

A* Pfadfindung für Anfänger

Von Patrick Lester (Aktualisiert 18. Juli 2005)

Aus dem [Englischen](#) übersetzt von [Walter Weber-Groß](#) (19. Juli 2005)

Dieser Artikel ist auch in das [Chinesische](#), [Französische](#), [Portugiesische](#), [Russische](#) und [Spanische](#) übersetzt worden. Weitere Übersetzungen sind willkommen (siehe die E-Mail-Adresse am Ende dieses Artikels).

Der A* (sprich: A-Stern) Algorithmus kann für Anfänger kompliziert sein. Obwohl es viele Artikel im Web gibt, die A* erklären, so sind doch die meisten von ihnen für Leute geschrieben, welche die Grundlagen bereits verstehen. Dieser Artikel ist für "echte" Anfänger.

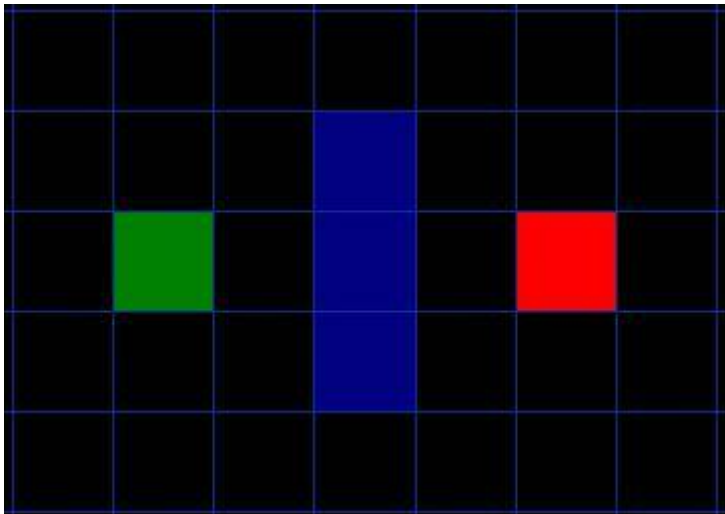
Dieser Artikel versucht nicht, "die" ultimative Arbeit zu diesem Thema zu sein, sondern beschreibt die nötigen Grundlagen um dafür gewappnet zu sein, die anderen Arbeiten zu diesem Thema verstehen zu können. Web-Verknüpfungen zu einigen der besten Arbeiten sind am Ende dieses Artikels unter "Weiterführende Artikel" zu finden.

Zu guter Letzt, dieser Artikel ist nicht programmspezifisch, Du solltest also in der Lage sein, das hier Beschriebene in jede Programmiersprache zu übersetzen. Dennoch habe ich - wie Du vielleicht erwartet hast -, am Ende des Artikels eine Webverbindung zu Programmbeispielen eingefügt. Dieses Beispieldpaket enthält zwei Versionen: Eine in C++ und eine in Blitz Basic; enthalten sind auch die entsprechenden ausführbaren Dateien, falls Du A* einfach mal in Aktion sehen möchtest.

Doch wir sind schon ein bisschen zu weit - lass uns zum Anfang zurück kehren ...

Einführung: Der Suchbereich

Angenommen, wir haben jemanden, der von Punkt A nach Punkt B möchte. Lass uns weiter annehmen, dass sich zwischen A und B eine Wand befindet, so wie es das Bild unten zeigt. Startpunkt A ist hier grün, Zielpunkt B ist rot und die blauen Quadrate zwischen A und B stellen die Wand dar.



[Bild 1]

Das Erste, das Du vermutlich bemerkst, ist die Unterteilung des Suchbereichs in ein quadratisches Gitter. Diese Vereinfachung des Suchbereichs ist der erste Schritt zur Pfadfindung. Dieses besondere Verfahren reduziert den Suchbereich auf eine einfache, 2-dimensionale Matrix. Jedes Element dieser Matrix repräsentiert eines der Quadrate im Gitter und sein Status kann entweder "begehrbar" oder "nicht begehrbar" sein. Der Pfad ergibt sich aus all den Quadraten, die wir für den Weg von A nach B benötigen. Sobald dieser Pfad gefunden ist, begibt sich die Spielfigur ausgehend vom Zentrum des Startquadrates A von einem Quadratzentrum zum nächsten und besucht auf diese Weise alle Quadrate des Pfades, bis es das Zentrum des Zielquadrates B erreicht hat.

Die Zentren der Quadrate werden Knoten genannt. Diesen Begriff wirst Du oft in Diskussionen über Pfadfindung wiederfinden. Doch warum belassen wir es nicht bei der Bezeichnung "Quadrate"? Einfach, weil es möglich ist, unseren Suchbereich auch in etwas anderes als Quadrate zu unterteilen; es können Rechtecke, hexagonale Felder, Dreiecke oder beliebige andere Formen sein. Die Knoten wiederum können überall innerhalb solch einer Form platziert sein: im Zentrum, an den Kanten oder sonst wo. Doch wir nehmen Quadrate, weil es die einfachste Form ist.

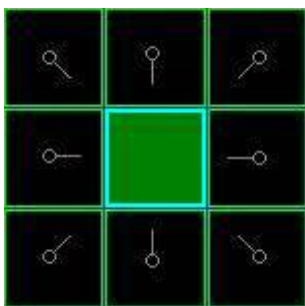
Die Suche beginnt!

Sobald wir den Suchbereich mit dem oben beschriebenen Quadratgitter in eine handhabbare Menge von Knoten gebracht haben, ist das nächste Ziel die Suche nach dem kürzesten Pfad einzuleiten. Wir tun dies, indem wir, beginnend bei Quadrat A, die umliegenden Quadrate eines Quadrates prüfen, um uns so suchend herauszubewegen, bis wir das Ziel, Quadrat B, gefunden haben.

Wir beginnen die Suche, indem wir Folgendes tun:

1. Beginne mit Startpunkt A und füge ihn einer "offenen" Liste von Quadraten hinzu. Diese Liste ist vergleichbar mit einer Einkaufsliste; zu diesem Zeitpunkt ist nur ein Element in der Liste, aber wir können weiter Elemente später hinzufügen. Sie enthält möglicherweise auch Quadrate, die nicht auf dem von Dir gewünschten Weg liegen. Aus diesem Grunde enthält diese Liste grundsätzlich Quadrate, die überprüft werden müssen.
2. Schaue Dir alle erreich- bzw. begehbaren Quadrate, die an das Startquadrat A grenzen, an, und ignoriere dabei Quadrate, welche Wände, Wasser oder anderes nicht begehbare Terrain darstellen. Füge sie der o.g. Liste hinzu und merke für jedes dieser Quadrate das Startquadrat A als seinen Vorgänger. Diese Beziehung eines Quadrates zu seinem Vorgänger wird später wichtig, wenn wir den Pfad entlanggehen wollen; dazu später mehr.
3. Wirf das Startquadrat A aus Deiner bisherigen, offenen Liste und füge es zu einer anderen, "geschlossenen" Liste von Quadraten hinzu, die Du Dir aber jetzt noch nicht anschauen brauchst.

