Fachhochschule Wedel

Seminar – Multimediale Anwendungen 2

Titel:

Informatik – Seminar (Spiele – KI)

Thema 4:

Anpassungen des A* - Algorithmus an reale Anwendungen

Jan Schliep – mi2173

Inhaltsverzeichnis

1 A* - Algorithmus

Während die Ziele und die grundlegende Theorie des A* - Algorithmus relativ leicht zu verstehen sind, ist die Implementierung meist sehr schwer zu realisieren. [Rab02] In den folgenden Punkten werden die grundsätzliche Funktionalität und die Implementierung der Wegfindung durch den A* - Algorithmus anhand einer 2D Karte erläutert. Zusätzlich wird ein wenig genauer auf die Heuristik eingegangen.

1.1 Grundlagen des A* - Algorithmus

Nehmen wir an, der grüne Punkt sei die Startposition, der rote das Ziel. Des Weiteren ist die blaue Markierung ein zu umgehendes und unpassierbares Hindernis. Damit ergibt sich für die Felder jeweils der Status „begehbar“ oder „nicht begehbar“. Die Karte ist zwecks der Vereinfachung des Suchbereiches in Quadrate unterteilt. Daraus ergibt sich eine zweidimensionale Matrix. Die Quadrate werden Knoten genannt. Dies dient der Pauschalisierung, da die Karte aus jeglichen Formen bestehen kann – zum Beispiel Rechtecke, hexagonale Felder oder auch Dreiecke. [Pfad07]

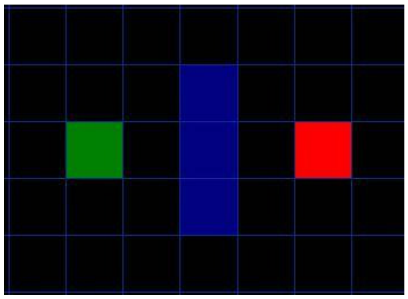

Abbildung 1.1.1 - Startposition

Zunächst wird der Startknoten einer so genannten „offenen Liste“ hinzugefügt. Dies bedeutet, dass dieser Knoten noch bei der Suche nach dem optimalen Pfad mit einbezogen werden muss. Ausgehend von der Startposition werden alle angrenzenden Knoten betrachtet, deren Status „begehbar“ ist. Auch diese Knoten werden der „offenen Liste“ hinzugefügt. Zusätzlich erhält jeder Knoten einen Zeiger auf den Vorgängerknoten, um den Pfad später zum Startknoten zurückverfolgen zu können. Abschließend wird der Startknoten aus der „offenen Liste“ entfernt und in eine „geschlossene Liste“ eingefügt. Dies bedeutet, dass nun alle angrenzenden Knoten des Startknotens entsprechend abgearbeitet und in die „offenen Liste“ eingefügt worden sind. [Pfad07]

Wie in Abbildung 1.1.2 aufgezeigt, ergibt sich daraus das folgende Bild. Der Knoten in der Mitte ist der Startknoten. Der blaue Rand bedeutet, dass sich dieser Knoten bereits in der „geschlossenen Liste“ befindet – alle Nachbarknoten wurden bearbeitet. Umliegend befinden sich die Nachbarknoten, die nun alle in der „offenen Liste“ stehen und einen entsprechenden Zeiger auf den Vorgängerknoten, also den Elternknoten besitzen. [Pfad07]

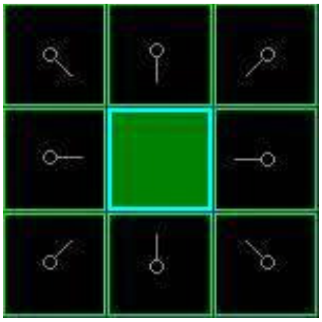

Abbildung 1.1.2 - Knotenbetrachtung

Nun wird als nächstes einer der angrenzenden Knoten betrachtet. Um den nächsten Knoten auszuwählen, werden drei Faktoren als Auswahlkriterium angeben:

g = Bewegungskosten – werden benötigt, um von dem Startknoten aus zu diesem spezifischen Knoten zu gelangen

h = Geschätzte Kosten – werden benötigt, um von diesem Knoten aus zu dem abschließenden Zielknoten zu gelangen

f = Summe der geschätzten Kosten und der anfallenden Bewegungskosten

Die Ermittlung des Wertes von h wird auch heuristisch genannt. Dies liegt an der Methode an sich, da die Kostenschätzung auf Vermutungen beruht. Für diese Schätzung gibt es unterschiedliche Methoden, die später genauer erläutert werden. Es wird nun der anscheinend optimale Pfad erzeugt, indem wir die „offene Liste“ abarbeiten und dabei immer jeweils den Knoten mit dem niedrigsten f Wert betrachten. Für die Betrachtung scheint dies der wahrscheinlichste Knoten zum Ziel zu sein. [Pfad07]

