Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Jörg Lischka

Wegfindung in einer 3D-Welt unter Beachtung von Springen und Fallenlassen

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Jörg Lischka

Wegfindung in einer 3D-Welt unter Beachtung von Springen und Fallenlassen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Michael Neitzke
Zweitgutachter: Prof. Dr. Thomas Thiel-Clemen

Eingereicht am: 27. Februar 2014

Jörg Lischka

Thema der Arbeit
Wegfindung in einer 3D-Welt unter Beachtung von Springen und Fallenlassen

Stichworte
Wegfindung, Spiele, 3D, Potentialfelder, Korridore, Navigation Mesh, Springen, Fallen

Kurzzusammenfassung
Diese Arbeit befasst sich allgemein mit der Wegfindung für 3 dimensionale Computerspiele. Betrachtet werden die Bereiche der Terrainanalyse, der Grapherstellung, die Suche auf dem Graphen und der Ablauf des Weges. Abschließend wird eine konkrete Umsetzung vorgestellt, welche auf einem Navigation Mesh basiert und zur Suche ein eigens erstelltes und hierarchisches Verfahren verwendet.
Als Zielsetzung dieser Arbeit soll die KI über das Wegfindungssystem etwas menschlicher wirken, indem eine gute Annäherung bezüglich des Bewegungsraumes und des Wegablaufs des menschlichen Spielers und des Computerspielers erreicht wird.

Jörg Lischka

Title of the paper
Pathfinding in a 3D-World allowing for jumping and falling

Keywords
Pathfinding, Games, 3D, Potential Fields, Corridors, Navigation Mesh, Jumping, Falling

Abstract
This thesis generally deals with the pathfinding for 3 dimensional computer games. Considering the areas of terrain analysis, graph creation, the search on the graph and the tread of the path. Finally, a concret implementation is presented, which is based on a navigation mesh and used an own created hierarchical search method.
The goal of this thesis is to let the AI to be a little more human like, by using a good approximation with respect to the movement space and the thread of the path of the human player and the computer player.

Inhaltsverzeichnis

Kapitel 1

Einführung

In dieser Arbeit wird erläutert, wie die Wegfindung in 3D-Spielen von Statten geht. Dabei wird der komplette Werdegang beschrieben. Beginnend bei der Terrainanalyse über die Wegsuche bis hin zur Wegbeschreitung. Hierbei werden vorhandene Methodiken vorgestellt. Des weiteren wird an einer Wegfindung geklügelt, welche dem NPC eine ähnliche Bewegungsfreiheit wie dem menschlichen Spieler bieten soll. Hierzu gehört selbstverständlich auch das Springen und Fallenlassen.

1.1 Motivation

Im Folgendem werden ein paar Probleme aus bekannten 3D Spielen vorgestellt.

In WoW[1] wird die Wegberechnung serverseitig durchgeführt. Das Ziel der Wegsuche ist die Position an der sich der Spieler in ca. einer halben Sekunde befinden wird. Bewegt man sich im schnellen Zickzack hin und her, dann hinkt der NPC[2] hinterher (aSp10), was eine Art Cheaten ermöglicht. Es kommt auch mal vor, dass Gegner durch Hindernisse einfach durchlaufen (Toz08b, 01:35). Des weiteren ist es in WoW möglich kleine Erhöhungen hinauf zu springen. Dieses kann der NPC nicht und nimmt statt dessen einen längeren Umweg zum Ziel (s. Abb. 1.1) (Toz08b, 00:50) (Toz08b, 02:05). Existiert kein Weg, dann werden Gegner zurückgesetzt[3].

Wenn man in Skyrim an einen Ort gelangt, zu dem der NPC keinen Weg findet, dann bleibt dieser einfach stehen, so dass man diesen in Ruhe aus der Ferne besiegen kann (s. Abb. 1.2). In Skyrim sind Begleiter nicht zuverlässig, weil diesen logische Pfade zum Spieler verborgen bleiben können (s. Abb. 1.3).

[1] World of Warcraft
[2] Computerspieler
[3] NPC kehrt zum Ausgangsort zurück und hat wieder 100% Leben

Abbildung 1.1: World of Warcraft: Der Gegner kann nicht springen (web11)

Abbildung 1.2: Skyrim: Eisbär kennt den Weg zum Spieler nicht (mup11)

In Oblivion gibt es unter anderem fliegende Gegner, welche nicht wirklich fliegen können, sondern nur ca 1 Meter über dem Boden dargestellt werden (s. Abb. 1.4). Auch scheint es dort ein Lokales Minima[4] Problem zu geben, weil Gegner gerne den direkten Weg zum Spieler gehen, obwohl ein nah gelegener Aufgang zum Ziel führt (Toz08b, 00:10).

Abbildung 1.3: Skyrim: Begleiter bleibt hängen (shi11)

Abbildung 1.4: Oblivion: Fliegen ist nur vorgetäuscht (Toz08b, 01:20)

In der Assassins Creed Reihe finden die Wachen so gut wie immer zum Spieler, obwohl diese sehr gefordert werden, weil Altair sehr akrobatisch ist. In Teil 1 der Reihe kennen die Wachen den Weg zu Altair, jedoch benötigen diese mehrere Anläufe bis zum Erfolg (s. Abb. 1.5). In Teil 2 bemerken die Wachen, dass der kürzeste Weg blockiert ist und nehmen einen größeren Umweg in Kauf. Auf dem Weg vergisst die KI, dass der kürzeste Weg blockiert ist und möchte daher diesen wieder beschreiten (Bod12). In Teil 3 sind folgende Probleme bekannt. Der Begleiter

[4] Die Wegsuche glaubt den besten Weg gefunden zu haben und führt in eine Sackgasse

dreht sich immer zu im Kreis (MrE13b, 00:11); hier sollte die Wegfindung korrekt arbeiten und somit handelt es sich hier eher um einen Bug. Manche Hindernisse sind an Orten, wo die KI es nicht vermuten würde; so geht eine Wache die ganze Zeit gegen eine Box (MrE13b, 04:52) bzw eine Tür (KS13, 02:33). Auch sind die Wachen nicht wirklich Teamfähig; jeder möchte am Rand des Stegs stehen, um Altair angreifen zu können und so kommt es vor, dass die Wachen sich gegenseitig vom Steg schubsen (KS13, 01:30). Wölfe sind zu hektisch und fallen auch mal von einem Baumstamm hinunter (MrE13a, 00:01 und 06:22).

Abbildung 1.5: Assassins Creed I: Wachen wollen zu Altair (ian09)

Abbildung 1.6: Assassins Creed III: Soldaten fallen vom Steg (KS13, 01:30)

Half Life 2 hat eine beschränkte Wegfindung, wenn es um Teamwork und dynamische Hindernisse geht.
Abschließend noch einen Blick auf die Wegfindung von Guild Wars 2. Die gefundene Lösung sollte die optimale des hinterlegten Suchgraphen sein, was jedoch nicht der optimale Weg ist, welchen ein menschlicher Spieler beschreiten wird.

Abbildung 1.7: Half Life 2: KI irrt ergebnislos umher (Cla13)

Abbildung 1.8: Guild Wars 2: Havok's Wegfindung (hav12)

Um die Mechanik der Wegsuche besser verstehen zu können sowie die Lösungsansätze der Spielehersteller nachvollziehen zu können, sollte diese Arbeit genau das Richtige sein.

1.2 Aufbau der Arbeit

In Kapitel 2 werden grundlegende Kenntnisse vermittelt und auf diese kurz eingegangen. Hierzu gehören unter anderem die Graphentheorie, Potentialfelder, die Repräsentation von 3D-Welten und eine genauere Betrachtung des A*-Algorithmus.
In Kapitel 3 werden unter Berücksichtigung der Wegfindungsfehler aus der Motivation (Abschnitt 1.1) Anforderungen gebildet. Dann werden die Komponenten vorgestellt, welche für das Wegfindungssystem dieser Arbeit benutzt werden, bevor das Kapitel durch Einschränkungen abgeschlossen wird. Detailliertere Beschreibungen und Umsetzungen folgen in den nächsten Kapiteln.
In Kapitel 4 geht es hauptsächlich um die Offline-Berechnungen. Hierfür wird der Suchgraph erstellt und mit Informationen angereichert.
In Kapitel 5 werden Methodiken beschrieben, um in angemessener Zeit ein gutes Resultat bei der Wegsuche zu erhalten.
In Kapitel 6 wird der gefundene Weg beschritten. Hier wird beschrieben, wie der NPC den Weg realistisch abläuft, wobei der gefundene Weg nicht 1:1 abgelaufen werden darf. Des weiteren wird die Corridor Maps Method kurz beschrieben, weil diese den Ablauf des Weges gut bewältigt.
In Kapitel 7 liegt der Hauptmerkmal auf der Umsetzung. Hierfür werden Verfahren der letzten Kapitel aufgegriffen und weiter verfeinert.
In Kapitel 8 wird der entstandene Prototyp mit dem eigenen Suchverfahren und dem des A*-Algorithmus ausgiebig unter mehreren Gesichtspunkten getestet.
Abschließend wird in Kapitel 9 das Resultat der Bachelor-Arbeit vorgestellt und bewertet, sowie ein paar Erweiterungsmöglichkeiten vorgestellt.

Kapitel 2

Grundlagen

Um diese Arbeit besser verstehen zu können, werden im Folgendem ein paar Grundkenntnisse vermittelt. Hierzu gehören Graphen, Potenzialfelder, der Aufbau einer 3D-Welt, Suchalgorithmen und deren Datenstrukturen sowie gängige Suchverfahren in Computerspielen.

2.1 Graphentheorie

2.1.1 Was Graphen sind

In der Graphentheorie bezeichnet ein Graph eine Menge von Knoten und Kanten. Hierbei verbindet jede Kante 2 Knoten miteinander.
Werden Knoten-A und Knoten-B miteinander verbunden, dann wird von der Kante-AB beziehungsweise Kante-BA gesprochen, wenn es sich um einen ungerichteten Graphen handelt. Somit kann geschrieben werden "Kante(A, B)" ist identisch zu "Kante(B, A)". Bei einem gerichteten Graphen wird hier unterschieden und somit ist "Kante(A, B)" ungleich "Kante(B, A)".
Graphen sind ein regelrechter Bestandteil unseres Lebens und werden unter anderem in Verkehrsnetz, Computerspielen, Rechnernetzen und Stammbäumen verwendet; also überall dort, wo Informationen verknüpft werden sollen.
Beim Stammbaum können Geschwister mit einer ungerichteten Kante verknüpft werden, wobei bei einer Vater-Kind-Beziehung ein bis zwei gerichtete Kanten verwendet werden müssen. Ein Beispiel wäre hier: "Homer ist der Vater von Bart und Bart ist ein Kind von Homer", wobei die jeweilige Beziehungsinformation den Kanten und die Personennamen den Knoten zugeordnet werden.
Bei einem U-Bahn-Netz sind die Haltestellen die Knoten und die Strecken von jeweils einer Haltestelle zur nächsten Haltestelle die Kanten, welchen die Fahrzeit als Information zugeordnet werden kann.
Beim Straßennetz sind die Kreuzungen, Abzweigungen und Straßenenden die Knoten und

die Straßen zwischen den jeweiligen Knoten sind die Kanten. Um detailliert genug zu sein, müssen weitere Knoten hinzukommen, weil es vorkommen kann, dass sich bei langen Straßen "plötzlich" der Name ändert. Auch bei Brücken, welche nur eine bestimmte Last tragen können oder bei Unterführungen, welche nur mit einer beschränkten Fahrzeughöhe zu passieren sind, müssen beim Betreten und beim Verlassen dieser zusätzliche Knoten hinzugefügt werden. Die Straßen, also die Kanten, benötigen im Straßennetz viele Informationen wie zum Beispiel die maximal erlaubte Geschwindigkeit, den Straßennamen, die Art der Straße, wie Autobahn, Landstraße oder Verkehrsberuhigter Bereich. Hier ist ein gerichteter Graph zu wählen, weil an manchen Straßen nur rechts abgebogen werden darf und es Einbahnstraßen gibt.
Das Internet ist ebenfalls ein gigantisch großer Graph. Ein Teilgraph dessen ist das lokale Heim-Netzwerk, in welchem sich Geräte wie Laptop, Drucker und Smartphone befinden. Diese haben den W-Lan-Router als zentralen Einstiegspunkt zum Internet. Innerhalb des Netzwerkes kann den Verbindungen zum Router, also den Kanten, die maximal mögliche Übertragungsgeschwindigkeit zugewiesen werden und den Geräten, also den Knoten, der Name sowie die freigegebenen Ressourcen.
In Computerspielen verwenden die Computerspieler oft Graphen zur Wegfindung, um dessen Einheiten an andere Orte zu führen. Hierbei gibt es mehrere gängige Methodiken, um die Knoten zu positionieren. Die Kanten enthalten als Information die Entfernung zwischen den Knoten, sowie je nach Spiel auch Informationen bezüglich der Art des Weges wie Wasser, Land oder Sumpf sowie strategische Informationen, um gefährliche Orte zu meiden beziehungsweise geeignete Orte für einen Hinterhalt zu erkennen.

2.1.2 Die Suche auf Graphen

Die Wegsuche findet auf Graphen statt und benötigt einen Startknoten sowie einen Zielknoten. Die Suche ist erfolgreich, wenn die Knoten miteinander verbunden sind und somit ein Weg vom Start- bis zum Zielknoten über Kanten und Knoten gegeben ist. Oft existieren mehrere Wege zum Ziel, wobei jeder Weg Kosten verursacht. Wird der kostengünstigste Weg gefunden, dann wird vom "Optimalen Weg" gesprochen. Kosten können unter anderem die Entfernung, die benötigte Zeit oder die Anstrengungen sein.
Handelt es sich um einen gerichteten Graphen, dann ist der optimale Weg von Knoten-A nach Knoten-B nicht unbedingt der optimale Pfad von Knoten-B nach Knoten-A. Betrachtet man das Straßennetz, dann ist diese Aussage plausibel, weil Einbahnstraßen nur in eine Richtung befahren werden dürfen.

Es existieren bereits mehrere Algorithmen zur Wegberechnung, welche sich an den Merkmalen Speicherverbrauch, Performanz und Qualität der Lösung unterscheiden. Diese drei Merkmale sind widersprüchlich, so dass man davon ausgehen kann, dass es keinen optimalen Algorithmus gibt. Je nach Problemfall ist mal der eine oder andere Algorithmus besser geeignet. Je Größer der Graph wird, desto wichtiger ist es performantere Algorithmen einzusetzen. Somit ist der Algorithmus von Dijkstra nicht geeignet, weil dieser eine Breitensuche verwendet und somit viel zu viele Knoten besucht, welche eigentlich von vornherein aufgrund der Lage ausgeschlossen werden konnten. Um diesen Performanzeinbußen entgegen zu wirken kann der A*-Algorithmus verwendet werden. Dieser beachtet die Distanz zum Zielknoten und somit wird ein Weg in dessen Richtung gesucht. Viele Knoten in entgegengesetzter Richtung müssen somit nicht mehr überprüft werden, was Rechenzeit einsparrt. Dieses ist in Spielen zwingend erforderlich, weil die Computerspieler in Echtzeit viele Einheiten auf der Landkarte bewegen müssen.
Weitere Informationen für Computerspiele relevanter Algorithmen für die Wegsuche werden im Verlaufe dieser Arbeit behandelt.

2.2 Potentialfelder

Potentialfelder repräsentieren negative oder positive Energiefelder unterschiedlicher Stärke und werden unter anderem bei automatischen Rasenmähern eingesetzt, um Hindernisse oder das Ende des Rasens zu symbolisieren. Die Potentialfelder werden von einem auf dem Rasen liegendem Kabel geliefert, dessen Spannung der Rasenmäher wahrnimmt.
Eine Wegfindung in Spielen anhand von Potenzialfeldern ist bedingt möglich. Das Ziel sendet eine starke positive Energie aus und ist damit sehr attraktiv für den Spieler, welcher sich somit auf diese zubewegt. Hindernisse und Gefahrenquellen hingegen senden eine negative Energie aus. Je näher der Spieler dieser Energie kommt, desto stärker wird dieser davon abgestoßen und somit dann vermieden werden, dass der Spieler in Gräben fällt oder gegen Hauswände läuft. Vorteil von Potenzialfeldern ist es, dass eine Vorabberechnung des Weges nicht stattfindet und somit Rechenleistung sowie Speicherplatz eingespart wird. Also eigentlich optimal, wenn es da nicht das lokale Minimum Problem geben würde. Unter lokales Minimum versteht man, dass der Spieler in eine Sackgasse geleitet wurde und diese aufgrund der starken Anziehungskraft des Zieles nicht verlassen kann. Des weiteren kann nicht davon ausgegangen werden, dass der zurückgelegte Weg des Spielers optimal ist.
Aufgrund der Performanz werden Potentialfelder trotz dessen in Computerspielen verwendet,

was dazu führt, dass Computerspieler stehen bleiben oder verzweifelt hin und her laufen. Dieses Verhalten ist gewiss für weniger intelligente Wesen geeignet, jedoch sollte dieses Verhalten nicht das Spiel bestimmen.

2.3 Wegsuche in Computerspielen

In Computerspielen verwendet die KI die Wegsuche meistens zur Einheitenverschiebung. Je nach Anzahl der Einheiten werden ziemlich viele Berechnungen getätigt, was viel Rechenleistung kostet. Als Beispiel dient die Wegsuche von Age of Empires II, welche auf damals üblicher Hardware 60 bis 70 Prozent der gesamten CPU-Leistung benötigt hat (Pot00, Seite 3). Der Stand der heutigen Hardware ist definitiv besser, jedoch sind auch die Anforderungen der Wegsuche gestiegen. Wo früher noch zweidimensionale rechteckige Landschaften begangen wurden, sind heutzutage dreidimensionale mehreckige Landschaften mit Höhlen, Tunneln, Brücken und Überhängen die Regel. Somit scheinen auch die Wegberechnungen um einiges komplexer zu sein, was zum Glück nur bedingt der Fall ist.
Die verwendeten Graphen enthalten in 2D-Welten meistens ungerichtete Kanten. Wird die Landschaft zu einer 3D-Welt erweitert, so dass es nun möglich ist auf Gegenstände hochzuspringen, sich von Klippen zu stürzen oder über Schluchten zu springen, dann muss der Graph modifiziert werden. Zum einen reicht ein ungerichteter Graph nicht mehr aus und es müssen die ungerichteten Kanten durch jeweils zwei gerichtete Kanten ausgetauscht werden und zum anderen werden neue Knoten und Kanten hinzugefügt. Die neuen Knoten können sich auf Klippenvorsprünge befinden, auf welche man sich nun runter fallen lassen kann. Die Verbindung zu diesen ist durch das Fallenlassen von weiter oben möglich und wird im Graphen als eine gerichtete Kante repräsentiert.
Das Hauptproblem von 2D zu 3D ist eher die Positionierung der Knoten im 3D-Raum riesiger 3D-Welten und die Verbindung dieser durch Kanten. In 2D-Grundrissen ist dieses noch klar ersichtlich und nachvollziehbar, wobei im 3D-Raum eine gute 3-dimensionale Vorstellung von Nöten ist. Die Positionierung der Knoten und Kanten kann von einer erfahrenen Person übernommen werden oder automatisiert von Statten gehen.
Zur Wegfindung gehörig ist auch der Ablauf des gefundenen Weges. Nachdem ein Weg berechnet wurde, wird dieser nicht eins zu eins von Knoten zu Knoten abgelaufen, weil die Knoten oft nicht in einer Linie liegen und ein reines Zickzack die Folge wäre, wie sich kein normaler Mensch oder Tier verhalten würde. Somit werden eckige Stellen soweit geglättet, wie es die Landschaft zulässt und den Ablauf natürlicher wirken lässt. In manchen Spielen ist es nicht

möglich durch andere Spieler hindurchzulaufen und somit muss auch die KI beim Ablauf des Weges ein paar Ausweichmanöver vornehmen. Hierfür ist es wichtig zu überprüfen, ob der Platz dafür ausreicht. Es kann jedoch auch vorkommen, dass eine Person einen schmalen Durchgang blockiert und nicht weggehen möchte. In diesem Fall muss eine erneute Wegfindung ausgeführt werden, um einen alternativen Weg zu ermitteln.
Wie schon unter Abschnitt 2.2 beschrieben, wird gerne auf Potentialfelder zurückgegriffen, wenn die Distanz zwischen dem Spieler und dessen Ziel eher gering ist.

2.4 Repräsentation einer 3D-Welt

Die Repräsentation einer 3D-Welt wird genutzt, um die Knoten für die Wegfindung automatisiert setzen zu können.
Aus Performanzgründen besteht eine 3D-Welt aus 3 Schichten. Die erste Schicht ist sehr detailliert wird als einzige Schicht grafisch dargestellt und dient dem Auge des Betrachters. Die zweite Schicht ist die Kollisionsschicht und dient dazu, dass Spieler nicht durch Wände laufen und durch den Boden fallen können. Die dritte Schicht ist die Navigationsschicht und dient den Computerspielern als begehbaren Bereich.