An diesem Punkt solltest Du etwas haben, was der folgenden Darstellung entspricht. In dieser Darstellung ist das dunkle, grüne Quadrat in der Mitte das Startquadrat A. Es ist durch eine blaue Umrisslinie hervorgehoben, um anzuzeigen, dass dieses Quadrat der geschlossenen Liste hinzugefügt wurde. Alle an A angrenzenden Quadrate befinden sich nun in der offenen Liste und harren der Überprüfung, weshalb sie mit einer hellgrünen Umrandung versehen sind. Zusätzlich ist jedes dieser Quadrate mit einem grauen Zeiger versehen, der auf ihren jeweiligen Vorgänger - in diesem Fall das Startquadrat A - verweist.



[Bild 2]

Als Nächstes wählen wir eines der angrenzenden Quadrate in der offenen Liste und wiederholen mehr oder weniger den vorangegangenen Prozess, wie er weiter unten beschrieben ist. Aber welches Quadrat nehmen wir nun? Das mit den niedrigsten F-Kosten.

Pfadbewertung

Der Schlüssel dazu, welche Quadrate für den Pfad in Frage kommen, ist folgende Gleichung:

$$F = G + H$$

wobei

- ⌋ G = Die Bewegungskosten, um vom Startpunkt A zu einem gegebenen Quadrat des Gitters unter Verwendung des dafür ermittelten Pfades zu gelangen.
- ⌋ H = Die geschätzten Kosten, um von dem gegebenen Quadrat zum Zielpunkt B zu gelangen. Dies wird oft heuristisch genannt, was ein bisschen verwirrend sein kann. Der Grund, warum dies so genannt wird, ist, dass diese Kosten auf

Vermutung beruhen, denn tatsächlich kennen wir die wirkliche Entfernung erst, wenn wir den Pfad dorthin gefunden und auf dem Weg liegende Hindernisse (Wände, Wasser, etc.) berücksichtigt haben. In diesem Artikel wird ein möglicher Weg gezeigt, wie H ermittelt werden kann; es gibt aber viele weitere, die in anderen Web-Artikeln beschrieben sind.

Wir erzeugen unseren Pfad, indem wir wiederholt unsere offene Liste durchschreiten und das Quadrat mit den geringsten F-Kosten wählen. Dieser Prozess wird weiter unten im Einzelnen beschrieben. Zuerst lass uns aber etwas näher betrachten, wie wir die Gleichung berechnen.

Wie oben beschrieben, sind G die Bewegungskosten um vom Startpunkt zu einem gegebenen Quadrat unter Verwendung des dorthin ermittelten Pfades zu gelangen. In diesem Beispiel weisen wir jedem in horizontaler oder vertikaler Richtung beschrittenen Quadrat einen Kostenwert von 10 und jedem diagonal beschrittenen Quadrat einen Kostenwert von 14 zu. Wir verwenden diese Werte, da sich die (abgerundete) Länge der Diagonale in dem Quadrat nach dem Satz des Pythagoras ergibt: $14,1421... = \text{Quadratwurzel aus } (10^2 + 10^2)$. Wir verwenden 10 und 14 der Einfachheit halber. Dies hat die Vorteile, dass wir nicht Wurzeln und Dezimalzahlen berechnen müssen und dennoch das Verhältnis zwischen vertikalen bzw. horizontalen und diagonalen Bewegungskosten genau genug ist. Wir machen dies nicht, weil wir dumm sind oder Mathe nicht mögen. Der Computer rechnet mit Ganzzahlen einfach schneller. Zudem kann Pfadfindung ziemlich schnell sehr langsam werden, wenn wir nicht solche Vereinfachungen verwenden.

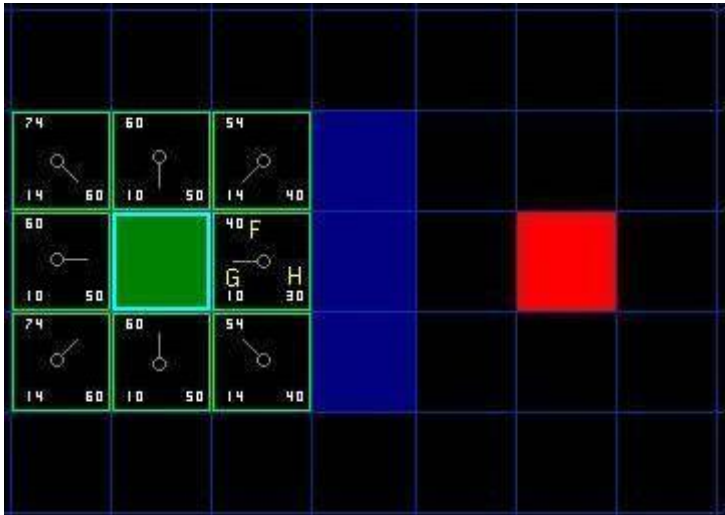
Da wir G entlang eines spezifischen Pfades zu einem gegebenen Quadrat berechnen, ermitteln wir G für dieses Quadrat, indem wir die G-Kosten seines Vorgängerquadrates nehmen und je nach Bewegungsrichtung 10 oder 14 addieren. Der Nutzen dieser Methode eröffnet sich ein bisschen später in diesem Beispiel, wenn wir uns mehr als ein Quadrat vom Startquadrat fortbewegen.

H kann auf vielerlei Art geschätzt werden. Die Methode, die wir hier verwenden, wird als "Manhattan-Methode" bezeichnet, mit der Du die Gesamtzahl von Quadraten, die in horizontaler oder vertikaler Richtung beschritten werden müssen, berechnest. Dabei werden diagonale Bewegungen und jedwede Hindernisse außer Acht gelassen. Wir multiplizieren die Summe dann mit 10. Dies wird (möglicherweise) die Manhattan-Methode genannt, weil sie dem Zählen von Stadtvierteln gleicht, wenn man von einem Platz zu einem anderen möchte und es keine diagonale Verbindung gibt.

Wenn Du diese Beschreibung liest, könntest Du auf den Gedanken kommen, dass die Heuristik hauptsächlich eine grobe Schätzung der verbleibenden Entfernung zwischen dem aktuellen Quadrat und dem Ziel in Luftlinie ist. Dies ist nicht der Fall. Wir versuchen tatsächlich die verbleibende Distanz entlang des Pfades zu schätzen (die normalerweise größer ist). Je näher unsere geschätzte Entfernung der tatsächlich verbleibenden Distanz ist, umso schneller wird der Algorithmus sein. Gleichwohl, wenn wir diese Distanz überschätzen, ist nicht garantiert, dass wir den kürzesten Pfad ermitteln. In solch einem Fall haben wir etwas, das "unzulässige Heuristik" genannt wird.

Technisch gesehen ist die Manhattan-Methode für dieses Beispiel unzulässig, da sie die verbleibende Distanz etwas überschätzt. Wir werden sie dennoch verwenden, da sie sehr viel leichter für unsere Zwecke zu verstehen ist und weil es nur eine leichte Überschätzung ist. In dem seltenen Fall, wo der resultierende Pfad nicht der kürzest mögliche ist, wird er dennoch nahezu kurz sein. Willst Du mehr darüber wissen? Dann findest Du Gleichungen und zusätzliche Anmerkungen [hier](#).

F wird berechnet, indem G und H addiert werden. Die Ergebnisse des ersten Schrittes unserer Suche sind in dem Bild unten dargestellt. Die F, G und H-Werte stehen in jedem Quadrat. Wie in dem Quadrat rechts des Startquadrats angezeigt ist, befindet sich F in der oberen linken Ecke, G in der unteren linken Ecke und H in der unteren rechten Ecke.