Für die Bewegung in eine Richtung verwenden wir in diesem Beispiel die Werte 10 (horizontal und vertikal) und 14 (diagonal), da sich diese ganz einfach annährend aus dem Pythagoras ergeben. g eines Knotens ergibt sich nun aus der Summe von g des Elternknotens und der Bewegung zu dem zu betrachtenden Knoten. h wird in diesem Beispiel durch die „Manhattan – Methode“ erzeugt. Diese Methode wird später genauer erläutert. Grundsätzlich sei genannt – je dichter die Schätzung der real verbliebenen Distanz liegt, desto schneller wird auch der verwendete Algorithmus sein. Abschließend wird f durch die Summe von g und h gebildet. In der Abbildung 1.1.3 befindet sich f in der oberen linken, g in der unteren linken und h in der unteren rechten Ecke. [Pfad07]

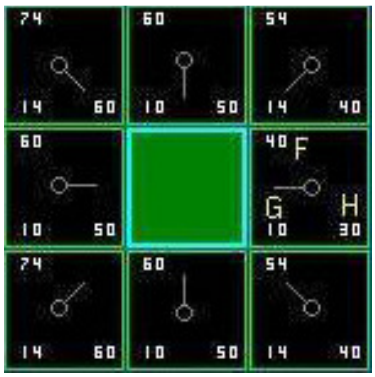

Abbildung 1.1.3 - Wegfindung

Für die Suche nach dem Pfad wird nun zunächst der Knoten mit dem niedrigsten f Wert betrachtet. In diesem Fall ist das der Knoten rechts neben dem Startknoten. Dazu wird der Knoten aus der „offenen Liste“ entfernt und der „geschlossenen Liste“ hinzugefügt. Danach werden alle „begehbaren“ angrenzenden Knoten der „offenen Liste“ hinzugefügt, die sich nicht bereits in einer der beiden Listen befinden. Sollte sich einer der angrenzenden Knoten bereits in der „offenen Liste“ befinden, wird überprüft, ob die Bewegungskosten für diesen angrenzenden Knoten geringer sind, wenn dieser von dem aktuell betrachteten Knoten aus betreten wird. Ist dies der Fall, wird der Zeiger auf den aktuellen Knoten als Elternknoten umgesetzt und jeweils der f und der g Wert neu berechnet. Sollte g nicht geringer werden, bleiben der Zeiger und die Werte des angrenzenden Knotens unverändert. [Pfad07]

In diesem Beispiel wären die Bewegungskosten nicht geringer, selbst wenn wir alle vier erlaubten angrenzenden Knoten betrachten. Die anderen vier Knoten sind entweder „nicht begehbar“ oder befinden sich bereits in der „geschlossenen Liste“. Diese können also damit aus der Betrachtung ausgeschlossen werden. Damit können wir den nächsten Nachbarknoten unseres Startknotens bearbeiten. Dazu wird der Knoten mit dem nächst niedrigen f Wert der noch verbliebenen sieben Knoten in der „offenen Liste“ verwendet. Dieser Prozess wird solange wiederholt, bis der Zielknoten in die „geschlossene Liste“ verschoben wird oder die „offene Liste“ leer ist – in diesem Fall konnte kein entsprechender Pfad gefunden werden. Wird der Zielknoten gefunden, wird über die Zeiger retrograd der Weg zum Startknoten zurückgegeben. [Pfad07]

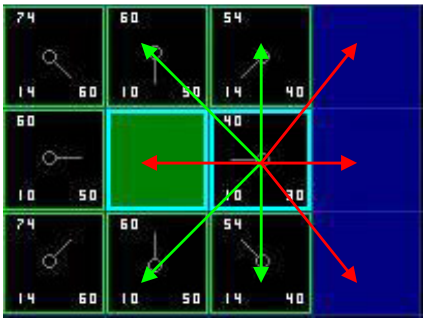

Abbildung 1.1.4 - Die Suche

Zusammenfassend lässt sich die Funktion des A* - Algorithmus also durch den folgend aufgezeigten Pseudocode beschreiben: [Rab02]

1. P = Startknoten
2. Ermittlung der g, h und f Werte für P
3. Füge P der „offenen Liste“ hinzu
4. B = erfolgversprechenster Knoten in der „offenen Liste“ (niedrigster f Wert)
   a) Wenn B der Zielknoten ist – brich ab (optimaler Pfad wurde gefunden)
   b) Wenn die „offene Liste“ leer ist – brich ab (kein Pfad gefunden)
5. C = gültiger Nachbarknoten von B
   a) Ermittlung der g, h und f Werte für C
   b) Überprüfung, ob C in der „offenen“ oder „geschlossnen Liste“ ist
      i. Wenn dem so ist – Überprüfung, ob der Weg effizienter ist (niedrigerer f Wert)
         -> Wenn dem so ist – Aktualisierung des Pfades
      ii. Ansonsten füge C der „offenen Liste“ hinzu
   c) Wiederhole Schritt 5 für alle gültigen Nachbarknoten von B
6. Wiederhole Schritt 4 und folgende Schritte

