



Computational Geometry and Virtual Reality - RunAR

Christian Schaf, Jannis Lindenberg, Vural Yilmaz

01. Januar 2019

Inhaltsverzeichnis

1	Einleitung	3
2	Grundlagen	4
2.1	A* Algorithmus	4
2.1.1	Funktionsweise	4
2.2	Unity	6
2.3	Vuforia	7
2.3.1	ImageTargets	7
2.3.2	Object Recognition	8
3	Recherche	9
3.1	A* Algorithmus	9
3.2	Imagetargets	10
3.3	Objekterkennung	10
3.3.1	Tensorflow	10
3.3.2	OpenCv	11
3.3.3	Vuforia Object Recognition	11
4	Konzept	12
4.1	Definition von Funktionsanforderungen	12
4.2	Spielkonzept	12
4.2.1	Entwicklung des Spielkonzepts	12
4.2.2	Finales Spielkonzept	12
5	Umsetzung	13
5.1	Implementierung des A* Algorithmus	13
5.2	ImageTargets	15
5.3	Erkennung von Objekten	15
5.4	UI	16
6	Zusammenfassung und Ausblick	17
6.1	Zusammenfassung	17
6.2	Ausblick	17
6.3	Fazit	17
7	Anhang	18
7.1	A* Beispiel	18

Abbildungsverzeichnis

2.1	Beispiel Node	4
2.2	A* Ausführungsschritte 1-3	5
2.3	A* Ausführungsschritte 4-6	5
2.4	A* Ausführungsschritte 7-9	5
2.5	A* Ausführungsschritte 10-12	6
2.6	A* Schritt 13	6
5.1	A* Implementation in Unity	13
5.2	StarGrid Settings	13
5.3	Binary-Min-Heap Beispiel	14
5.4	PathRequestManager Settings	14
5.5	Vuforia Datenbank	15
5.6	Eingescanntes Objekt mit Nullpunkt	15

Kapitel 1

Einleitung

Im Rahmen des Moduls “Computational Geometry and Virtual Reality” wird eine Augmented Reality App entwickelt. Das Ziel dieses Projekts ist es, Objekte in einer durch AR erweiterten Realität zu erkennen und in die Spielumgebung zu integrieren. Der Algorithmus-gesteuerte Spieler soll durch den A* Algorithmus möglichst effizient den kürzesten Weg zwischen Start und Ziel nutzen, dabei variabel auf die sich ständig ändernde Spielumgebung reagieren - wird ein neues Hindernis erkannt, muss der Weg des Computergesteuerten Spielers zum Ziel neu berechnet werden. Die Motivation hinter dieser Projektidee ist unter anderem im Rahmen dieses Projekts einen kleinen Einblick in die Entwicklung von Augmented Reality zu erhalten. Zusätzlich soll in Erfahrung gebracht werden, wie weit Augmented Reality Anwendungen mit der echten Umgebung interagieren können.

Kapitel 2

Grundlagen

2.1 A* Algorithmus

Der A*-Algorithmus ist ein Wegfindungs-Algorithmus, der von Peter Hart, Nils Nilsson und Bertram Raphael entwickelt wurde. Sein Ziel ist es den schnellsten Weg in einem Grafen vom Startknoten zum Zielknoten zu finden.

2.1.1 Funktionsweise

Der Ausgangspunkt bildet ein zwei-dimensionales Array in dem es Felder gibt, die entweder Pfad und Hindernis darstellen. Es wird ein Start- sowie Zielfeld festgelegt. Beide sind dem Algorithmus während der Ausführung bekannt. Sobald das Spielfeld erstellt sowie ein Start- und Zielfeld gewählt wurde, sind die Mindestvoraussetzungen erfüllt. Ein Feld wird auch als Node bezeichnet und hat drei wichtige Attribute (siehe Abbildung 2.1). Eines der Attribute sind die *G-Kosten*, welche die Distanz zum Start-Node darstellen bzw. die Kosten für das Beschreiten des bisherigen Wegs. Weiterhin gibt es die *H-Kosten*, welche die Distanz zum Ziel darstellen. Das dritte Attribut sind die *F-Kosten*. Diese werden berechnet indem man die *H-Kosten* mit den *G-Kosten* addiert (siehe Formel 2.1).

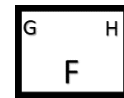


Abbildung 2.1:
Beispiel Node

$$F_{cost} = G_{cost} + H_{cost} \quad (2.1)$$

Für die *H-Kosten* wird eine Heuristik verwendet. Hierbei wurde sich aus Simplizität für die *Manhattan-Methode* entschieden. Bei dieser Methode ignoriert man den Node-Typ (Pfad oder Hindernis) und berechnet einfach die Kosten als würde man direkt, ohne Diagonalen, zum Ziel gehen. Für die Kostenberechnung allgemein muss man sich auf einen numerischen Wert für das horizontale bzw. vertikale und diagonale Bewegung zum nächsten Node festlegen. Als Wert für die horizontale bzw. vertikale Bewegung wird 10 ($1 * 10$) festgelegt. Für die diagonale Bewegung wird 14 ($\sqrt{2} * 10$) verwendet. Mit diesen Werten lassen sich nun die Gesamtkosten eines Nodes berechnen.