[Bild 3]

Betrachten wir nun einige dieser Quadrate. In dem Quadrat, in dem sich die Buchstaben befinden, ist $G = 10$, weil es genau ein Quadrat vom Startquadrat in horizontaler Richtung entfernt ist. Die Quadrate unmittelbar über, unter und links vom Startquadrat haben aus diesem Grunde alle denselben G-Wert von 10. Die diagonal liegenden Quadrate haben alle den G-Wert 14.

Die H-Werte ergeben sich aus der "Manhattan-Distanz" zum roten Zielquadrat. Diese Distanz wird, wie weiter oben beschrieben, überwunden, indem man sich ausschließlich in horizontaler oder vertikaler Richtung zum Ziel bewegt, ohne irgendwelche Hindernisse auf diesem Weg zu berücksichtigen. Mit dieser Methode ergibt sich für das Quadrat unmittelbar rechts neben dem Startquadrat eine Entfernung von 3 Quadraten zum Zielquadrat, was einem H-Wert von 30 entspricht. Entsprechend ergibt sich für das Quadrat oberhalb des gerade betrachteten Quadrats ein H-Wert von 40, da hier zusätzlich noch ein weiteres Quadrat in vertikaler Richtung beschritten werden müsste. Nun erschließt sich für Dich wahrscheinlich, wie sich die H-Werte der übrigen Quadrate um das Startquadrat herum ergeben.

Der F-Wert ergibt sich, wie schon erwähnt, einfach aus der Addition von G- und H-Wert.

Fortsetzung der Suche

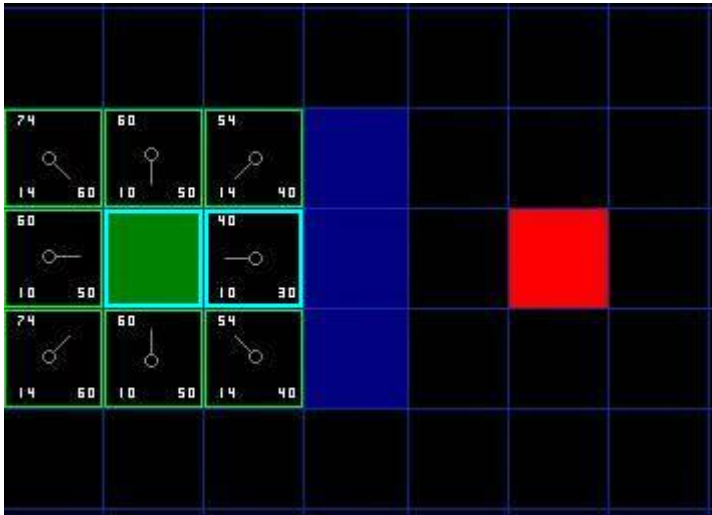
Um die Suche fortzusetzen, wählen wir aus der offenen Liste einfach das Quadrat mit dem niedrigsten F-Wert. Mit diesem (aktuellen) Quadrat machst Du Folgendes:

4. Entferne es aus der offenen Liste und füge es der geschlossenen Liste hinzu.
5. Prüfe alle angrenzenden Quadrate und füge sie der offenen Liste hinzu, sofern sie:
 - | kein Hindernis darstellen, also begehbar sind
 - | sich nicht bereits in der offenen Liste befinden
 - | sich nicht in der geschlossenen Liste befinden

und trage für jedes dieser Quadrate das aktuelle Quadrat als Vorgängerquadrat ein.

6. Falls eines der umliegenden Quadrate sich bereits in der offenen Liste befindet, prüfe, ob der Pfad vom aktuellen Quadrat zu solch einem Quadrat ein besserer ist, mit anderen Worten, ob der G-Wert für dieses Quadrat geringer wäre, wenn wir vom aktuellen Quadrat aus dorthin gehen würden. Wenn nicht, unternimm gar nichts. Falls jedoch die G-Kosten dieses Quadrates geringer würden, wenn wir vom aktuellen Quadrat aus dorthin gehen würden, ändere das Vorgängerquadrat dieses Quadrats auf das aktuelle Quadrat (richte dann den Zeiger im Diagramm oben auf das aktuelle Quadrat). Schließlich berechne sowohl den F- als auch den G-Wert dieses Quadrats neu. Falls dies etwas verwirrend sein sollte, findest Du eine Darstellung weiter unten.

Ok, mal sehen, wie das nun funktioniert; von unseren anfänglichen 9 Quadraten verbleiben 8 in der offenen Liste, nachdem wir das Startquadrat in die geschlossene Liste übertragen haben. Von diesen ist das mit den niedrigsten F-Kosten das unmittelbar rechts vom Startquadrat. Seine F-Kosten betragen 40. Also wählen wir dieses Quadrat als unser nächstes aktuelles Quadrat. Es ist im folgenden Bild hellblau umrandet.



[Bild 4]

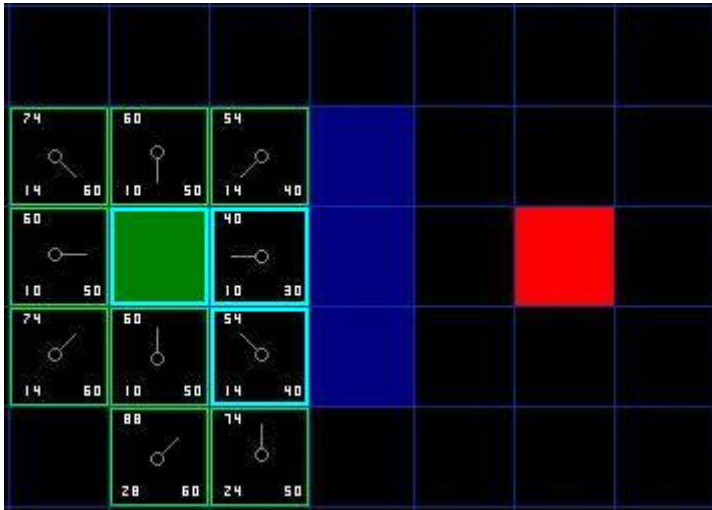
Zuerst entfernen wir es aus unserer offenen Liste und fügen es unserer geschlossenen Liste hinzu (darum ist es hellblau umrandet). Dann prüfen wir seine angrenzenden Quadrate. Gut, die Quadrate zur Rechten sind alles Wandquadrate, wir können sie also ignorieren. Das Quadrat zur unmittelbar Linken ist das Startquadrat. Da es sich in der geschlossenen Liste befindet, können wir auch dieses Quadrat ignorieren.

Die anderen vier Quadrate sind bereits in der offenen Liste, also müssen wir prüfen, ob die Pfade zu diesen Quadraten besser sind, wenn wir vom aktuellen Quadrat dorthin gehen. Zur Beurteilung verwenden wir den G-Wert eines jeden Quadrats. Wenn wir uns das Quadrat genau über unser unserem aktuellen Quadrat betrachten, merken wir uns seinen G-Wert von 14. Würden wir nun vom aktuellen Quadrat zu diesem Quadrat gehen, wäre dessen G-Wert 20 (10, um zum aktuellen Quadrat zu gelangen, plus weitere 10, um dann vom aktuellen Quadrat zu diesem oberen Quadrat zu gelangen). 20 ist größer als 14, also ist dies bestimmt nicht der günstigere Pfad. Das ergibt sich auch anschaulich aus dem Bild, denn es ist kürzer, vom Startquadrat direkt, also diagonal, als rechtwinklig über unser aktuelles Quadrat zu diesem Quadrat zu gehen.