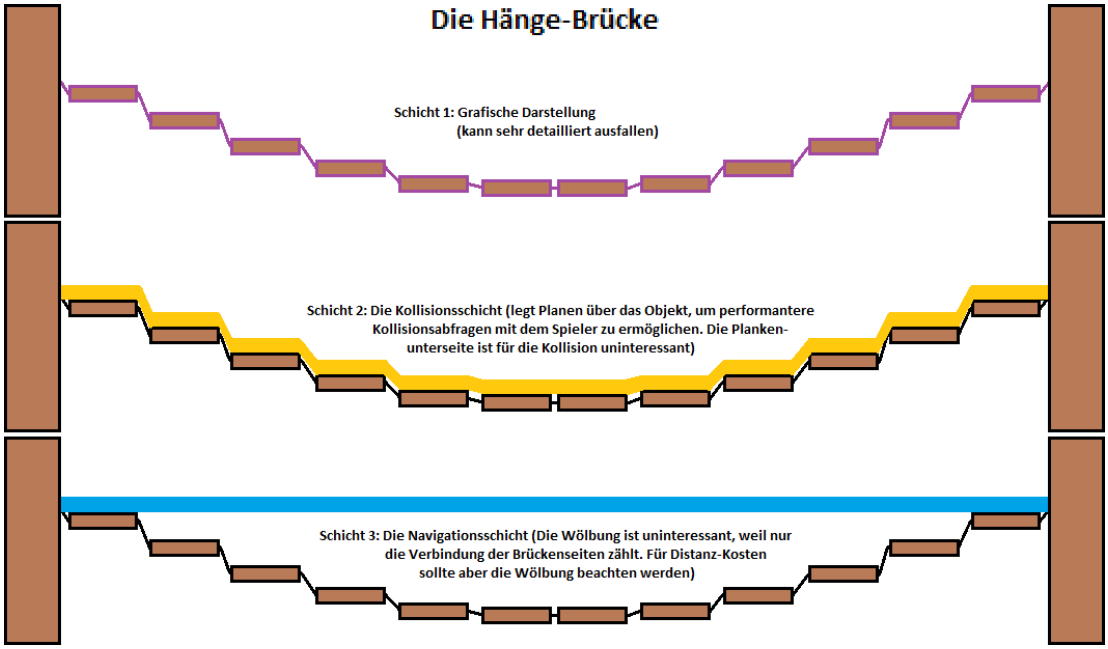

Abbildung 2.1: Die verwendeten drei Schichten einer virtuellen 3D-Welt

Alle Schichten bestehen aus vielen Punkten, wobei immer 3 Punkte eine Ebene bilden. Diese Ebene oder besser gesagt dieses Dreieck hat jeweils bis zu drei Nachbardreiecke. Ein Würfel mit sechs Seiten besteht aus zwölf Dreiecken. Bei einer Hängebrücke besteht die erste Schicht aus vielen durch Seile verbundenen Planken, wobei jede einzelne Planke ebenfalls aus zwölf Dreieicken besteht. Die zweite Schicht besteht aus einer 2D-Plane, welche auf die Hängebrücke gelegt wird und besteht je nach der Wölbung der Brücke aus wenigen oder mehreren Dreiecken. Die dritte Schicht muss die Wölbung der Hängebrücke nicht beachten und besteht somit nur aus zwei Dreiecken.
Die für die Wegsuche relevante Schicht ist die dritte Schicht und je kleiner diese gehalten wird, desto kostengünstiger ist auch die Wegfindung.

2.5 Der A*-Algorithmus

Der wohl bekannteste Suchalgorithmus ist der A*-Algorithmus, welcher bei der Wegsuche den bereits zurückgelegten und den geschätzten Restweg beachtet. Für die Schätzung wird eine Heuristik benötigt, welche die Eigenschaft hat, dass die Kosten des geschätzte Restweg nicht die tatsächlichen Kosten überschätzen darf. Als Heuristik kann der Euklidische Abstand genutzt werden, welcher als geschätzte Kosten die Luftlinie vom aktuellen Knoten zum Zielknoten verwendet.
In dieser Arbeit kommt der A-Stern-Algorithmus in Verbindung mit der Euklidischen-Abstand Heuristik noch häufiger vor und sollte somit genauestens verstanden sein. Damit dieses gewährleistet ist, wird diese Konstellation im Folgendem noch genauer untersucht.

Der A*-Algorithmus verwendet eine Breitensuche, welche durch die gewählte Heuristik zielgerichteter wird. Wird eine Heuristik verwendet, welche immer die Kosten von Null zurückgibt, dann ist diese zulässig, weil die Kosten nicht überschätzt werden. Jedoch wird dadurch der A*-Algorithmus zu einer reinen Breitensuche und entspricht somit dem Dijkstra-Algorithmus. Für die Euklidische-Abstand Heuristik wird für jeden Knoten eine geographische Koordinate vorausgesetzt, damit der Abstand der Punkte bestimmt werden kann.

Es wird eine Open- und eine Closed-List verwendet, in welchen die noch zu untersuchenden beziehungsweise die bereits untersuchten Knoten befinden. Der Ablauf von A* ist am Ende des Abschnittes als eigener Pseudocode dargestellt. Die dortige Methode "DecreaseKeyInOpenList" entfernt zunächst den Knoten aus der Open-List und fügt diesen danach mit dem neuen Wert

wieder hinzu.
Die Closed-List kann in den Programmiersprachen als HashSet umgesetzt werden, welche einen schnellen Zugriff von O(1) bietet und somit eine schnelle Verifikation gegeben ist, ob ein Knoten Teil dieser Menge ist. Die Open-List kann als SortedMap umgesetzt werden, wobei der Schlüssel die Kosten und der Wert die entsprechenden Knoten sind. Hierbei sollte beachtet werden, dass mehrere Knoten identische Kosten haben können. Eine mögliche Implementierung in Java wäre zum Beispiel "TreeMap<Float, Set<Node»", wobei die Kosten als Gleitkommazahl abgespeichert werden. Die benötigten Operationen auf der TreeMap werden in O(log(n)) ausgeführt.
Der A*-Agorithmus sollte somit durch das Big-O Verhalten der Open- und Closed-List nicht ausgebremst werden und hat als einzigen Performanz-Nachteil die Anzahl der expandierten Knoten, welche durch die Heuristik beschränkt wird. Die Euklidische-Abstand Heuristik versagt aber, wenn es sich im 2D-Raum um ein Labyrinth oder im 3D-Raum um Flüsse, Tunnel, übereinander liegenden Ebenen oder ähnlichem handelt. Befindet sich hier der Spieler im Erdgeschoss eines südlich liegenden Zimmers und möchte in den ersten Stock in ein ebenfalls südlich liegendes Zimmer und die Treppe befindet sich im nördlichen Teil des Hauses, dann ist davon auszugehen alle Zimmer im Erdgeschoss besucht werden, bevor im ersten Stockwerk das Ziel schnell gefunden wird. Als Beispiel dient ein Labyrint, in welchem die besuchten Knoten hervorgehoben worden sind (s. Abb. 2.2). Betrachtet wird nun der Speicherverbrauch anhand

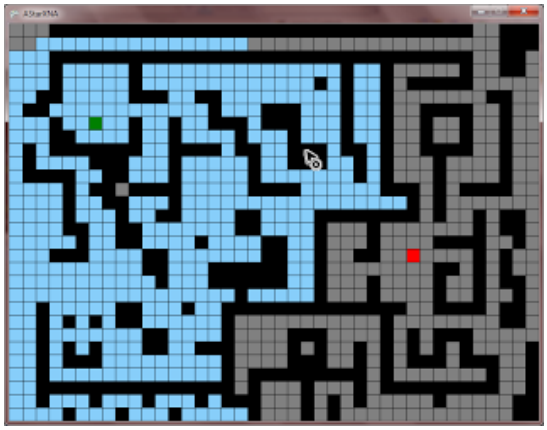

Abbildung 2.2: Besuchte Knoten beim Ablauf von A* (Gre10)

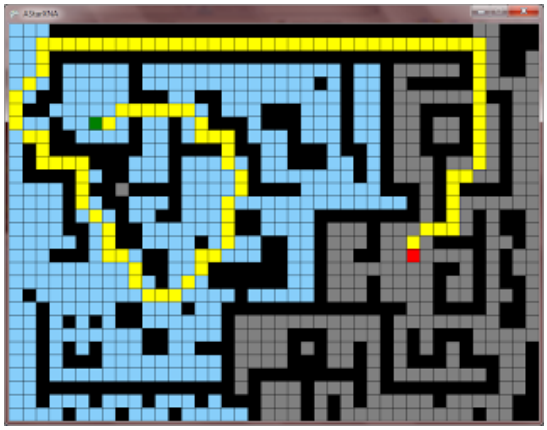

Abbildung 2.3: Resultat vom A*-Algorithmus im Labyrinth

der Abbildung 2.3 während des Ablaufes vom A*-Algorithmus. Es ist davon auszugehen, dass die Open-List während des Ablaufes bis zu 25 Knoten vorübergehend angesammelt hat, bevor es zum Ende hin bis zu 5 sein sollten. Die Closed-List ist zum Ende hin gut befüllt und belegt

mit ungefähr 60 Knoten am meisten Speicherplatz. Wird nur die Closed-List beachtet und optimistisch nur die Objekt-Referenz von 4 Byte genommen wird, dann ergeben sich hierbei 240 Byte Speicherbedarf.
Die Realität großer Welten sieht jedoch meistens anders aus und es können weit über eine Million von Knoten existieren. Bei einem Szenario mit 1.000.010 Knoten sind eine Millionen Knoten miteinander verbunden und die restlichen 10 haben keine Verbindung zu diesen. Im "worst-case" befindet sich der Zielknoten unter den 10 Knoten und der Startknoten unter den der einen Millionen. Nun terminiert der Algorithmus mit der Information, dass kein Weg gefunden wurde und hält zu dieser Zeit eine Million Knoten in der Closed-List, was 4 Megabyte entspricht. Bei heutzutage üblichen 3-4 Gigabyte Arbeitsspeicher sollte es auch bei einer größeren Menge von Einträgen in den beiden Listen nicht zu einer OutOfMemory-Exception kommen. Somit sollte der A*-Algorithmus bezüglich des Arbeitsspeicherbedarfs keine Probleme machen.

Durch die Analyse des A*-Algorithmus wurde gezeigt, dass dieser Schwachstellen besitzt. Zum einen kann es zu einem "worst-case" Verhalten kommen, wenn alle Knoten bis auf den nicht erreichbaren Zielknoten expandiert worden sind. Die Wegsuche läuft somit voll ins Leere und verbraucht unnötig viel Rechenleistung. Um dieses zu vermeiden, wäre es sinnig gewesen gleich vorweg unerreichbare Knoten auszuschließen. Dieses kann durch hierarchische Ansätze umgesetzt werden, indem jeder Knoten einen Parent bekommt. Haben der Start- und der Zielknoten unterschiedliche Parents, dann ist kein Weg vorhanden. Wie beim Labyrinthbeispiel kann es vorkommen, dass unnötig viele Knoten expandiert werden. In diesem Fall liegt das an der verwendeten Heuristik und somit ist der Euklidische-Abstand nicht immer die beste Wahl und eine andere Heuristik wäre sinniger gewesen. Beim Labyrinth würde sich der HPA*[1] anbieten, welcher in Unterabschnitt 5.1.1 noch genauer beschrieben wird. Der Speicherbedarf von A* ist eher gering und kann in den meisten Fällen ignoriert werden. Handelt es sich jedoch um sehr große Graphen und der noch freie Arbeitsspeicher ist eher gering, dann bieten sich andere Algorithmen an, welche in Abschnitt 5.2 beschrieben werden.

[1] Hierarchical Pathfinding A*

Algorithm 2.1 $A* - Algorithmus$

Require: $start : Startknoten, ziel : Zielknoten$
Ensure: $current : aktueller\ Knoten$

```
AddToOpenList(0, start)
repeat
  current ← RemoveMinimumFromOpenList()

  if current = ziel then
    return Gefundener Weg
  end if

  AddToClosedList(current)

  for each nachbar = Nachbarknoten(current) do
    if nachbar ∈ ClosedList then
      CONTINUE
    end if

    kante ← (current, nachbar)
    realeKostenBisNachbar ← realeKosten[current] + kosten[kante]

    if realeKostenBisNachbar ≥ realeKosten[nachbar] then
      CONTINUE
    end if

    parent[nachbar] ← current
    realeKosten[nachbar] ← realeKostenBisNachbar

    kostenDesWeges ← realeKostenBisNachbar + Heuristik(nachbar, ziel)
    if nachbar ∈ OpenList then
      DecreaseKeyInOpenList(nachbar, kostenDesWeges)
    else
      AddToOpenList(nachbar, kostenDesWeges)
    end if
  end for
until OpenList ≠ ∅

return Kein Weg gefunden
```

Kapitel 3

Die Planung der Wegfindung

In diesem Kapitel wird ein Wegfindungssystem vorgestellt, welches sich über den Rest dieser Arbeit hinwegzieht. Aufgrund der Komplexität eines solchen Systems werden in den nachfolgenden drei Kapiteln vorhandene und eigene Verfahren besprochen, bevor in Kapitel 7 eine konkrete Umsetzung getätigt wird.
Zunächst werden Anforderungen gebildet, die verwendeten Komponenten kurz vorgestellt sowie Einschränkungen getätigt.

3.1 Anforderungen

In der Motivation am Anfang dieser Arbeit wurden bereits mehrere Wegfindungsfehler vorgestellt, welche definitiv vermieden werden sollten und somit in die Anforderungen mit einfließen. Im Folgendem werden die Anforderungen detailliert aufgeführt.

1. Es muss schnell berechnet werden können, ob ein Ziel unerreichbar ist.
   Diese Anforderung ist definitiv mit hierarchischen Ansätzen schnell berechenbar und wird in Kapitel 5 genauer beschrieben.

2. Jeder Weg, welcher vom menschlichen Spieler gewählt werden kann, muss auch von der KI mit dem gleichen Spieler-Typ gewählt werden können.
   Um alle Möglichkeiten abzudecken, muss die Wegfindung einen ganz großen Graphen verwenden, welcher die 3D-Welt gut widerspiegelt. Je größer der Graph, desto schlechter ist aber die Performanz und somit untauglich für Echtzeitspiele. Des weiteren müsste für jeden Spielertyp ein eigener Graphen erstellt werden, weil man deren Fähigkeiten wie Teleportation, Springen oder Sprinten mit einbeziehen muss, um dem menschlichen Spieler näher zu kommen. Anhand dieser Tatsachen ist klar, dass es hier zu einen größern Arbeitsspeicherbedarf kommen wird, welcher jedoch mit aktueller Hardware erfüllt sein

sollte. Die Erstellung des Graphen wird in Kapitel 4 sowie eine Performanzsteigerung wird in Kapitel 5 genauer beschrieben.

3. Ist es der KI nicht möglich einen Weg zum menschlichen Spieler zu finden, dann darf diese nicht in Fernkampfreichweite verharren.
   Dieser Punkt ist je nach Intelligenz der Gegner mal stärker und mal schwächer zu beachten. Die Berechnung, ob ein Weg zum Ziel existiert, ist definitiv der Wegfindung zu zu schreiben. Die Knoten können als Information die geographische Position besitzen und somit ist ein Nachbarknoten zu wählen, welcher von der Gefahrenquelle weiter weg ist. Falls der Gegner sich jedoch in Fernkampfreichweite in einer Sackgasse befindet, dann muss sich dieser eventuell vorerst dem Spieler nähern, um anschließend das Weite zu suchen. Die Umsetzung könnte mit Potentialfeldern oder zusätzlichen Kanten- / Knotengewichtungen statt finden und wird in Kapitel 6 genauer betrachtet.

4. Fliegende Kreaturen müssen sich frei bewegen können.
   Die Bewegungsfreiheit von fliegenden Kreaturen ist eine viel größere und somit sollte der Graph mehr Knoten und Kanten enthalten. Trotz dessen ist die Performanz der Wegsuche meistens sehr gut, weil ab einer gewisser Flughöhe so gut wie keine Hindernisse mehr vorkommen. Fliegt der Gegner nur knapp über dem Boden und überfliegt bei Bedarf ein paar Hindernisse, dann ist mit Performanz-Einbußen im Vergleich zu Bodentruppen zu rechnen. Das Tauchen ist dem Fliegen ähnlich und könnte mit gleicher Methodik umgesetzt werden.

5. Ermittelte Wege müssen von der KI gleich beim ersten Anlauf bewältigt werden können.
   Entweder biete die Wegfindung Wege an, welche an einer kleinen Stelle inkorrekt sind oder spielerspezifische Attribute wie die Bewegungsgeschwindigkeit, die Sprungkraft oder den Cooldown von Fähigkeiten werden nicht beachtet. Gehe über eine nicht vorhandene Brücke, gehe durch eine nicht eingeplannte Mauer oder springe mit zu niedriger Geschwindigkeit über einen Graben, müssen definitiv vermieden werden. Behandelt wird dieser Punkt in Kapitel 4 mit anschließender Überprüfung in Kapitel 8.

6. Die KI darf nicht durch Hindernisse durchlaufen können, weil der menschliche Spieler dieses auch nicht kann und als cheaten empfindet.
   Hier laufen gleich zwei Sachen schief. Erstens dürfte die Wegfindung keinen Weg durch Hindernisse anbieten und zweitens sollte spätestens die Kollisionsabfrage zwischen Spieler und Hindernis die Aktion blockieren. Wie im Punkt weiter oben muss bei der

Suchgrapherstellung auf Richtigkeit geachtet werden. Auch eine geeignete Kollisionserkennung sollte genutzt werden.

7. Der Weg muss des menschlichen Spielers nachempfunden werden, welcher sich nicht von Wegpunkt zu Wegpunkt begiebt.
   Diese Anforderung hat mit der Suche auf einem Graphen nichts zu tun. Hierbei handelt es sich um das Begehen eines Weges und wird in Kapitel 6 vorgestellt.

8. In 3D-Landschaften kann unter anderem ein Abhang oder eine Fallluke nur in eine Richtung betreten werden.
   Für diese Wegbeschreitung in nur eine Richtung wird ein gerichteter Graph benötigt. Somit wird an Stellen der beidseitigen Wegbeschreitung statt einer Kanten gleich zwei Kanten benötigt und wirkt sich somit auf den Speicherbedarf aus. Jedoch nicht auf die Performanz.

3.2 Das System

Das Wegfindungssystem besteht aus mehreren Planungsphasen und Komponenten.
Die Planungsphasen werden durch die nächsten drei Kapitel repräsentiert und betreffen die Terrain Analyse zur automatischen Weggrapherstellung über dessen Optimierung bis hin zum Ablauf des Weges.
Die Komponenten werden im Folgenden aufgeführt und beschrieben.

1. Einlesen der Spiel-Objekte.
   Diese Komponente lädt die Objekte in den Arbeitsspeicher und bietet diese gut leserlich an.

2. Erstellen der Kollisionsschicht.
   Objeke besitzen des häufigeren ein oder mehrere Kollisionsobjekte, welche weniger detailliert sind, um die Zeit für Kollisionsabfragen zu reduzieren. Existieren keine Kollisionsobjekte, dann müssen diese aus den grafischen Objektdaten erstellt werden.

3. Erstellen der Navigationsschicht.
   Aus den Kollisionsobjekten untereinander kann der begehbare Bereich errechnet werden. Bei diesem Bereich können sich die Spielfiguren hin und her bewegen, ohne durch Wände zu gehen oder durch die Luft zu schweben. Diese Schicht besteht aus miteinander verbundenen Ebenen, wobei jede Ebene als Dreieck repräsentiert wird. Alle Bereiche zusammen ergeben ein Navigation-Mesh, welches eine zweidimensionale Oberfläche hat.

Hierdurch wird das 3D-Problem auf ein 2D-Problem reduziert. Weil sich die KI immer auf der Navigationsschicht befinden muss, sind teure Kollisionsabfragen nicht mehr von Nöten.

4. Spielertypspezifische Erweiterung der Navigationsschicht.
   Je nach Spielertyp werden Fähigkeiten wie Springen und Teleportation beachtet und somit ergeben sich weitere Verbindungen zwischen davor noch nicht verbundenen Ebenen. Zur Berechnung dessen wird die Kollisionsschicht zur Navigationsschicht hinzugezogen. Ebenen, welche bereits verbunden sind, müssen nur neu verbunden werden, wenn sich dadurch eine Abkürzung ergibt. Jede neu gesetzte Verbindung muss als Information die Fähigkeit als Bedingung besitzen.

5. Die Graph-Erstellung.
   Für jeden Spielertyp wird eine eigener Graph verwendet, welcher aus der jeweiligen Navigationsschicht extrahiert wird. Je nach Spielervolumen und dessen Fähigkeiten variiert die Größe des Graphen. Der Graph kann unterschiedlich aufgebaut sein. Sobald der Graph existiert, können auf diesen Suchverfahren angewandt werden. Die Grapherstellung erfolgt einmal vorab und kann daher wie die Terrainanalyse sehr Kostenintensiv sein.

6. Optimierung der Suche.
   Bei einer guten Repräsentation einer großen und detaillierten Landschaft als Graph ist damit zu rechnen, dass der Graph aus unzählig vielen Knoten und Kanten besteht. Als Zielsetzung soll bei jeder Berechnung nur ein geringer Teil des Graphen betrachtet werden, um schnell ein Resultat zu erhalten. Hierfür werden unter anderem hierarchische Verfahren verwendet.

7. Die "Zickzack"-Glättung.
   Beim Ablauf des Weges soll ein Zickzack beim Abarbeiten der Knoten vermieden werden. Hierbei muss beachtet werden, dass eine solche Glättung den Spieler nicht für ein paar Augenblicke über einen Abgrund oder durch Kollisionsobjekte führt. Mit der Navigationsschicht und Potenzialfeldern kann der gefundene Weg geglättet werden. Hierfür ist davon auszugehen, dass existierende Wegpunkte durch mehrere andere ersetzt werden müssen.

8. Ablenkende Quellen.
   In Spielen wie auch im richtigen Leben kommt man mal von einem geplanten Weg ab, weil man sich von den aktuellen Umweltzuständen leiten lässt. Nachts, also bei möglicher

Gefahr, ist man ungern allein unterwegs und wenn man eine Münze auf dem Boden liegen sieht, nimmt man einen kleinen Umweg in Kauf. Bei düsteren Gestalten hingegen wird ein Sicherheitsabstand um diese gemacht. Solche ablenkenden Quellen können als Potentialfelder angesehen werden. Von gegnerischen Einheiten hält man sich fern, es sei den man ist in der Übermacht. Ist die Lage angespannt, dann fühlt man sich zu seinen Kumpels angezogen. Mit dieser Komponente soll der begehbare Bereich besser ausgenutzt werden und der Graph mit zusätzlichen Informationen angereichert werden.