Der Algorithmus arbeitet mit zwei Listen, der *Open List* und der *Closed List* also die offene und geschlossene Liste. In der *Open List* befinden sich die Felder, die für einen möglichen Pfad in Frage kommen. In der *Closed List* sind die Nodes, welche schon beschritten wurden.

In Schritt 1 von Abbildung 2.2 sind die hellblauen Felder als Start und Zielfeld markiert. Das Startfeld wird nun im zweiten Schritt (Abbildung 2.2b) in die offene Liste aufgenommen. Von ihm werden jetzt die Nachbarn ermittelt und diese der offenen Liste hinzugefügt. Das Startfeld wird dann der geschlossenen Liste hinzugefügt, weil es jetzt als Beschrifteten gilt. Aus der offenen Liste wird nun das Feld mit den geringsten F-Kosten beschrifteten (Abbildung 2.2c). Der Node rechts vom Start hat die geringsten Kosten, wird nun beschrifteten (in Lila gekennzeichnet). Dieser Node merkt sich seinen Parent, um später seinen Weg zurück verfolgen zu können. Die Parent Nodes erkennt man anhand der Pfeile, die auf sie gerichtet sind. Die Nachbarn des eben beschrifteten Nodes sind bereits in der offenen Liste, weshalb sich an dieser Liste nichts ändert.

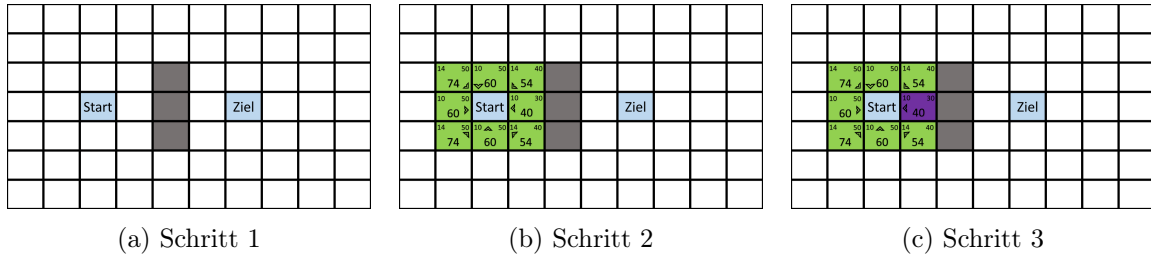


Abbildung 2.2: A* Ausführungsschritte 1-3

Nun wird wieder der Node mit den geringsten Kosten gewählt. Er wird in die geschlossene Liste aufgenommen und seine Nachbar werden zur offenen Liste hinzugefügt Schritt 1 von Abbildung 2.3. Dieser Vorgang wiederholt sich nun neun weitere Male (siehe Abbildung 2.4, 2.5 und 2.6).

