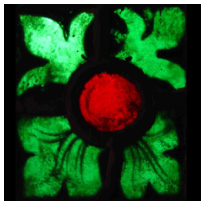

Otto-Friedrich-Universität Bamberg

Lehrstuhl für Angewandte
Informatik in den Kultur-,
Geschichts- und Geowissenschaften

Hausarbeit

Im Rahmen des Seminars

Spatial Planning

Zum Thema:

A* Engines

Vorgelegt von:

Jan Petendi

Betreuer: Prof. Dr. Christoph Schlieder

Bamberg, SS 07

Inhaltsverzeichnis

1 Einleitung

Für das Lösen von Pfadsuchproblemen wird fast ausschließlich der A* Algorithmus oder eine auf ihn basierende Optimierung verwendet. Entwickler investieren häufig viel Zeit darin, eine auf ihr Suchproblem und Laufzeitumgebung optimierte Implementierung zu entwickeln. A* ist als generischer Algorithmus definiert [1] und kann somit auf eine Vielzahl von Problemen angewendet werden. Da A* zur Klasse der Breitensuchverfahren zählt, er also die bei der Suche generierten Knoten im Speicher hält, stand die Optimierung der Speichernutzung bei einer Implementierung häufig im Vordergrund. Während die technischen Voraussetzungen heutiger Computersysteme im Allgemeinen ausreichen, um den Speicheranforderungen einer A* Suche für Desktopanwendungen gerecht zu werden, kämpfen Entwickler für mobile Endgeräte mit eben diesen Problemen. Die Relevanz mobiler Endgeräte hat dabei in letzter Zeit enorm zugenommen und einer aktuellen Studie des Marktforschungsinstituts Gartner [Gar07] zufolge könne man damit rechnen, dass auf Grund der Allgegenwärtigkeit mobiler Endgeräte mobile Spiele in Zukunft mehr Nutzer erreichen werden, als dies bei traditionellen PC- oder Konsolenspielen der Fall sei. Dem gegenüber existieren noch wenig Lösungsansätze, um A* auf mobilen Endgeräten optimiert anwenden zu können.

Vorliegende Arbeit präsentiert auf Basis von Anforderungen, die in der Spiele-Industrie an das Pfadfinden mit A* gestellt werden, eine generische objektorientierte Architektur für den A* Algorithmus – eine sog. *A* Engine* – und eine prototypische Implementierung für mobile Endgeräte auf Basis des dotNet Compact Frameworks mit C#[2].

2 allgemeine Anforderungen

Um die Anforderungen an eine *A* Engine* herauszuarbeiten, ist es zunächst erforderlich zu definieren, an welcher Stelle in der Spielarchitektur sich diese integriert und inwiefern sie mit den übrigen Komponenten der Anwendung interagiert (siehe Abbildung 1). Ganz grob lässt sich jede Anwendung dieser Art unterteilen in eine Komponente zur graphischen Interaktion mit dem Benutzer, der Anwendungslogik und den Anwendungsdaten, auf welchen beide Komponenten arbeiten und die zum Teil den Suchraum der *A* Engine* darstellen . Die *A* Engine* ordnet sich dabei in die Schicht der Anwendungslogik ein und hat somit schon dadurch mit einigen Einschränkungen zu kämpfen. Der Benutzer erwartet von solch einer Anwendung als oberste Priorität eine reaktionsschnelle, gut zu bedienende und gut aussehende Oberfläche – die Hauptrechen- und Speicherleistung wird also schon durch das Bereitstellen dieser Funktionalität in Anspruch genommen. Dem zur Folge werden die Anwendungsdaten, auf welchen wie gesagt auch die *A* Engine* in Teilen arbeiten muss, vor allem optimiert für eine effiziente graphische Darstellung. Gleichzeitig ist die Benutzeroberfläche allerdings abhängig von den Ergebnissen der Anwendungslogik. Im besten Fall integriert sich eine *A* Engine* also nahtlos und möglichst unauffällig in das sie umgebende System, d.h. sie muss sowohl den Anforderungen **einer schnellen Reaktion**, dem **effizienten Arbeiten mit den bereitgestellten Anwendungsdaten**, als auch den **Einschränkungen von Rechen- und Speicherressourcen** gerecht werden.

[1] vgl. hierzu [RN95, S.96ff.]

[2] `http://msdn2.microsoft.com/en-us/netframework/aa731542.aspx`, zuletzt besucht am 25.Juni 2007

Der A* Algorithmus birgt dabei neben dem in der Einleitung bereits erwähnten größten Problem der Speicherkomplexität auch ein Problem, welches sich auch wieder durch die Anforderungen einer gut aussehenden Benutzeroberfläche erklären lässt.Hauptaufgabe des A* Algorithmus in Spielen ist das Auffinden von Pfaden für Spieleinheiten; die gefundenen Pfade sind zwar immer die kürzesten, wirken aber gerade dadurch unnatürlich und müssen daher für eine gelungene graphische Darstellung noch optimiert werden. Lösungsansätze für dieses Problem werden hier allerdings nicht weiter betrachtet werden [3].