Je nach Umsetzung aller Komponenten kann das resultierende System verschieden ausfallen. Eine schnelle Wegplanung mit einem Resultat, welches das des menschlichen Spielers nahe kommt, soll das System erfüllen.

3.3 Einschränkungen

Wegfindungen sind meistens dem jeweiligen Problemfall angepasst, um einen guten Mix aus Performanz und Akzeptanz zu erreichen.
Im Folgendem werden die Einschränkungen erläutert, welche in dieser Arbeit nicht weiter behandelt werden.

1. Fliegende Gegner.
   Für jeden fliegenden und tauchenden Gegnertyp müsste ein eigener Graph angefertigt werden, wobei die Terrain Analyse andere Gesichtspunkte beachten muss, weil es sich ja ansonsten nur um Bodentruppen handelt. Des weiteren ist hierfür das Navigation-Mesh ohne weiteres nicht mehr zu verwenden. Aufgrund des Mehraufwandes, welcher den Umfang dieser Arbeit sprengen würde, wird hierauf verzichtet und somit fällt die Anforderung Nummer 4 weg.

2. Spielerdarstellung.
   Der Spieler wird einer genauen geographischen Koordinate zugeordnet und besitzen ein Spielervolumen. Dieses wird in Form eines Zylinders und darunter ein auf dem Kopf stehenden Kegels dargestellt. Beim menschlichen Spieler muss genau das gleiche Spielervolumen verwendet werden, um Kollisionen identisch zu vermeiden beziehungsweise zu begegnen. Andere Spielervolumen werden in dieser Arbeit nicht besprochen.

3. Dynamische Objekte.
   Dynamische Objekte also temporäre Hindernisse wie zerstörbare Mauern, Gebäude oder andere Spieler wird verzichtet. Eine Erweiterung dessen soll aber möglich sein. Oft

werden statische Graphen verwendet und durch dynamische Sub-Knoten und -Kanten erweitert, um die Berechnungskosten gering zu halten.

4. Fernkampfreichweite von nicht erreichbaren Gegnern vermeiden.
   Das Vermeiden der Fernkampfreichweite kann durch Potentialfelder umgesetzt werden. Hierbei könnten Knoten vermieden werden, welche sich in der Fernkampfreichweite der Gegners befinden. Eine konkrete Umsetzung zur Anforderung Nummer 3 wird es nicht geben. Statt dessen wechselt der Spieler sein Ziel. Ist kein Ziel erreichbar, dann begibt sich der Spieler zu seiner Ausgangsposition.

5. Anzahl der Gegnertypen.
   Die Anzahl der Gegnertypen wird auf vier beschränkt. Diese Gegnerklassen sind Krieger, Schurke, Magier und Priester. Die Fähigkeiten dieser sind Anstürmen, Sprinten, Teleportieren und Verlangsamen. Weitere für die Wegsuche relevante Fähigkeiten werden nicht umgesetzt.

6. Beschleunigungs- und Verlangsamungseffekte.
   Ändert sich die Bewegungsgeschwindigkeit des Gegners, dann kann dieser theoretischerweise mal weiter und mal nicht ganz so weit springen. Bei knapp bemessenen Wegübergängen, kann es somit vorkommen, dass diese mit Verlangsamungseffekten nicht benutzbar sind. Diese Bewegungsgeschwindigkeitseffekte sind zeitlich befristet und daher muss der Graph zeitliche Aspekte beachten können. Die Graphen sollten die Fähigkeits-Kanten, welche die Geschwindigkeit beachten, effektiv ein- und ausblenden können. Kombinationen wie Sprinten und Verlangsamen können gleichzeitig aktiv sein. Damit für die möglichen Kombinationen nicht zu viele Kanten erstellt werden müssen, wird der prozentuale Effekt identisch sein, so dass sich die Kombination Sprinten und Verlangsamen neutralisiert.

7. Bewegungsgeschwindigkeit.
   Auf Beschleunigungen wie in Rennspielen wird verzichtet, um zusätzliche Kanten und Bedingungen zu vermeiden. Wenn der Spieler aus dem Stand los läuft, dann ist dieser gleich bei 100% der möglichen Bewegungsgeschwindigkeit.

3.4 Fazit und Ausblick

Es soll ein Wegfindungsystem entstehen, welches statisch vorberechnet wird und zur Laufzeit ein schnelles Resultat liefert. Die Wegabdeckung muss zu 100% gewährleistet sein, damit

die KI bezüglich der Bewegungsfreiheit nicht benachteiligt wird. Für die Wegberechnung werden Wege erst grob und dann iterativ immer feiner berechnet. Bei diesem hierarchischen Optimierungsverfahren kann es vorkommen, dass der kürzeste Weg nicht gefunden wird, was jedoch nicht schlimm ist, wenn die Wegkosten nah genug an die des optimalen Weges ran kommen. Einmal berechnete Wege müssen von der KI fehlerfrei ablaufbar sein. Wenn der jeweilige Spieler durch gegnerische Fähigkeiten verlangsamt wird, dann muss eine neue Wegberechnung gestartet werden, weil es vorkommen kann, dass der geplante Weg mit einer Verlangsamung nicht zu bewältigen ist. Es gibt insgesamt vier Klassen, welche auf Grund deren Fähigkeiten eigene Graphen für die Wegsuche benötigen. Hinzu kommen zusätzlich Kanten für geschwindigkeitsverändernde Effekte. Als bewegliche Objekte gibt es nur die Spieler, welche untereinander nicht kollidieren und somit nicht von der Wegfindung beachtet werden müssen. Der Ablauf des Weges wird dem menschlichen Spieler nachempfunden und soll somit normal wirken. Durch Potentialfelder sollen die Spieler beeinflusst werden können, um sicherheitshalber Umwege zu nehmen oder um nebenbei schwache Gegner auszuschalten.

Das System soll dann auf das Warsong "Capture the Flag" Szenario aus World of Warcraft angewandt werden. Ziel ist es jeweils drei gegnerische Flaggen aus der gegnerischen Basis zu stehlen und in der eigenen Basis abzugeben, um zu gewinnen. Die Spielertypen besitzen nur Leben und cooldownbedingte Fähigkeiten; also kein Mana oder ähnliches. Die Anzahl der Fähigkeiten ist auch stark beschränkt, damit Fähigkeiten betreffend der Wegfindung oft an die Reihe kommen. Hierbei soll jeder Spieler individuell sein und bekommt verschiedene Werte für folgende Variablen. Beschützer der eigenen Flagge, Flaggendiebstahlkommando und Gegnerschlächter. Je nach Situation soll ein menschliches Verhalten erkennbar sein. Hierbei spielt auch die Reaktionszeit eine Rolle. Der Mensch hat eine Reaktionszeit von ungefähr einer viertel Sekunde. Um dem nahe zu kommen, müsste jede viertel Sekunde überprüft werden, ob sich das Ziel des Spielers ändern soll, die Position des Zieles muss erfasst werden und falls diese von der letzten Position abweicht, müsste eine neue Wegberechnung angestoßen werden. Bei Warsong gibt es zwei Teams mit jeweils zehn Spielern. Hierbei muss das Spiel flüssig laufen und somit dürfen die Berechnungen pro Person nur ein paar Millisekunden betragen.
In Kapitel 8 wird an mehreren spiel- und wegfindungsbedingten Variablen rumgeschraubt und das Verhalten getestet. Des weiteren wird die Wegfindung mit der des A*-Algorithmus verglichen und bewertet.

Kapitel 4

Erstellung des Suchgraphen

In diesem Kapitel werden existierende Techniken zur Graph-Erstellung beschrieben. Anschließend wird eine eigene automatisierte Technik vorgestellt, welche eine ausführliche Terrain Analyse ausführt, um die begehbaren Bereiche für das Navigation-Mesh zu ermitteln und die benötigten Graphen zu erstellen.

4.1 Bekannte Methodiken

4.1.1 Rasterbasiertes Vorgehen

Wie in Age of Empires 2 oder Civilization 5 wird die Welt in quadratische oder auch hexagonale Raster aufgeteilt. Dank des Rasteraufbaues ist schnell ermittelt, in welchem Feld sich ein Spieler befindet. Jedes Raster kann als Knoten in einem Graphen angesehen werden und ist entweder begehbar oder blockiert. Somit muss kein Zusätzlicher Graph erstellt werden, weil dieser ja bereits existiert. Auch Performanzoptimierungen aufgrund von Hierarchien lassen sich hier schnell umsetzen. Das Raster wird grober dargestellt, indem vier Raster zu einem kombiniert werden. In 3D-Welten kann ein Octree verwendet werden. Hierdurch hat ein Raster statt 8 genau 26 Nachbarn.
Um die Knotenanzahl gering zu halten werden mehrere Raster iterativ zu Clustern zusammenfassen (s. Abb. 4.1). Jeder entstandene Knoten hat als Eigenschaft die Clustergröße und ob der Bereich komplett oder überhaupt nicht begehbar ist. Alle entstandenen verschieden großen Cluster werden mit denen der Nachbarn verbunden, wenn diese ebenfalls begehbar sind. Hierbei wird der Mittelpunkt jedes Clusters als Kontenpunkt im Graphen genommen, was sich jedoch meistens negativ auf die Qualität des Weges auswirkt, weil Wege unnötigerweise immer über die Clustermittelpunkte geführt werden. Hierbei handelt es sich um keinen hierarchischen Ansatz. Damit dieses der Fall wäre, müsste zuerst auf großen Clustern und dann iterativ auf immer kleineren Kind-Clustern gesucht werden.

Vorteil bei dieser Welt-Modellierung ist, dass kein wirklich komplexes Wegfindungssystem

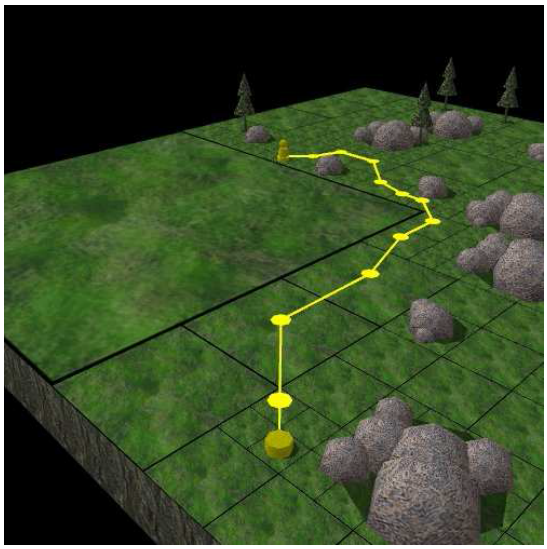

Abbildung 4.1: Quadtree splitting (Hal05)

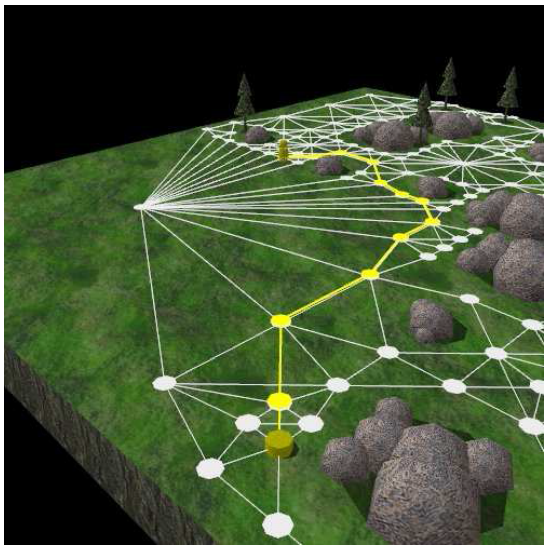

Abbildung 4.2: Accompanying Searchtree

von Nöten ist. Der Graph existiert bereits und meistens erfüllt der HPA*[1]-Algorithmus seine Arbeit, was in Unterabschnitt 5.1.1 noch genauer erwähnt wird.
Nachteil ist dass meistens unzählig viele Knoten entstehen. Bei einer 256x256 Landschaft sind das bereits 65536 Knoten und bei einer 3D-Landschaft sind es definitiv viel mehr. Zu beachten ist, dass die Anzahl der expandierten Knoten sich negativ auf die Performanz der Suche auswirkt.

4.1.2 Korridorbasiertes Vorgehen

Bei Tunnelsystemen, Autorennspielen oder in städtischen Fußgängernetzen bieten sich Korridore an. Als erfolgreiche Methodik hat sich die Corridor Maps Method herauskristallisiert. Hierbei wird immer mittig im Gang oder auf dem Fußgängerbürgersteig eine Kante gelegt. Ändert sich die Richtung oder die Breite, dann werden dort Knoten platziert. Jeder Knoten hat einen Radius, in welchem sich kein Hindernis befinden darf. Die Kanten verbinden diesen begehbaren Bereich der zwei Knoten miteinander.
Die Kanten den Graphen werden als Backbone-Path bezeichnet, weil diese mittig liegen (s. Abb. 4.3). Die Spieler werden als Kreise definiert. Ein einziger Graph kann ganz viele verschie-

[1] Hierarchical Pathfinding A*

dene Spieler aufnehmen, wobei der Spieler nur Knoten passieren kann, dessen Radius größer als der eigene Radius ist. Es wird unterstützt, dass Spieler nicht durch andere durchgehen können und somit diesen ausweichen. Crowd-Controlling unterstützt diese Methodik ebenso, genau wie das Abglätten von eckigen Wegen über einen Faktor.

Vorteil ist, dass der benötigte Backbone-Graph über Voronoi-Diagrammen oder sogar in

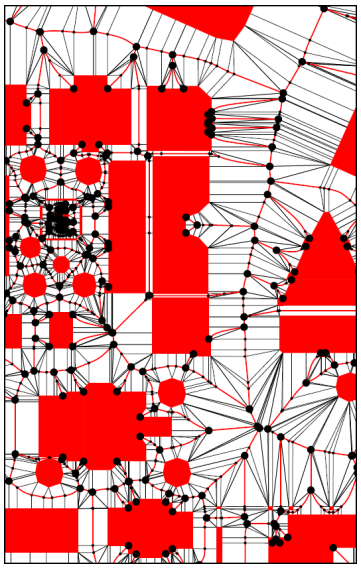

Abbildung 4.3: Corridor map and closest points (Ger10)

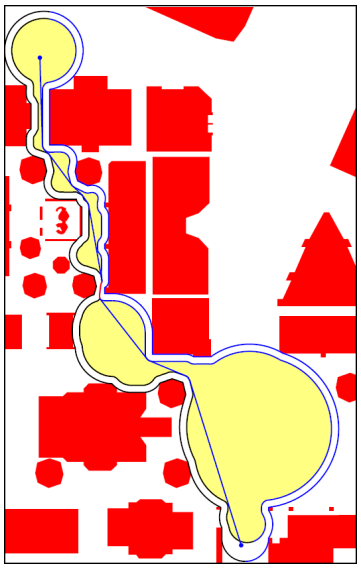

Abbildung 4.4: Shortest path

Abbildung 4.5: Smooth path

kürzester Zeit über die Grafikkarte automatisiert erstellt werden kann. Ein begehbarer Bereich zum ausweichen anderer Spieler ist definiert.

Nachteil ist, dass die Teilnehmer als Kreise dargestellt werden müssen, was bei länglichen Fahrzeugen nicht wirklich möglich ist. Ein weiterer Nachteil ist, dass diese Methodik in ihrer Form nicht wirklich 3D-tauglich ist[2].

4.1.3 Navigation-Mesh-basiertes Vorgehen

Für dieses Verfahren werden die begehbaren Dreiecke zu ungefähr gleich großen konvexen Bereichen vereint, wobei mehrere nebeneinanderliegende Bereiche das Navigation-Mesh bilden. In diesen Bereichen können sich die Spieler beliebig hin und her bewegen, sowie dieses über

[2]Die Corridor Maps Method wird in Unterabschnitt 6.1.1 noch genauer untersucht

die am Rand befindende Kanten zu benachbarten Bereichen verlassen. Diese Repräsentation ist dem der 3D-Welten am nächsten und somit lässt sich der komplette begehbare Bereich mit diesem Verfahren abdecken. Im abstrakten Sinn sind die Bereiche die Knoten und die Verbindungen zu den Nachbarbereichen die Kanten.
Vorteil ist eine 100%ig mögliche Abdeckungsquote und dass der begehbare Bereich klar definiert ist. Anders als bei der CMM lassen sich auch eckige dynamische Hindernisse platzieren. Nachteil ist die komplexe Erstellung eines korrekten Graphens, welcher je nach Spielervolumen und kollidierbaren Hindernissen wie tiefen schrägen Decken, in der Luft hängenden Hindernissen oder in den Raum geneigte Wände anders ausfällt.

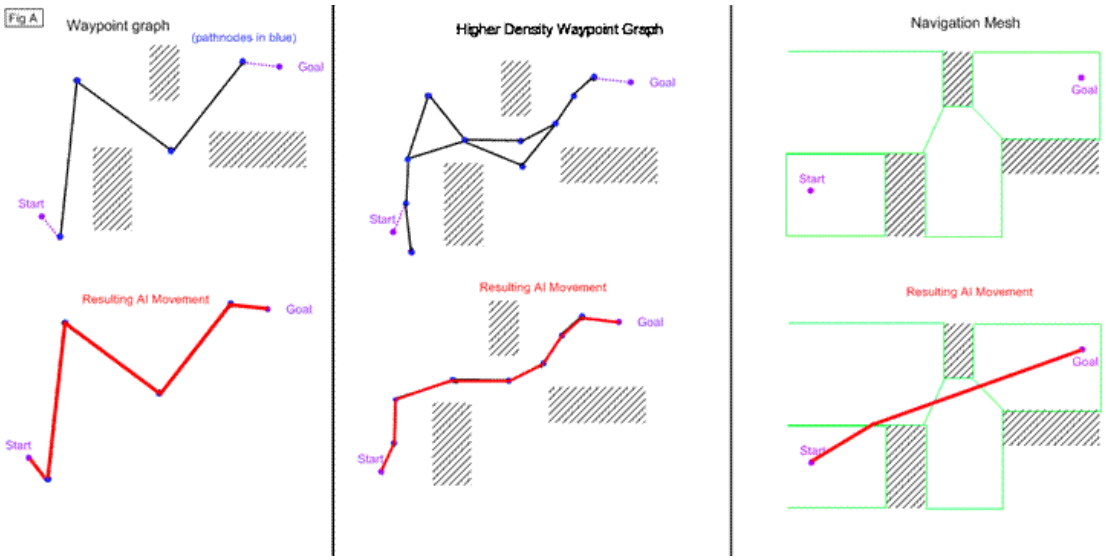

Abbildung 4.6: Vergleich zwischen einem Waypointgraph und Navigation-Mesh (Tec)

4.1.4 Wegpunkt-basiertes Vorgehen

Bei diesem Verfahren wandert der Spieler von Knoten zu Knoten, welche durch gerichteten oder ungerichteten Kanten verbunden sind. Damit der Ablauf nicht zu eckig erscheint, müssen mehrere Knoten vorhanden sein. Die Knoten sind meist in und an Ecken platziert, wo sich auch der Spieler befinden kann, um wiederum die Anzahl gering halten zu können. Eine schnelle automatisierte Umsetzung in 3D-Gelände mit regelmäßig angeordneten Dreiecken, wird unter anderem so umgesetzt, dass der Mittelpunkt jedes begehbaren Dreieckes mit denen der Nachbarn verbunden wird.

Vorteil dabei ist, dass dieses Verfahren leicht verständlich und je nach Umsetzung schnell zu implementieren ist.
Nachteil ist, dass je nach Umsetzung zu viele oder zu wenige Knoten vorhanden sind und dadurch ein hoher Speicherverbrauch sowie Performanzeinbußen oder ein Zickzack der Wege zu erkennen ist.

4.1.5 Händisches Vorgehen

Alle Verfahren außer das des rasterbasierten lassen sich auch händisch erzeugen. Dieses sollte jedoch nur von einem erfahrenen Designer übernommen werden und benötigt einen dafür erstellten Editor.
Vorteil dabei ist, dass dem Ersteller bewusst ist, welche Wege von Nöten sind. Die Knoten werden gezielt gesetzt und unnötige, falls vorhanden, entfernt. Dieses wirkt sich sehr Positiv auf die Performanz aus.
Nachteil ist die manuelle Arbeit, welche sehr zeitintensiv ist. Des weiteren können ein paar wichtige Kanten vergessen worden sein, welche auf dem ersten Blick eher unwichtig erschienen.

4.2 Eigene Terrainanalyse

Die Terrainanalyseverfahren sind eher schlicht gehalten oder sind einfach nicht frei zugänglich. Daher muss eine eigene Terrainanalyse her, welche den Anforderungen 2, 5, 6 und 8 gerecht wird (s. Abschnitt 3.1).