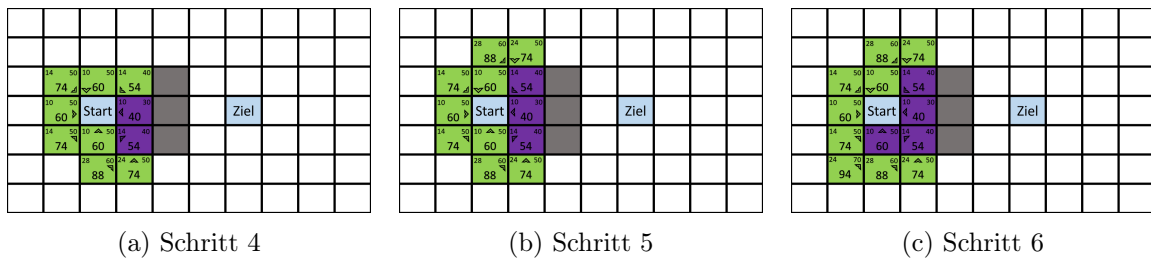


Abbildung 2.3: A* Ausführungsschritte 4-6

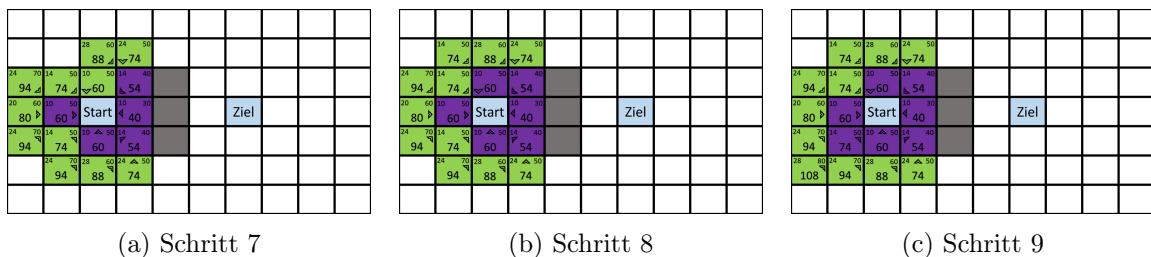


Abbildung 2.4: A* Ausführungsschritte 7-9

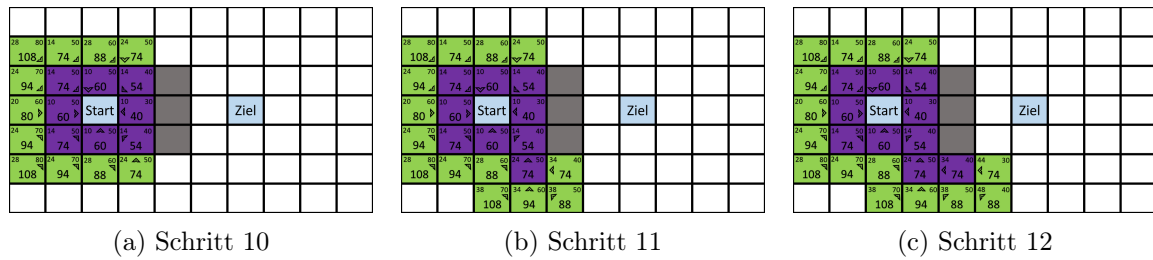


Abbildung 2.5: A* Ausführungsschritte 10-12

Befindet sich das Ziel in der offenen Liste so ist das Ziel gefunden bzw. erreicht worden. Sollte sich kein Node mehr in der *Open List* befinden gibt es keinen Pfad vom Start bis zum Ziel. Hierbei handelt es sich auch zugleich um den *Worst Case*, bei dem alle Nodes besucht wurden. Der kürzeste Pfad aus den beschrifteten Nodes kann nun mit der Zurückverfolgung anhand der Parents herausgefunden werden. Im Detail heißt das, das man vom Ziel zu dessen Parent dann zum nächsten Parent usw. geht, bis man wieder auf das Startfeld trifft. Damit hat man den kürzesten Weg vom Start bis zum Ziel bestimmt. (Im [Anhang](#) ist eine Animation eingebettet, welche mit dem Adobe Reader abgespielt werden kann.)

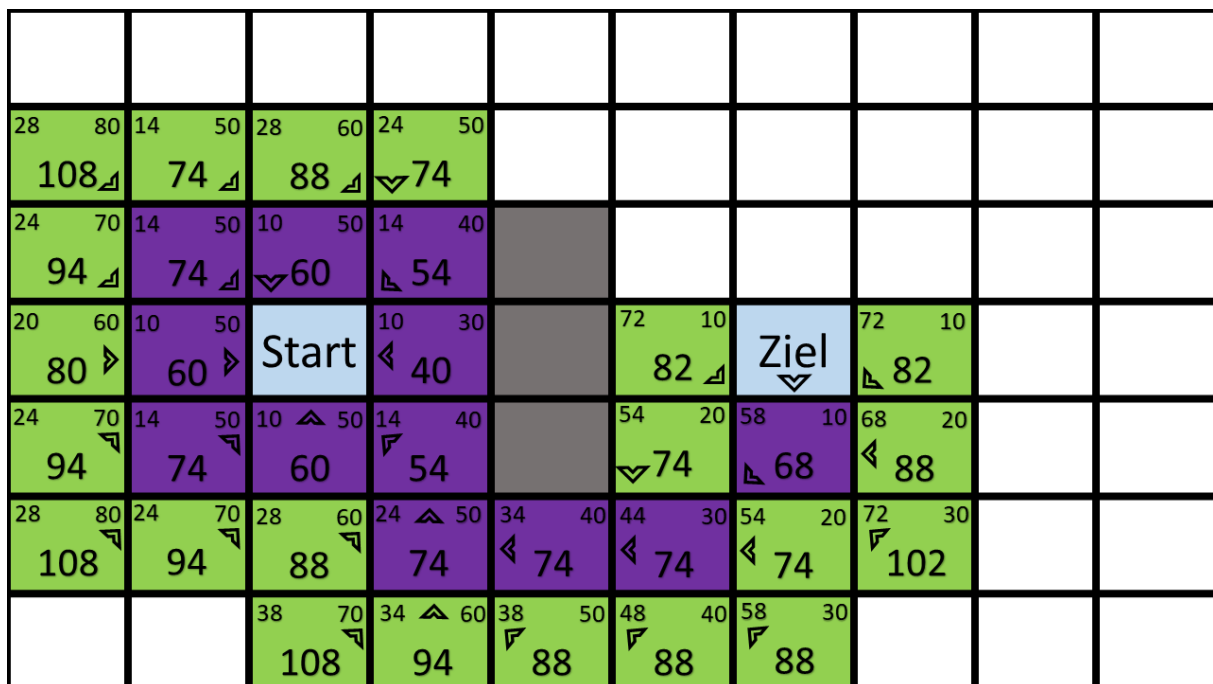


Abbildung 2.6: A* Schritt 13

2.2 Unity

Unity ist eine Laufzeit- und Entwicklungsumgebung von *Unity Technologies*. Sie eignet sich besonders gut zur Entwicklung von Spielen für unterschiedliche Endgeräte. Zielplattformen für die mit Unity entwickelt werden kann sind unter anderem Desktop Anwendungen, mobile Endgeräte