Grob verfolgt meine Lösung also folgende Hauptziele:

geringe Reaktionszeit je nach konkreter Anwendung muss die Komponente nicht nur Anfragen sequentiell verarbeiten, sondern ist mit einer Vielzahl von parallelen Suchanfragen konfrontiert, die im seltensten Fall bereits vorgeordnet sind und für deren Bearbeitung auch nur eine (bereits belastete) CPU zur Verfügung steht. Es ist also Aufgabe der *A* Engine*, Anfragen in eine geeignete Reihenfolge zu bringen und diese auch möglichst dynamisch auf Basis von Spiel-Ereignissen zu verarbeiten; die Verarbeitung der Anfragen sollte im Hintergrund geschehen, es dürfen also vor allem keine Auswirkungen der CPU-Nutzung bei der Darstellung der Benutzeroberfläche sichtbar werden [4].

Abstraktion von der Suchraumrepräsentation wie bereits erwähnt stellen Teile der Anwendungsdaten den Suchraum für die *A* Engine* dar, diese können je nach Anwendung und geforderter Funktionalität unterschiedlich realisiert sein. Die Komponente sollte so modelliert werden, dass sie von der jeweiligen Darstellung des Suchraums abstrahiert und somit auf beliebigen Suchraumrepräsentationen arbeiten kann.

effiziente Verwendung von Rechen- und Speicherressourcen die Speicherkomplexität ist das größte Problem des A* Algorithmus, die Komponente muss Methoden

[3] für weiterführende Informationen empfiehlt sich hierfür z.B. [Rab00a]

[4] dieser bei Anwendern wohl bekannte und gehasste Effekt des Einbruchs der Bildwiederholfrequenz unter die im Allgemeinen verwendete mindeste Bildwiederholfrequenz von 25 Bildern pro Sekunde sollte in allen Fällen vermieden werden

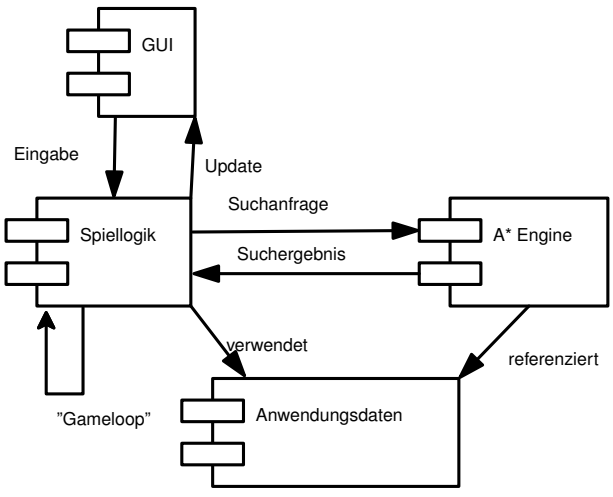

Abbildung 1: allgemeine Spielarchitektur

unterstützen, um auch auf großen Suchräumen effizient arbeiten zu können.

3 Stand der Forschung

Es existiert viel Literatur zu Forschungsergebnissen mit dem A* Algorithmus, dessen Anwendungen und Optimierungen. Der Großteil befasst sich dabei allerdings damit für ein spezielles Problem eine gut optimierte Lösung anzubieten. Demgegenüber existieren zu den generischen Eigenschaften und Anwendungsmöglichkeiten des A* Algorithmus nur sehr wenige Ergebnisse; allen voran sei hier Daniel Higgins erwähnt, auf dessen Artikel [Hig02a] meine Arbeit aufbaut.

Alle Forschungsergebnisse bieten Lösungen an, um den Speicher- und Zeitaufwand des A* Algorithmus zu optimieren. Für die Entwicklung einer generischen Lösung ist es also notwendig die grundlegendsten Methoden zu strukturieren; der nachfolgende Abschnitt soll nun einen kurzen Abriss von Optimierungen an unterschiedlichsten Stellen aufzeigen und somit eine Grundlage für die Struktur der *A* Engine* bilden. Bewertungen der betrachteten Ansätze würden den Rahmen dieser Arbeit bei weitem sprengen und werden deshalb von mir ausgeklammert werden.

3.1 Suchraumoptimierungen

Die größten Einsparungen für den Speicheraufwand lassen sich in der Repräsentation des Suchraums erzielen. Es existieren sehr viele Optimierungen, die abhängig von der Suchumgebung jeweils zu sehr guten Ergebnissen führen. [Rab00b] und [Sto00] nennen die am häufigsten verwendeten und vergleichen diese. In einem ersten Schritt unterscheiden sie lineare Suchräume und hierarchische.