4.2.1 Auffrischung: Repräsentation einer 3D-Welt

Durch Terrainanalyse werden wichtige Informationen aus der Umgebung extrahiert, welche der Wegfindung zugute kommen.
In 3D-Spielen wird es oft so gehandhabt, dass der Boden der Landschaft separat erstellt wird und als Highmap hinterlegt ist. Mauern, Dekoration und ähnliches existiert jeweils als eigenes Objekt und kann somit mehrmals auf die Landschaft platziert werden. Durch die platzierten Hindernisse kann die Struktur des Landschaftsbodens nicht eins zu eins für die Wegfindung genutzt werden. Wege müssen um die Hindernisse herum geführt werden und bei großen Landschaften mit nur wenigen Hindernissen ist die Struktur des Landschaftsbodens zu detailliert und würde die Wegfindung nur ausbremsen. In 2D-Welten reicht es aus, um die Gegenstände

der Landschaft ein Kollisionslinie zu ziehen. In 3D-Welten kann gegen niedrige Gegenstände wie Kisten gegen gelaufen werden, jedoch kann auch auf diese hochgesprungen werden, um eventuell höher gelegenes Gelände erreichen zu können. Daher sind mehrere Wege möglich, dessen Anzahl jedoch überschaubar ist.

4.2.2 Bestimmung der begehbaren Bereiche

Nun zu den geometrischen Berechnungen, wofür die Kollisionsebene verwendet wird. Es werden zunächst die absoluten Koordinaten der 3D-Landschaftsboden und der darauf platzierten Objekte berechnet. Hierzu werden die relativen Koordinaten mit der Rotations-, Skalierungs- und Transformationsmatrix verrechnet. Mit den resultierenden absoluten Koordinaten können die Dreiecke auf Kollisionen mit anderen Dreiecken der Welt/Objekte überprüft werden.
Nun wird die 3D-Landschaft mit dem Kollisionsvolumen der Spieler geflutet, welche in Unterabschnitt 4.2.5 noch genauer beschrieben wird. Diese Flutung gibt den für den Spieler begehbaren Bereich zurück. Hierdurch wurden die Dreiecke mit einem positivem Up-Vektor zum Teil in kleinere zerlegt. Ist der Up-Vektor Null, dann handelt es sich um eine steile Wand und kann ignoriert werden. Ist der Up-Vektor negativ, dann handelt es sich um eine "Decke" und ist somit ebenfalls nicht relevant für die Wegfindung. Es sei denn, es wird eine Wegfindung für Spinnen erstellt.

Alle nicht begehbaren Dreiecke werden entfernt und den übrig bleibenden wird ein Typ zugeordnet. In beide Richtungen begehbare, in eine Richtung herunterrutschbare und als Hindernis dienende Dreiecke. Anschließend werden alle begehbaren Dreiecke so zusammengefasst, dass die entstehenden Bereiche konvex sind und dass die Bereiche so gut wie möglich gleich groß sind. Herunterrutschbare Dreiecke dienen als verlängerten Bereichsübergang und verbinden konvexe Bereiche miteinander. Hierbei kann es vorkommen, dass die herunterrutschbaren Dreiecke weiter zerlegt werden müssen, wenn diese in mehreren konvexen Bereichen beginnen oder enden. Die entstandenen konvexen Bereiche mit deren Verbindungen werden zu einem Navigation-Mesh zusammen gefasst.

4.2.3 Beachtung der Spieler-Fähigkeiten

Im Folgendem werden die Fähigkeiten beachtet. Diese sollen nur zusätzliche Verbindungen liefern, wenn dadurch Bereiche kostengünstiger erreicht werden können. Zu den Standard-

Fähigkeiten gehören Springen und Fallenlassen. Diese sind mit einer mathematischen Funktion bestimmbar, welche als Parameter die Spielergeschwindigkeit entgegennimmt.
Die Funktion für das Fallenlassen wird an den Bereichsrändern ausgeführt. Bei der Anwendung wird das Spielervolumen mit der Fall-Funktion von A nach B verschoben. Zunächst wird nur die Funktion ausgeführt und auf Kollisionen mit anderen Bereichen getestet. Existiert eine solche Kollision, welche dem Spieler nützlich ist, dann wird anschließend überprüft, ob das Spielervolumen ebenfalls diesen Weg nehmen kann. Wenn ja, dann wird eine Verbindung zwischen diesen Bereichen hergestellt. Diese Verbindungen wird in Form von Unterbereichen dargestellt, welche auf die benötigte Fähigkeit sowie die Blickrichtung hinweisen. Je nach Falltiefe kann der Spieler Leben verlieren oder sogar sterben und daher sollte der Verbindung dieses mitgeteilt werden.
Die Funktion für das Springen ist ähnlich dem des Fallenlassens. Einzige Unterschiede sind, dass diese nicht nur am Bereichsrand angewandt werden muss, sondern auch innerhalb des Bereiches und dass das Volumen in welchem gesprungen werden kann ein größeres ist und am Anfang auch in die Höhe geht. Diese kostenintensive Berechnung wird nur einmalig zur Informationsgewinnung getätigt und sollte somit akzeptabel sein.
Im Folgendem werden noch die Fähigkeiten durchgegangen, welche für diese Arbeit relevant sind:

- Anstürmen soll nur möglich sein, wenn ein direkter Weg zum Ziel vorhanden ist, wobei zusätzlich auch steilere Bereiche passierbar sind.
- Sprinten erhöht die Bewegungsgeschwindigkeit und wirkt sich somit auf alle anderen Fähigkeiten aus, welche als Parameter die Spielergeschwindigkeit benötigen. Unter anderem kann dann weiter gesprungen werden, wodurch neue Verbindungen entstehen können.
- Durch Teleportation soll sich der Spieler mehrere Meter gerade nach vorne katapultieren. Die Reichweite dieser Fähigkeit wird vermindert, wenn Hindernisse im weg sind, weil eine freie Luftlinie von Nöten ist.
- Beim Verlangsamen ist es wie beim Sprinten. Alle Fähigkeiten bezüglich der Spielergeschwindigkeit müssen mit der Bewegungsgeschwindigkeit verrechnet werden.

4.2.4 Weitere mögliche Eigenschaften

Nun können alle Bereiche sowie Unterbereiche im abstrakten Sinne als Knoten angesehen werden und die Verbindungen zwischen diesen als Kanten. Alle Bereiche zusammen ergeben

das Navigation Mesh, welches in dieser Arbeit genutzt wird.
Zusätzlich können je nach Spiel und Umgebung Bereiche mit Eigenschaften erstellt werden. Somit könnte verseuchter Boden Schaden zufügen oder Wasser die Bewegungsgeschwindigkeit verringern. Zu beachten ist, dass je mehr Informationen zur Laufzeit ausgewertet werden müssen, desto mehr Zeit benötigt auch die Wegberechnung.

4.2.5 Volumenkollisionen

Dieser Teil wird in Unterabschnitt 4.2.2 erwähnt und aufgrund der Komplexität am Ende des Abschnittes aufgeführt.
Der Spieler wird als Zylinder dargestellt, welcher im unteren Bereich spitz zuläuft und oberhalb aus einem Halbkreis besteht (s. Abb. 4.7). Der spitze Punkt ist die Spielerposition in der 3D-Welt und die Steigung von diesem bis zum vollen Spielerradius ist die Steigung, welche der Spieler begehen kann. Ist die Steigung der Welt steiler, dann erkennt der Spieler dieses als eine Hindernis wie eine Wand und bleibt davor stehen. Diese Repräsentation des Spielers ermöglicht es an Rändern von Klippen entlang zu gehen und nicht zum Teil durch Felswände zu laufen.
Für jedes begehbare Dreieck wird das Spielervolumen platziert und auf Kollisionen überprüft. Existieren Kollisionen, dann wird das begehbare Dreieck dementsprechend zerteilt (s. Abb. 4.8). Der begehbare Raum wird dadurch weiter eingeschränkt und übrig bleibt der tatsächlich begehbare Bereich.
Für Fähigkeiten muss das Zielgebiet das Spielervolumen aufnehmen können und auch der Weg dahin darf je nach Fähigkeiten-Typ nicht blockiert sein.
Die Kollisionsschicht muss den Spieler ebenso als solches Volumen wahrnehmen, damit der KI und dem menschlichen Spieler die gleiche Bewegungsfreiheit gegeben ist.

Im Folgendem werden nun die Kollisionsberechnungen zwischen dem Spielervolumen und den Dreiecken beschrieben.
Zunächst kann eine Sphere um das Spielervolumen und um das Dreieck gesetzt werden. Wenn diese zwei Spheren nicht kollidieren, was leicht durch die Distanzberechnung zu überprüfen ist, dann muss keine zusätzliche Rechenleistung zur genauen Kollisionsberechnung verwendet werden. Findet eine Kollision statt, dann wird der nächstgelegenste Punkt zwischen dem Dreieck und des als Linie repräsentierten Spielerrumpfes ermittelt (s. Abb. 4.7 und hilfreiche Literatur (Eri04)). Ist der Punkt weiter Weg als der Spielerradius lang ist, dann existiert keine Kollision. Ist der Punkt in Höhe des Zylinders, dann existiert eine Kollision. Andernfalls muss

der auf dem Kopf stehende Kegel überprüft werden, falls keine Kollision stattfindet, ist eine letzte Überprüfung der Linie des Zylinders von Nöten, um eine endgültige Kollision auszuschließen.

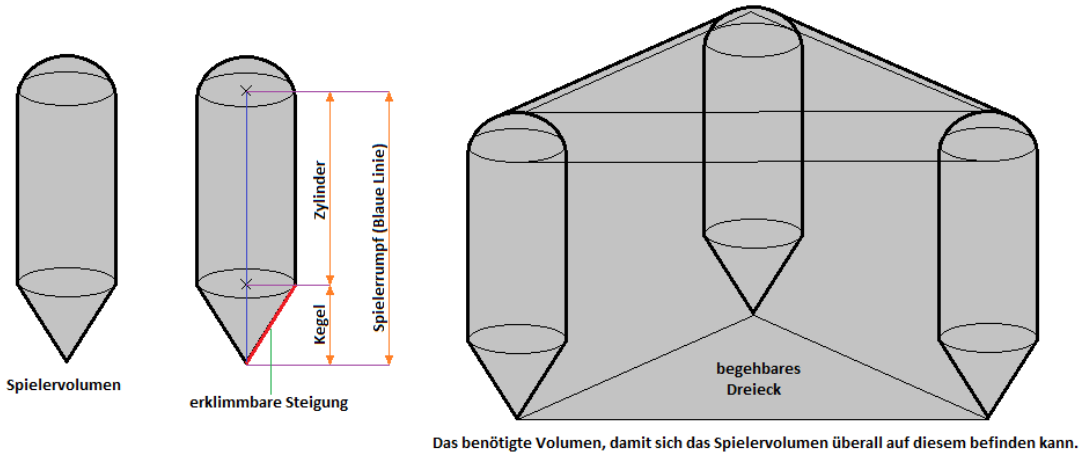

Abbildung 4.7: Spielervolumen und Volumen des begehbaren Dreieckbodens

Nun wird das begehbare Dreieck untersucht.
Hierfür wird zunächst an jeder der drei Ecken das Spielervolumen platziert und anschließend werden diese verbunden. Es entsteht ein neues Volumen, welches wieder in eine Kollisions-Sphere verpackt wird. Wenn Dreiecke mit der Sphere und dem Volumen kollidieren, dann werden diese zunächst gesammelt. Anschließend geht es an das Zerstückeln des begehbaren Dreiecks. Eine bekannte Methodik ist ein Rasterierungsbeschuss mit Linien, welche bei Hindernissen eine Kollision melden und begehbare sowie nicht begehbare Rasterkästchen ermitteln (s. (Tec)). Dieses Verfahren soll nicht verwendet werden, weil dieses im Vergleich zur Kollisionserkennung zu ungenau ist und bei höherer Präzision zu viel Speicher verbraucht. Statt dessen finden komplexere Berechnungen statt, welche die Zerstücklung vornehmen und anschließend eine Validierung ausführen müssen, weil kleine Ungenauigkeiten akzeptiert werden (s. Abb. 4.8).

Bei den Fähigkeiten sieht das wie folgt aus.
Es wird eine Funktion angewandt, welche ebenfalls im Spiel verwendet wird, um die neue Position des Spielers während des Fallenlassens, Springens oder ähnlichem berechnet. Diese wird während des Spielens öfters aufgerufen, wobei eine höhere Rate die Performanz ausbremst. Somit muss ein guter Mittelwert ermittelt werden. Für die Bereichsübergänge der Fähigkeiten werden gleich alle Punkte vom Start bis zum Ziel ermittelt, welche mit Linien verbunden werden

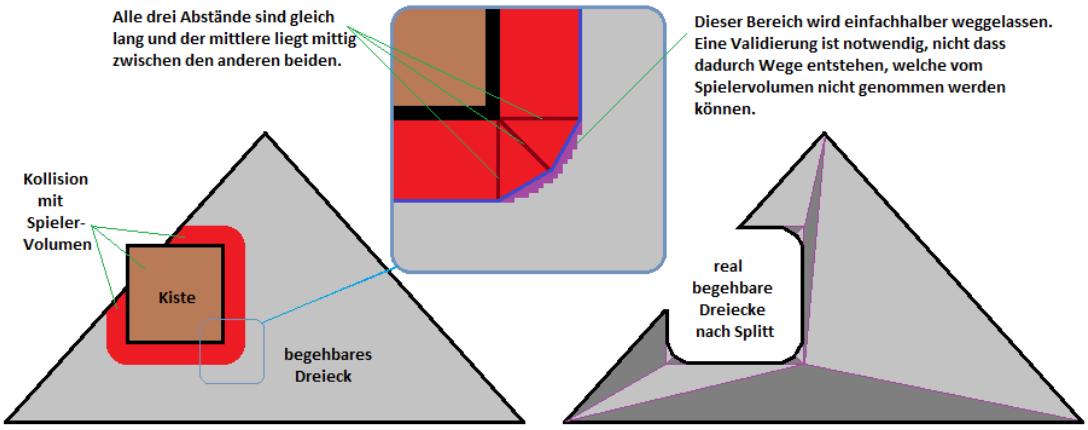

Abbildung 4.8: Beispiel einer Zerlegung eines begehbaren Dreiecks unter Beachtung des Spielervolumens durch eine Kiste

und zusammen eine krumme Linie bilden. Beim Fallenlassen wird an den Ecken des begehbaren Dreiecks jeweils paralell diese Linie angebracht und Ebenen zwischen diesen gebildet, welche anschließend mit Kollisionen anderer Welt-Dreiecke überprüft werden. Existieren Kollisionen, dann ist das Fallenlassen entweder nicht möglich, zum Teil nicht möglich oder berührt zum Teil schon früher den Boden. Je nachdem muss dann der Fähigkeitenübergang zerstückelt oder ganz gestrichen werden. Die Fähigkeit Springen kann in zwei Überprüfungen zerlegt werden. Kann auf andere in der Nähe befindenen begehbaren Bereiche hoch gesprungen werden oder kann durch die Sprungreichweite ab der maximal erreichten Höhe ähnlich des Fallenlassens ein anderer Bereich erreicht werden. Nachdem die "Brücken" zwischen den Bereichen kollisionsfrei ist und in eventuelle Bahnen zerlegt wurde, findet auf der Oberfläche eine dem begehbaren Dreiecken ähnliche Zerlegung in Bezug auf das Spielervolumen statt.
Unter anderem sollte die Funktion des Fallenlassens in verschiedenen Winkeln vom begehbaren Dreieck gestartet werden, wobei im Normalfall die zwei Extremata ausreichend sein sollten, wenn dadurch kein zusätzliche begehbarer Bereich ausgelassen wird (s. Abb. 4.9).

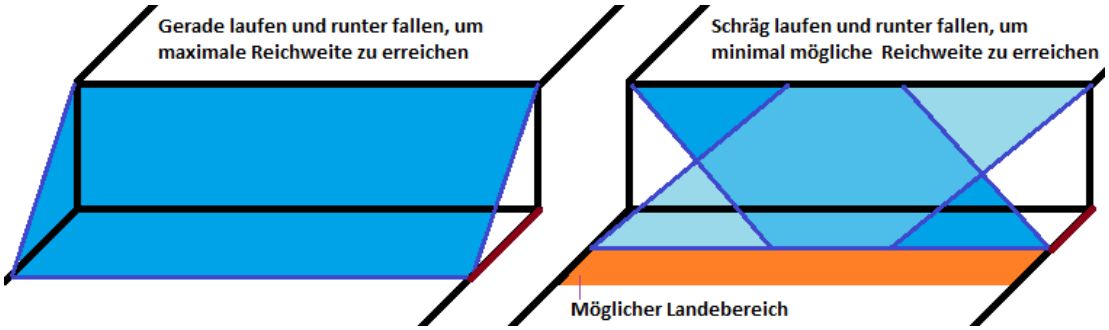

Abbildung 4.9: Der mögliche Landebereich beim Fallen durch die Extremata

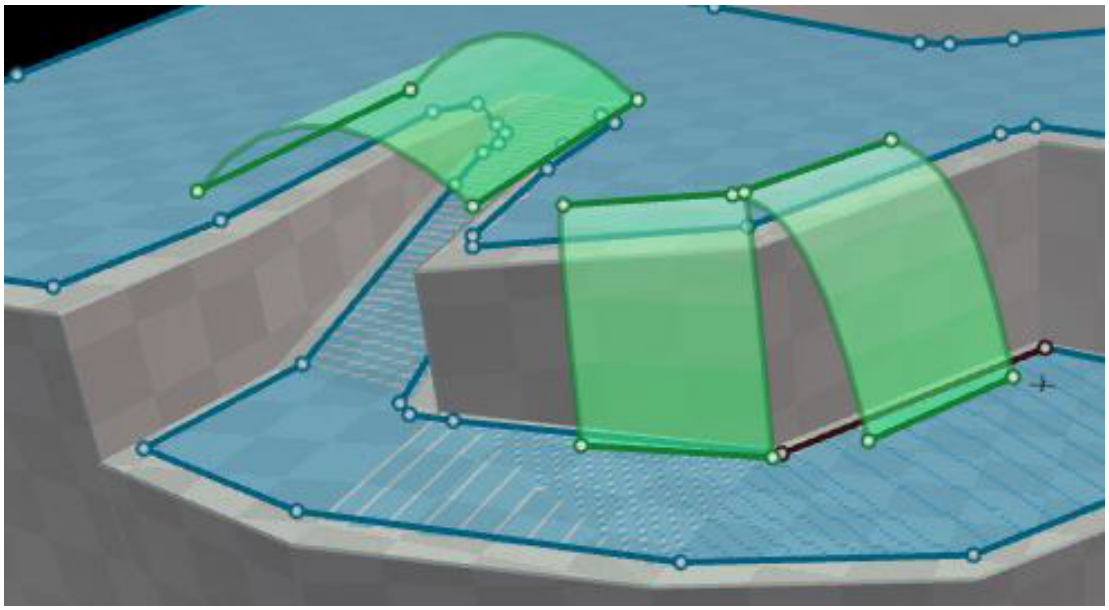

Abbildung 4.10: Ein paar mögliche Bereichsübergänge des Typs Springen und Fallenlassen (pra)

Kapitel 5

Optimierung der Suche

Werden Suchverfahren eins zu eins aus dem Lehrbuch umgesetzt, dann wird dieses in einer 3D-Welt nicht unbedingt ein zufriedenstellendes Resultat liefern und sollte daher noch optimiert werden. Optimierungen werden durchgeführt, um die Performanz zu steigern, den Arbeitsspeicherbedarf zu senken oder die Qualität zu beeinflussen. Das Ziel ist es in möglichst kurzer Zeit einen akzeptablen Weg mit durchschnittlichem Arbeitsspeicherbedarf zu berechnen.

5.1 Performanz

Bei der Wegsuche wird Rechenleistung für folgendes benötigt. Anders als in der Lehrbuch-Graphentheorie wird nicht ein Start- und ein Zielknoten ausgewählt. Statt dessen wird die Position des Spielers und des Ziels genommen und von beiden der best gelegenste Knoten des Graphen bestimmt. Zur Bestimmung dieser müssen häufig kostenintensive Berechnungen durchgeführt werden, bevor die Wegsuche überhaupt starten kann. Es sei denn, dass ein Raster-Graph für die Suche verwendet wird oder dass die Spieler die eigene Position im Graphen stetig aktualisieren. Die Kosten für die Wegsuche auf dem Graphen sind je nach gewähltem Algorithmus und der Anzahl der Informationen verschieden hoch. Des weiteren können oft getätigte Suchen gemerkt werden, so dass bei einer wiederholten Suchabfrage die Suche auf dem Graphen überflüssig wird. Werden zu viele Wege gecacht, dann wird der Arbeitsspeicher nicht mehr ausreichen und es wird zwangsläufig zu Fehlern und Programmabstürzen kommen.

Wie bereits schon in Abschnitt 2.5 beschrieben, bietet sich die Konstellation des A*-Algorithmus mit der Euklidischen-Abstand Heuristik aus Performanzgründen nicht an, weil es mehrere "worst-case"-Szenarien gibt. Durch andere Heuristiken oder Algorithmen kann eine Performanzsteigerung erreicht werden. Unter anderem bieten sich hier die ALT[1]-Algorithmen an,

[1] ALT: A*, Landmarks und Triangle inequality

welche im Vergleich zu etablierten Verfahren je nach Anwendungsfall um 6 bis 30 mal effizienter arbeiten (s. (GH03) und ebenfalls empfehlenswert ist (Fir08)). Dieses jedoch auf Kosten des Speicherbedarfs, weil jeder der verwendeten Landmarken alle Kosten zu allen anderen Knoten hält. Des weiteren werden auch hierarchische Strukturen zur Performanzsteigerung genutzt.

5.1.1 Hierarchische Strukturen

Für jede Abstraktion eines Graphen wird zusätzlicher Speicher benötigt. Von Ebene zu Ebene nimmt die Qualität des Graphen ab. Zunächst wird ein grober Weg geplant und iterativ immer feiner ausgearbeitet. Ziel der Abstraktion ist, dass viel weniger Knoten betrachtet werden müssen und somit positiv aus die Performanz auswirkt. Jedoch ist es auch möglich, dass durch zu komplexe Abstraktionen kein Performanzgewinn erreicht wird, weil dieser durch das Abarbeiten der Ebenen drauf geht. Des weiteren wird unter Umständen nicht der beste Weg gefunden.