1.2 Heuristische Verfahren

Grundsätzlich wird für den A* - Algorithmus zwischen zwei Arten der Heuristik unterschieden – *zulässig* und *monoton*. *Zulässige* Heuristiken dürfen zwar nie die Kosten bis zum Ziel überschätzen, wohl aber die Kosten von einem Knoten zum nächsten. *Monotone* Heuristiken dagegen dürfen weder die einen noch die anderen Kosten überschätzen. Die Monotonie ist also eine stärkere Eigenschaft als die Zulässigkeit. Die wichtigste Konsequenz dieser Unterscheidung findet man direkt in der Implementierung des A* - Algorithmus. Sollte lediglich eine *zulässige* Heuristik verwendet werden, so kann es passieren, dass die Kosten zwischen zwei Knoten u und v überschätzt werden. Dadurch ist die berechnete Entfernung zwischen diesen beiden Knoten nicht optimal, da der Knoten v unter Umständen über einen anderen Knoten w besser erreicht wird. Ist eine Heuristik lediglich *zulässig* aber nicht *monoton*, kann nicht gewährleistet werden, dass der gefundene Weg auch wirklich der optimale ist. Dies müsste dann zusätzlich überprüft werden. Eine gute Heuristik stellt einen Kompromiss zwischen Berechnungsaufwand für das Verfahren und der Genauigkeit dar. [Wiki07]

1.2.1 Manhattan – Methode

Bei dieser Methode wird die Gesamtzahl der Knoten betrachtet, die in horizontaler oder vertikaler Richtung beschritten werden müssen, um an den Zielknoten zu gelangen. Dabei werden diagonale Bewegungen sowie jegliche Hindernisse außer Acht gelassen. Abschließend wird die Summe mit dem Faktor 10 multipliziert. [Pfad07] Grundsätzlich gibt diese Heuristik immer den kürzesten Weg zum Ziel an. Ein Nachteil dieser Methode ist aber die Übergewichtung des Wertes von g bei dem Einsatz einer Influence Map oder ähnlicher Feldmodifikationen. Diese Methode ist lediglich *zulässig*. [Heu07]

```
h = 10 * ( abs ( currentX – targetX ) + abs ( currentY – targetY ) )
```

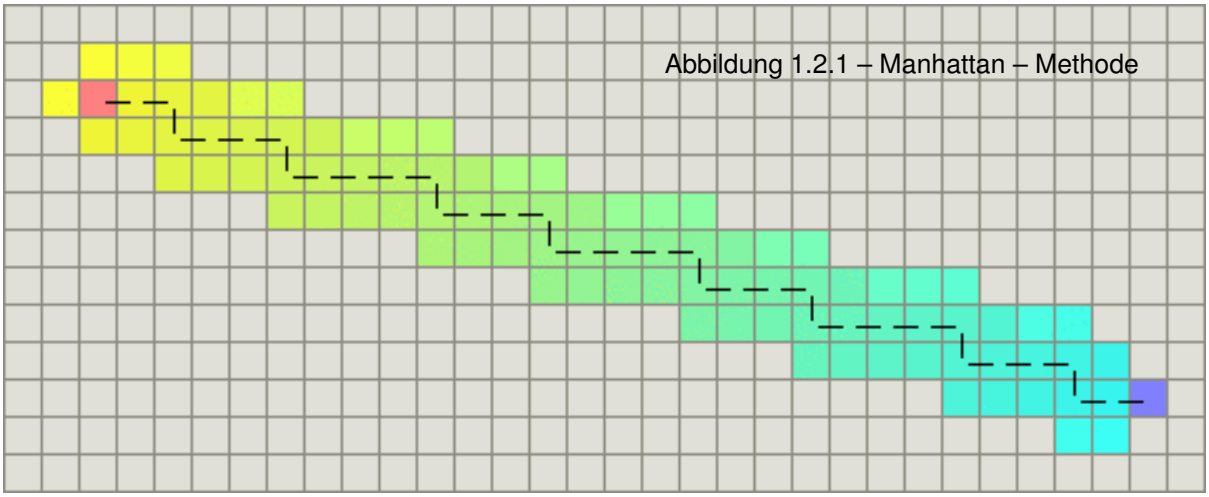

Abbildung 1.2.1 – Manhattan – Methode

1.2.2 Diagonal Shortcut

Diese Methode gleicht das Verhältnis zwischen g und h grundsätzlich aus. Sie ist ein wenig langsamer als die Manhattan – Methode, berücksichtigt aber eher kleine Modifikationen der Karte, die zum Beispiel durch eine Influence Map entstehen. Diese Methode ist lediglich *zulässig*. [Heu07]

```
xDistance = abs ( currentX – targetX )
yDistance = abs ( currentY – targetY )
if xDistance > yDistance
h = 14 * yDistance + 10 * ( xDistance – yDistance )
else
h = 14 * xDistance + 10 * ( yDistance – xDistance )
end if
```

