Erste Schritte ins Pathfinding mit C-Script

für A5.20

von Philipp Walser

Inhalt

Vorwort 3
1. Einleitung
1.1 Pathfinding – das Problem 4
2. Erste Schritte
2.1 Hindernisumgehung 4
2.2 Künstliche Intelligenz 6
2.3 Lösungsansätze umsetzen 7
3. Der A* - Ein Suchalgorithmus
3.1 Suchalgorithmen und Baumstrukturen 8
3.2 Suche nach einem Suchalgorithmus 1 - Die Tiefensuche 8
3.3 Suche nach einem Suchalgorithmus 2 - Die Breitensuche 9
3.4 Heuristik! 10
3.5 Suche nach einem Suchalgorithmus 3 - Best-First-Search 11
3.6 Heuristik die Zweite 11
4. Implementierung
4.1 Node-Definitionen 12
4.2 Multidimensionale Arrays 14
4.3 Heapsort 15
4.4 Das eigentliche Script 19
4.5 Ohne Terrain... 32
Anhang: Optionen, optimieren, erweitern...
I. OPEN und CLOSED 36
II. Postprocessing 37
III. Unterschiedliche Höhenebenen 37
IV. Bei Verwendung mehrerer Terrains... 38
V. Weitere Quellen 38

Vorwort

Bei dem in diesem Tutorial behandelten Thema handelt es sich um einen Teilbereich der Künstlichen Intelligenz. Die Programmierung der KI ist einer der schwierigsten Abschnitte in der Entwicklung eines Computerspiels. Dieses Tutorial ist also eindeutig nicht für Einsteiger zu empfehlen. Es wird mit Multidimensionalen Arrays, Rekursion und Parameterübergaben gearbeitet. Ich versuche zwar, alles so gut es geht zu erklären, doch ich kann nicht garantieren dass dies schlussendlich für jeden Verständlich rüberkommt.
Wir werden versuchen, einer Entity das Wahrnehmen und automatische Umgehen von einem oder mehreren Hindernissen „beizubringen“. Natürlich gibt es hierfür viele verschiedene Lösungsansätze, und gerade was KI und Pathfinding betrifft wird man keine perfekte Lösung finden. Jeder Programmierer hat für sein Spiel eigene Vorstellungen, wie sich seine computergesteuerten Figuren verhalten sollen.
Ich versuche, in diesem Tutorial einen Pathfinding-Algorithmus anzubieten, der vielseitig einsetzbar ist, dadurch aber auch auf die eigenen Wünsche angepasst und erweitert werden muss. Sehen Sie dieses Tutorial also nicht als eine Anleitung zum Erstellen eines perfekten Pathfindings an, sondern als ein Beispiel dafür, was mit WDL/C-Script möglich ist. Ein Pathfinding-Algorithmus lässt sich mit C++ sicherlich effektiver gestalten, aber nicht jeder hier verfügt über das SDK. Doch auch für diejenigen, die den SDK haben kann das Tutorial interessant sein. Denn auch wenn dieser Pathfinding-Algorithmus in C-Script geschrieben wurde lässt sich das Prinzip selbstverständlich auch in C++ umsetzen. Und wahrscheinlich ist das C-Script um einiges leichter zu verstehen als ein vergleichbarer C++ Sourcecode!
Ich hoffe, dass ich Sie nach diesem Tutorial mit den Grundlagen des Pathfindings vertraut gemacht habe und Sie anhand der Beispiele schon bald ihre eigenen Wege im Pathfinding beschreiten. Vielleicht finden Sie ja vollkommen neue Lösungsansätze, oder haben Sie weitere Verbesserungsvorschläge für meine Ansätze? Wer sein Feedback loswerden möchte kann mir also gerne eine eMail an pathfinding@philippw.de schreiben

Noch ein paar Worte zum mitgelieferten Sourcecode: Ich habe lange überlegt, ob ich wirklich eine sofort benutzbare Version des Pathfindings dem Tutorial hinzufüge. Schließlich lernt man am besten, indem man sich seinen Code selber schreibt. Ich habe mich dann aber doch dazu entschieden, eine komplett funktionstüchtige Version zur Verfügung zu stellen. Zum einen weil somit denjenigen, denen dieses Tutorial zu schwer ist, den Pathfinding-Algorithmus trotzdem anwenden können. Zum anderen weil ich somit ein Beispiel hinzufügen konnte, wie man auf einfache Weise Start- und Ziel-Koordinaten festlegen, und das Ergebnis, das dieser Algorithmus liefert, in einem Spiel nutzen kann. Auch wenn diese Dinge indirekt mit dem Pathfinding zu tun haben (es handelt dich schließlich um ein Anwendungsbeispiel), halte ich es nicht für nötig, sie im Rahmen dieses Tutorials zu erklären. Das beigefügte Script-Beispiel reicht meiner Meinung nach vollkommen aus, um den Umgang mit dem Algorithmus zu verstehen. Durch diese und weitere kleine Änderungen unterscheidet sich der mitgelieferte Sourcecode von dem im Tutorial vorgestellten geringfügig.