Ein bekanntes vorgehen ist HPA* (Hierarchical Pathfinding A*), welches mehrere Raster zu einem abstrakten zusammenfässt. Existieren Übergänge zu benachbarten abstrakten Rastern, dann werden diese durch "Brücken" miteinander verbunden, welche als Knoten repräsentiert werden (s. Abb. 5.1). Anschließend werden alle Brücken-Knoten innerhalb eines abstrakten Rasters durch Kanten verbunden, wenn zwischen diesen ein Weg existiert. Dieses Verfahren ist für Rasterlandschaften ausgelegt und somit nicht weiter interessant. Des weiteren ist die Genauigkeit des abstrakten Graphen eher schlecht, wenn es sich um offenes Gelände handelt.

Wird mit einem Navigation-Mesh gearbeitet, dann können die konvexen Bereiche als Knoten dienen. Je nach Lage dieser können mehrere Zusammengefasst werden, um eine erneute Abstraktion zu bilden. Alle Bereiche in Sackgassen wie Räume lassen sich zusammenfassen und anschließend der Flur zwischen den Räumen. Danach lässt sch noch der Flur und alle Räume, welche an diesem liegen und Sackgassen sind zu einer weiteren Abstraktion vereinen. In offenen Landschaften, wo es eher selten Sackgassen gibt, können nah beieinanderliegende Gebiete gleichen Typs vereint werden. Der Aufwand im Vergleich zum Raster ist um einiges teurer. Dafür handelt es sich hier aber um eine einmalige Berechnung und benötigt während der Laufzeit viel weniger Arbeitsspeicher und wird daher in dieser Arbeit weiter verfolgt.

Handelt es sich um einen Wegpunkt-Graphen, dann können ähnlich wie beim Navigation-Mesh alle Knoten einer Sackgasse / eines Flures zusammengefasst werden.

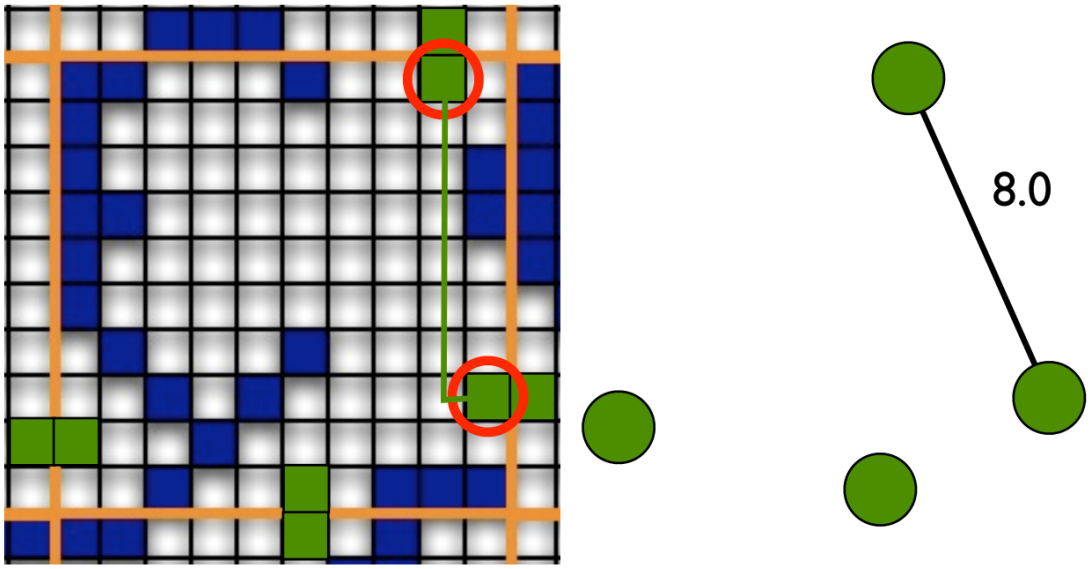

Abbildung 5.1: Übergänge zu benachbarten abstrakten Rastern werden gesetzt und verbunden (Har09)

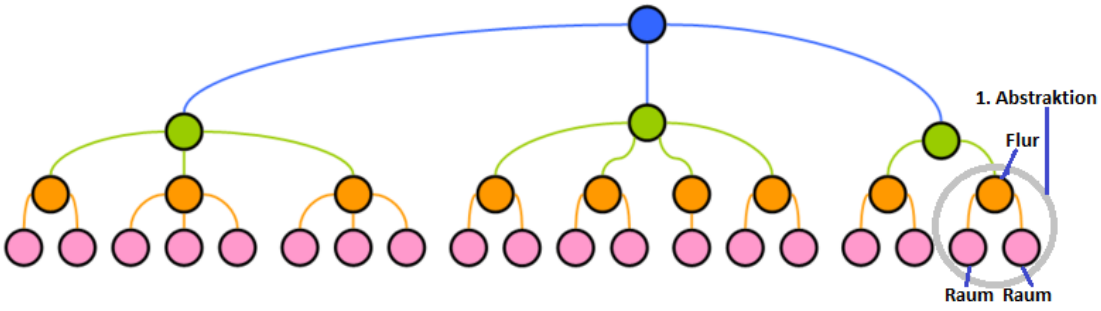

Abbildung 5.2: Graph mit 3 abstrakten Ebenen (weba)

5.2 Speicherbedarf

Arbeitsspeicher wird für den Graphen und während der Wegsuche benötigt, um schnellen Zugriff auf die Daten zu haben. Enthalten die Graphen mehrere Millionen Knoten, wie es bei großen Straßennetzen der Fall ist, dann müssen sich unter Umständen Teile des Graphen auf der Festplatte beziehungsweise dem Flashspeicher befinden. Bezüglich des Straßennetzes wird mit Landmarken gearbeitet, wobei jede Landmarke die Entfernung zu jedem Knoten hält. Aufgrund der Arbeitsspeicherbegrenzung werden nur die benötigten Landmarken in den Arbeitsspeicher geladen. In Spielen hingegen dürfen die Graphen nicht zu groß sein, weil die Berechnung der Weg nicht in wenigen Sekunden, sondern in wenigen Millisekunden erfolgen müssen.
Anders als im Straßenverkehr wird bei Spielen darauf geachtet, dass es Übergänge zwischen größeren Gebieten gibt. Hierbei wird ein hierarchischer Graph verwendet. Abstraktere Ebenen werden immer im Arbeitsspeicher gehalten und detaillierte Ebenen werden nur geladen, wenn der Spieler sich in diesen befindet oder wenn der gefundene Weg durch diese führt.
Neben den statischen Daten wie dem Graphen und eventuellen Landmarken oder "Look Up Tables" gibt es auch dynamische Daten wie die Open-List oder die Closed-List. Je nach Graphaufbau, Heuristik und Weglänge können diese gut befüllt sein. Um dieses zu verhindern, können dementsprechende Algorithmen verwendet werden. Im Folgendem werden zwei dieser vorgestellt.

- IDA* oder auch Iterative Deepening A* ist eine Variante der iterativen Tiefensuche. Hierbei handelt es sich um eine Kombination aus A* und DFS oder auch Depth-first search. Das Limit der Suche beträgt am Anfang mindestens die geschätzten Kosten und wird bei jeder Iteration erhöht. Übersteigen die Kosten des zurückgelegten und geschätzten Rest-Weges das Limit, dann wird dieser nicht expandiert. Durch dieses Vorgehen wird die durch die Heuristik eingeschränkte Breitensuche vom A* zu einer Tiefensuche, welche dank dem Limit keine endlose Suche durchführt. Die Closed-List kann gut befüllt sein, wobei die Open-List nicht benötigt wird. Bei A* kann die Open-List gut befüllt sein und ist aufgrund geforderter Sortierung bei jeder Änderung eine teure Operation, was beim IDA* eher nicht der Fall ist.

- RBFS oder auch Recursive Best-First Search beschränkt den Speicherplatzverbrauch linear zur Länge des Weges. Hierbei werden im Vergleich zum IDA* weniger Knoten besucht, diese jedoch öfters. Gestartet wird beim Startknoten und es werden alle Nachbarn expandiert. Der Kostengünstigste wird genommen und weiter expandiert. Sind alle Kosten der Nachbarknoten teurer als ein noch nicht expandierter Nachbarknoten des

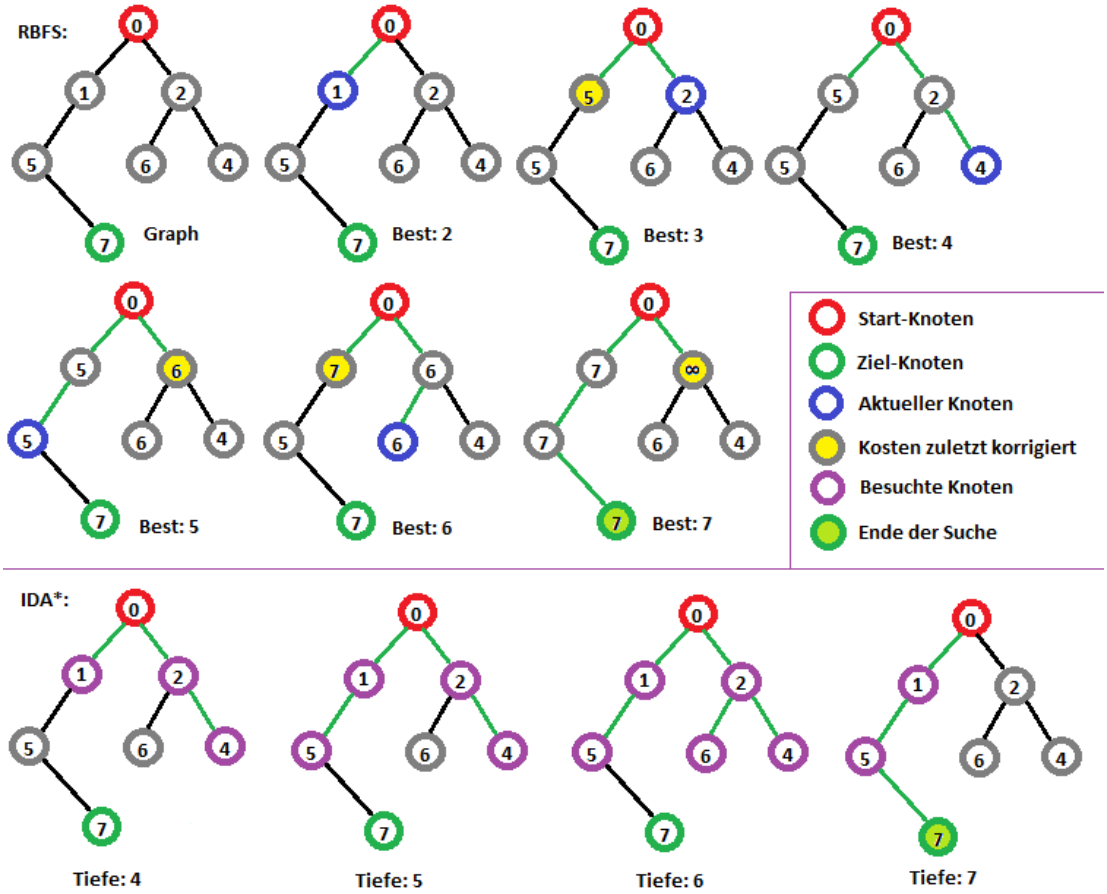

Abbildung 5.3: Beispiele für Recursive Best-First Search und Iterative Deepening A*, wobei RBFS den worst-case hat und IDA* bei der letzten Iteration gleich den optimalen Weg findet

Startknotens, dann werden die Kosten des aktuelle Knotens auf die Kosten des günstigsten Nachbarknotens gesetzt und alle Nachbarn vergessen. Es wird wieder zurück zum Startknoten gegangen und der kostengünstigste Nachbar expandiert. In diesem Fall ist nun ein anderer Nachbar der kostengünstigste und wird expandiert. So geht es die ganze Zeit weiter, bis der Zielknoten gefunden wurde. Hierbei kann ein Knoten sehr oft besucht werden, was auf die Performanz geht. Eine Closed-List ist nicht von Nöten und die Openlist enthält dank dem Vergessen besuchter Knoten nur wenige Elemente.

5.3 Qualität

Um die Qualität der Wegsuche messen zu können, muss diese erstmal definiert werden. Die Wegsuche ist qualitativ, wenn diese in kürzt möglichster Zeit einen optimalen Weg findet, wenn dieser existiert. Die benötigten Ressourcen wie der Arbeitsspeicherverbrauch sind Anforderungen der Wegfindung und nicht Teil der Qualität. Trotz dessen wird ein geringer Ressourcenbedarf angestrebt, was jedoch oft zur Verschlechterung der Qualität führt.
Meistens reicht ein quasi-optimaler Algorithmus aus, damit ein guter Weg ermittelt werden kann. Somit lassen sich hierarchische Graphen verwenden, weil der optimale Weg auf abstrakter Ebene nicht unbedingt dem optimalen Weg auf unterster Ebene entspricht. Die Ermittlung des Weges soll in kürzester Zeit von Statten gehen, was jedoch meistens nur mit hohem Speicherbedarfs umsetzbar ist.
Visuell ist Qualität durch den Ablauf des Weges erkennbar. Läuft der Spieler ohne jeglichen Grund Zickzack oder in Schlangenlinien, dann ist dieses gewiss nicht dem menschlichen Spieler nachempfunden und somit nicht qualitativ. Durch Glätten und Verkürzen von gefundenen Wegen, lässt sich ein akzeptables Wegablauf ermöglichen.

Kapitel 6

Ablauf des Weges

Nachdem ein Weg zum Ziel ermittelt wurde, wird dieser abgearbeitet. Hierbei ist es wie in Abschnitt 5.3 bereits erwähnt unsinnig diesen eins zu eins abzulaufen. Bei einem reinen Wegpunkt-Graphen ist dieses unmöglich, weil die Korrektur des Weges den ermittelten Weg zwangsläufig verlassen wird und ungültig sein kann. Damit eine Korrektur möglich ist, muss ein Bewegungsfreiraum gegeben sein, was bei Korridoren und beim Navigation-Mesh der Fall ist. Entweder wirken Potentialfelder und lassen den Spieler um die Ecken gleiten oder es werden zusätzliche Wegpunkte hinzugefügt. Für 3D-Welten bietet sich eher das Navigation-Mesh an, weil dort gerichtete Kanten gut umgesetzt werden können, was bei der Corridor Maps Method nicht der Fall ist.

6.1 Realitätsnahe Wege

Hier stellt sich die Frage, wie man selbst in einem Computerspiel vorgehen würde. Das Handeln sollte so gut wie immer zielgerichtet sein. Die Welt will erkundet werden, vor starken Gegnern wird geflohen, Gegenstände werden eingesammelt und die Schwachstellen der Gegner werden ausgenutzt. Je nach Spielertyp läuft man dafür gemütliche Wege und wartet auf Transportfahrzeuge oder nimmt steinige Wege für Abkürzungen in Kauf und springt mutig von hohen Klippen herunter.
Der Mensch nimmt die Spielumgebung mit den Augen wahr und ermittelt so den aktuellen Aufenthaltsort. Das Ziel der Begierde liegt in Sichtweite, es wird danach gesucht oder die Erinnerungen weisen einem den Weg. Ohne das es dem menschlichen Spieler meistens bewusst ist, wird der zurückzulegende Weg geplant.
Die KI ermittelt den Weg anhand gegebener Informationen. Ist ein Weg gefunden, dann wird dieser über das Navigation-Mesh zunächst begradigt und anschließend werden die Eckpunkte des Weges durch ein paar Wegpunkte ersetzt, welche diese etwas runder wirken lassen. Die Eck-

punkte eines Weges lassen sich aber auch durch Potenzialfelder abrunden, was in Abbildung 6.1 veranschaulicht wird. Hierbei muss auch sichergestellt werden, dass der modifizierte Weg nicht das Navigation-Mesh verlässt. Diese Korrekturen benötigen zusätzliche Rechenleistung, welche es gering zu halten gilt.

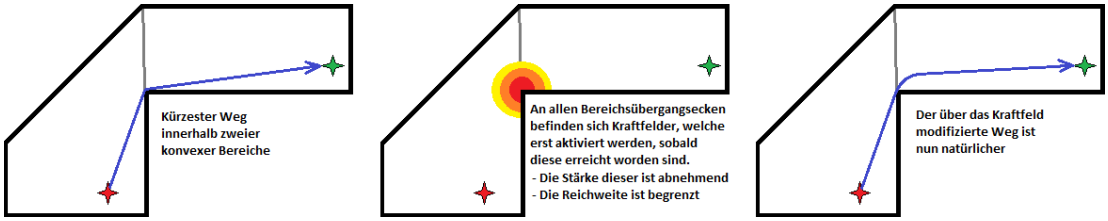

Abbildung 6.1: Durch Potentialfeld modifizierter Weg.

6.1.1 Corridor Maps Method

Unter anderem liefert die Corridor Maps Method gute Glättungen und somit geschwungene Wege und sollte daher besser betrachtet werden, damit ein allgemeines Verständnis gegeben ist. Dieses dient als Vergleich zum Navigation-Mesh obwohl es zu Anfang des Kapitels ausgeschlossen worden ist und wird unter diesem Kapitel aufgeführt, weil diese Methodik den Wegablauf ziemlich gut unterstützt.

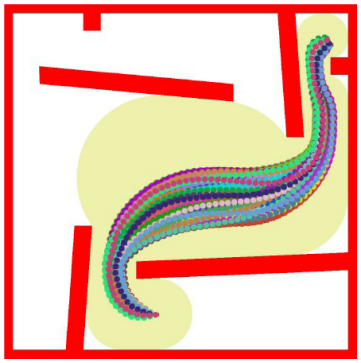

Abbildung 6.2: Hundert zufällige Wege

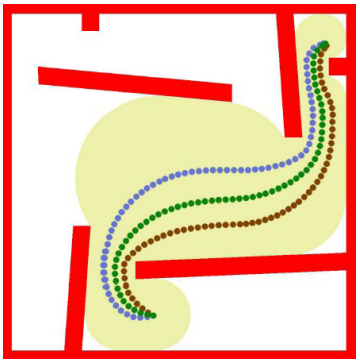

Abbildung 6.3: Mögliche Weganordnungen

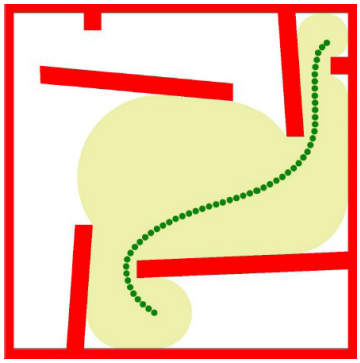

Abbildung 6.4: Kürzester Weg (Egg08)

Wie bereits in Unterabschnitt 4.1.2 erwähnt wird der begehbare Bereich durch die Fläche mit und zwischen zweier Kreise definiert. Alle Akteure sind als Punkt mit einem Radius defi-

niert und werden durch die Potentialfelder der Knoten angezogen. Wenn um Ecken gegangen werden soll, dann werden die Akteure dort rumgezogen, wobei der am nächsten am Ziel befindende Knoten die stärkste Anziehungskraft ausstrahlt. Je nach prozentualer Gewichtung ist der Weg mal mehr und mal weniger geschwungen, was in Abbildungen 6.2 gut zu erkennen ist. Dynamischen Hindernissen, wie andere Spieler, wird ausgewichen, indem diese eine abstoßende Kraft aussenden, welche immer stärker wird. Kurz bevor eine Kollision zweier Spieler überhaupt passieren kann, sind die abstoßenden Kräfte jedes Spielers so hoch, dass diese entweder unwahrscheinlich aneinander prallen und zum Stillstand kommen oder aneinander vorbei gleiten. Hierbei können die Akteure unterschiedliche Radien besitzen.

In 3D-Spielen ist CMM interessant, wenn es sich bei dem Terrain eher um Korridore handelt. Entweder ist die Umgebung so aufgebaut oder passive NPCs wie Fußgänger wird nur ein solcher Bereich in einer offenen Landschaft zugesprochen. Für Straßen ist es jedoch nicht zu empfehlen, weil die Autos auf deren Spur fahren sollen und nicht irgendwo auf der Straße. In Süßwasser-Gewässern für Wasserlebewesen und Bote ist dieses schon eher passend.

Eine 100%ige Abdeckung des begehbaren Bereiches ist mit vielen Knoten möglich, wobei die Anzahl der Knoten größer ausfallen kann, als die Anzahl aller Punkte der Umgebung. Die Abbildung 6.5 zeigt die Komplexität der Corridor Maps Method im Vergleich zum Navigation-Mesh.