Lineare Suchräume Die beiden bei Spielanwendungen am häufigsten verwendeten sind:

- Rechteckiges oder Hexagonales Gitter: ein 2D-Gitter wird über die Welt gelegt, wobei sich die Größe einer Gitterzelle an dem kleinsten in der Welt vorkommenden Objekt orientiert
- Polygonaler Untergrund: hier werden entweder diejenigen Polygone von den bereits als Daten zum Rendern der 3D-Welt vorhandenen markiert, die den Untergrund darstellen, auf welchem die Einheiten sich bewegen können oder es werden vereinfachte Polygone nur zum Zweck des Pfadfindens erstellt

Hierarchische Suchräume Für den häufigen Fall, dass der Suchraum nicht komplett im Speicher gehalten werden kann, existieren auch Methoden obige Suchräume hierarchisch zu clustern und auf diesem eine viel speichereffizientere Suche ausführen zu können. Gerade auf mobilen Endgeräte ist eine hierarchische Repräsentation eine sehr gute Möglichkeit, den geringen Speicherkapazitäten Herr zu werden. Die genaue Vorgehensweise

soll an dieser Stelle nicht weiter betrachtet werden, für Näheres hierzu sei deshalb z.B. verwiesen auf [Rab00b] und [BMS04].

3.2 Open- und ClosedList

Zum Verwalten des Zustands der Suche müssen für jeden Knoten folgende Informationen verfügbar sein :

- ein Verweis auf den Elternknoten
- die Kosten, um zu diesem Knoten zu kommen
- die geschätzten Restkosten
- ob dieser Knoten schon besucht wurde
- ob der Knoten noch besucht werden muss

Dafür werden 2 Datenstrukturen verwendet:

- die OpenList : enthält alle noch zu besuchenden Knoten
- die ClosedList : enthält alle schon besuchten Knoten

Die OpenList muss optimiert sein auf häufiges Sortieren und Entfernen, daher wird hierfür als Implementierung häufig eine `PriorityQueue` auf Basis eines `Binary Heap Trees` verwendet, während die ClosedList, welche vor allem verwendet wird, um zu überprüfen, ob ein Knoten bereits besucht wurde, mit einer `HashTable` verwaltet werden sollte.

[Sto00] schlägt als Optimierung hierfür zusätzlich vor diese Datenstrukturen zu vereinen in einer `HashTable`, um die Informationen, auf welcher Liste sich der Knoten befindet mit linearem asymptotischen Aufwand `(O(n))` zu erhalten und zusätzlich die OpenList zu verwalten. Da lediglich mit Objektreferenzen gearbeitet wird, stellt die Redundanz durch die doppelt referenzierten Knoten auf der OpenList und der `HashTable` nur einen geringen Overhead dar. Für die Implementierung der OpenList schlägt er ebenfalls eine `PriorityQueue` vor, da dort sowohl Einfüge- als auch Löschoperationen nur `O(log(n))` benötigen.

3.3 Kostenfunktion und Heuristik

Das Suchverhalten von A* lässt sich mit einer richtigen Balanzierung von Kostenfunktion und heuristischer Restkostenabschätzung genau auf ein Suchproblem optimieren. In Spiele-Anwendungen ist allerdings eine reine Unterteilung des Suchraumes in *frei* und *blockiert* häufig unzureichend. Man will erreichen, dass die berechneten Pfade sich möglichst dynamisch an der gegenwärtigen Spielsituation orientieren. Als mögliche zusätzliche Attribute kämen z.B. Terraintypen, Höhenunterschiede, Nähe zu Feinden etc. in

Betracht. Aufgrund der Fülle von Variationen bei der Kostenfunktion ist auch das Abschätzen der Restkosten viel komplexer, als dies schon im Allgemein bei A* der Fall ist . Meist optimiert man deshalb auf Basis einer bewährten Heuristik unter der Prämisse,dass ein schnell berechneter jedoch möglicherweise suboptimaler Pfad für den Großteil der Fälle ausreicht. Für eine sehr anschauliche Heranführung an diese Problemstellung empfiehlt sich `http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html` (zuletzt besucht am 18.07.2007) ([Pat07]).

3.4 Entlastung der A* Engine

Neben all den Optimierungen, wie man eine A* Engine speicher- und zeiteffizienter gestalten kann, sei natürlich ebenfalls erwähnt, dass einige Methoden existieren diese in ihrer Arbeit zu unterstützen bzw. zu entlasten.

Filtern überflüssiger Suchanfragen Das Formulieren einer Suchanfrage, die keinen gültigen Pfad zwischen Start und Ziel erzeugen kann, führt zu dem Verhalten, dass die Suche erst stoppt, wenn die *OpenList* leer ist, was je nach Größe des Suchraums einiges an Zeit- und vor Allem Speicheraufwand bedeutet. Neben der Bestimmung einer Höchstzahl von Suchschritten (vgl. 4.1.2) sollten geeignete Vorverarbeitungsschritte in Betracht gezogen werden, um die Durchführung solcher Suchanfragen zu minimieren (siehe auch 5).