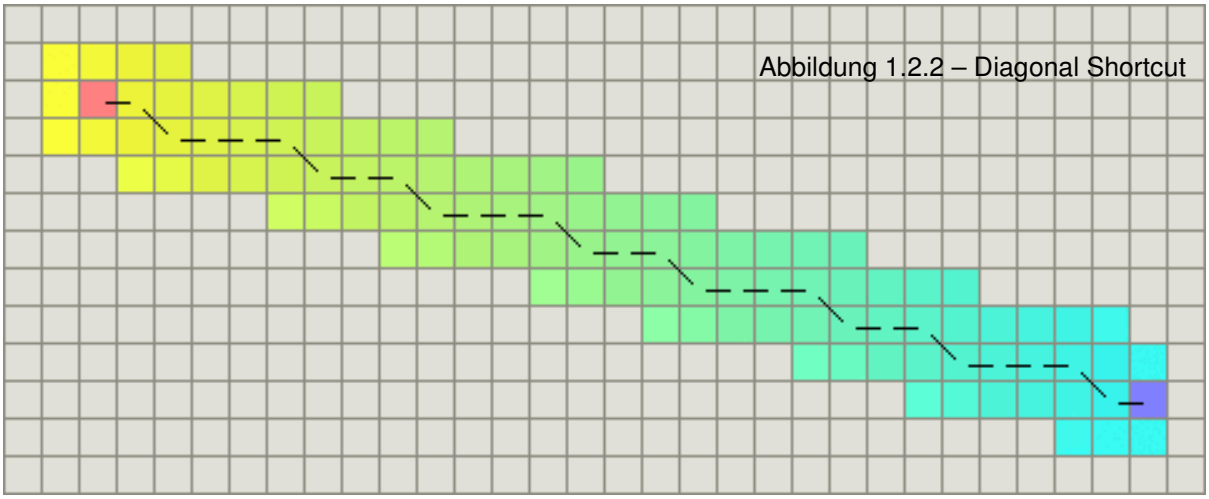

Abbildung 1.2.2 – Diagonal Shortcut

Exkurs:

Die folgenden Beispiele orientieren sich an der Entwicklung des Computerspiels *Empire Earth*. Bei dem Spiel handelt sich um ein Echtzeit Strategiespiel mit dem Ziel, eine Zivilisation erfolgreich durch die verschiedenen Zeitepochen der Erde zu führen.

Dabei stehen einem Spieler zu Beginn der Partie eine geringe Anzahl an Bürgern, einige Ressourcen und ein Kapitol, also quasi ein Haupthaus zur Verfügung. Nun müssen weitere Ressourcen abgebaut werden, um neue Bürger zu erschaffen, Gebäude zu bauen und Heere auszuheben. Die zu erstellenden Einheiten sind immer von der jeweiligen Epoche abhängig.

Das Spiel gilt als gewonnen, wenn man einen militärischen Sieg über die anderen Zivilisationen erringen kann. Alternativ können auch so genannte Weltwunder erbaut und dann für eine festgelegte Zeit gehalten werden.

Das Spiel ist mit acht Spielern im LAN spielbar. Zu Beginn des Spiels kann das Limit für die Anzahl der Einheiten auf der Karte festgelegt werden. Der Maximalwert beträgt dabei 1200 Einheiten. Die Engine für dieses Spiel ist von den *Stainless Steel Studions* entwickelt worden.

2 Generic A* Machine

Der A* - Algorithmus kann für weitaus mehr genutzt werden als für die reine Wegfindung. In dem Spiel *Empire Earth* wurde der Algorithmus unter anderem auch für die Terrainanalyse, die Choke – Point Detection, die Planung der KI Militärrouten, das Erschaffen und die Bewegung des Wetters, das Bauen von Mauern und Gebäuden durch die KI, die Planung und Bewegung von Tieren und natürlich auch für die Wegfindung verwendet. [Rab02]

Anstelle der reinen Suche nach dem kürzesten Pfad kann der A* - Algorithmus auch dafür verwendet werden, die negativen Seiten eines Weges aufzuzeigen. In diesem Fall werden dann alle Wegpunkte betrachtete, die nicht zum direkten Weg gehören. Ein Beispiel der Terrainanalyse wäre das Erschaffen eines Waldes. Betrachtet man nun einen Baum und möchte alle mit diesem Baumknoten verbundenen Bäume finden und abbilden, wird dafür eine Einheit auf diesen ersten Baum gesetzt. Diese Einheit erhält die Regel, dass sie sich nur auf Knoten mit Bäumen bewegen kann. Abschließend wird der Einheit ein Knoten als Ziel angegeben, den sie nicht erreichen kann. Das bedeutet, der Algorithmus wird sämtliche Knoten mit einem Baum betrachten und dann abbrechen, da der Zielknoten nicht erreicht wurde. In der „geschlossenen Liste“ befinden sich nun alle Knoten mit einem Baum. Die gesamte „geschlossene Liste“ ist dann also der abzubildende Wald. [Rab02]

Sinn der A* Engine ist es, die Anwendung des A* - Algorithmus möglichst flexibel zu halten. Es soll eben nicht nur eine A* Engine zur reinen Wegfindung entwickelt werden, sondern ein generisches Konstrukt, das dann auf viele unterschiedliche Weisen genutzt werden kann.