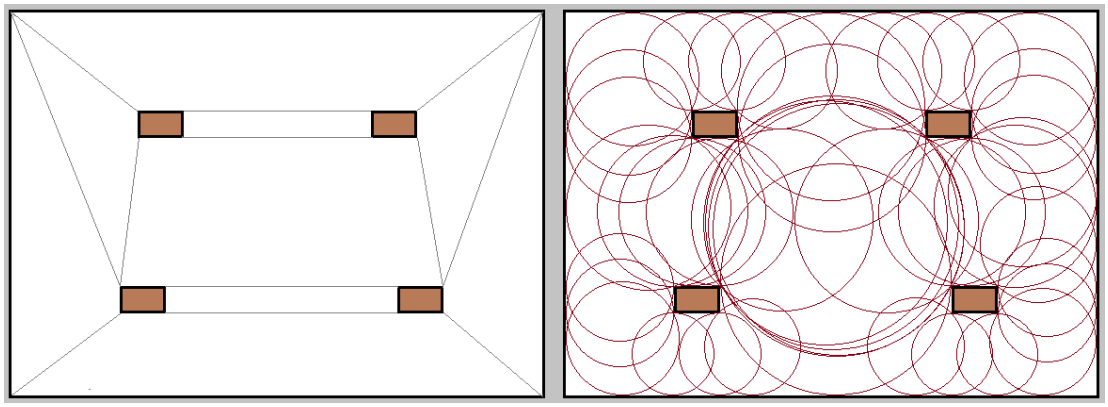

Abbildung 6.5: Links sind die konvexen Bereiche eines Navigation-Meshes und rechts alle Knoten der Corridor Maps Method

Für Spiele, in denen sich der Spieler frei bewegen kann, es mehrere Ebene gibt, über Seile gegangen werden kann oder der Spieler nicht als Kreis dargestellt werden soll, ist die CMM nicht zu empfehlen. Ein Seil hat einen kleineren Durchmesser als der NPC und somit könnte das Seil diesen nicht aufnehmen. Nach dem Motto entweder alles oder nichts ist dieses auch bei Felswänden so. Der menschliche Spieler kann zum Teil über den Felsrand ragen, wenn der Gebirgspfad zu eng ist und fällt nicht runter. Bei niedrigen Brücken kann mit einem Navigation-Mesh ein Fallenlassen-Übergang zu der darunterliegenden Ebene leicht erstellt werden, wobei eine solche Umsetzung beim CMM undenkbar ist.

6.2 Verwendung zusätzlicher Potentialfelder

Potentialfelder wirken anziehend oder abstoßend und können innerhalb von Bereichen angewendet werden. Des weiteren können Bereiche des Navigation-Meshes mit einem Gefahren- oder Begehren-Multiplikator belegt werden. Handelt es sich um größere Bereiche und nur ein Teilbereich davon ist zu betroffen, dann könnte dieser mit einem Potentialfeld versehen werden.
Hierdurch wird folgendes erreicht:

- Als sicher geltende Wege werden statt dem kürzesten Weg bevorzugt
- Mitglieder des eigenen Teams suchen Anschluss, um kein leichtes Opfer darzustellen und um die gebündelte Angriffsstärke zu erhöhen
- Schwache feindliche Truppen werden bevorzugt angegriffen
- Starke oder eine Vielzahl von Gegnern werden gemieden

Die genannten Errungenschaften erweitern den Graphen um Informationen, damit der Spieler je nach Kanten-Gewichtung einen anderen optimalen Weg einschlagen kann. Beim Navigationsgerät ist dem so ähnlich, jedoch wird hier nach dem kürzesten oder Sprit-sparensten Weg gesucht.

Kapitel 7

Konkrete Umsetzung

In den letzten drei Kapiteln wurde bereits genauer beschrieben, welches Methodiken es gibt, um ein Wegfindungssystem umzusetzen zu können. In diesem Kapitel werden die priorisierten Methodiken gewählt und in Verbindung der Implementierung mit Java genauer beschrieben. Am Ende dieses Kapitels wird dann der resultierende Prototyp vorgestellt.

7.1 Technische Informationen

Die Wahl ist auf die Programmiersprache Java gefallen, weil die Umsetzung hauptsächlich auf einem Android-Tablet statt findet und ebenfalls auf einem Heim-PC für komplexere Entwicklungsphasen zur Verfügung stehen soll. Des weiteren ist Android auf Java ausgelegt und bietet eine einfache Schnittstelle zu OpenGL[1] an, welche für die grafische Ausgabe verwendet wird. Eine grafische Ausgabe ist für spätere Präsentationen sinnvoll und für eine halbwegs schnelle und aussagekräftige Validierung während der Umsetzung notwendig. Des weiteren werden nicht alle Androidgeräte unterstützt, weil nicht jede Grafikkarte mit dem importierten Bildmaterial umgehen kann.
Die Wahl ist auf das Gebiet Warsong-Schlucht von World of Warcraft gefallen, weil dieses für die Wegfindung sehr interessant ist. Es existiert offenes Gelände, Tunnel, Klippen und mehrere Etagen innerhalb der Gebäude. Die Spieler können springen und sich von Klippen oder höheren Etagen hinunterstürzen und genießen die Wahlfreiheit mehrere Wege nutzen zu können, welche zum Teil sehr beschränkt oder auch recht offen sind. Die Beschränkung des Spielfeldes wird durch unerklimmbare Klippen gut beschränkt, damit ein übersichtliches Szenario gegeben ist.

[1] OpenGL: Open Graphics Library. Eine Konkurrenz zu Microsofts Direct3D, welche meistens eine bessere Treiberunterstützung genießt, jedoch nicht auf den Spielekonsolen Microsofts läuft (webb).

7.2 Erstellung des Navigation-Mesh

Zunächst werden die geographischen Informationen benötigt, welche durch einen eigens geschriebenen Parser aus den benötigten "World of Warcraft"-Dateien eingelesen werden. Angefangen bei den Dateien für die Landschaft bis hin zu den dort platzieterten Objekten. Diese werden anschließend gut leserlich als Java-Klassen angeboten.

Die Daten werden nun verarbeitet. Alle relativen Koordinaten werden per Matrizen in absolute Koordinaten umgewandelt. Diese werden zur Generierung der Dreiecke verwendet. Die Klasse "Dreieck" besteht somit aus drei Positions-Punkten und einem Up-Vektor, welcher aus den Positionspunkten erstellt wird. Der Up-Vektor wird in den nachfolgenden Berechnungen noch häufig benötigt und wird somit jetzt schon vorberechnet. Die Klasse "Dreieck" hat zusätzlich noch bis zu drei Nachbardreiecke. Ein Nachbardreieck ist durch zwei gemeinsame Punkte mit dem Dreieck definiert. Diese und folgende Berechnungen bezüglich der Dreiecke werden auf einem Heim-PC ausgeführt, weil dieser über mehr Leistung verfügt, was bei über einer Millionen von Dreiecken zu Beginn ratsam ist.

Nun werden alle Dreiecke mit positivem Up-Vektor ausgewählt, weil diese begehbar oder herunterrutschbar sind. Diese werden nun mit dem Spielervolumen geflutet. Kann sich die Spielfigur auf dem kompletten Dreieck aufhalten, dann werden keine Unterteilungen benötigt. Ist dem nicht der Fall, dann muss das zu untersuchende Dreieck in mehrere kleinere Dreiecke zerlegt werden. Bei diesen müssen auch die Nachbardreiecke korrekt gesetzt werden. Als Resultat dieser Prozedur wird eine Sammlung von Dreiecken zurückgegeben, welche das Spielervolumen aufnehmen kann.

Der Up-Vektor repräsentiert die Steigung, um welche es im Folgendem geht. Alle Dreiecke mit starkem Up-Vektor sind in alle Richtungen begehbar und die mit schwachem können in eine Richtung heruntergerutscht werden. Diese Aufteilung wird vorgenommen und anschließend werden die in alle Richtung begehbaren Dreiecke zu konvexen Bereichen verbunden, welche gemeinsam das Navigation-Mesh bilden. Die herunterrutschbaren Dreiecke dienen als Übergänge zweier konvexer Bereiche. Hierbei kann es vorkommen, dass die herunterrutschbaren Dreiecke zerlegt werden müssen. Diese Übergänge zwischen zweier konvexer Bereiche über herunterrutschbare Dreiecke werden bei der Betrachtung der Fähigkeiten weiter betrachtet.

Damit die nächste Schritte erfolgreich sind, muss zunächst der Graph gebaut und der A*-Algorithmus implementiert werden. Alle Eckpunkte eines jeden Bereiches werden durch zwei Kanten verbunden, wobei die Kosten die Distanz der beiden Knoten ist. Die Knoten, welche sich die Bereiche teilen, werden zu einem Knoten zusammengefasst. Hierdurch entsteht ein großer gerichteter Graph. Der A*-Algorithmus wird hierfür Lehrbuch-konform umgesetzt.

Alle Spielertypen können standardmäßig gleich schnell laufen und springen identisch hoch und weit. Um diese Fähigkeiten mit einzubeziehen wird durch alle Bereiche iteriert und diese mit den Funktionen verrechnet. Als Input dient der jeweilige Bereich, alle anderen Bereiche und die Menge aller nicht begehbaren und herunterrutschbaren Dreiecke. Wenn Übergänge vom Bereich zu einem anderen Bereich kostengünstiger möglich sind, als es zur Zeit der Fall ist, dann werden in beiden Bereichen zwei Punkte festgelegt. Diese Linien sind als Start- oder Ziellinie definiert, welche zusätzlich an den Punkten Richtungsvekoren besitzen, welche die Blickrichtung einschränken. Ebenso wird die Fähigkeit als Bedingung der Benutzbarkeit festgelegt. Dieses wurde bereits in der Abbildung 4.10 gezeigt.
Jeder Spielertyp hat ebenso seine eigenen Fähigkeiten, mit welchen genau so wie denen der Standardfähigkeiten verfahren wird. Die entstehende Startlinie enthält somit als Bedingung die Spielertyp-Fähigkeit.

Am Ende enthält das Navigation-Mesh mehrere konvexe Bereiche, wovon manche Bereiche Linien für fähigkeitsbasierte Übergänge besitzen können. Der existierende Graph wird um die Knoten dieser Linien erweitert.
Weil die Fähigkeiten Cooldown-basiert sind, kann es vorkommen, dass eine Fähigkeit noch nicht bereit ist. Bei der Suche auf dem Graphen muss daher beachtet werden, dass nach ein paar Metern die Fähigkeit bereit ist und genutzt werden kann. Bei der Implementierung wird nicht auf Fähigkeiten gewartet, bevor der Weg beschritten wird, weil dadurch unter anderem weniger Aktion im Spiel ist. Dabei kann es aber vorkommen, dass bei einer nachfolgenden Wegsuche ein anderer Weg eingeschlagen wird, weil eine nützliche Fähigkeit benutzbar geworden ist.

7.3 Erstellung von Hierarchien

Weil es während der Laufzeit auf jede Millisekunde ankommt, wurde ein eigenes Hierarchisches-Modell entwickelt.
Die ALT-Algorithmen, welche auf Landmarken spezialisiert sind, werden nicht genommen,

weil das Landmarken-Setz-Problem existiert und die Wege starr vorberechnet werden. Hier sind jedoch fähigkeitsbasierte Teilwege möglich, wenn die cooldownbasierte Fähigkeit bereit steht. Des weiteren müssen die Kosten der Wege bezüglich Gefahr und Sicherheit modifiziert werden können.

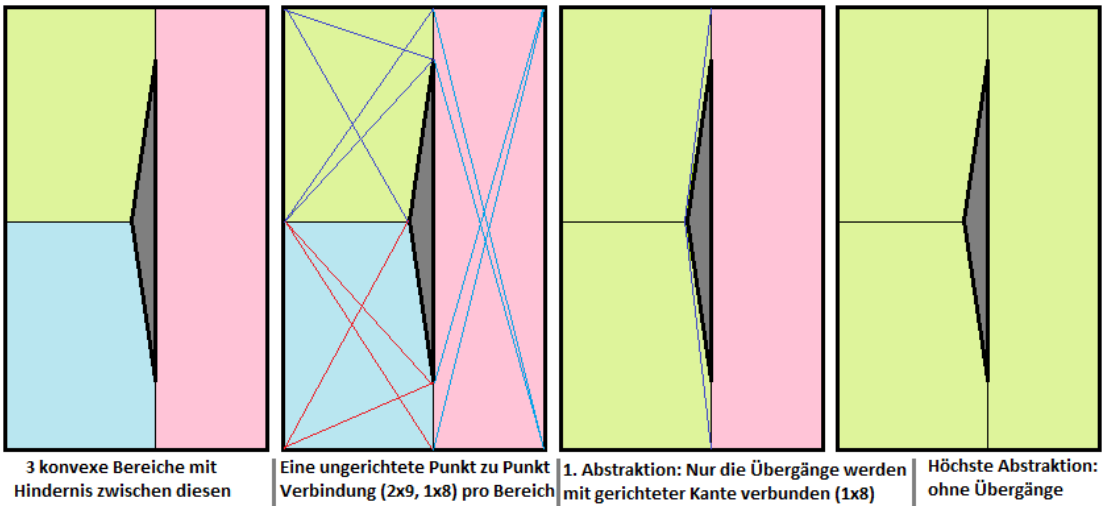

Abbildung 7.1: Merge Prozess und verwendete Kanten pro Ebene

Für das eigene System werden zunächst die Hierarchien erzeugt, indem ein Merge-Prozess verwendet wird (s. Abb. 7.1), bei welchem jeder Bereich mit denen der Nachbarbereiche verglichen wird. Die zwei günstigsten Bereiche werden dann rekursiv zusammengefügt, bis sich keine Bereiche mehr zusammenfügen lassen. Zum ermitteln der zwei günstigsten Bereiche werden zwei Kriterien beachtet, welche zum einen die maximalen Kosten und zum anderen die Anzahl der neuen Nachbarn sind.

Die maximalen Kosten können durch den A*-Algorithmus ermittelt werden, indem die Wege von den Knoten des einen Bereiches zu jedem Knoten des anderen Bereiches ermittelt werden. Dieses muss beidseitig geschehen, weil es sich um einen gerichteten Graphen handelt. Als Resultat wird dann den zwei Bereichen die Kosten des teuersten Weges zugewiesen. Hierbei handelt es sich auf unterster Ebene um nur einen Bereichsübergang über welchen die Wege geführt werden und daher kann die Wegermittlung in zwei Teile geteilt werden, um Rechenleistung einzusparen. Einmal vom Start zu den Übergangsknoten und einem von den Übergangsknoten zu den Endknoten. Bei abstrakteren Ebenen muss es sich nicht nur um einen Übergang handeln, sondern kann beliebig viele besitzen. In diesem Fall müssen alle Übergänge

beachtet werden.
Die Anzahl der neuen Nachbarn ist ebenso ausschlaggebend, weil dadurch vermieden wird, dass auf abstrakterer Ebene zwei Gebiete mit unzähligen Übergängen untereinander existieren können. Je mehr Übergänge existieren, desto mehr Informationen werden benötigt, was sich wiederum negativ auf den benötigten Arbeitsspeicher auswirkt. Die Informationen sind die Entfernungen aller Bereichsübergangsknoten untereinander. Dieses ermöglicht, dass die Kosten der Wegteile, welche durch abstrakte Bereiche führen ohne dort zu starten oder zu enden, sofort gegeben sind und wertvolle Rechenzeit einsparen.

Es entsteht eine Baumstruktur, welche optimal eine Wurzel hat. Ist dieses nicht der Fall, dann handelt es sich um Teilbereiche, welche entweder nicht oder nur durch eine gerichtete Kante verbunden sind. Jeder Knoten besitzt zwei Unterknoten, welche in beliebiger Tiefe zu Blättern enden. Weist das Gelände eine regelmäßige Struktur auf, dann ist damit zu rechnen, dass alle Blätter ungefähr in der gleichen Baumtiefe vorhanden sind.

Damit die Suche den optimalen Weg trotz verwendeter Hierarchien finden kann, darf nicht nur innerhalb des abstrakten gemeinsamen Knotens von Start und Ziel gesucht werden, womit sich nun auch der nächste Abschnitt befasst.

7.4 Erstellung der endgültigen Suche

Hier geht es um die Performanz. Es entstehen Kosten für das Verknüpfen der Start- und Zielposition als Knoten an den Graphen. Damit diese Kosten gespart werden, werden alle Spieler und Zielobjekte mit dem Graphen verknüpft. Die Position der Spieler ändert sich stetig und muss daher beim Graphen aktualisiert werden. Diese Aktualisierung ist kostengünstiger, weil die neue Position in der Nähe der alten Position auf dem Navigation-Mesh liegen muss. Somit wird das Wegfindungsproblem um das des Verknüpfens reduziert, wodurch die Suche über die Start- und Zielobjekte die Knotenpunkt des Graphen schnell ermitteln kann.

Wie in Abschnitt 7.3 bereits beschrieben wird mit Hierarchien gearbeitet. Bei der Suche wird zunächst vom Start- und Zielknoten nach oben gegangen, bis diese sich den gleichen Knoten in einer hierarchischen Ebenen teilen. Ist dieses nicht der Fall, dann ist der Weg zwischen den beiden Knoten nur in eine Richtung möglich oder es existiert gar kein Weg, was sich auf abstrakter Ebene schnell ermitteln lässt. Teilen sich der Start- und Zielknoten einen abstrakten

Knoten, dann kann nicht davon ausgegangen werden, dass der optimale Weg nur innerhalb des abstrakten Knotens liegt. Aus diesem Grund müssen auch die Nachbarknoten des abstrakten Knotens genauer untersucht werden, indem sich der Suchalgorithmus Ebene für Ebene nach unten durchhangelt. Bei diesem Vorgehen werden auf jeder Ebene ermittelte abstrakte mögliche Wege aussortiert und neue der nicht aussortierten abstrakteren Wege ermittelt, welche auf der nächst niedrigeren Ebene untersucht werden.
In der Abbildung 7.2 wird ein kleines Beispiel dargestellt.

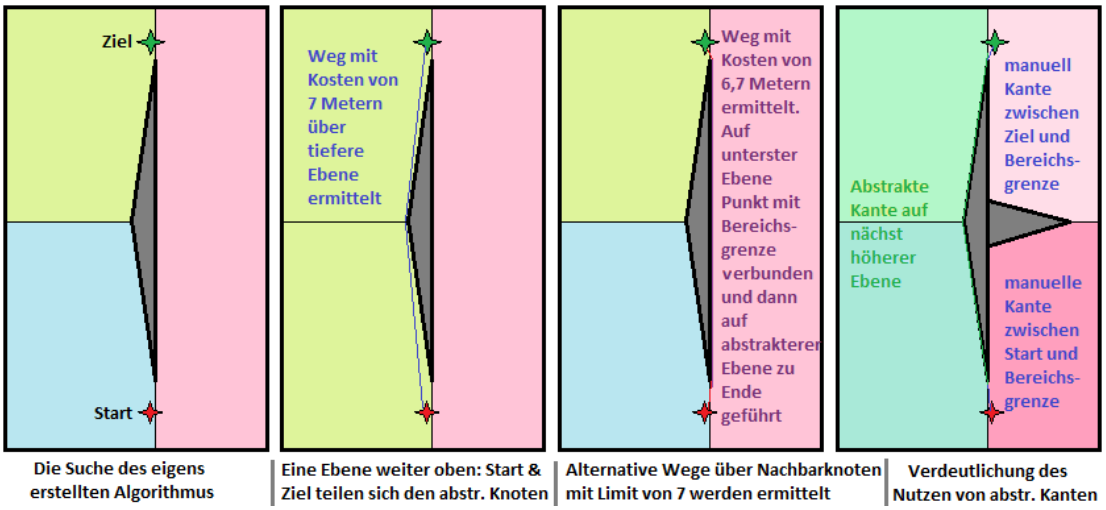

Abbildung 7.2: Merge Prozess und verwendete Kanten pro Ebene

Wird dieses Vorgehen mit dem A*-Algorithmus verglichen, welcher bei der Heuristik nicht überschätzen darf, dann ist zu erkennen, dass hier genau das Gegenteil gemacht wird. Durch diese komplett andere Vorgehensweise wird kein "worst-case" Verhalten erkennbar sein, es wird immer der optimale Weg gefunden, eine Teilparallelisierung ist möglich und auch die Kosten können zur Laufzeit performant modifiziert werden. Diese Vorteile bringen als Nachteil einen erhöhten Arbeitsspeicherverbrauch mit sich, welcher sich aber in Grenzen halten sollte, was in Kapitel 8 zu beweisen ist.

7.5 Korrekturen des ermittelten Weges

Die Korrekturen sind zum einen die Wegverkürzung und die Glättung der Ecken. Beide sind anhand der Informationen des begehbaren Gebietes möglich, welcher durch die konvexen

Bereiche des Navigation-Mesh gegeben sind.

Da mit einem Navigation-Mesh gearbeitet wird, werden die Wegverkürzungen über einen "funnel"-Algorithmus durchgeführt, welcher als Input die Bereiche benötigt, durch welche der gefundene Weg läuft. Bereich für Bereich wird sich nun durchgehangelt und der gefundene Weg verkürzt. Als Resultat führt dann der Weg entweder direkt vom Start- bis zum Zielknoten oder führt an beliebig vielen Bereichsecken vorbei, damit nicht durch Hindernisse durchgelaufen wird.

Im Folgendem wird der Ablauf eines Funnel-Algorithmus gezeigt. Abbildung 7.3 zeigt den via Graph ermittelten optimalen Weg und die folgenden Abbildungen (Abb. 7.4 bis Abb. 7.14 (Mil06)) zeigen den Berechnungsablauf des Algorithmus.

