share&t=7129.
- [23] *How does videogame AI make its decisions? (FSM, Behaviour Trees, BDI, GOAP) / Bitwise*. Jan. 2023. URL: https://www.youtube.com/watch?v=ValJk151_y8.
- [Rey02] Craig Reynolds. „Steering Behaviors For Autonomous Characters“. In: (Juni 2002).