Zusammenfassung von Gruppenbewegungen Die Pfadsuche vieler Einheiten, die gemeinsam ein Ziel erreichen wollen, sollte keinesfalls so realisiert werden, dass für jede Einheit einzeln der Pfad berechnet wird, vielmehr sollten in der Spiellogik geeignete Mechanismen eingebaut werden, um diese Gruppenbewegung anderweitig zu unterstützen.Eine einfache Lösung wäre z.B. für eine Einheit den Pfad zu berechnen und die übrigen auf diesem Pfad folgen zu lassen; hierbei treten allerdings andere Probleme auf, wie mögliche Kollisionen untereinander oder mit der Umwelt, deshalb sei an dieser Stelle nur kurz darauf verwiesen, dass eben dieser Sachverhalt der Simulation von Massenbewegung auch in vielen Forschungsarbeiten behandelt wird.

intelligent Movement Die Pfadsuche in Spielen wird überwiegend dazu genommen Einheiten einen Weg zu generieren, welchem diese folgen sollen. Diese Einheiten sind im häufigsten Fall Implementierungen intelligenter Agenten, es ist also ein Leichtes diesen zusätzlich ein Verhalten zu spezifieren, damit sie noch vor Rückgabe des kompletten Pfades ihre Bewegung in grobe Zielrichtung starten und diese bei Bedarf auf den finalen Pfad korrigieren können [5].

[5] vgl. hierzu [Hig02c]

4 eigener Lösungsansatz

4.1 allgemeine Struktur einer A* Engine

Die Grobstruktur der *A* Engine* ist [Hig02a] entnommen, letztendlich repräsentiert sie in dieser Detailstufe die genaue Abbildung des A* Algorithmus auf ein objektorientiertes Modell.

Neben dem A* Algorithmus selbst benötigt man einen *Speicher*, der die benötigten Daten enthält, eine *Suchanfrage*, die definiert, welches Suchproblem gelöst werden soll und schließlich den *Suchraum*, auf welchem die Suche vollzogen werden soll. Dieser besteht aus *Knoten*, die miteinander verbunden sein können. Abbildung 2 illustriert diese Struktur.

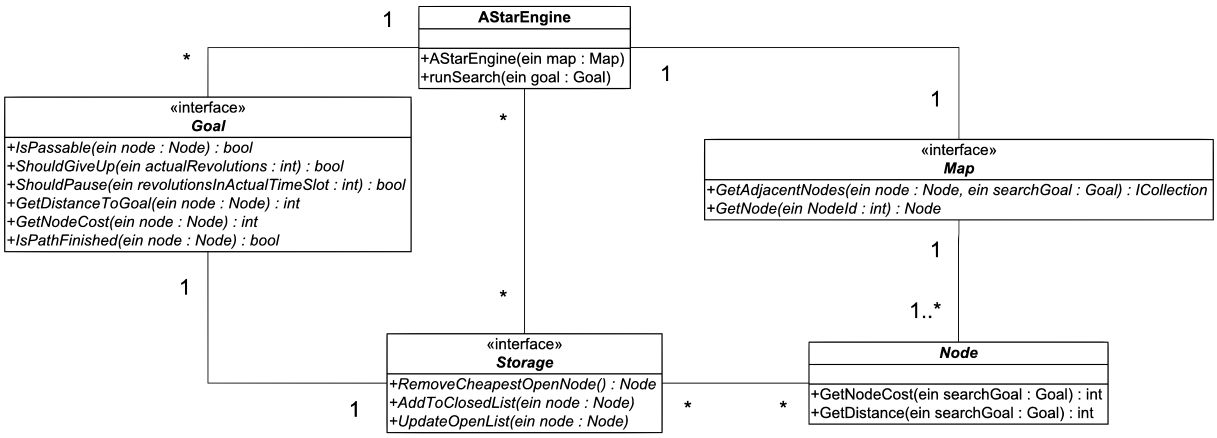

Abbildung 2: Klassendiagramm einer A* Engine

4.1.1 Speicher

(Storage) Der Speicher vereint die Funktionalität von *ClosedList* und *OpenList*, also die Menge der schon besuchten und noch zu besuchenden Knoten bei Bearbeitung einer Suchanfrage. Dafür sollte man immer Datenstrukturen verwenden, die genau auf die Laufzeitumgebung und das Suchproblem optimiert sind (vgl. 3.2).

Die benötigten Methoden sind:

- `RemoveCheapestOpenNode()` : entfernt den für die aktuelle Suchanfrage günstigsten Knoten [6] von der *OpenList* und gibt ihn zurück
- `UpdateOpenList(Node)` : fügt den Knoten zur OpenList hinzu bzw. aktualisiert diesen
- `AddToClosedList(Node)` : fügt den Knoten zur ClosedList hinzu
- `IsOnClosedList(Node)` : prüft, ob der Knoten bereits besucht wurde

[6] dieser ist im Allgemeinen durch die bei A* verwendete Funktion ($Gesamtkosten = KostendesKnotens + geschaetzteRestkostenzumZielknoten$) definiert

4.1.2 Suchanfrage

(Goal) Die Suchanfrage enthält alle Informationen darüber, wonach gesucht wird und unter welchen Bedingungen diese Suche vollzogen werden soll. Da die (Rest-)Kosten eines Knotens anfragespezifisch sind,befindet sich die Berechnung dieser auch hier und nicht, wie man vielleicht annnehmen sollte, in Methoden des Suchraums.