2.1 Der Speicher

Im Beispiel von *Empire Earth* wurde eine Speicherklasse verwendet, um alle traversalen Daten sowie die „offene“ und die „geschlossene Liste“ zu speichern. Dieser Speichercontainer macht den größten Performanceunterschied im A* Prozess aus. Bei der Terrainanalyse wurde ein sehr schneller aber speicheraufwändiger Speichercontainer verwendet. Dieser kann benutzt werden, da der Speicher nach der Analyse dem System wieder zur Verfügung steht. Würde der Standardspeicher, der auch im Spiel verwendet wird, für die Analyse angewendet werden, würde der Prozess zu lange dauern. Mit dem Speichercontainer hingegen konnte die Terrainanalyse für die gesamte Karte innerhalb weniger Sekunden mehrfach durchgeführt werden. Eine zusätzliche Variante zur Analyse ist es, nicht den günstigsten Knoten als nächstes zu betrachten, sondern immer den ersten in der „offenen Liste“. Für die Terrainanalyse sind die Kosten irrelevant und können von daher vernachlässigt werden. [Rab02]

2.2 Das Ziel

Die Zielklasse bestimmt, was in der A* Engine wirklich passieren soll. Der Container enthält unter anderem Daten wie die Einheiten an sich, die Quelle, die Bestimmung und das zu verwendende heuristische Verfahren. Das interessante an der generischen Zielklasse ist, dass sie alle Informationen enthält, die für eine bestimmte A* Anwendung benötigt werden. [Rab02]

2.3 Die Karte

Die Karte ist der eigentliche Suchbereich des A* - Algorithmus. Es kann dabei die Karte der Welt im Computerspiel sein oder ein Array. Um zum Beispiel Choke – Points zu ermitteln, könnte ein Array mit Choke – Point Flags versehe sein. Dann wird die A* Machine darauf angewendet und sammelt die Flags in eine Gruppe von Punkten. Nachdem diese Punkte ermittelt worden sind, kann für diesen Bereich zum Beispiel ein High – Level Pfadfindungssystem integriert werden, um eine gute Navigation für diese Punkte gewährleisten zu können. [Rab02]

3 Design Architektur

Wenn mehrere hundert Einheiten über die Karte manövrieren sollen, wird das System dabei extrem belastet. Anhand des Beispieles *Empire Earth* wird aufgezeigt, wie es möglich ist, die Pfadfindung komplex anzuwenden ohne das System dabei in die Knie zu zwingen. Dabei wird darauf eingegangen, wie Pfade während der Ermittlung aufgeteilt werden können und der Anwender trotzdem den Eindruck hat, die Pfade würden genau in dem Augenblick vollständig generiert werden.

3.1 Das Problem

Wenn eine Einheit einen langen Weg auf der Karte zurücklegen soll, muss der dazugehörige Pfad zunächst ermittelt werden. Werden keine entsprechenden Maßnahmen dazu ergriffen, bedeutet dies, dass die Einheit nach dem Anklicken zunächst auf der Stelle verbleibt bis der optimale Pfad ermittelt worden ist und sich erst danach in Bewegung setzt. Dieses „Einfrieren“ der Einheiten ist ein fundamentales Problem, das es zu lösen gilt. Ein Ansatz zur Lösung ist dabei die Aufteilung des Pfads über die Zeit. Dies gewährleistet, dass, egal wie viele Einheiten sich bewegen sollen, die Framerate dabei nicht beeinflusst wird. In dem Spiel *Empire Earth* wird dies durch eine Dreistufeneinteilung realisiert – Quick Path, Full Path und Splice Path. [Rab02]

3.2 Quick Path

Wenn ein Benutzer einer Einheit einen Befehl gibt, dann erwartet er, dass sich die Einheit umgehend in Bewegung setzt. Der Quick Path wurde dazu entwickelt, genau dies zu realisieren. Dazu wird für den Quick Path ein highspeed Pfadfinder für kurze Distanzen verwendet. Die Einheit wird 3 bis 15 Tiles bewegt, um dem A* - Algorithmus Zeit zu geben, den vollständigen Pfad zu ermitteln. Dazu wird die Pfadfindung für den Quick Path umgehend gestartet, stoppt aber nach einer gewissen Anzahl an Durchläufen. Dann wird der Punkt, der als nächstes am Ziel liegt, als vorläufiger Zielknoten angenommen. Die Einheit bewegt sich auf diesen Knoten zu. Es wird der Eindruck vermittelt, die Einheit würde den Befehl ausführen. [Rab02]

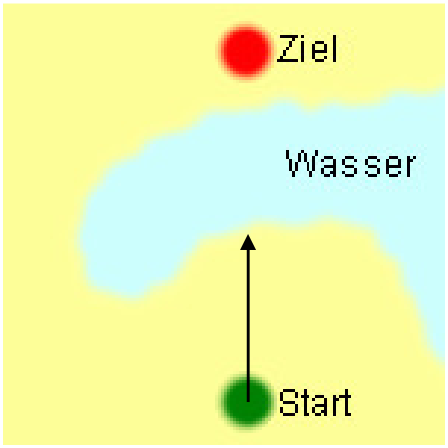

Abbildung 3.2.1 - Quick Path

3.3 Full Path

Während sich die Einheit anhand des Quick Path auf das vorläufige Ziel zu bewegt, wird im Hintergrund der eigentliche A* - Algorithmus ausgeführt. Um den Vorgang zu vereinfachen sollte eine Regel in der A* Machine implementiert werden – wird einem Knoten der Status „begehbar“ oder „nicht begehbar“ zugewiesen, sollte dies für den gesamten folgenden Prozess unverändert bleiben. Durch das Cachen eines Status muss während des Suchvorgangs nicht jeder Knoten wieder einzeln abgefragt werden. Die CPU wird damit im Spielbetrieb weniger belastet. Um einen vollständigen Pfad gewährleisten zu können, beginnt man mit der Pfadfindung an dem vorläufigen Zielknoten des Quick Path. Das Ergebnis zeigt sich in der Abbildung 3.3.1 – Full Path. [Rab02]