Wenn wir diesen Prozess für alle 4 angrenzenden Quadrate, die ja bereits in der offenen Liste sind, wiederholen, erkennen wir, dass keiner dieser Pfade dadurch verbessert wird, dass wir über unser aktuelles Quadrat gehen; also ändern wir gar nichts. Da wir nun alle angrenzenden Quadrate geprüft haben, sind wir mit dem aktuellen Quadrat fertig und bereit, auf unser nächstes Quadrat zu gehen.

Dazu betrachten wir unsere nunmehr auf 7 Quadrate geschrumpfte offene Liste und wählen aus diesen wieder das mit dem geringsten F-Kosten. Interessanterweise gibt es in diesem Fall zwei Quadrate mit einem F-Wert von 54. Welches nehmen wir nun? Es ist tatsächlich egal. Aus Gründen der Geschwindigkeit könnte es besser sein, das der offenen Liste zuletzt hinzugefügte zu nehmen. Dies richtet die Suche zugunsten zuletzt hinzugefügter Quadrate aus, sobald Du näher am Ziel bist, aber es ist nicht wirklich wichtig (unterschiedliche Behandlungen von Verbindungen ist aber der Grund dafür, warum zwei Versionen des A*-Algorithmus unterschiedliche Pfade gleicher Länge finden können).

Nehmen wir also das Quadrat, das sich genau unter dem aktuellen Quadrat und rechts unterhalb des Startquadrats befindet, wie im folgenden Bild gezeigt.

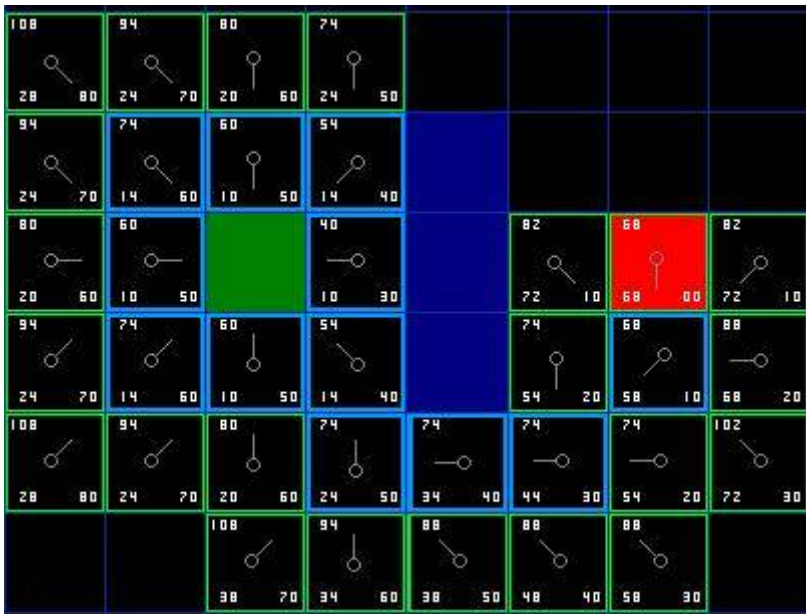


[Bild 5]

Wenn wir nun die angrenzenden Quadrate prüfen, sehen wir, dass das Quadrat zur Rechten ein Wandquadrat ist, wir können es also ignorieren. Das gleiche gilt für das Quadrat darüber. Wir ignorieren ebenfalls das Quadrat darunter. Warum? Weil wir nicht direkt (also diagonal) zu diesem Quadrat gelangen können, da wir sonst die Wand schneiden würden. Wir müssten also um die Ecke, d.h., vom aktuellen Quadrat aus zuerst nach unten und dann nach rechts gehen, um dieses Quadrat zu erreichen (Hinweis: Diese Regel ist kein Muss, sondern hängt davon ab, wie Deine Knoten platziert sind).

Somit bleiben fünf andere Quadrate übrig. Die anderen beiden Quadrate unterhalb des aktuellen Quadrats sind bereits in der offenen Liste, deshalb ändern wir deren Vorgängerquadrat auf das aktuelle Quadrat. Von den anderen drei Quadraten sind zwei bereits in der geschlossenen Liste (das Startquadrat und das Quadrat genau über dem aktuellen Quadrat, beide im Bild hellblau umrahmt), auch sie können wir ignorieren. Bleibt das letzte Quadrat, genau links vom aktuellen Quadrat, für das wir prüfen, ob sich seine G-Kosten verringern, wenn wir vom aktuellen Quadrat dorthin gehen würden. Klar, eine Antwort erübrigt sich. Wir sind also fertig mit dem aktuellen Quadrat und können nun das nächste Quadrat in der offenen Liste prüfen.

Wir wiederholen diesen Prozess, bis wir das Zielquadrat in unsere geschlossene Liste eintragen. An diesem Punkt sieht das Ganze wie im folgenden Bild aus:

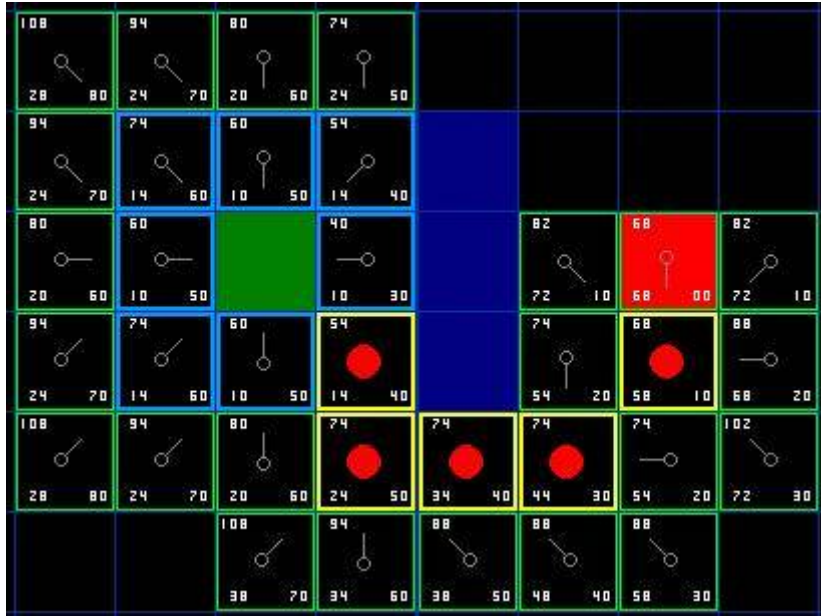


[Bild 6]

Beachte, dass sich das Vorgängerquadrat für das Quadrat zwei Quadrate unterhalb des Startquadrats von dem vorherigen Bild unterscheidet. Vorher hatte es einen G-Wert von 28 und zeigte zurück auf das Quadrat rechts darüber. Nun hat es einen G-Wert von 20 und zeigt auf das Quadrat direkt darüber. Diese Änderung geschah irgendwann während der Suche, als sich beim Prüfen des G-Wertes herausstellte, dass er dann geringer wurde, wenn der Pfad zu ihm über das zu dem Zeitpunkt aktuelle Quadrat gewählt wurde. Also wurden das Vorgängerquadrat geändert und G sowie F neu berechnet. Obwohl diese

Veränderung in diesem Beispiel unbedeutend zu sein scheint, gibt es doch viele mögliche Situationen, in denen solch eine Veränderung auf Grund der konstanten Überprüfung den wesentlichen Unterschied beim Finden des besten Pfades darstellt.