Die benötigten Methoden sind:

- `IsPassable(Node)` : prüft, ob der Knoten in den Pfad einbezogen werden kann
- `GetDistanceToGoal(Node)` : gibt die heuristischen Restkosten zum Ziel an
- `GetNodeCost(Node)` : gibt die Kosten dieses Knotens an
- `IsPathFinished(Node)` : prüft, ob der Zielknoten erreicht wurde
- `ShouldGiveUp(ActualRevolutions)` : prüft, ob die Höchstanzahl von Suchschritten erreicht wurde und die Suche abgebrochen werden soll
- `ShouldPause(RevolutionsInActualTimeSlot)` : prüft, ob die Höchstanzahl von Suchschritten im aktuellen Zeitfenster erreicht wurde und die Suche pausiert werden soll

4.1.3 Suchraum

(Map) Es existieren eine Vielzahl von Suchraumrepräsentationen , die je nach Problemstellung optimal eingesetzt werden können (vgl. 3.1). Abstrahiert man von der jeweiligen Implementierung, sucht man immer in einem Graphen.Der Algorithmus benötigt dabei lediglich die Information, welche Knoten zu einem bestimmten Knoten in direkter Nachbarschaft stehen. Zudem muss man auch direkt auf Knoten im Suchraum zugreifen können – z.B. zum Festlegen der Start- und Endknoten einer Suchanfrage. Selbstverständlich sollte es zudem noch Methoden zur Verwaltung des Suchraums geben (wie Anzahl der referenzierten Knoten oder eventuelles Caching von Knoten, wenn der Suchraum nicht komplett im Speicher gehalten werden kann), welche aber je nach Anforderungen an das System verschieden ausfallen können und deshalb hier nicht ausspezifiert werden. Für weiterführende Informationen sei hier nochmals verwiesen auf Abschnitt 3.1.

Die benötigten Methoden sind:

- `GetAdjacentNodes(Node)`: gibt alle Knoten in direkter Nachbarschaft zurück
- `GetNode(NodeId)`: gibt den Knoten mit der betreffenden Id zurück

4.1.4 Knoten

(Node) Er repräsentiert die Daten, welche im Suchraum strukturiert abgelegt sind. Zur effizienten Wiederverwendung der durch die Suchanfrage definierten Kosten beim Suchen

und Anordnen im Speicher sollten diese hier verwaltet werden. Neben den unten angegebenen Methoden, sollten an dieser Stelle alle notwendigen Attribute zu Unterstützung der Kostenfunktion und Heuristik bereitgestellt werden (siehe 3.3).

Die benötigten Methoden sind:

- `GetDistance(Goal)` : gibt die anfragespezifisch berechneten Restkosten an
- `GetNodeCost(Goal)` : gibt die anfragespezifisch berechneten Kosten an

4.2 Erweiterung für parallele Anfragebearbeitung

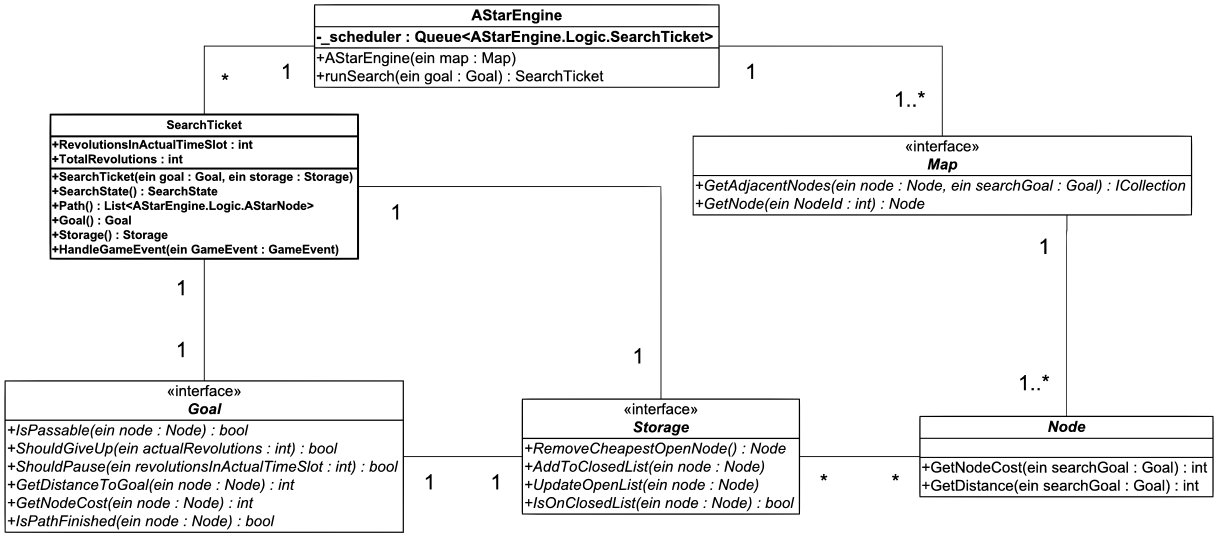

Abbildung 3: Klassendiagramm einer A* Engine, die parallele Anfragebearbeitung erlaubt

Aufbauend auf obige Struktur soll die Architektur nun erweitert werden, damit eine parallele Anfragebearbeitung möglich wird (siehe Abbildung 3). Dafür benötigt man zusätzlich eine Klasse, die den Zustand jeder Suchanfrage kapselt und einen Scheduling-Algorithmus, der verwaltet, in welcher Reihenfolge die wartenden Suchanfragen abgearbeitet werden.