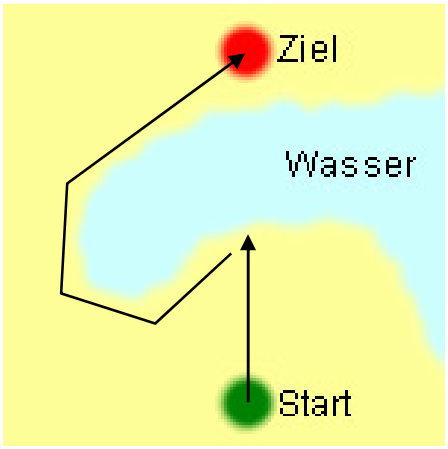

Abbildung 3.3.1 - Full Path

3.4 Splice Path

Abschließend muss noch ein Weg generiert werde, der eher an die realen Anforderungen angepasst ist. Der Splice Path behebt nun die Ungenauigkeiten, die durch den Einsatz des Quick Path auftreten. Dieses Verfahren verwendet den gleichen highspeed Pfadfinder wie der Quick Path. Als Startknoten wird die aktuelle Position der Einheit und als Zielknoten ein Knoten auf dem Full Path verwendet. Um den besten Punkt auf dem Full Path auswählen zu können benötigt es ein wenig Erfahrung. Als mögliche Kriterien könnte genannt werden, dass eine gewisse Entfernung zum Full Path erreicht sein muss oder eine bestimmte Anzahl an Wegpunkten auf dem Full Path abgeschritten wurde. [Rab02]

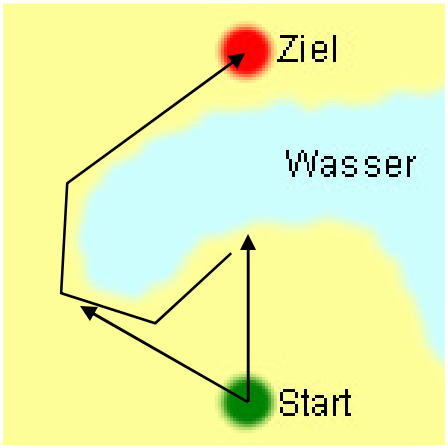

Abbildung 3.4.1 - Splice Path

3.5 Entfernen der Extra – Waypoints

Um die Bewegung dann dem generierten Pfad anzupassen, müssen abschließend alle überflüssigen Waypoints entfernt werden. Dadurch ergibt sich der in Abbildung 3.5.1 dargestellte Weg. [Rab02]

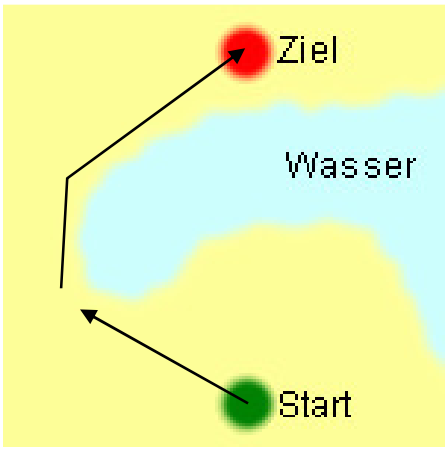

Abbildung 3.5.1 - Vollständiger Pfad

4 Performance Optimierung

Wenn in einem Programm der A* - Algorithmus, bzw. die A* Engine einwandfrei funktionieren, gelangt man erst an das eigentlich Problem der Pfadfindung – die Performance. Um jedoch eine A* Engine zu implementieren, die auch unter Echtzeitbedingungen optimal funktioniert, erfordert es viel Kreativität, Entwicklungszeit und vor allem Optimierungsarbeit. Es gilt dabei abzuwägen, ob der Weg sehr sauber sein soll, dies dann aber unter Umständen die dreifache CPU Belastung bedeuten würde. [Rab02]

4.1 Intel® VTune™

Intel® VTune™ Performance Analyzer ist ein Profiler zum Aufspüren und Beseitigen von Leistungsengpässen in Programmen. Intel® VTune™ kann für jedes Programm das exakte Zeitverhalten ermitteln. [Tune07] Bei der Entwicklung von *Empire Earth* wurden zu 80% In – Code Timer und in etwa 20% der Fälle dieses Programm verwendet. Dabei wurden die Timer dazu verwendet, das Verhalten des Spiels unter Echtzeitbedingungen zu testen. Intel® VTune™ wurde dann dazu eingesetzt, um unter anderem Slowdowns und Cache Verluste ausfindig zu machen. [Rab02]

4.2 Flood Insurance

Um die Anforderungen an die Performance zu ermitteln, gilt es zunächst einmal die Funktion des Algorithmus zu analysieren. Bei diesem Verfahren wird überprüft, wie der Algorithmus auf dem Graphen arbeitet. Dabei werden nach dem Durchlauf die „offene“ und die „geschlossene Liste“ durchgegangen und dann alle besuchten Knoten auf der Karte grafisch ausgegeben. Dadurch kann ermittelt werden, welche Bereiche der Karte überhaupt überprüft worden sind und wie anspruchsvoll die Suche auf der Karte bezüglich der CPU Belastung war. Unter normalen Umständen kann es sein, dass eine Karte den Anforderungen des Algorithmus angepasst werden muss. Ziel der Optimierungsarbeit ist es aber nun, das Grafikteam nicht in seiner Kreativität zu beeinflussen und den Algorithmus auf fast alle Karten anwendbar zu machen. [Rab02]