Wie bestimmen wir nun den Pfad? Einfach, indem wir vom roten Zielquadrat aus den Richtungspfeilen Quadrat für Quadrat rückwärts folgen. Dies führt Dich schließlich zum Startquadrat und das ist dann Dein Pfad. Es sollte wie das nachfolgende Bild aussehen. Das Gehen von Startquadrat A zum Zielquadrat B ist dann nur noch eine Angelegenheit des Bewegens vom Zentrum (dem Knoten) eines Quadrats zum Knoten des nächsten Pfad-Quadrats bis das Ziel erreicht ist.



[Bild 7]

Zusammenfassung der A*-Methode

Gut. Jetzt, wo Du Dich durch die Erklärung gearbeitet hast, wollen wir die einzelnen Schritte noch einmal zusammenfassen:

- 1) Füge das Startquadrat der offenen Liste hinzu.
- 2) Wiederhole das Folgende:
 - a) Suche in der offenen Liste nach dem Quadrat mit dem niedrigsten F-Wert. Wir bezeichnen dieses Quadrat im Folgenden als das aktuelle Quadrat.
 - b) Verschiebe es in die geschlossene Liste.
 - c) Für jedes der 8 an das aktuelle Quadrat angrenzenden Quadrate:
 - ┆ Wenn es nicht begehbar ist oder sich bereits in der geschlossenen Liste befindet, ignoriere es; andernfalls mach das Folgende:
 - ┆ Wenn es nicht in der offenen Liste ist, füge es der offenen Liste hinzu. Trage das aktuelle Quadrat als Vorgängerquadrat dieses Quadrats ein. Trage zusätzlich die Werte für die F-, G- und H-Kosten dieses Quadrates ein.
 - ┆ Falls es bereits in der offenen Liste ist, prüfe, ob der Pfad vom aktuellen Quadrat zu ihm - gemessen am G-Wert -, besser ist, als der Pfad von seinem eingetragenen Vorgängerquadrat (ein geringerer G-Wert bedeutet einen besseren Pfad). Falls dem so ist, ändere sein Vorgängerquadrat auf das aktuelle Quadrat und berechne seine Werte für G und F neu. Sofern Du Deine offene Liste nach dem F-Wert sortiert hast, ist möglicherweise eine Neusortierung dieser Liste erforderlich, um dieser Veränderung Rechnung zu tragen.
 - d) Beende den Prozess, falls:
 - ┆ Du das Zielquadrat in die geschlossene Liste verschoben hast; in diesem Fall hast Du den Pfad ermittelt
 - ┆ kein Zielquadrat gefunden werden konnte und die offene Liste leer ist; in diesem Fall gibt es keinen Pfad.
- 3) Sichere den Pfad. Der Pfad erschließt sich, indem Du vom Zielquadrat aus Quadrat für Quadrat rückwärts schreitend das Startquadrat erreichst.

Etwas Gelaber

Du magst mir verzeihen, wenn ich etwas abschweife, aber es scheint mir sinnvoll, darauf hinzuweisen, dass es in verschiedenen Diskussionen über die A*-Methode im Web und in den zahlreichen Foren gelegentlich jemanden gibt, der sich mit seinem Quelltext-Beitrag auf die A*-Methode bezieht, obwohl es nichts damit zu tun hat. Um die A*-Methode zu verwenden, werden die Elemente benötigt, die oben diskutiert werden - speziell offene und geschlossene Listen und die Pfadbewertung über F-, G- und H-Werte. Es gibt eine Menge anderer Pfadfindungsalgorithmen, aber diese Methoden sind eben nicht die A*-Methode, welche allgemein als die beste von allen betrachtet wird. In seinem Artikel, der am Ende dieses Artikels erwähnt wird, diskutiert Bryan Stout viele von ihnen, einschließlich einiger ihrer Pros und Contras. Manchmal sind alternative Methoden unter bestimmten Umständen besser, aber Du solltest dann auch wissen, warum. Gut, genug geschwätzt; nun zurück zum Artikel.

Anmerkungen zu Implementierung

Jetzt, da Du die grundlegende Methode verstehst, gibt es noch einige zusätzliche Dinge, an die es zu denken gilt, wenn Du Deine eigenen Programme schreibst. Einige der folgenden Materialien beziehen sich auf die Programme, die ich in C++ und Blitz Basic geschrieben habe, aber sie gelten ebenso für in anderen Sprachen geschriebene Programme.

1. Andere Einheiten (Kollisionsvermeidung): Wenn Du Dir meine Beispielquelltexte genau ansiehst, wirst Du bemerken, dass dort andere Einheiten auf dem Bildschirm vollständig ignoriert werden. Die Einheiten bewegen sich einfach durch andere Einheiten hindurch. Abhängig vom Spiel kann das akzeptabel sein oder eben nicht. Falls Du andere Einheiten während der Pfadfindung berücksichtigen möchtest, so dass sie einander ausweichen, schlage ich vor, dass Du nur die Einheiten berücksichtigst, die entweder still oder zu dem Zeitpunkt, an dem der Pfad berechnet wird, direkt neben der "Pfadfindungseinheit" stehen, so dass Du deren Quadrate als unbegehrbar betrachten kannst. Für angrenzende Einheiten, die sich bewegen, kannst Du die Gefahr von Kollisionen verringern, indem Du die Knoten, die sich entlang der entsprechenden Pfade befinden, benachteiligst und so der Pfadfindungseinheit die Möglichkeit gibst, eine alternative Route zu finden (mehr dazu unter 2.).

Falls Du andere sich bewegende Einheiten berücksichtigen willst, die sich nicht in der Nähe der Pfadfindungseinheit befinden, wirst Du eine Methode entwickeln müssen um vorherzusagen, wann sie sich zu einem bestimmten Zeitpunkt wo befinden, so dass die Einheiten einander ausweichen können. Andernfalls ergeben sich möglicherweise seltsame Pfade, auf denen sich Einheiten im Zick-Zack bewegen, nur um anderen Einheiten auszuweichen, mit denen sie gar nicht mehr kollidieren können, weil diese sich längst woanders befinden.

Ebenso wirst Du natürlich Methoden zur Kollisionsentdeckung entwickeln müssen, denn egal, wie gut der Pfad zur Berechnungszeit ist, so verändern sich doch die Dinge mit der Zeit. Sobald eine Kollision zu erwarten ist, muss entweder ein neuer Pfad berechnet werden, oder eine Einheit muss, sofern sich die andere Einheit nicht frontal nähert, kurz beiseite treten, um nach Passieren der anderen Einheit ihren Weg fortzusetzen.

Diese Hinweise sind möglicherweise genug, damit Du endlich loslegen kannst. Wenn Du mehr zu diesem Thema wissen möchtest, sind vielleicht folgende Seiten nützlich:

- | [Steering Behavior for Autonomous Characters](#) : Craig Reynold's Arbeit zur Steuerung unterscheidet sich eine bisschen von der Pfadfindung, kann aber in die Pfadfindung integriert werden, um das Bewegungs- und Kollisionsvermeidungssystem zu vervollständigen.
- | [Long and Short of Steering in Computer Games](#) : Eine interessante Studie der Literatur über Steuerung und Pfadfindung. Dies ist eine PDF-Datei.
- | [Coordinated Unit Movement](#) : Der erste Teil einer zweiteiligen Serie von Artikeln über Formations- und gruppenbasierte Bewegung vom Age of Empires-Designer Dave Pottinger.
- | [Implementing Coordinated Movement](#) Der zweite Teil in Dave Pottingers zweiteiliger Serie.