und Webbrowser. So kann die GameEngine unter anderem verwendet werden um Spiele sowohl für Apples Betriebssystem iOS wie auch für Android Geräte zu builden. Unity bietet die Möglichkeit Anwendungen sowohl in 2D als auch in 3D zu entwickeln. Die Entwicklungsumgebung ist für Windows, Linux und MacOS verfügbar.

Die Entwicklungsumgebung erinnert an gängige 3D-Entwicklungsplattformen. Die Oberfläche besteht aus einem Hauptfenster in dem die Spielszene zu finden ist. In dieser Scene können 2D und 3D Objekte ausgewählt, skaliert und verschoben werden. Objekte in der Scene werden Game-Objekt genannt und in einer hierarchisch organisierten Sidebar aufgelistet. Einem GameObjekt können weitere Komponenten wie Sounds, Materials oder Skripte zugeordnet werden. Einem GameObjekt zugeordnetes Skript kann das Verhalten eines GameObjekts bestimmen. Ein Skript bestimmt die Spiellogik und wird in C# geschrieben. Der Standard Editor von Unity ist Visual Studio. Zum entwickeln kann aber auch jeder andere Editor verwendet werden. Skripte, Assets wie zum Beispiel auch 3D Modelle können als sogenannte *Prefabs* zusammengefasst werden. Einem Prefab kann bestimmtes Verhalten zugeordnet werden. Dies ist von Vorteil, wenn das gleiche Objekt mehrmals in einer Scene verwendet werden soll.

Quelle:
???

2.3 Vuforia

Bei Vuforia handelt es sich um ein Augmented Reality SDK [h] für mobile Endgeräte. Mit diesem SDK lassen sich Augmented Reality Anwendungen für iOS und Android entwickeln. Mit Vuforia lassen sich reale Bilder (ImageTargets)[h] und einfache 3D Objekte (3D Scans) in Echtzeit erkennen und verfolgen. Vuforia bietet dem Entwickler mehrere APIs [h]. Darunter APIs in C++, Java, Objective C++ und als Erweiterung für Unity auch in C#. Vuforia kann in Unity als Erweiterung die Möglichkeit der Scene eine AR-Camera hinzuzufügen und ermöglicht so eine Augmented Reality Umgebung in der Anwendung. Vuforia erweitert die Auswahl von GameObjects in Unity. Der Scene können durch Vuforia, GameObjects wie ImageTargets, ModelTargets, 3D Scans oder VuMarks hinzugefügt werden.

evtl. einmal aus-schreiben

immer zusammen schreiben!

einmal aus-schreiben!!!

Quelle: <https://www.vuforia.com>

Image

2.3.1 ImageTargets

Ein ImageTarget ist ein reales Objekt, welches von der Vuforia AR-Camera erkannt werden kann. Wird ein ImageTarget in der realen Welt vor der Kamera des mobilen Endgeräts platziert, erkennt das Vuforia SDK durch die Bilderkennung das ImageTarget. Auf dem erkannten ImageTarget können so virtuelle Objekte wie zum Beispiel 3D-Modelle platziert werden. Die Position und Orientierung des 3D-Modells richten sich hierbei nach dem erkannten ImageTarget. Ein Bild muss benötigt hierbei keine speziellen schwarz-weiß regionen wie zum Beispiel ein QR-Code. Das Bild wird mit einem, in einer Vuforia Datenbank hinterlegten Bildes, abgeglichen und so anhand seiner natürlichen Merkmale erkannt. Befindet sich das Bild im Blickwinkel der Kamera des Endgerätes, wird dieses erkannt und so lange getrackt, bis es den Blickwinkel der Kamera verlässt. Bilder können im JPG oder PNG Format in RGB oder Graustufen in die Vuforia Datenbank geladen werden. Ein Bild darf hierbei nicht größer als 2MB sein. Beim upload in die Datenbank muss die Breite des Bilds in Meter angegeben werden. Die Datenbank mit den Referenzbildern

kann für unterschiedliche Plattformen heruntergeladen werden. Darunter unter anderem Android Studio, xCode, Visual Studio oder Unity. Das heruntergeladene Package kann dann in der jeweiligen Plattform importiert werden.