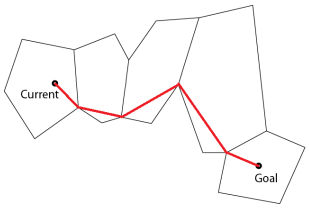

Abbildung 7.3: Ermittelter kürzester Weg

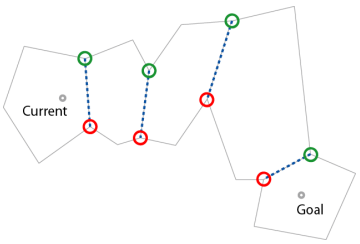

Abbildung 7.4: Portale zwischen Bereichen

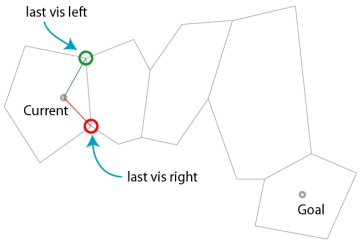

Abbildung 7.5: Start des Funnel-Algorithmus

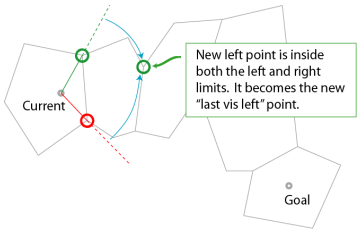

Abbildung 7.6: 1. Iteration

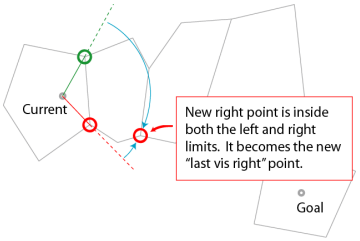

Abbildung 7.7: 1. Iteration

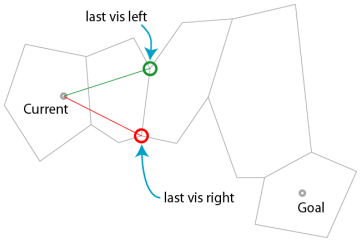

Abbildung 7.8: 1. Iteration

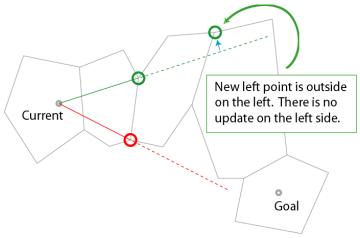

Abbildung 7.9: 2. Iteration

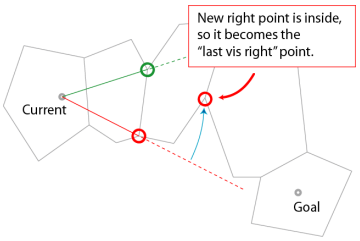

Abbildung 7.10: 2. Iteration

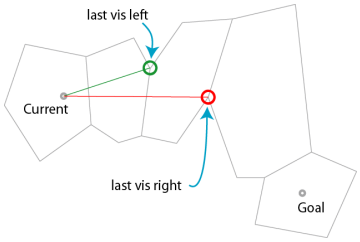

Abbildung 7.11: 2. Iteration

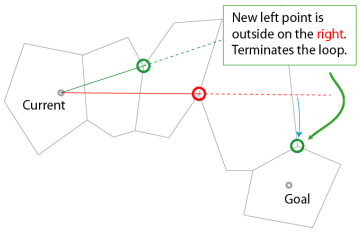

Abbildung 7.12: 3. Iteration

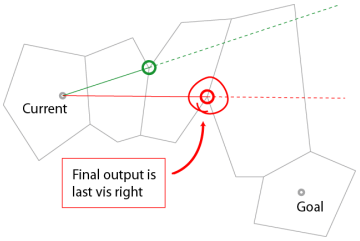

Abbildung 7.13: 3. Iteration

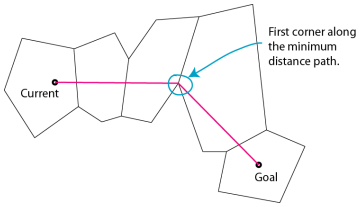

Abbildung 7.14: ..., Ziel

Nun zu den Glättungen der Ecken, welches durch Potentialfelder geschieht. Beim Navigation-Mesh müssen nur die Ecken betrachtet werden, welche an mindestens zwei konvexen Bereichen angrenzen. Wenn nach der Wegverkürzung über einen solchen Knoten gegangen wird, dann läuft die Glättung wie folgt ab. Der NPC wird vom vorherigen Knoten zum Eckknoten angezogen. Wird dieser erreicht, dann wird der Richtungsvektor zu einer durch einen Radius definierten Länge des Eckpunktes abschwächend fortgeführt, während zusätzlich die Anziehungskraft des nachfolgenden Knotens wirkt. Befindet sich der nachfolgende Knoten innerhalb des durch den Radius definierten Kreises, dann wird der Radius für diese Berechnung auf den Abstand der Punkte heruntergesetzt. Die Abschwächung des ersten Richtungsvektors bis hin zum zweiten kann durch eine beliebige Funktion umgesetzt werden, welche die erste Kraft vorerst leicht und dann immer stärker ausbremst.

Bei der Wegsuche spielen die zusätzlichen Wegekosten bezüglich der Gefahr oder der Sicherheit eine Rolle.
Beim Wegablauf hingegen wird auf die in Potentialfelder der Spieler geachtet. Hierbei ist nicht die Luftlinie als Abschwächung zu nehmen, sondern der tatsächliche Weg zwischen diesen. Hierbei ist aufgrund des gerichteten Graphen eine Suche in beide Richtungen von Nöten, welcher durch ein Limit beschränkt wird. Ist das Limit erreicht, dann wird das Potentialfeld nicht weiter beachtet. Die Potentialfelder modifizieren den Bewegungsvektor des Spieler leicht,

wobei darauf zu achten ist, dass das Navigation-Mesh nicht verlassen wird und der nächste geplante Wegpunkt über einen leichten Umweg erreicht werden kann. Da alle 0,25 Sekunden eine erneute Wegsuche gestartet wird, kann es vorkommen, dass sich aufgrund des Umweges ein kostengünstiger Weg finden lässt.
Damit die Spieler jeweils individuell agieren können, erhalten diese Variablen, welche die Stärke der Potentialfelder via Faktorisierung modifizieren.

7.6 Der Prototyp

Der Prototyp besteht aus dem Wegfindungssystem, welches in diesem Kapitel beschrieben wurde und stellt die eigene hierarchische und den A*-Algorithmus zur Verfügung, zwischen welchen gewechselt werden kann.
Das verwendete CTF[2]-Szenario ermöglicht einen automatisierten Ablauf und liest zu Beginn eine Property-Datei ein, in welcher jeder einzelne Spieler bezüglich Risikobereitschaft und Aufgabenbereich individuell konfiguriert werden kann. Während der Laufzeit kann durch die Spieler gewechselt werden, um Ausgaben für die jeweiligen Spieler zu erhalten, deren Attribute nachträglich zu modifizieren oder dass die Kamera auf diesem platziert wird. Ansonsten lässt sich die Kamera auch frei in der Welt positionieren und verschieben.
Geloggt werden können die Dauer der Wegberechnung und der Speicherbedarf des jeweiligen Wegfindungssystems bezüglich des Optimums, des Durchschnitts und des katastrophalsten Ergebnisses, damit diese vergleichbar und bewertbar sind.

Abbildung 7.15: Vogelperspektive

Abbildung 7.16: Flaggenraub

[2]CTF: Capture the Flag

Aufgrund des Umfanges wird nur das CTF-Szenario auf dem Androidgerät (grafisch) umgesetzt. Eine technische Bewertung wird aufgrund der getätigten Ausgaben ermöglicht.

7.7 Hindernisse bei der Umsetzung

- Ungenauigkeit bei der Gleitkommazahl "Float"
  Alle geografischen Punkte der Highmap des Landschaftsbodens resultieren zu gebogenen Platten. Weil diese Platten unterschiedlich verschoben werden, um zur Nachbarplatte zu passen, kommt es ab und zu vor, dass die nebeneinander liegenden Platten sich nicht zu 100% berühren. Der Unterschied beträgt maximal ein Millimeter, wobei die Einheiten in Metern angegeben werden. Durch diesen Unterschied berühren sich die benachbarten Dreiecke nicht und werden somit nicht als Nachbardreiecke erkannt. Um den begehbaren Bereich zu ermitteln, muss dieses jedoch der Fall sein und somit habe ich Punkte auf ein paar wenige Nachkommastellen gerundet, damit Nachbardreiecke erkannt werden können.
  Wegen der Ungenauigkeit der Gleitkommazahl muss auch beim Bilden von konvexen Bereichen darauf geachtet werden. Somit sind die konvexen Bereiche nur quasi-konvex, weil minimale Innenwölbungen toleriert werden.

- Langsame Speicherverwaltung bei Android.
  Java übernimmt die Speicherverwaltung der Objekte, welche angefordert bzw. nicht mehr benötigt werden. Die "Garbage Collection" kümmert sich darum Speicher wieder frei zu geben. Ziel ist es den Arbeitsspeicherbedarf des aktuellen Programmes zu reduzieren. In Android steht jeder App standardmäßig nur wenig Arbeitsspeicher zu und beträgt je nach maximalen Arbeitsspeicher 16 oder 24 Megabyte. Wird mehr benötigt, dann kann der Programmierer das Flag "largeHeap=true" in der App-Konfiguration setzen. Dieses ist bei grafischen Anwendungen nur bedingt nötig, weil alle grafischen Objekte und Bilder via OpenGL extern, also außerhalb von Java, gehandhabt werden und der benötigte Speicherplatz von Java nicht verwaltet werden muss.
  Probleme mit dem GC habe ich bekommen, als die Dreiecke, die Kollisionsboxen, die konvexen Bereiche und die hierarchischen Ebenen geladen werden. Ungefähr alle 6 Sekunden wird die App-Ausführung für 1,5 Sekunden lahmgelegt, weil auf den GC gewartet werden muss. Dieses ist der Fall, weil der GC von Android immer nur ein wenig Arbeitsspeicher mehr bereitstellt und immer wieder versucht den maximalen Arbeitsspeicher zu reduzieren, obwohl dieser ungenutzte Speicher in Kürze wieder verwendet wird.

Leider lässt sich bei Android kein minimaler Arbeitsspeicherbedarf festlegen, wie es zum Beispiel bei Heim-PC's der Fall ist.
Um dieses Problem zu reduzieren musste der Code so umgeschrieben werden, dass Objekte nicht immer wieder neu erstellt werden, sondern in Recycling-Pools gehalten werden. Aus diesen werden die Objekte genommen und mit neuen Werten versehen. Nach dieser Umsetzung ist zu erkennen, dass die Ruckler nach dem Setzen des Flags "largeHeap=true" während der Laufzeit stark zurückgegangen sind. Beim initialen Laden ist dieses jedoch kaum zu vermeiden.
In diesem Zuge hat sich ein klares Bild zwischen Heim-PC und Android-Gerät herauskristallisiert. Beim Heim-PC werden die kostenintensiven Berechnungen zum Verbinden der Dreiecke, Erstellen der Kollisionsboxen, Bilden der konvexen Bereiche und der hierarchischen Ebenen im Wegwerfstil programmiert bei denen mit riesigen HashSets, HashMaps, TreeMaps und Array-Listen gearbeitet wird, um in jeweils nur wenigen Sekunden mit hundert-tausenden Objekten jonglieren zu können. Beim Androidgerät hingegen muss mit eigener Recycling-Verwaltung gearbeitet werden. Alleine bei der Umwandlung von Zahlen zu Texten werden unnötig viele Objekte neu erstellt, was effektiv unterbunden werden konnte. Über Pools und dem hin und her switchen von Buffern wird zwar mehr Speicherplatz belegt, jedoch wird der GC entlastet, was sich bezüglich der Performanz bemerkbar macht.

- Doppelter Boden.
  Zum Teil sind Landschaftsobjekte wie Häuser dem Boden so dicht, dass diese nach der Rundung der geografischen Punkte identisch sind. Hierdurch konnten Dreiecke als Nachbarn erkannt werden, wodurch ungewöhnliche Szenarien ermöglicht wurden. Zwei Dreiecke sind benachbart, wenn diese zwei identische Punkte besitzen. Zieht man eine Linie zwischen den identischen Punkten, dann liegen die zwei übrigen Punkte auf verschiedenen Seiten; einer links und einer rechts von der Linie. Nun ist es jedoch auch möglich, dass sich die zwei nicht identischen Punkte auf der gleichen Seite der Linie befinden, was bei der Berechnung der konvexen Bereiche zunächst nicht beachtet worden ist. Nach der Identifizierung dieses Verhaltens wurde die Berechnung umgeschrieben, um ein akzeptables Resultat zu erhalten.

Algorithm 7.1 $Vorschau : Hierarchischer - Algorithmus$

Require: $start : Startknoten, startGebiet, ziel : Zielknoten, zielGebiet$
Ensure: $Q : open\ list, H : predecessors$

```
while startGebiet ≠ zielGebiet do
  if startGebiet.tiefe > zielGebiet.tiefe then
    startGebiet ← startGebiet.parent
  else if zielGebiet.tiefe > startGebiet.tiefe then
    zielGebiet ← zielGebiet.parent
  else
    startGebiet ← startGebiet.parent
    zielGebiet ← zielGebiet.parent
  end if
end while

limit ← lookUpLimitInsideHierarchy(startGebiet, zielGebiet, start, ziel)
if limit < 0 then
  return Kein Weg gefunden
end if

return searchPath(startGebiet, zielGebiet, start, ziel, limit)
```

```
function lookUpLimitInsideHierarchy(startGebiet, zielGebiet, start, ziel)
    startGebiet ← startGebiet.deepBackwards
    zielGebiet ← zielGebiet.deepBackwards
    limit ← MAX
    if zielgebiet ∈ startGebiet.nachbarschaft then
      for each portal = getPortal(startGebiet, zielGebiet) do
            pathCost ← lookUpLimitInsideHierarchy(startGebiet, zielGebiet, start, portal)+
            lookUpLimitInsideHierarchy(startGebiet, zielGebiet, portal, ziel)
            if pathCost < limit then
                limit ← pathCost
            end if
      end for
    else if zielgebiet ∈ startGebiet.hierarchischerNachbar.nachbarschaft then
      Wie erstes inklusive einer Verbindung zwischen zwei anstelle einem Portal
    else if zielgebiet = startGebiet.hNachbar.nachbarschaft.hNachbar then
      Wie erstes inklusive zweier Verbindung zwischen drei anstelle einem Portal
    end if
    return limit
end function
```

```
function searchPath(startGebiet, zielGebiet, start, ziel, limit)
    Wie die Limitberechnung nur das auch die Nachbarn mit beachtet werden
    LookUp geschiet auf abstrakter Ebene. Nur falls der Wert < limit
      dann verbinde mit startGebiet und zielGebiet sowie start und ziel
end function
```

Kapitel 8

Test

In diesem Kapitel wird der Prototyp unter die Lupe genommen. Hierbei wird unter anderem der eigene mit dem A*-Algorithmus verglichen. Hauptmerkmale sind Speicherverbrauch und Performanz so wie die Anzahl der untersuchten Knoten. Alle Tests werden auf dem Android-Gerät ausgeführt, welche über das Interface des Prototyps zum Teil gesteuert und ausgewertet werden können. Die Auswertung findet über Textausgaben statt und bietet zur Laufzeit eine visuelle Repräsentation im 3D-Raum. Die Ergebnisse des Wegfindungssystems sowie des Spielverhaltens werden analysiert und hinsichtlich mehrerer Kriterien bewertet.

8.1 Testbeschreibung

Damit ein Vergleich der Wegfindungsverfahren möglich ist, könnten viele zufällige Suchen ausgeführt werden. Die Start- und Zielknoten sollten dabei nicht immer über einen Weg verbunden sein. Gegen dieses Vorgehen wurde sich entschieden, weil es rein theoretisch ist und somit Fehler enthalten könnte, welche auf Anhieb nicht auffallen. Statt dessen wird ein existierendes 3D-Szenario genommen, um die Algorithmen in der Praxis zu testen und um die Ergebnisse nachvollziehen zu können. Damit die Wegfindung auch ordentlich ausgelastet wird, ist die Entscheidung auf das "Capture the Flag"-Szenario "Warsong Schlucht" aus World of Warcraft gefallen, weil dort die sinnigen Ziele der Wegfindung beweglich sind, es sich um offenes sowie geschlossenes Gelände handelt und die Fähigkeiten Springen und Fallenlassen gut genutzt werden können.

Nun zum Spielablauf, welcher im Prototyp nachgestellt wird. In der Warsong Schlucht befinden sich zwei gegenüberliegende verfeindete Basen, zwischen welchen sich eine offene Landschaft befindet. Ziel ist es die Flagge des Feindes aus deren Basis zu rauben und in die eigene Basis zu bringen. Die Abgabe der feindlichen Flagge passiert in der eigenen Basis und

ist nur möglich, wenn die dort eigene Flagge vorhanden ist. Wurde die eigene Flagge geraubt, dann muss diese zunächst wiederbeschafft werden, bevor die Abgabe ermöglicht wird. Wird die Flagge wiederbeschafft, dann wird diese sofort in die eigene Basis teleportiert. Jede Basis ist über zwei Wege betretbar, welche sich vor dem Flaggenraum jeweils gabeln und bieten zum Teil mal mehr und mal weniger Bewegungsfreiheit. Erzielt ein Team einen Punkt, dann dauert es ein paar Sekunden, bevor die Flaggen wieder geraubt werden können. Pro Team gibt es 10 Spieler und das Team, welches als erstes 3 Flaggen erobert, gewinnt das Match.

Für die Tests stehen insgesamt zwei Interfaces zu Verfügung (s. Abb. 8.1). Das eine Interface dient als allgemeiner Überblick, über welches die Kamera beliebig im 3D-Raum bewegt werden kann. Des weiteren können alle Werte bezüglich der Performanz und des Speicherverbrauches ausgelesen werden und zwischen den zu verwendenen Algorithmen hin und her gesprungen werden. Eine Positionierung der Flaggen ist auch möglich, welches das Spiel stark beeinflusst. Hierdurch kann der "worst-case", die Reaktionsfähigkeit und komplexe Wege erzwungen werden, welche es zu analysieren gibt. Das andere Interface bezieht sich auf den jeweiligen Spieler, über welches sich die Spieler-Attribute ablesen und modifizieren lassen. Des weiteren kann die Kamera auf dem Spieler platziert und der ermittelte Weg grafisch widergespiegelt werden. Hierdurch kann unter anderem beim Ablauf des Weges die Kantenglättung bewertet werden.

Abbildung 8.1: Admin-Menü

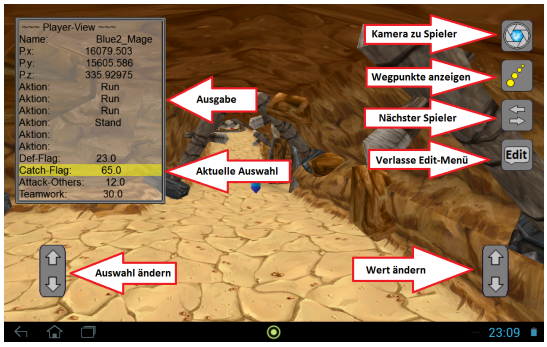

Abbildung 8.2: Spieler-Modifikations-Menü

Somit werden Performanz, Speicherverbrauch und Qualität während der Laufzeit getestet. Tests für die einmaligen Berechnungen während der Informationsgewinnung und der Grapherstellung werden nicht beachtet, weil alle Berechnungen innerhalb weniger Minuten stattfinden und das Resultat auf dem Androidgerät visuell überprüft werden kann.

Die Tests werden auf einem Iconia-Tablet A210 von Acer ausgeführt, welcher einen NVIDIA Tegra 3 Mobilprozessor, einen Quad-Core ARM Cortex -A9 und 1 Gigabyte Arbeitsspeicher zu Verfügung stellt. Auf diesem läuft das Android-Betriebsystem mit der Version 4.1.1 und hat eine Auflösung von 1280x800.
Im Gegensatz zum Original "Warsong Schlucht" werden nicht alle 3D-Modelle wie Laternen, Sträucher und weitere Dekoration verwendet, um die Komplexität niedriger zu halten.
Das Wegfindungssystem während des Testens läuft wie folgt ab. Alle 0,25 Sekunden werden die Extra-Kosten des Graphen neu berechnet. Danach werden die Wege von jedem Spieler zu jedem Spieler berechnet, wenn ein festgelegtes Limit nicht überschritten wird. Anschließend berechnen die Spieler den Weg des Verlangens. Sind die Kosten zu hoch und ein Gegner ist in akzeptabler Entfernung, dann wird dieser unter Umständen attackiert. Bei 10 Spielern pro Seite ergeben sich somit 19 Weganfragen pro Spieler und insgesamt 380 Suchen, zu welchen pro Spieler noch zusätzlich 2 hinzukommen könnten, wenn beide Flaggen noch in deren Basen stehen und beachtet werden sollen, welches je nach Begierde kein Limit verwendet. Somit wären das im schlimmsten Fall 420 Suchanfragen. Bei einer Wiederholung von 250 Millisekunden müsste somit jede Abfrage in weniger als einer Millisekunde abgearbeitet werden. Aufgrund der Verwendung eines Quad-Cores und der Verwendung eines Limits wird ein flüssiger Ablauf für möglich gehalten, was zu überprüfen ist.

8.2 Anforderungen

Es werden folgende technische und spielbedingte Anforderungen beachtet:

1. Alle nicht ausgeschlossenen Anforderungen aus Abschnitt 3.1 müssen erfüllt worden sein.

2. Die Wegfindungen müssen performant laufen, damit ein flüssiges Spiel ermöglicht wird.

3. Es sollte ein Ansatz erkennbar sein, ob die NPC's durch menschliche Spieler ersetzt werden können. Damit dieses endgültig der Fall ist, müsst der KI-Teil noch weiter ausgeschmückt werden.

4. Teamwork bezüglich der Potentialfelder muss erkennbar sein.

5. Je nach Individualität der Spieler soll deren Verhalten erkennbar sein. Hierzu gehören die Aufgabenbereiche "Eigene Flagge beschützen", "Gegner aufhalten" und "Gegnerische Flagge beschaffen".

6. Je nach Individualität der Spieler muss das "Capture the Flag"-Spiel erfolgreich verlaufen. Hierbei muss das Bestreben zu Gewinnen erkennbar sein. Unter Umständen müssen eher defensive Spieler zum Flaggendiebstahl aufbrechen.

7. Die wenigen Fähigkeiten, welche Cooldown-bedingt sind, sollten von den Spielern intelligent eingesetzt werden und die Angriffsziele sollten eher dem "Schere-Stein-Papier"-Prinzip entsprechen.