2. Variable Terrain-Kosten: In diesem Tutorial und meinem begleitenden Programm ist Terrain entweder begehbar oder nicht begehbar. Was aber, wenn Du Terrain hast, das zwar begehbar, aber höhere Bewegungskosten hat? Sumpf, Hügel, Treppen in einem Verlies, etc., dies alles sind Beispiele für Terrain, das zwar begehbar ist, aber höhere Bewegungskosten hat als flacher, offener Boden. In ähnlicher Weise hat z.B. eine Straße geringere Bewegungskosten als das umgebende Gelände.

Dieses Problem wird einfach dadurch gelöst, dass Du bei der Berechnung der G-Kosten die Terrainkosten hinzurechnest. Addiere einfach zusätzliche Kosten für solche Knoten. Der A*-Pfadfindungsalgorithmus ist bereits so gestaltet, dass der den Pfad mit den geringsten Kosten findet, so dass er dieses Problem leicht mit berücksichtigen kann. In dem einfachen Beispiel, das ich beschrieben habe, in dem Terrain also nur begehbar oder unbegehrbar ist, wird A* nach dem kürzesten, direktesten Pfad suchen. Aber in einer kostenvariablen Landschaft wird der kostengünstigste Pfad möglicherweise länger als der direkteste Pfad sein, da zum Beispiel die Straße um einen Sumpf herum genommen wird, statt den Weg direkt durch den Sumpf zu gehen.

Eine interessante, zusätzliche Betrachtung ist etwas, das die Profis "Einflusskartierung" (engl.: influence mapping) nennen. Ähnlich wie bei den oben beschriebenen variablen Terrain-Kosten könntest Du ein zusätzliches Punktesystem bei der Pfadfindung hinsichtlich der KI anwenden. Stell Dir vor, Du hast eine Karte mit einer Gruppe von Einheiten, die einen Pass verteidigen, der durch eine Gebirgsregion verläuft. Jedesmal, wenn der Computer eine Einheit auf den Weg durch diesen Pass schickt, wird diese Einheit vernichtet. Du könntest nun eine Einflusskartierung erstellen, welche die Kosten für solche Knoten erhöht, bei denen hoher Schaden zu erwarten ist. Dies würde den Computer veranlassen, sicherere Pfade zu wählen, und ihm helfen, solch dumme Situation zu vermeiden, in denen Einheiten in den sicheren Tod geschickt werden, nur weil der Pfad zum nicht erreichbaren Ziel kürzer ist ...

Eine andere Verwendung für die Kostenerhöhung von Knoten betrifft solche Knoten, die auf Pfaden von sich beieinander bewegend Einheiten liegen. Einer der Kehrseiten von A* ist, dass, wenn eine Gruppe von Einheiten versucht, gleichzeitig einen Pfad zu benachbarten oder identischen Zielen zu finden, es normalerweise eine signifikante Überschneidung gibt, da eine oder mehrere Einheiten versuchen, den selben oder ähnliche Wege zum Ziel zu benutzen. Wenn nun die Kosten für die von einer oder mehreren Einheiten bereits beschrittenen Knoten erhöht werden, kann dies helfen, einen bestimmten Grad der Trennung zu sichern und die Anzahl der Kollisionen zu verringern. Jedoch sollten solche Knoten nicht als unbegehrbar gekennzeichnet werden, da es schon sein kann, dass sich viele Einheiten in einem Schub durch enge Passagen quetschen müssen. Es sollten auch nur solche Pfade mit höheren Kosten belegt werden, die sich nahe der Pfadfindungseinheit befinden, nicht etwa alle, da Du sonst das seltsame Gebaren von Einheiten erlebst, die Frontalzusammstöße mit Einheiten verursachen, weil sie anderen Einheiten ausweichen möchten, die sich aber zu diesem Zeitpunkt überhaupt nicht in ihrer Nähe befinden. Des Weiteren solltest Du auch nur Knoten "bestrafen" - also mit höheren Kosten belegen -, die noch zu beschreiten sind; also nicht jene, die auf dem Weg zum Ziel bereits beschritten wurden.

3. Behandlung unerforschter Gebiete: Hast Du jemals ein PC-Spiel gespielt, in dem der Computer stets ganz genau wusste, welcher Pfad zu nehmen ist, obwohl die Karte noch nicht erforscht war? Je nach Spiel kann zu gute Pfadfindung unrealistisch wirken. Glücklicherweise kann dieses Problem ziemlich einfach gehandhabt werden.

Die Antwort auf dieses Problem ist, einen separaten "Begehrbarkeitsbereich" für jeden der verschiedenen Spieler und Computergegner zu erstellen (wohl gemerkt: nicht für jede Spielfigur - dies würde erheblich mehr Arbeitsspeicher erfordern). Jeder dieser Bereiche würde Informationen darüber enthalten, welche Gebiete der Spieler bereits erforscht hat; dabei wird für den Rest der Landschaft die Begehrbarkeit solange angenommen, wie sich durch Erkundung das Gegenteil noch nicht herausgestellt hat. Mit diesem Ansatz geraten Einheiten auch in Sackgassen und treffen ähnlich falsche Entscheidungen, bis sie durch Erkunden den richtigen Weg kennen gelernt haben. Sobald die Karte aber erkundet ist, läuft die Pfadfindung normal.

4. Schönerer Pfad: Zwar erzeugt A* automatisch den kürzesten, kostengünstigsten Pfad, aber dieser Pfad ist nicht automatisch der schönste Pfad. Ein Blick auf den Pfad in Bild 7 zeigt als ersten Schritt den zum Quadrat rechts unterhalb des Startquadrats. Würde es nicht schöner aussehen, wenn der erste Schritt auf das Quadrat direkt unterhalb des Startquadrats erfolgte?

Es gibt diverse Wege um dieses Problem anzugehen. Während Du den Pfad berechnest, könntest Du einen Knoten "bestrafen", falls dort eine Richtungsänderung erfolgt, indem Du seinem G-Wert einen Malus hinzufügst. Alternativ könntest Du nach Berechnung des Pfades diesen durchlaufen und nach benachbarten Knoten schauen, die diesen Pfad besser ausschauen lassen. Um mehr zu diesem Thema zu erfahren, schaue Dir folgende Seite im Web an: [Toward More Realistic Pathfinding](#), ein (freier, aber registrierungspflichtiger) Artikel bei Gamasutra.com von Marco Pinter.

5. Nichtquadratische Suchbereiche: In unserem Beispiel verwendeten wir ein schlichtes, zweidimensionales Feld, bestehend aus Quadraten. Du bist nicht an diesen Ansatz gebunden, denn Du könntest unregelmäßig geformte Bereiche verwenden. Nimm z.B. das Spiel "Risiko" mit seinen Ländern. Du könntest Dir ein Pfadfindungsszenario für ein Spiel wie dieses ersinnen; um dies zu tun, bräuchtest Du eine Tabelle, welche die Informationen darüber speichert, welche Länder aneinander grenzen, und entsprechende G-Kosten, die mit der Bewegung von einem Land zum anderen verbunden sind. Weiterhin würdest Du eine Methode benötigen, um H zu ermitteln. Alles weitere könnte so wie in dem Beispiel oben behandelt werden. Statt Quadraten würdest Du einfach nach aneinander grenzenden Ländern in der Tabelle schauen, wenn Du neue Elemente in Deine offene Liste schiebst.