2.3.2 Object Recognition

Die Vuforia Object Recognition funktioniert ähnlich wie die Erkennung von ImageTargets. Ein reales Objekt wie zum Beispiel eine Streichholzschachtel kann anhand von bestimmten Merkmalen und eines Referenzobjektes in einer Vuforia Datenbank zur Laufzeit erkannt und verfolgt werden. Damit die Objekterkennung durch Vuforia gut funktioniert, sollte das Objekt undurchsichtig und unbeweglich sein. Die Oberfläche des Objekts sollte viele kontrastbasierte, eindeutige Merkmale aufweisen, die als Referenzpunkte bei der Erkennung dienen können.

Referenzobjekte, die in die Vuforia Datenbank geladen werden können, müssen mit dem Vuforia Object Scanner erstellt werden. Der Scanner ist eine Applikation für Android geräte, die als APK von Vuforia zur Verfügung gestellt wird. Das zu scannende Objekt wird hierbei auf einer definierten Unterlage platziert und eingescannt. Ergebnis des Scanvorgangs ist eine Object Data Datei (.od), die in die Vuforia Datenbank geladen werden kann. Wie bei den Image Targets beschrieben, kann auch diese Datenbank heruntergeladen und in die Entwicklungsumgebung importiert werden.

Quelle:
<https://library.vuforia.com/developing/creating-object-scanner>

Quelle:
<https://library.vuforia.com/developing/creating-object-scanner>

Kapitel 3

Recherche

3.1 A* Algorithmus

Das Thema Wegfindung spielt in diesem Projekt eine wichtige Rolle. Der Computergegner soll einen Wegfindungs-Algorithmus benutzen, um den kürzesten Weg zum Ziel finden. Dabei soll die Wegfindung schnell und zuverlässig sein, auch wenn dynamisch ein oder mehrere Hindernisse auftauchen. In Kapitel [2.1](#) wird die Funktionsweise des Algorithmus erläutert. An dieser Stelle soll deutlich werden warum sich in diesem Projekt für den A* Algorithmus entschieden wurde.

Zu den bekanntesten Algorithmen der Wegfindung gehören Dijkstra und A*. In Tabelle [3.1](#) werden die Vor- und Nachteile dieser Algorithmen gegenüber gestellt. Die Entscheidung für den A* Algorithmus ist aufgrund der Optimierung für die Suche mit nur einem Ziel. Das erhöht die Performanz, was bei einem Spiel zu einem besseren Spielfluss führen kann.

Algorithmus	Vorteile	Nachteile
Dijkstra	<ul style="list-style-type: none"> • Das Ziel muss nicht bekannt sein. • Gut geeignet bei mehreren Zielen 	<ul style="list-style-type: none"> • Es wird der kürzeste Weg gefunden, bis dahin werden aber auch unnötige Pfade gegangen. • Liefert fehlerhafte Ergebnisse bei negativen Kanten
A*	<ul style="list-style-type: none"> • Verwendung einer Heuristik, um zielgerichtet zu suchen. • Höhere Performanz durch zielgerichtete Suche. • Er ist komplett, da er immer einen Weg findet, solange einer existiert • Kann in einen anderen Path-Finding Algorithmus umgewandelt werden, indem man die Heuristik wechselt 	<ul style="list-style-type: none"> • Nicht hilfreich wenn es mehrere Ziele gibt, da man nicht sagen kann welches das naheliegendste Ziel ist.

Tabelle 3.1: Vergleich A* und Dijkstra

3.2 Imagetargets

Jannis

3.3 Objekterkennung

Da die Objekterkennung einen großen Anteil des RunnAr Projekts ausmacht, wurde hierzu viel Recherche betrieben. Da die Objekte vom Benutzer in Echtzeit platziert und manipuliert werden können, wurde besonderes Augenmerk auf die Performanz der Implementierungsmöglichkeiten gerichtet.

3.3.1 Tensorflow

Tensorflow ist ein von Google entwickeltes Framework. Es wird oftmals in Programmen für maschinelles Lernen genutzt. Implementiert ist es in Python und C++. Da Unity von Haus aus

mit C Sharp Skripten arbeitet und keine alternative Sprache genutzt werden kann, muss für Unity das TensorFlowSharp Plugin genutzt werden. Dies stellt Wrapper zur Verfügung die den nativen C++ Code in C Sharp Code umwandeln. Auf Github finden sich einige Beispielprojekte, die den Fokus auf Objekterkennung setzen. Dies ermöglicht einen schnellen Einblick auf die Funktionalitäten von Tensorflow in Kombination mit Unity. Tensorflow ermöglicht es zwar durch trainierte Agenten eine große Variante an Objekten zu erkennen und zu klassifizieren, jedoch braucht dies seine Zeit. Die Kamera muss sich recht nah am Objekt befinden und Tensorflow verliert immer wieder den Fokus auf das Objekt. Zusätzlich ist die bounding Box, die sich um das Objekt dargestellt wird, recht grob. Bei diesem Ansatz fehlte für das Projekt also die Performanz und die Genauigkeit. Zusätzlich passte die Nähe die die Kamera zu dem Objekt haben musste nicht zu unserem Spielkonzept.