4.2.1 Suchticket

(SearchTicket) Um der Bedeutung gerecht zu werden, dass diese Klasse einerseits einen Zustand kapselt, andererseits aber auch die Schnittstelle zwischen der *A* Engine* und der Komponente, die die Anfragen formuliert, darstellt, wurde von mir die Metapher des *Tickets* eingeführt.

Die Klasse kann folgende Zustände annehmen, deren Zusammenhang in Abbildung 4 aufgezeigt wird :

- CANCELED : die Suchanfrage wurde abgebrochen, bevor ein Pfad ermittelt werden konnte
- FAILED : es wurde kein Pfad gefunden

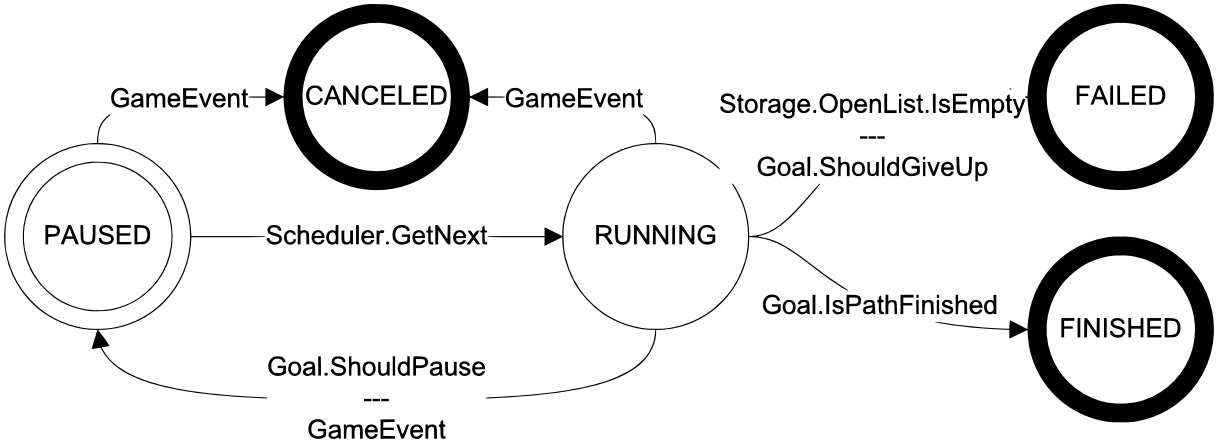

Abbildung 4: Zustände der Klasse SearchTicket

- FINISHED: die Suchanfrage wurde erfolgreich abgeschlossen (ein Pfad wurde gefunden)
- PAUSED: die Suchanfrage pausiert
- RUNNING : die Suchanfrage wird gerade bearbeitet

Die benötigten Eigenschaften sind:

- SearchState : repräsentiert den Zustand, in welchem sich die Suchanfrage momentan befindet
- Storage : referenziert den Speicher der Suchanfrage
- Goal : referenziert die Suchanfrage selbst
- Path : referenziert den Pfad, der ermittelt wurde, hier müssen geeignete Maßnahmen getroffen werden, damit der Zugriff nur im Zustand `FINISHED` gewährt wird
- TotalRevolutions : enthält die Gesamtanzahl von Suchschritten, die für diese Suchanfrage aufgewendet wurden
- RevolutionsInActualTimeSlot : enthält die Anzahl von Suchschritten, die zwischen dem letzten Wechsel vom Zustand `PAUSED` in `RUNNING` aufgewendet wurden

4.2.2 Scheduling

Der Schedulingalgorithmus verwaltet die Liste der Suchanfragen und legt fest in welcher Reihenfolge diese abgearbeitet werden. Man kann dabei keine pauschale Antwort darauf geben wie das Scheduling auszusehen hat. Grobe Ansätze für die Realisierung des Schedulingalgorithmus sollen aber zumindest kurz aufgezählt werden:

- Priorität von Anfragen, deren Pfad im sichtbaren Bereich beginnt
- Priorität von Anfragen, die vom Benutzer direkt erzeugt wurden

- Priorität von Anfragen, die einen kurzen Pfad erzeugen

Ein Hauptziel besteht zusammenfassend also darin, die Reaktionzeit der *A* Engine* merklich zu erhöhen.

5 Einsatzgebiete

Dieser Abschnitt beschreibt aufbauend auf allgemeinen Eigenschaften der *A* Engine* beispielhaft einige Anwendungsmöglichkeiten über die reine Wegfindung hinaus.[7].

Die *A* Engine* hat neben dem berechneten Pfad ebenfalls Zugriff auf die ClosedList des A* Algorithmus und auf alle Teile des Suchraumes; durch eine Überwachung des Suchfortschritts lässt sich zudem erfahren, in welcher Reihenfolge die Knoten während der Bearbeitung einer Suchanfrage expandiert wurden. Die Markierung gefundener Pfade (siehe 6.1, `Priority`) kann zusätzlich zur Verwendung bei der heuristischen Restkostenabschätzung für verschiedenste Anwendungsmöglichkeiten genutzt werden.