Auf ähnliche Weise könntest Du ein Wegepunktesystem auf einer fixen Landkarte erstellen. Wegepunkte sind allgemein besuchte Punkte eines Pfades, z.B. auf einer Straße oder einem Hauptgang in einem Verlies. Als Spiele-Designer könntest Du diese Wegepunkte vorher zuweisen. Zwei Wegepunkte würden als angrenzend betrachtet werden, wenn kein Hindernis auf direktem Weg zwischen ihnen läge. Wie bei dem "Risiko"-Beispiel würdest Du diese Information in etwas wie einer Nachschlagetabelle speichern und auswerten, sobald Du ein neues Element in Deine offene Liste einträgst. Zusätzlich würdest die damit verbundenen G-Kosten (ermittelt z.B. durch die Luftlinienentfernung zwischen beiden Knoten) und H-Kosten (ermittelt z.B. durch die Luftlinienentfernung vom Knoten zum Ziel) eintragen. Alles andere würde wie schon beschrieben durchgeführt.

Amit Patel hat einen kurzen [Artikel](#) verfasst, in dem er sich eingehender mit einigen Alternativen befasst. Ein anderes Beispiel für das Suchen auf einer isometrischen Rollenspielkarte unter Verwendung eines nichtquadratischen Suchbereichs zeige ich in meinem Artikel [Two-Tiered A* Pathfinding](#).

6. Einige Tipps zur Geschwindigkeit: Wenn Du Dein eigenes A*-Programm schreibst oder das übernimmst, das ich geschrieben habe, wirst Du schließlich bemerken, dass Pfadfindung ein gerüttelt Maß an Prozessorzeit benötigt, insbesondere, wenn Du eine anständige Zahl von Pfadfindungseinheiten auf dem Plan hast und die Karte eine vernünftige Größe hat. Wenn Du die Beiträge im Netz liest, wirst Du sehen, dass dies auch für die Profis gilt, die Spiele wie Starcraft oder Age of Empires entwickeln. Sobald Du siehst, dass sich der Ablauf verlangsamt, findest Du hier einige Ideen, wie sich die Dinge beschleunigen lassen:

- | Ziehe in Betracht, eine kleinere Karte oder weniger Einheiten zu nehmen.
- | Führe die Pfadfindung nicht für mehr als ein paar Einheiten zur gleichen Zeit durch. Stattdessen packe sie in eine Warteschlange und verteile sie über mehrere Spielzyklen. Wenn Dein Spiel bei, sagen wir mal, 40 Zyklen pro Sekunde läuft, wird es keiner jemals merken. Aber man wird es merken, wenn das Spiel ab und zu in die Knie geht, sobald eine Gruppe von Einheiten alle zur gleichen Zeit ihre Pfade berechnen.
- | Ziehe in Betracht größere Quadrate (oder welche Form auch immer Du verwendest) für Deine Karte zu verwenden. Dies reduziert die Gesamtanzahl von Knoten, die bei der Padsuche angefasst werden müssen. Wenn Du sehr ambitioniert bist, kannst Du auch zwei oder mehr Pfadfindungssysteme erfinden, die in verschiedenen Situationen, abhängig von der Länge des Pfades, angewendet werden können. Dies ist das, was die Profis tun: große Bereiche für lange Pfade zu verwenden, um dann unter Verwendung kleinerer Quadrate/Bereiche zur verfeinerten Suche umzuschalten, sobald sie sich in der Nähe des Ziels befinden. Falls Du an diesem Konzept interessiert bist, schaue Dir doch meinen Artikel [Two-Tiered A* Pathfinding](#) an.
- | Für längere Pfade bietet es sich auch an, vorberechnete Pfade zu verwenden, die fest ins Spiel verdrahtet sind
- | Versuche, Deine Karte "vorzukompilieren", um die Bereiche herauszufiltern, die vom Rest der Karte unerreichbar sind. Ich nenne solche Bereiche "Inseln". In der Wirklichkeit können es wirkliche Inseln sein oder jeder andere Bereich, der auf die eine oder andere Weise unerreichbar ist. Einer der Kehrseiten von A* ist, dass, wenn Du damit einen Pfad zu solch einem Bereich suchst, die ganze Karte durchsucht wird und die Suche erst dann beendet wird, wenn jeder erreichbare Knoten durch die offene und geschlossene Liste gewandert ist. Das kann ziemlich viel Prozessorzeit verschwenden. Dem kann aber dadurch vorgebeugt werden, dass (z.B. über eine Füllroutine oder Ähnliches) solche unerreichbaren Gebiete in einem gesonderten Bereich gemerkt und vor der Padsuche geprüft werden.
- | In einer vollgestopften Umgebung gleich der eines Irrgartens ist es ratsam, Knoten, die nirgendwo hinführen, als Sackgassen zu markieren. Solche Bereiche können per Hand vorher im Karteneditor markiert werden; Ambitionierte Entwickler werden vielleicht einen Algorithmus entwickeln, um solche Bereiche automatisch zu identifizieren. Jeder Knoten in einem Sackgassenbereich könnte mit einer eindeutigen Nummer versehen werden, damit bei der Pfadfindung solche Bereiche sicher vermieden werden; eine Ausnahme bilden dabei Start- und Zielknoten, die sich in solch einer Sackgasse befinden.

7. Verwaltung der offenen Liste: Dies ist tatsächlich eine der zeitraubendsten Bestandteile des A*-Pfadfindungsalgorithmus. Jedesmal, wenn Du auf die Liste zugreifst, suchst Du das Quadrat mit den niedrigsten F-Kosten. Es gibt verschiedene Wege dies zu tun. Die einfachste, aber für lange Pfade sehr zeitaufwändige Methode ist, die Elemente, so wie sie anfallen, zu speichern und bei der F-Kosten-Suche stets die gesamte Liste zu durchsuchen. Eine verbesserte Methode ist, die Liste nach Eintrag eines Pfadelements aufsteigend nach dem F-Wert zu sortieren, um dann bei der F-Kosten-Suche schlicht das erste Element der Liste abzugreifen. Als ich meine Programme schrieb, was dies die erste Methode, die ich verwendete.

Dies funktioniert für kleine Karten ordentlich, aber es ist nicht die schnellste Lösung. Ernsthafte A*-Programmierer, die auf hohe Geschwindigkeit Wert legen, verwenden etwas, das "Binäre Halde" (engl.: binary heap) genannt wird, und dies ist das, was ich in meinem Programm verwende. Nach meiner Erfahrung ist dieser Ansatz in den meisten Situationen mindestens 2-3 Mal schneller und geometrisch schneller (10 Mal so schnell oder noch schneller) bei langen Pfaden. Falls Du daran interessiert bist, mehr über Binäre Halden herauszufinden, schau' Dir meinen Artikel [Using Binary Heaps in A* Pathfinding](#) an.