3.3.2 OpenCv

OpenCV ist eine freie Programmbibliothek zur Bildverarbeitung. Der Ansatz bei OpenCv wäre, dass Umrisse von auf dem Tisch platzierten Gegenständen erkannt werden sollten. Dies müsste jedoch auf einem hellen Untergrund geschehen, an dem sich die Objekte deutlich abheben. Der Vorteil gegenüber Tensorflow wäre hierbei, dass man keine trainierte Agenten bräuchte, die die Objekte zusätzlich klassifizieren würden. OpenCV hat jedoch den Nachteil, dass der Einsatz mit Unity kostenpflichtig ist. Die Kombination mit Unity ohne Kosten gestaltet sich als schwierig. Da Unity nur mit C Sharp Skripten arbeiten kann, müssten Wrapper erstellt werden, die die Kommunikation zu Unity ermöglichen würden. Fragwürdig wäre jedoch wie performant die Erkennung der Objekte mit OpenCv gewesen wäre. Viel Zeit und Aufwand hätten in die Entwicklung eines C++ Projekts mit OpenCv fließen müssen, bei der am Ende immer noch die Frage im Raum gewesen wäre, ob dies überhaupt flüssig in Unity laufen würde. Diesen Ansatz haben wir nach mehreren Tagen erfolgloser Recherche und Implementierungsversuchen für ineffizient eingestuft.

3.3.3 Vuforia Object Recognition

Vuforia verfügt nativ über Object Recognition. Hierbei müssen die Objekte, welche erkannt werden sollen, vorher durch eine kostenlose App für Android Geräte eingescannt werden. Hierfür kann man sich auf der Webseite von Vuforia eine Schablone ausdrucken, auf der man die zu scannenden Objekte platziert. Danach nutzt man die App und scannt die Objekte mit der Kamera auf dem mobilen Endgerät ein. Es wird virtuell ein Gitter um das platzierte Objekt dargestellt und die Bereiche die fertig gescannt wurden, werden grün markiert. Dies wiederholt man solange bis das Gitter komplett grün gefärbt ist. Beim einscannen sollte man keine Objekte nehmen die zu klein sind, da diese recht lange brauchen um eingescannt zu werden. Zusätzlich ist die Erkennung kleiner Objekte nicht sehr effektiv, da Vuforia bei kleinen Objekte nicht ausreichend Vergleichspunkte hat um die Gegenstände zu erkennen. Die Performanz der eingescannten Objekte ist sehr gut, sodass die Einschränkung, dass nur eingescannte Objekte erkannt werden können, in Kauf genommen wurde. Auch die Erkennung über weite Entfernungen ist gegeben. Nach dem Einscannen, können diese in eine Datenbank importiert und problemlos in Unity eingebunden werden. Mehr zu diesem Thema im Kapitel Umsetzung.

Kapitel 4

Konzept

Nachdem sich die Idee des Projektes verfestigt hat, muss genau ausdefiniert werden, welche Funktionen die Anwendung am Tag der Abgabe erfuellen soll. Nicht alle Visionen lassen sich in dem vorgegebenen Zeitraum umsetzen, sodass eine Liste aus Funktionsanforderungen gefertigt wird. Unumgaenglich war jedoch die Wahl fuer einen Algorithmus zur Merkmalserkennung, der das Herz der Anwendung darstellt. Somit ist bei dessen Wahl besondere Vorsicht geboten, weswegen hierfuer eine Gegenueberstellung von moeglichen Algorithmen ausgearbeitet wurde. Als Bibliothek wurde sich fuer OpenCV entschieden, welche mit der Sprache C++ genutzt wird. Der Grund fuer diese Wahl sind die deutlich weitreichenderen Funde bei Recherchen, sie uebersteigen die von Java oder Python deutlich.

Jannis
überar-
beiten

4.1 Definition von Funktionsanforderungen

Bei der Abgabe des Projektes sollen folgende Anforderungen Teil des Funktionsumfang sein.
Donnerstag zusammenmachen

Christian
- evtl als
Tabelle

4.2 Spielkonzept

Auf der Skizze aus Abbildung 1 ist ein Musterspielfeld abgebildet. Die Spielfigur (gruener Pull-over), welche sich zu Beginn des Spiels am unteren Rand des Spielfelds befindet, muss den Hindernissen ausweichen. Hindernisse koennen zum Einen aus Image Targets bestehen, auf die in der Spielumgebung 3D Objekte gemappt werden, oder zum Anderen aus realen Objekten, die Im Spielfluss erkannt werden und so ebenfalls ein 3-dimensionales Hindernis fuer den Spieler darstellen. Der Start und das Ziel sind ebenfalls durch Muster gekennzeichnet.