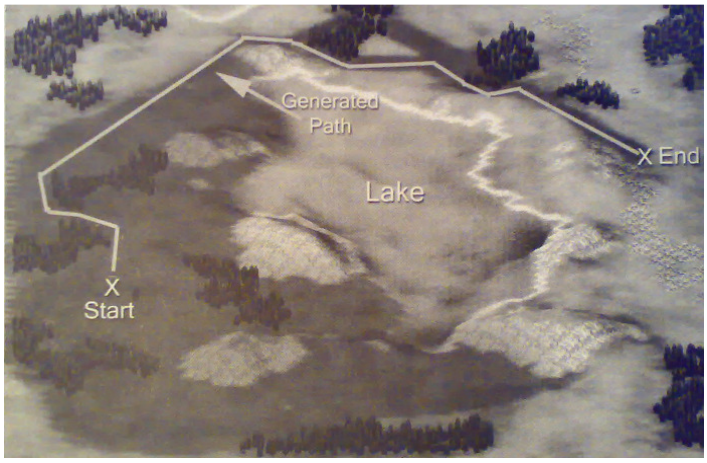

Abbildung 4.2.1 – Flood Insurance

4.3 Individuelle Speicherklasse

Das Spiel *Empire Earth* ist mit dem C++ ANSI Standard entwickelt und realisiert worden. In den meisten Fällen reicht die Datenstruktur der Standard Template Library von C++ aus. In diesem besonderen Fall wurde aber eine speziell verkettete Listen Struktur entwickelt. Diese Struktur wurde dazu geschaffen, um kleinere Tricks mit den einzelnen Knoten der Listen zu ermöglichen, die mit der

Standard Template Library ansonsten nicht möglich sind – jede Mikrosekunde zählt. Dadurch kann zum Beispiel überprüft werden, welche Knoten bei der Suche überhaupt besucht worden sind. Daraus kann geschlossen werden, wie groß der entsprechende Memory Pool für einen bestimmten Prozess ist, bzw. maximal sein kann. Dadurch kann jedem Prozess ein gewisser Memory Pool zugewiesen werden. Außerdem können nach dem Beenden des Spiels diverse Statistiken aufbereitet und dem Benutzer zur Verfügung gestellt werden. Auch wurde die Datenstruktur der einzelnen Knoten bezüglich des Zugriffs stark vereinfacht. Dazu wurde ein Statusarray mit 1 – Byte Flags geschaffen:

Clear – Der Knoten ist ungeprüft und befindet sich in keiner Liste
Passable – Der Knoten wurde überprüft und ist passierbar
Blocked – Der Knoten wurde überprüft und ist unpassierbar
Open – Der Knoten befindet sich in der „offenen Liste“
Closed – Der Knoten befindet sich in der „geschlossenen Liste“

Dadurch kann jeglicher Status eines Knotens anhand simpler AND und OR Bitoperationen abgelesen werden. Dies spart Ressourcen bezüglich der einzelnen Knotenzugriffe. [Rab02]

5 Tipps zur Optimierung

Es gibt weitere Möglichkeiten der Optimierung, um den Prozess des A* - Algorithmus noch effizienter zu gestalten. Die folgenden Punkte sind Erfahrungswerte und Möglichkeiten, die während der Entwicklungsarbeit von *Empire Earth* aufgetreten sind.

5.1 Iterative Tiefe

Bei der iterativen Tiefe wird dem Aufruf des A* - Algorithmus ein Parameter mit übergeben. Dieser Parameter kann jegliches künstliche Limit bezüglich der Wiederholung des Prozesses sein – der maximal verwendete Speicher, die Anzahl an Durchläufen der Hauptschleife oder die zurückgelegte Weglänge einer Einheit. Der Prozess wird dann sooft wiederholt, bis das Limit erreicht worden ist. Diese Methode verwendet weniger Speicher, da effektiv weniger Knoten besucht werden. Wurde nach dem Erreichen des Limits kein entsprechender Weg gefunden, wird der Prozess ein weiteres Mal mit einem leicht erhöhten Limit aufgerufen. Gerade in Situationen, in denen die A* Wegfindung nicht umgehend geschehen muss, ist diese Lösung optimal. Nehme man zum Beispiel eine Gruppe von Agenten, die einen Raum betreten möchte. Einer der Agenten blockiert nun die Tür. Alle weiteren Agenten würden jetzt nach einem nicht vorhanden Weg suchen und der Speicher wird unnötig belastet. Wird nun aber das Limit erreicht, wird die neue Suche erst nach einem Delay ausgeführt. Bis dahin ist der Weg wieder freigegeben und die anderen Agenten können den Raum betreten. [Rab02]