Terrainanalyse und Floodfill Die Daten der *ClosedList* können dazu benutzt werden zusammenhängende Regionen bestimmter Ausprägungen im Suchraum zu ermitteln (Terrainanalyse); die ermittelten Regionen können so z.B. als Grundlage zur Klassifizierung von Suchanfragen hergenommen werden, die über unüberwindbare Regionengrenzen gestellt werden. Mit einer zusätzlichen Überwachung der Reihenfolge von abgearbeiteten Knoten kann ebenso ein sog. Flutungseffekt (Floodfill) erzeugt werden [8].

Vorgehensweise ist hierbei immer Folgende:

1. Formulierung einer unschaffbaren Anfrage an die *A* Engine*, die in der zu analysierenden Region beginnt und Bewegung nur auf dem zu markierenden Terraintyp erlaubt
2. Warten auf den Rückgabewerte `FAILED` (bei ausschließlicher Verwendung der Menge aller durchsuchten Knoten kann aus Effizienzgründen auf eine Ordnung der OpenList verzichtet werden)
3. Weiterverarbeitung der Daten der ClosedList

Straßen- und Städtebau Die *A* Engine* bearbeitet eine Vielzahl von Anfragen und kann somit ohne Mehraufwand die Informationen bereits gefundener Pfade nutzen, um einerseits die Heuristik zu unterstützen (Knoten, die Bestandteil gefundener Pfade waren, werden sehr stark bevorzugt). Gerade im Bereich der Echtzeitstrategiespiele ist es meist notwendig Handelsrouten zu erschaffen, bzw. geeignete Plätze für die Errichtung von Siedlungen / Städten zu finden. Durch die simple Annahme, dass in Bereichen, die häufig Teil eines Pfades sind, explizit geeignete Wege erschaffen werden sollten und an

[7] Grundlage meiner Ausführung sind [Hig02a, S.114] und [Pat07]

[8] mit Hilfe einer geeigneten Visualisierung lassen sich so z.B. realistisch wirkende Überschwemmungseffekte o.ä. in der Spielumgebung simulieren

Kreuzungspunkten dieser wiederum ein geeigneter Platz für Siedlungen ist, kann man durch eine Markierung der Knoten einiges an Mehrwert über das reine Pfadfinden hinaus generieren.

Planung Anwendungen, die Methoden der künstlichen Intelligenz verwenden, benötigen meist auch eine allgemeine Komponente für die Planung. Letztendlich kann man die im Suchraum verwalteten Knoten mit beliebigen Informationen anreichern. Es existieren Verfahren, die eine spezifierte `Planning Domain` so verarbeiten, dass diese mit einer beliebigen heuristischen Suche bearbeitet werden kann. Für einen Einstieg in dieses komplexe Themengebiet sei z.B. [BG01] erwähnt.

6 Evaluierung

6.1 prototypische Implementierung

Aufgrund der zeitlichen Begrenzung einer Seminararbeit beschränkt sich meine Implementierung lediglich darauf meine spezifizierte Architektur prototypisch umzusetzen und damit aufzuzeigen, dass sie auch geeignet ist, um auf mobilen Endgeräten Verwendung zu finden. Um Anfragen an die *AStarEngine* zu stellen und das Suchverhalten beobachten zu können, wird auch eine rudimentäre GUI [9] angeboten.

Die genaue technische Umsetzung kann man dabei dem kommentierten Quellcode entnehmen, der meiner Arbeit beigelegt ist; dieser Abschnitt soll lediglich unterstützend grundlegende Teile der Implementierung erläutern.

GridMap4Adjancencies Vor allem aus Zeitgründen wurde von mir als Suchraumrepräsentation ein Gitter gewählt, welches intern als 2dimensionales Array mit den Maßen 100×100 verwaltet wird und auch keinerlei Optimierung der Speicherverwaltung bereitstellt. Von einem Knoten ist lediglich die 4 Nachbarschaft erreichbar, d.h. ein diagonaler Pfad ist nicht möglich.

AStarNode Aufgrund der Erweiterung für parallele Anfragebearbeitung kennt diese Klasse die Zustände `SINGLETHREADED` und `MULTITHREADED`, die jeweils optimiert auf eine bzw. mehrere parallele Suchanfragen sind. Zur Unterstützung von Kostenfunktion und Heuristik werden die Attribute `PathType` (Terraintyp, kann neben `FREE` und `BLOCKED` noch zusätzliche Werte annehmen) und `Priority` (speichert, wie häufig der Knoten Teil eines gefundenen Pfades war) verwaltet.

AStarEngine Hier werden die Suchanfragen formuliert und durchgeführt. Die Durchführung der Suche wird durch einen Hintergrundthread vollzogen; diese Umsetzung garantiert zum einen, dass das Laufzeitverhalten des gesamten Systems nicht zu stark von der

[9] Teile davon basieren auf einem frei verfügbaren anderen Projekt zur Implementierung des A* Algorithmus (`http://www.codeguru.com/csharp/csharp/cs_misc/designtechniques/print.php/c12527__2/`) , die von mir entnommene und modifizierte Klasse wurde dabei gesondert markiert

Suche beeinträchtigt wird, andererseits kann man so bei vorhandener technischer Unterstützung auch wirkliche Parallelität erzeugen [10]. Der zugrunde liegende A* Algorithmus ist der ursprünglich von Nilsson definierte ([Nil98, S.141f.], die Durchführung eines Suchschrittes steht in der Methode `revolute(SearchTicket ticket)`, die Generierung des Pfades in `SearchTicket.buildPath()`.