Jannis

4.2.1 Entwicklung des Spielkonzepts

4.2.2 Finales Spielkonzept

Kapitel 5

Umsetzung

Jannis

5.1 Implementierung des A* Algorithmus

Allgemein

Die Implementierung des A* Algorithmus erfolgt in einem separaten Projekt, um mögliche Abhängigkeiten zu minimieren. Dieses Projekt ist in Abbildung 5.1 zu sehen. In der Grafik ist das Spielfeld, also das Grid mit seinen Nodes zu sehen, dort als graue oder rote Kacheln unter den Objekten dargestellt. Ein Node ist dementsprechend entweder als Pfad (grau) oder ein Hindernis (rot) gekennzeichnet.

Für das Bestimmen der Hindernisse auf dem Spielfeld wird die Funktionalität der Layers von Unity verwendet. Mit ihr lassen sich Spielobjekte logisch voneinander treffen. So sind die roten Quader auf dem selbst definierten Layer *unwalkable*.

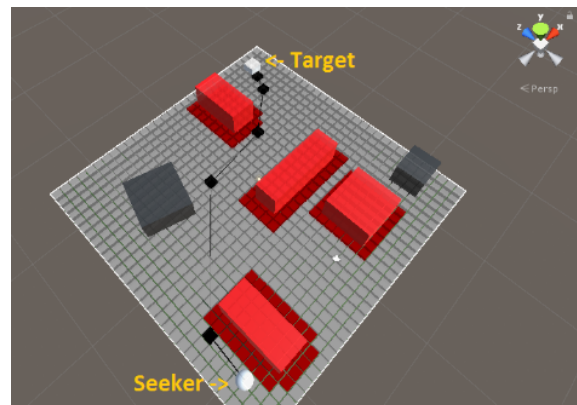


Abbildung 5.1: A* Implementation in Unity



Abbildung 5.2: StarGrid Settings

Wird das Spielfeld durch die *StarGrid* Klasse generiert, kann gezielt nach den Hindernissen mit diesem Layer gesucht werden. Dann bekommen die Nodes, die entsprechende Eigenschaft ob sie begehbar oder nicht begehbar sind.

Damit dieses Skript im Kontext von Unity eingebunden werden kann, wird ein leeres *GameObject* erstellt, was alle zum Algorithmus gehörenden Skripte zugewiesen bekommt. Dort lassen sich dann zum Beispiel Parameter wie die Größe des Grids, den Radius einen einzelnen Nodes

oder ein Wert für einen Puffer setzen (siehe Abbildung 5.2). Der Puffer ist dann auf die Größe der Hindernisse drauf gerechnet, damit die Spielfigur einen größeren Abstand bei Verbeilaufen hat.

Sobald der A* den kürzesten Pfad berechnet hat bekommt er eine Liste mit Node-Position, welche dann verwendet werden, um die Spielfigur von Node zu Node laufen zu lassen aber auch den Pfad zu visualisieren.

Binary-Min-Heap

Im Grundlagenkapitel [A* Algorithmus \(2.1\)](#) wird erläutert, dass der A* Algorithmus eine *Open List* verwendet, die benötigt wird, um alle Nodes zu speichern, die als möglicher Wegpunkt in Frage kommen. Da dessen Nodes häufig sortiert und aktualisiert werden muss, wird dafür ein Binary-Min-Heap als Priority Queue verwendet. Diese Datenstruktur kann effizient mit einem linearem Zeitaufwand von $O(n)$ erzeugt werden. Das Hinzufügen oder Entfernen von Elementen hat im schlimmsten Fall eine Laufzeit von $O(\log n)$. Der Vorteil bei dieser Datenstruktur gegenüber einem herkömmlichen Arrays ist das *in-place* Sortieren und wenn intern mit einem Array gearbeitet wird, ist der Speicherverbrauch gering und die Zugriffe sind schnell. Die Gesamtkomplexität für das Sortieren wäre im *Worst-Case* $O(n * \log n)$.

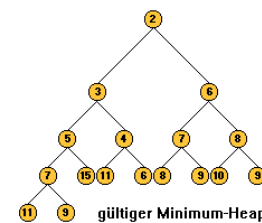


Abbildung 5.3: Binary-Min-Heap Beispiel

PathRequestManager

Der *PathRequestManger* ist eine Klasse, die implementiert wurde, um mehrere Wegberechnung nacheinander asynchronen abarbeiten zu lassen. Dabei verwaltet er alle Pfadanfragen und lässt alle nacheinander berechnen. Das sorgt dafür, dass der Spielfluss nicht durch das Blockieren des Hauptthreads unterbrochen wird.

Turret

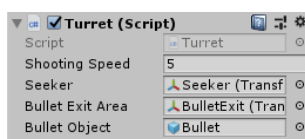


Abbildung 5.4: PathRequest-Manager Settings

Zum Einen sind das die Schussgeschwindigkeit, der Austrittsbereich, aus dem das Projektil geschossen wird und das eigentliche Projektil. Zum Anderen ist dort das Feld *Seeker*, welche eine Referenz zur Spielfigur ist, also das Objekt auf das geschossen werden soll. Diese Komponente hat es noch nicht in das aktuelle Spiel geschafft, was aber mit dieser Klasse einfach nachzuholen ist.