Ein anderer möglicher Flaschenhals ist die Art und Weise, wie Du zwischen den Funktionsaufrufen zur Pfadfindung die Datenstrukturen löschst und verwaltest. Ich persönlich ziehe es vor, alles in fest dimensionierten Bereichen zu speichern. Obwohl die Knoten dynamisch verwaltet werden können, benötigt meiner Erfahrung nach das Erzeugen und Löschen der Knoten einen zusätzlichen, unnötigen Aufwand, der den gesamten Ablauf verlangsamt. Doch auch wenn Du fest dimensionierte Bereiche verwendest, wird es erforderlich sein, dass Du zwischen den Funktionsaufrufen Dinge aufräumt; das Letzte, was Du in solchen Fällen aber brauchst, ist das Initialisieren von dynamischen Listen, besonders, wenn Du eine große Karte hast.

Ich vermeide diesen Zusatzaufwand, indem ich einen zweidimensionalen Bereich verwende - ich nenne ihn `whichList(x,y)` -, der jeden Knoten der Karte als der offenen oder geschlossenen Liste zugehörig kennzeichnet. Nach einer Pfadfindung nulle ich diesen Bereich nicht aus, sondern setze die Werte der Variablen `onClosedList` und `onOpenList` bei jeder Pfadfindung zurück, indem ich sie um 5 oder einen ähnlichen Wert erhöhe. Auf diese Weise kann der Pfadfindungsalgorithmus alle Werte, die aus vorherigen Pfadfindungsversuchen resultieren, ignorieren. Genauso merke ich mir die F-, G- und H-Werte in fest dimensionierten Bereichen und überschreibe sie einfach bei jedem neuen Pfadfindungsversuch. Damit kann ich mir das Löschen dieser Werte nach jeder Pfadfindung sparen.

Daten in vielen fest dimensionierten Bereichen zu speichern ist dennoch ein Kompromiss, da ja mehr Speicher verbraucht wird. Letztendlich wirst Du selber herausfinden, welche Methode Dir am besten liegt.

8. Dijkstras Algorithmus: A* wird allgemein als der beste Algorithmus zur Pfadfindung betrachtet (siehe Gelaber oben), doch es gibt mindestens einen anderen Algorithmus, der seinen Nutzen hat - Dijkstras Algorithmus. Dieser ist grundsätzlich der gleiche wie A*, außer, dass er keine Heuristik verwendet; H ist stets 0. Da er keine Heuristik verwendet, sucht er gleichmäßig expandierend in alle Richtungen. Wie Du Dir vorstellen kannst, untersucht Dijkstras Algorithmus ein wesentlich größeres Gebiet, bevor das Ziel gefunden wird. Dies macht diesen Algorithmus langsamer als A*.

Warum ihn also verwenden? Manchmal wissen wir nicht, wo sich unser Ziel befindet. Nehmen wir an, wir haben eine ressourcensuchende Einheit, die irgendeinen Rohstoff sammeln soll. Sie mag wissen, wo sich verschiedene Rohstoffgebiete befinden, aber sie möchte das naheliegendste finden. Hier ist Dijkstras Algorithmus besser als A*, denn wir wissen nicht, welches das naheliegendste Gebiet ist. Unsere Alternative ist, wiederholt A* anzuwenden, um die Entfernung zu jedem Gebiet zu ermitteln, um dann dessen Pfad dorthin zu wählen. Es gibt möglicherweise zahllose ähnliche Situationen, in denen wir die Eigenschaft des Zieles, nach dem wir vielleicht suchen werden, kennen, das naheliegendste finden wollen, aber nicht wissen, wo es sich befindet bzw. welches nun das naheliegendste sein könnte.

Weiterführende Artikel

Gut, jetzt kennst Du die Grundlagen und hast einen Eindruck von den fortgeschritteneren Konzepten. An diesem Punkt empfehle ich Dir, Dich durch meine Quelltexte zu arbeiten. Das Paket enthält zwei Versionen, eine in C++, die andere in Blitz Basic. Beide Version sind ausgiebig dokumentiert und sollten relativ leicht nachzuvollziehen sein. Du findest sie unter nachfolgender Verbindung.

- | [Sample Code: A* Pathfinder \(2D\) Version 1.9](#)

Falls Du keinen Zugang zu C++ oder Blitz Basic hast, enthält die C++-Version zwei kleine exe-Dateien. Zum Ausführen der Blitz Basic-Version kannst Du Dir die freie Demo-Version von "Blitz Basic 3D" (nicht Blitz Plus) von der [Blitz Basic](#)-Webseite herunterladen. Eine Online-Demo von Ben O'Neill findet sich [hier](#).

Empfehlenswert sind auch die folgenden Webseiten. Sie sollten jetzt nach diesem Tutorial viel leichter zu verstehen sein.

- | [Amits A* Pages](#): Dies ist eine gut referenzierte Seite von Amit Patel, sie kann jedoch etwas verwirrend sein, falls Du Dieses Tutorial nicht vorher gelesen hast. Sehr empfehlenswert. Besonderes Augenmerk verdienen Amits eigene [Gedanken](#) zu diesem Thema.
- | [Smart Moves: Intelligent Path Finding](#): Dieser Artikel von Bryan Stout auf Gamasutra.com erfordert zu Lesen die Registrierung, die aber kostenlos ist und durchgeführt werden sollte, einfach um diesen Artikel, weniger die anderen Quellen, die dort zu finden sind, zu lesen. Das von Brian in Delphi geschriebene Programm half mir A* zu lernen, und es ist inspirierte mein A*-Programm. Es beschreibt auch einige Alternativen zu A*.
- | [Terrain Analysis](#): Dies ist ein fortgeschrittener, aber interessanter Artikel von Dave Pottinger, einem Profi bei den Ensemble Studios. Er koordinierte die Entwicklung von Age of Empires und Age of Kings. Erwarte nicht, alles in diesem Artikel zu verstehen, aber es ist ein interessanter Artikel, der Dich zu eigenen Ideen anregen kann. Er schließt einige Diskussionen über mip-Kartierung ("mip" ist ein Akronym für das lateinische "multum in parvo" und heißt soviel wie "Vieles im Kleinen"), die Einflusskartierung (influence mapping) und einige andere fortgeschrittenere KI-/Pfadfindungskonzepte ein.

Einige andere empfehlenswerte Seiten im Web:

- | [aiGuru: Pathfinding](#)
- | [Game AI Resource: Pathfinding](#)
- | [GameDev.net: Pathfinding](#)

Zusätzlich empfehle ich sehr die folgenden Bücher, die eine Menge Artikel über Pfadfindung und andere KI-Themen

enthalten. Sie werden mit Quelltexten auf CD ergänzt, und ich besitze beide. Falls Du sie über die folgenden Adressen kaufst, erhalte ich (Peter Lester, Anm. d. Ü.) ein paar Pennies von Amazon. :)



[AI Game Programming](#)

[Wisdom](#)

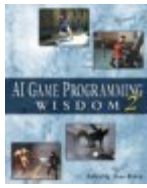
Steve Rabin

[Best Price \\$37.94](#)

or Buy New [\\$44.07](#)



[Privacy Information](#)



[AI Game Programming](#)

[Wisdom 2](#)

Steve Rabin

[Best Price \\$29.51](#)

or Buy New [\\$44.07](#)



[Privacy Information](#)

Gut, das war's. Wenn Du dazu kommen solltest ein Programm zu schreiben, das irgendeines dieser Konzepte anwendet, würde ich mich freuen, es zu sehen. Ich kann unter folgender Adresse erreicht werden:

patrick@policyalmanac.org

Bis dann, und viel Glück!