5.2 Knoten schließen

Unter normalen Umständen wird der Knoten mit dem niedrigsten f Wert aus der „offenen“ in die „geschlossene Liste“ verschoben und die umliegenden Knoten werden betrachtet. Bei der Betrachtung der Kindknoten werden aber auch Knoten mit extrem hohen Kosten in die „offene Liste“ aufgenommen. Dies können entsprechend der Regeln auch unpassierbare Knoten sein. Diese können nun aber umgehend in die „geschlossene Liste“ verschoben werden, da diese wohl nie betrachtet werden. Durch eine kleinere „offene Liste“ kann der Zugriff auf diese schneller erfolgen. [Rab02]

5.3 Generierung von Teilwegen

Diese Methode findet ihre Anwendung in den Fällen, in denen kein Weg gefunden werden kann oder das künstliche Limit bei einem Prozess erreicht worden ist. In diesem Fall wird ein Teil des Weges als Ergebnis zurückgegeben. Dies kann zum Beispiel der Weg mit den niedrigsten Kosten oder der mit minimal verbleibenden Restkosten sein. Diese Methode kann auch in Kombination mit der iterativen Tiefe dafür verwendet werden, um die Agenten des Spiels besser reagieren zu lassen. Die A* Wegfindung kann mit einem niedrigen Limit aufgerufen werden und damit einen Teilweg zurückgegeben. Während der Weg zurückgelegt wird könnten sich aber die Situation und das Umfeld ändern. Der nächste Teilweg passt sich dann dieser neuen Situation an. Der Agent erscheint menschlicher und scheint auf aktuelle Situationen zu reagieren. [Rab02]

5.4 Cachen von ungültigen Wegen

Eine weitere Möglichkeit ist es, fehlgeschlagene Anfragen bezüglich der Wegfindung zu speichern. Dies kann solange passieren, bis sich etwas auf der Karte ändert. Dies ist sinnvoll, wenn mobile Hindernisse den Weg blockieren können oder ein Agent stecken geblieben ist. Der Agent wird immer wieder neue Anfragen bezüglich der Wegfindung stellen. Werden fehlgeschlagene Anfragen gespeichert, muss dieser Weg nicht erst wieder entsprechend generiert werden. In dem Augenblick, in dem sich etwas auf der Karte ändert, muss auch der Cachespeicher zurückgesetzt werden. [Rab02]

5.5 Waypoints

Der menschliche Spieler kann sehr gut Wege erkennen. So kann der Mensch dem Agenten quasi dabei helfen, seinen Weg zu finden. Dazu kann der Mensch Wegpunkte auf der Karte verteilen, die die Grundlage für die Wegfindung bilden. Anstatt lange Wege über die Karte zu suchen müssen nun mehrere kleine Wege zwischen den einzelnen Wegpunkten generiert werden. Zusätzlich scheint der Agent auch bei der Suche nach längeren Wegen schneller zu reagieren. Um einen kürzeren Weg zu finden wird weniger Zeit benötigt. Der Agent setzt sich also umgehend in Bewegung. [Rab02]

6 Konklusion

Die Anwendung des A* - Algorithmus ist oft ein ineffizienter Prozess, der das System extrem ausbremsen kann. Außerdem ist die Komplexität des A* - Algorithmus unter machen Umständen schwer zu verstehen. Auf dem Papier sieht der Algorithmus einfach aus. Auch wenn man sich die Teile eines Codes anderer Programmierer anschaut erscheint die Lösung einfach. Aber das vollständige Verstehen kann manchmal abschreckend sein. [Rab02] Es bedarf bei der komplexen Anwendung des A* - Algorithmus viel Entwicklungs- und Optimierungsarbeit, um möglichst alle Eventualitäten und Performanceoptimierungen mit einzubeziehen.

Man muss sich darüber im Klaren sein, wie genau und sauber der zu generierende Weg sein soll und welche Performanceressourcen dem Prozess zur Verfügung gestellt werden sollen. Auch sollte man bedenken, dass der A* - Algorithmus nicht nur für die reine Wegfindung verwendet werden kann. Bei einer generischen Umsetzung der Lösung gibt es ein großes Spektrum an Anwendungsmöglichkeiten. Erst bei der eigenständigen und intensiven Entwicklung komplexer A* Anwendungen und dem damit verbundenen Befassen mit der Materie ergeben sich neue Optimierungsansätze – wenn zum Beispiel 1200 Einheiten gleichzeitig über eine Karte bewegt werden sollen.

7 Literaturverzeichnis

[Rab02] Steve Rabin, AI Game Programming Wisdom, Thomson Delmar Learning, 2002

[Wiki07] A* - Algorithmus – Wikipedia – Betrachtet am 24. Mai 2007
http://www.wikipedia.de/ – Suchbegriff: A*

[Pfad07] A* Pfadfindung für Anfänger – Betrachtet am 20. Mai 2007
http://www.policyalmanac.org/games/aStarTutorial_de.html

[Heu07] Heuristics and A* Pathfinding – Betrachtet am 20. Mai 2007
http://www.policyalmanac.org/games/heuristics.htm

[Tune07] Intel® VTune™ Performance Analyzer – Betrachtet am 27. Mai 2007
http://www.intel.com/cd/software/products/asmo-na/eng/vtune/vpa/219898.htm

8 Abbildungsverzeichnis