Für das im Kapitel ?? (??) erwähnte Feature einer schießenden Einheit ist das *Turret* Skript geschrieben worden. Dieses Skript wird einem *GameObject* zugewiesen, welches dann auch im Spiel Projektil verschießen soll. Als Projektil kann ein beliebiges *GameObject* verwendet werden. Sollte keines ausgewählt sein, wird ein Standard Projektil aus den Prefabs geladen. Wie in Abbildung 5.4 zu sehen, gibt es vier Parameter, welche angepasst werden können.

5.2 ImageTargets

Design der Images und Erkennung der Images

5.3 Erkennung von Objekten

Wie in Kapitel 3.3.3 beschrieben müssen die eingescannten Objekte in eine Vuforia Datenbank importiert werden (siehe Abb. 5.5). Diese Datenbank wird im Vuforia Developer Portal zur Verfügung gestellt und ist mit wenigen Klicks eingerichtet. Diese Datenbank lässt sich für unterschiedliche Entwicklungsplattformen herunterladen. Unter anderem auch für die Unity IDE als package. Um die Objekte nun in der Unity Scene verwenden zu können, muss das Package per Import Package Funktion in Unity geladen werden.



<input type="checkbox"/> Target Name	Type	Rating	Status ▼	Date Modified
<input type="checkbox"/>  cigarettes	Object	n/a	Active	Jan 10, 2019 09:49
<input type="checkbox"/>  matches	Object	n/a	Active	Jan 09, 2019 19:51

Abbildung 5.5: Vuforia Datenbank

Vuforia bietet die Möglichkeit in Unity 3d Scan Objekte als Gameobject anzulegen. Bei dem angelegten Objekt kann nun die importierte Datenbank und das wiederzuerkennende Objekt ausgewählt werden.

In der Szene erscheint nun eine Bounding Box. Laut der Vuforia Dokumentation [1] sollte die Bounding Box den Maßen des eingescannten Objektes entsprechen. Dies ist bei der Durchführung jedoch nicht der Fall gewesen. Die Objekte haben einen Nullpunkt. (siehe Abb. 5.6) Der Nullpunkt wird von der Schablone welche zum Einscannen genutzt wird, übernommen. Die Bounding Box bietet sich gut an, um Objekte auf dem eingescannten Objekt zu platzieren. Jedoch ist die Länge vom Nullpunkt bis zum Ende des Objektes nicht bekannt. Auch rechteckige Objekte werden durch eine quadratische Bounding Box dargestellt. Durch diesen Umstand kann keine präzise Bounding Box generiert werden, die das reale Objekt in der Scene widerspiegelt.



Abbildung 5.6: Eingescanntes Objekt mit Nullpunkt

Dem in der Scene befindlichen 3d Scan Game Object kann nun der *unwalkable Layer* zugewiesen werden siehe [5.1 Implementierung des A* Algorithmus](#). Im Test reagierte der A* Algorithmus nicht auf die realen Objekte, da diese sich nicht auf der selben Ebene befinden wie das Spielfeld. Durch diesen Umstand haben wir den *DefaultTrackableEventHandler* von Vuforia erweitert. Hierfür wurde ein eigenes Script implementiert, welches die eingescannten Objecte bei jedem Update auf die Position 0 der y-Achse transformiert. So wird sichergestellt, dass sich das reale Objekt und das Spielfeld auf der selben Ebene befinden. Das reale Objekt wurde daraufhin von den Nodes des A* Grids als *unwalkable* erkannt.

Um die Objekte in die Scene des Spielers zu integrieren muss sich die Kamera exakt über dem Spielfeld befinden. Eine Third Person Ansicht ist nicht möglich, da die erkannten Objekte von der Position der Kamera ausgehen. Steht die Kamera Beispielsweise auf xyz(0,-20, -20), erhält das gescannte Objekt ebenfalls diese Koordinaten. Wird das Objekt in der realen Welt bewegt, verändern sich die Koordinaten, allerdings ausgehend von der Kamera Position. Durch diesen Umstand muss die Kamera exakt in der Mitte des Spielfelds befinden, damit die realen Objekte ausgehend von diesem Punkt platziert werden können.

5.4 UI

Kapitel 6

Zusammenfassung und Ausblick

6.1 Zusammenfassung

Christian

6.2 Ausblick

Christian

6.3 Fazit

Christian

Kapitel 7

Anhang

7.1 A* Beispiel

Literaturverzeichnis

- [1] Vuforia, “How to use Object Recognition in Unity,” 2019. [Online]. Available: <https://library.vuforia.com/articles/Solution/How-To-Use-Object-Recognition-in-Unity.html>