8.3 Ergebnisse

In diesem Abschnitt werden die Ergebnisse der Tests aufgeführt. Zunächst wird mitgeteilt, in welchem Umfang die Anforderungen erfüllt worden sind. Bevor auf entdeckten Auffälligkeiten eingegangen wird.

8.3.1 Anforderungsbezogen

- Anforderung 1.1) Es muss schnell berechnet werden können, ob ein Ziel erreichbar ist. Diese Anforderung ist klar erkennbar erfüllt. Beim A*-Algorithmus sind Performanzeinbußen durch den Worst-Case-Wert zu erkennen, wobei der Hierarchische Algorithmus neue Bestzeiten erreicht.

- Anforderung 1.2) Jeder Weg, welcher vom menschlichen Spieler gewählt werden kann, muss auch von der KI mit dem gleichen Spielertyp gewählt werden können.
  Dieses ist nicht wirklich zu überprüfen, weil eine Spielersteuerung nicht möglich ist. Durch das Setzen der Flagge über die Admin-Ansicht kann jedoch bestätigt werden, dass die Spieler den Weg dorthin finden, wenn dieser erreichbar scheint.

- Anforderung 1.5) Ermittelte Wege müssen von der KI gleich beim ersten Anlauf bewältigt werden können.
  Es konnte nichts widersprüchliches entdeckt werden.

- Anforderung 1.6) Die KI darf nicht durch Hindernisse durchlaufen können, weil der menschliche Spieler dieses auch nicht kann und als cheaten empfindet.
  Beim Endtest ist dieses Verhalten nicht mehr aufgetreten und Hindernisse werden umlaufen.

- Anfoderung 1.7) Der Weg muss des menschlichen Spielers nachempfunden werden, welcher sich nicht von Wegpunkt zu Wegpunkt begibt.

Die Wegverkürzung und die Glättung an Ecken schaut gut aus. Jedoch ist aufgefallen, dass der Spieler bei Fähigkeitsübergängen urplötzlich die Blickrichtung ändert. Somit ist dieser Punkt mit dem aktuellen Stand nur zum Teil erfüllt.

- Anforderung 1.8) In 3D-Landschaften kann unter anderem ein Abhang oder eine Fallluke nur in eine Richtung betreten werden.
  Sind gegnerische Fernkämpfer auf einer Klippe oder bei den Basen eine Etage weiter oben und greifen einen unten stehenden Feind an, dann ist zu erkennen, dass dieser entweder ein Umweg zu diesen nimmt, weil dieser nicht so hoch springen kann oder einfach stehen bleibt um die eigene Flagge nicht außer acht zu lassen. Nahkämpfer hingegen lassen sich von den oberen Etagen oder Klippen runter fallen.

- Anforderung 2) Die Wegfindungen müssen performant laufen, damit ein flüssiges Spiel ermöglicht wird.
  Auch ohne aktivierter Wegfindung ist das Szenario nicht wirklich flüssig; es handelt sich um ca. 20 Bilder die Sekunde. Durch die Wegsuche ist nur eine leichte Verschlechterung spürbar.

- Anforderung 3) Es sollte ein Ansatz erkennbar sein, ob die NPC's durch menschliche Spieler ersetzt werden können.
  Zur Zeit ist diesem eher nicht der Fall und die KI verhält sich eher wie Bots. Visuell ist dem menschlichen Spieler schnell klar, welches Richtung der Gegenspieler einschlagen wird. Hier kommen die Ausweichmanöver erst viel zu spät.

- Anfoderung 4) Teamwork bezüglich der Potentialfelder muss erkennbar sein.
  Wird die Spielervariable "Teamwork" hoch angesetzt, dann ist zu erkennen, dass Anschluss zum Team gesucht wird. Aufgefallen ist, dass wenn der Teamplayer als erstes Vorwegläuft, dieser ab und zu die Richtung ändert und zurück zum Team läuft. Dieses passiert meistens dann, wenn es mindestens zwei Wege zum Ziel gibt, welche ungefähr die gleiche Distanz haben. Ein Dead-Lock, so dass die Entscheidung nie getroffen werden kann und immer leicht hin und her gerannt wird, konnte zum Glück nicht provoziert werden.

- Anforderung 5) Je nach Individualität der Spieler soll deren Verhalten erkennbar sein. Je nach prozentualer Einstellung der Spielervariablen handelt der Spieler unterschiedlich. Beim Verteidigen der Flagge ist aufgefallen, dass die Verteidiger sich von Fernkämpfern killen lassen, wenn der Weg zu diesen zu lang erscheint. Hier hätte die Umsetzung der ausgeschlossenen "Anforderung 1.3" geholfen. Des weiteren ist aufgefallen, dass die

Spieler, welche gerne Gegner bekämpfen alleine auf große Gegnertruppen zulaufen. Erst viel zu spät schein der Spieler zu bemerken, dass das nicht gut war und versucht auszuweichen. Dieses klappt je nach Gegner-Konfiguration gut, wobei jedoch ein unnützliger Umweg getätigt wurde. Trotz dessen ist dieser Punkt erfüllt.

- Anforderung 6) Die wenigen Fähigkeiten, welche Cooldown-bedingt sind, sollten von den Spielern intelligent eingesetzt werden und die Angriffsziele sollten eher dem "Schere-Stein-Papier"-Prinzip entsprechen.
  Dass Heiler vor Kriegern fokussiert angegriffen werden konnte nicht wirklich beobachtet werden. Ein leichter Ansatz ist erkennbar, wenn beide Gegnertypen so gut wie in Reichweite sind.

8.3.2 Auffäligkeiten

- Sind zu viele Spieler auf beiden Seiten zum Verteidigen eingeteilt, dann resultiert dieses zu einem Endlosspiel. Wenn zulange kein Punkt erzielt wird, dann wäre es sinnig, dass der Verteidigungswert immer schwächer beachtet wird und somit Verteidiger zu Flaggenträgern werden, um den Ball am laufen zu halten.

- Bei graden Wegen ist davon auszugehen, dass der A*-Algorithmus bessere Resultate erzielt. Tests nach Spielstart haben jedoch andere Resultate gezeigt. Der Hierarchische Algorithmus kann mithalten und liefert sogar schneller die Resultate. Dieses ist damit zu erklären, dass der Hierarchische Algorithmus auf abstraktere Ebene weniger Knoten und Kanten beachtet.

8.3.3 Vergleich der Suchalgorithmen

- Die Bestzeiten vom A* verbessern sich, wenn das Ziel immer näher kommt. Die Bestzeiten von der Hierarchischen Suche verbessern sich, wenn entweder das Ziel unerreichbar ist oder wenn ebenfalls das Ziel immer näher kommt. Der A*-Algorithmus erhält dicht gefolgt von Hierarchischen Algorithmus die beste Suchzeit.

- Die durchschnittlichen Suchzeit des A*-Algorithmus variiert zu beginn stärker und pendelt sich langsam ein, wobei die durchschnittlichen Suchzeit von der Hierarchischen Suche eigentlich gleich zu Beginn schon feststeht. Die durchschnittlichen Suchzeiten von der Hierarchischen Suche sind je nach Spieler-Modifikation und manueller Flaggensetzung mal besser und mal deutlich besser als die A*-Suche.

- Die schlechteste Suchzeit erhält klar und deutlich der A* Algorithmus, wenn die Flagge an nicht erreichbaren Orten platziert wird. Der Hierarchische Algorithmus ist nur ein klein wenig schlechter als dessen eigene durchschnittliche Wartezeit und ist somit quasi Worst-Case frei.

- Der Speicherbedarf bezieht sich auf den Worst-Case und liegt beim A*-Algorithmus bei 365 Tausend besuchten Knoten und beim Hierarchischen Algorithmus bei 730 besuchten Knoten. Hierbei ist zu beachten, dass die hierarchischen Ebenen zusätzlichen Speicherplatz benötigen. Der zusätzliche Speicherplatz beträgt ungefähr Speicherplatz des Graphen plus dessen Hälfte und wiederum plus dessen Hälfte usw. und ist somit noch akzeptabel und liegt unterhalb des doppelten Speicherplatzes vom Graphen.

Kapitel 9

Zusammenfassung

In dieser Arbeit wurde ein allgemeines Verständnis für die Wegsuche in 3D-Spielen vermittelt. Beginnend bei der Terrainanalyse, dem Bilden von Bereichen, der Grapherstellung und Optimierung bis hin zur Graphanreicherung und dem Ablauf des Weges. Hierfür wurde ein Navigation-Mesh verwendet, welches den begehbaren Bereich widerspiegelt und auf welchem via Graph gesucht werden kann. Aufgrund dessen sind Kollisionsabfragen auf der Kollisionsebene nicht mehr von Nöten. Des weiteren wurde eine hierarchische Struktur verwendet, um die Berechnungszeit gering zu halten, was auch den "worst-case" betrifft. Zum Schluss wurde ein Prototyp gebaut, welcher ausführlich getestet und bewertet wurde.

9.1 Der Hierarchische-Algorithmus

Während der Arbeit wurde an einem neuen Algorithmus geforscht, welcher ein schnelles und robustes Laufzeitverhalten aufweist. Hierbei ist ein hierarchischer Algorithmus entstanden.
Der Algorithmus verwendet Bereiche, welche iterativ zusammengefasst werden, bis nur noch ein Bereich übrig bleibt. Handelt es sich um eine Stadt, dann könnten Straßen zu Stadtteilen und Stadtteile zur Stadt zusammengefasst werden. Bei der Suche wird sich von oben nach unten durchgehangelt und überprüft, ob sich der Start- oder der Zielknoten in den jeweiligen Bereichen befindet. Bei der Stadt wären das folgende Überprüfungen. Befinden sich Start und Ziel in der Stadt, wenn ja, in welchem Stadtteil und anschließend in welcher Straße des jeweiligen Stadtteils. E wird zunächst überprüft ob ein Weg existiert und welche Länge dieser aufweist. Danach wird die ermittelte Weglänge genommen, um auf abstrakterer Ebene nach kürzeren Wegen zu suchen. Hierdurch wird sichergestellt, dass trotz der Verwendung von Hierarchien ein sehr guter Weg gefunden werden kann.
Des weiteren ist aufgrund der Hierarchien eine gute Skalierbarkeit geboten und aufgrund des

Algorithmus wird eine neben-läufige Umsetzung ermöglicht.

9.2 Bewertung

Durch die Testergebnisse hat sich der Hierarchische Algorithmus als eindeutiger Sieger herauskristallisiert. Ein "worst-case" Szenario wie beim A*-Algorithmus existiert gar nicht und die Performanz ist aufgrund weniger Berechnungen sehr gut. Des weiteren werden Gebietsübergänge wie Springen und Fallenlassen gut unterstützt.
Die Kanten des verwendeten hierarchischen Graphen enthalten die Wegkosten, welche um weitere dynamische Kosten erweitert worden sind, damit der Gruppenzusammenhalt des Teams und die Angst vor Feinden während der Wegberechnung beachtet werden kann. Tests zeigten hier, dass unter anderem die Übermacht von Feinden zunächst unterschätzt worden ist und Ausweichmanöver erst spät angestoßen worden sind. Daher wäre es gewiss sinnig den Wert und die Berechnung der dynamischen Kosten noch weiter zu verfeinern, um realistischere Ergebnisse zu erhalten.
Neben der Wegsuche wurde der Weg durch mehrere Techniken modifiziert. Als erstes wurde der gefundene Weg durch einen Funnel-Algorithmus begradigt und somit verkürzt. Danach wirken anziehende und abstoßende Kräfte von Teammitgliedern und Gegnern und verändern leicht den Wegablauf. Damit der Weg jedoch garantiert begehbar ist, mussten viel zu teure Berechnungen getätigt werden. Kostengünstigere Berechnungen oder eine andere Repräsentation von Hindernissen könnte hier eventuell den gewünschten Effekt erzielen. Ein weiterer Modifikator wirkt als Potentialfeld sobald ein Wegpunkt erreicht worden ist und kümmert sich um geschwungene Wege, was den Wegablauf gut erkennbar realistischer wirken lässt.

9.3 Ausblick

Diese Wegfindung kann in vielerlei Hinsicht um noch mehrere Aspekte erweitert. Unter anderem:

- Mit mehreren anderen Suchverfahren verlgeichen.
  Hierbei sollten mehrere verschiedene Geländearten ausgewählt werden, weil davon ausgegangen werden kann, dass je nach Typ ein anderer Algorithmus Leistungsstärker ist. Gute Test-Kriterien sind Performanz und Speicherverbrauch in den Phasen der initialen Grapherstellung / -anpassung sowie zur Laufzeit.

- Sich dynamische ändernde Umgebungen mit einbeziehen und bewerten.
  Zugbrücken, Aufzüge, zerstörbare Hindernisse oder herabgestürzte Felsbrocken haben Auswirkungen auf den Graphen. Oft reicht es aus, nur den betroffenen Bereich neu zu berechnen, wobei es bei genutzten Hierarchien kostenintensiver sein könnte. Dieses zu beurteilen und effektive Maßnahmen zu untersuchen, wäre gewiss sinnig. Geländeänderungen in World of Warcraft passieren meistens in indirekter Abwesenheit des Spielers, wodurch voraussichtlich einfach nur der aktuelle Graph für die Wegfindung in den Speicher geladen wird. In diesem Fall handelt es sich nur um reines Austauschen und hat mit diesem Punkt eher wenig zu tun.

- Kollisionen zwischen Spielern ermöglichen.
  Spieler oder bewegliche Objekte können den Weg versperren oder sogar neue Wege ermöglichen, weil diese als eine Art Treppenstufe genutzt werden können. Die Form des Volumens kann ebenso beliebig sein. Die Corridor Maps Method erlaubt nur Kreisbasierte Repräsentationen, wobei das Navigation-Mesh besser eckige aufnehmen kann. Beim Navigation-Mesh könnten die Formen der Spieler genauer untersucht werden und auch deren Effekte bezüglich des Wegverlaufes, der Wegblockade oder als Wegbildung. Eine Kombination aus Navigation-Mesh und CMM könnte auch untersucht werden.

- Den angesprochenen KI-Teil ausschmücken.
  Bei diesem Punkt sollten die Bereiche Performanzminderung und Nutzsteigerung genauestens beobachtet werden. Meist muss der Graph mit weiteren Informationen angereichert werden. Dieses könnten unter anderem interaktive Objekte sein, wie es in Sims der Fall ist (Rab02). Soll der Aufzug benutzt werden, dann wird nach Erreichen dessen der Türöffner gedrückt und auf den Aufzug gewartet. Ist der Aufzug da, dann wird dieser betreten und weitere Anweisungen folgen. In diesem Fall sagt nicht der Spieler, was getan werden muss, sondern die interaktiven Objekte. Interessant wäre auch der kämpferische KI-Teil, bei welchem der NPC genau wie der menschliche Spieler Leben, Mana und Cooldown-basierte Fähigkeiten verwendet, welche zum Teil mal mehr und mal weniger Mana benötigen. Hierbei sollte die Wahl der Fähigkeiten der Situation entsprechend ausgewählt werden. Auch das aggressive und defensive Verhalten bezüglich des Kampfes und der Wegfindung sollte beachtet werden. Ebenso wäre es sinnig, wenn die KI sich gute Positionen merken kann, um aus der Landschaft im Kampf einen Vorteil zu gewinnen. Dieses wird unter anderem durch Influence Maps (Pot00) umgesetzt.

Literaturverzeichnis

[aSp10] aSpaceInvader: *Tutorial: How to Kite/Kill ANY Melee mob in WoW*. http://www.youtube.com/watch?v=GBOCOfSyORc. Version: August 2010. – [Abruf: Januar 2014]

[Bod12] Bodewes, Veerle: *Path planning in Assassin's Creed II*. http://www.docstoc.com/docs/124766089/Path-planning-in-Assassins-Creed-II. Version: July 2012. – [Abruf: Januar 2014]

[Cla13] Clavus: *Path planning in Half-Life 2*. http://steamcommunity.com/sharedfiles/filedetails/?id=153501648. Version: Juni 2013. – [Abruf: Januar 2014]

[Egg08] *Kapitel* Using the Corridor Map Method for Path Planning for a Large Number of Characters. In: Egges, Arjan: *Motion in Games - First International Workshop, MIG 2008, Utrecht, The Netherlands, June 14-17, 2008, Revised Papers*. 2008. Aufl. Berlin, Heidelberg : Springer, 2008. – ISBN 978–3–540–89219–9, S. 11–22

[Eri04] Ericson, Christer: *Real-Time Collision Detection*. Har/Cdr. Boca Raton, Fla : CRC Press, 2004. – ISBN 978–1–558–60732–3

[Fir08] Firbach, Michael: *Echtzeitfähige Wegplanung in Computerspielen: Diplomarbeit Hochschule für angewandte Wissenschaften FH München, Fakultät 07 Informatik, Mathematik*. Hochsch. für angewandte Wissenschaften, 2008 http://books.google.de/books?id=UgtMcgAACAAJ. – [Abruf: Januar 2014]

[Ger10] Geraerts, Roland: *Planning Short Paths with Clearance using Explicit Corridors*. http://www.staff.science.uu.nl/~gerae101/pdf/ecm.pdf. Version: 2010. – [Abruf: Januar 2014]

[GH03] Goldberg, Andrew V. ; Harrelson, Chris: *Computing the Shortest Path: A* Search Meets Graph Theory*. http://www.avglab.com/andrew/pub/soda05.pdf. Version: 2003. – [Abruf: Januar 2014]

[Gre10] Grech, Andreas: *The A* algorithm in XNA*. http://blog.dreasgrech.com/2010/12/a-algorithm-in-xna.html. Version: December 2010. – [Abruf: Januar 2014]

[Hal05] Hall, Matt: *Pathfinding in an Entity Cluttered 3D Virtual Environment*. http://www.comp.leeds.ac.uk/fyproj/reports/0405/HallM.pdf. Version: 2005. – [Abruf: Januar 2014]

[Har09] Harabor, Daniel: *Beyond A*: Speeding up pathfinding through hierarchical abstraction*. http://harablog.files.wordpress.com/2009/06/beyondastar.pdf. Version: 2009. – [Abruf: Januar 2014]

[hav12] havokchannel: *Havok AI in Guild Wars 2 Presented by Brett Vickers Part 1*. http://www.youtube.com/watch?v=8ZzaxBs62ag. Version: July 2012. – [Abruf: Januar 2014]

[ian09] ianxofour: *The Pathfinding Charm: Majd Addin and William de Montferrat*. http://www.youtube.com/watch?v=PRdq6ioX8NE. Version: September 2009. – [Abruf: Januar 2014]

[KS13] KS: *Assassins creed 3, Glitches and funny moments (Part 4)*. http://www.youtube.com/watch?v=j0LzyNxDI_w. Version: February 2013. – [Abruf: Januar 2014]

[Mil06] Miles, David: *Crowds In A Polygon Soup Next-Gen Path Planning*. http://www.navpower.com/gdc2006_miles_david_pathplanning.ppt. Version: March 2006. – [Abruf: Januar 2014]

[MrE13a] MrEdxwx: *Assassin's Creed 3 Funny Silly Crazy Stuff Part 2*. http://www.youtube.com/watch?v=iW-1E9Viiro. Version: February 2013. – [Abruf: Januar 2014]

[MrE13b] MrEfedrin: *Assassin's Creed III: Funny moments, glitches, stupid NPC AI and other...* http://www.youtube.com/watch?v=4tSvtTOOk8o. Version: May 2013. – [Abruf: Januar 2014]

[mup11] MUPP33N: *The Elder Scrolls V: Skyrim | AI Pathfinding is shit!* http://www.youtube.com/watch?v=IAiPYaj8J3o. Version: November 2011. – [Abruf: Januar 2014]

[Pot00] POTTINGER, Dave C.: Terrain Analysis in Realtime Strategy Games. (2000). http://web.archive.org/web/20081221121604/http://www.gamasutra.com/features/gdcarchive/2000/pottinger.doc. – [Abruf: Januar 2014]

[pra] *Lecture 6.* http://cs.brown.edu/courses/cs195u/lectures/06.pdf. – [Abruf: Januar 2014]

[Rab02] RABIN, Steve: *AI Game Programming Wisdom.* Har/Cdr. Clifton Park, NY : Cengage Learning, 2002. – ISBN 158–4–500–778

[shi11] SHIVNZ: *Skyrim companion pathfinding fail (Blackreach).* http://www.youtube.com/watch?v=8_3llpoHDvQ. Version: November 2011. – [Abruf: Januar 2014]

[Tec] TECHNOLOGY, Unreal: *Navigation Mesh Reference.* http://udn.epicgames.com/Three/NavigationMeshReference.html. – [Abruf: Januar 2014]

[Toz08a] TOZOUR, Paul: *Fixing Pathfinding Once and For All.* http://www.ai-blog.net/archives/000152.html. Version: July 2008. – [Abruf: Januar 2014]

[Toz08b] TOZOUR, Paul: *Pathfinding Bugs in Modern Games.* http://www.youtube.com/watch?v=lw9G-8gL5o0. Version: July 2008. – [Abruf: Januar 2014]

[Tur12] TURBOROADER: *Assassin's Creed III - New and improved AI and path-finding (Official Devlog Update).* http://www.youtube.com/watch?v=pLdazwuGLgw. Version: November 2012. – [Abruf: Januar 2014]

[weba] *Hierarchisches Layout.* http://de.wikipedia.org/wiki/Hierarchisches_Layout. – [Abruf: Januar 2014]

[webb] *OpenGL.* http://de.wikipedia.org/wiki/OpenGL. – [Abruf: Januar 2014]

[web11] *WoW-JK pathfinding demonstration.* http://www.youtube.com/watch?v=NVWr-Fl5dK8. Version: Mai 2011. – [Abruf: Januar 2014]

Abbildungsverzeichnis

Algorithmenverzeichnis

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 27. Februar 2014 Jörg Lischka