SimpleStorage Die Optimierungen aus 3.2 konnte ich aus Zeitgründen nicht realisieren, sondern musste auf Datenstrukturen ausweichen, welche im dotNet Compact Framework bereits vorhanden waren - für ClosedList benutze ich eine einfach verkettete Liste, für OpenList eine PriorityQueue, die ebenfalls auf einer einfach verketteten Liste aufbaut.

6.2 Laufzeitanalyse

Für die abschließende Laufzeitanalyse wurden von mir 2 Versionen erstellt, eine für ein mobiles Gerät und die andere für eine Desktopumgebung. Da das dotNet Compact Framework eine Teilmenge des dotNet Frameworks darstellt, konnte mein Code bis auf eine marginale Veränderung für die Umstellung auf eine Bedienung mit Maus (anstatt dem Stift für ein mobiles Gerät) ohne Änderungen am Quellcode portiert werden. Anschließend testete ich die Korrektheit beider Versionen mit Hilfe von gleichlautenden Testfällen (zu finden im Paket TestCasesmeiner Abgabe); auf empirische Analysen zur Geschwindigkeit verzichtete ich, da aussagekräftige Ergebnisse erst mit einer (zu diesem Zeitpunkt noch ausstehenden) Optimierung der Datenstrukturen erreicht würden. Aussagen zur geforderten effizienten Speichernutzung können von mir ebenfalls erst nach Implementierung einer optimierten Suchraumrepräsentation getätigt werden.

[10] multicore Architekturen würden so z.B. eine Ausführung auf allen zur Verfügung stehenden Kernen erlauben; diese technischen Grundlagen, welche in DesktopPCs sich mittlerweile etabliert haben,sind auf mobilen Endgeräten allerdings nicht vorhanden

Literatur

[BG01] Bonet, Blai ; Geffner, Hector: Planning as heuristic search. In: *Artificial Intelligence* 129 (2001), Nr. 1-2, 5-33. citeseer.ist.psu.edu/bonet01planning.html

[BMS04] Botea, Adi ; Müller, Martin ; Schaeffer, Jonathan: Near Optimal Hierarchical Path-Finding. In: *Journal of Game Development* Bd. 1. 2004, S. 7–28. – online verfügbar unter http://www.cs.ualberta.ca/~mmueller/ps/hpastar.pdf; zuletzt besucht am 22.06.2007

[Gar07] Gartner Press Release: *Gartner Says Worldwide Mobile Gaming Revenue to Grow 50 Percent in 2007.* Website, 2007. – online verfügbar unter http://www.gartner.com/it/page.jsp?id=507467; zuletzt besucht am 22.06.2007

[Hig02a] Higgins, Daniel: Generic A* Pathfinding. In: *Game Programming Wisdom* (2002), S. 114–121

[Hig02b] Higgins, Daniel: How to Achieve Lightning-Fast A*. In: *Game Programming Wisdom* (2002), S. 133–145

[Hig02c] Higgins, Daniel: Pathfinding Design Architecture. In: *Game Programming Wisdom* (2002), S. 122–132

[KM06] Kauko, Jarmo ; Mattila, Ville-Veikko: Mobile Games Pathfinding. In: *Proceedings of the Ninth Scandinavian Conference on Artificial Intelligence (SCAI 2006)*, 2006. – online verfügbar unter http://www.stes.fi/scai2006/proceedings/176-182.pdf; zuletzt besucht am 22.06.2007

[Nil98] Nilsson, Nils J.: *Artificial intelligence: a new synthesis.* San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1998. – ISBN 1–55860–467–7

[Pat07] Patel, Amit: Amit's Game Programming Information. (2007). http://www-cs-students.stanford.edu/~amitp/gameprog.html

[PB06] Pontevia, Pierre ; Baur, Alain: Kynapse 4.0 Large Scale A.I. / Kynogon. 2006. – Forschungsbericht. – online verfügbar unter http://www.kynogon.com/images-blog/Documents/WPG3large_scale_ai.pdf; zuletzt besucht am 22.06.2007

[Rab00a] Rabin, Steve: A* Aesthetic Optimizations. In: *Game Programming Gems* (2000), S. 264–271

[Rab00b] Rabin, Steve: A* Speed Optimizations. In: *Game Programming Gems* (2000), S. 272–287

[Rab02] Rabin, Steve: *AI Game Programming Wisdom.* Rockland, MA, USA : Charles River Media, Inc., 2002

[RN95] Russell, Stuart ; Norvig, Peter: *Artificial Intelligence: A Modern Approach.* Prentice-Hall, Englewood Cliffs, NJ, 1995

[Sto00] Stout, Bryan: The Basics of A* for Path Planning. In: *Game Programming Gems* (2000), S. 254–263