1. Einleitung

1.1 Pathfinding – Das Problem

Beispielsituation: Mittlerweile haben wir schon einige Erfahrungen mit C-Script gemacht und für Player und Actor evtl. schon eigene Movement-Scripte geschrieben. Auch die Bewegung zu vorgegebenen Start- und Zielpunkten konnte vergleichsweise einfach realisiert werden. Doch was passiert, wenn wir ein Hindernis zwischen den Start- und Zielpunkt setzen? Die ausgewählte Entity (im Pathfinding wird zwar normalerweise der Begriff „Unit“ benutzt, aber einen schon vertrauten Begriff zu benutzen kann ja nicht schaden) prallt auf das Hindernis auf und kommt nicht mehr weiter, bestenfalls gleitet sie noch eine Weile an dem Hindernis entlang. Durch die Kollisionsabfrage hat die Entity das Hindernis zwar als solches erkannt, weiß aber ansonsten nicht weiter. Man könnte meinen, unsere Entity ist stark kurzsichtig und erkennt ein Hindernis erst wenn es kurz davor steht.

Die Aufgabe des Pathfindings besteht nun darin, der Entity zur Weitsichtigkeit zu verhelfen. Hierbei geht es zunächst darum, dass unsere Entity überhaupt einen Weg um das Hindernis herum findet. Danach soll sie wieder den alten Kurs aufnehmen und den Zielpunkt erreichen. Bei einer späteren Optimierung kann man dann versuchen, diese „Weitsichtigkeit“ weiter auszubauen, so dass die Entity den Weg um das Hindernis so früh wie möglich findet. Weiterhin sollte es der Entity nicht nur möglich sein, ein kleines Hindernis zu umgehen, sondern auch mehrere auf einmal b.z.w. sehr komplexe Hindernisstrukturen wie Labyrinthe zu erkennen und zu meistern. Denn schließlich möchten wir unserer Entity nicht dabei zugucken, wie sie nach wie vor an zu komplexen Strukturen hängen bleibt oder absolut unnötige Wege einschlägt die sie nur auf Umwegen zum Ziel führen. Vorzugsweise soll die Entity natürlich nicht nur irgendeinen Weg um das Hindernis herum finden, sondern den besten (welches nicht immer der kürzeste sein muss). An diesen Anforderungen kann man sehen, dass wir es beim Pathfinding mit KI zu tun haben: Wir müssen der Entity beibringen, auf die erkannten Hindernisse einigermaßen „intelligent“ zu reagieren. Egal ob Shooter-, Strategie- oder Adventure-Game: Unsere Entities sollen sich nicht vor dem Spieler blamieren!

2. Erste Schritte

2.1 Hindernisumgehung

Den Anfang der KI macht die Hindernisumgehung. Die Kollisionsabfrage der A5-Engine sorgt zwar dafür, dass unsere Entity ein Hindernis erkennt und an ihm stehen bleibt, doch wie bekommen wir die Entity von diesem Hindernis wieder los? Wenn man gar keine Ideen hat kann man ja erst einmal nachsehen, was andere „Pathfinder“ schon für Ideen hatten. Pathfinding ist natürlich ein sehr wichtiges Element in der Spielentwicklung und Sie können sich sicher sein, dass es schon andere vor Ihnen gab, die sich dem Problem angenommen haben.

Das in Ausgabe 2 des AUM (Acknex User Magazin, http://www.gameresource.de/aum/) vorgestellte „Pathfinding“ von Caveman, welches zu dem Code-Snippet von einem Strategiespiel gehört, ist ein gutes Beispiel dafür, wie ein Anfang aussehen kann. Die Methode, die verwendet wird nennt sich **Random Bounce**: Sobald die Entity ein Hindernis erreicht hat wird ihre Richtung in eine zufällige Richtung geändert, so dass sie sich wieder ein Stück vom Hindernis entfernen kann und nicht an diesem hängen bleibt. Danach wird sie wieder in die Richtung des Zielpunktes gebracht. Wenn Sie sich das ganze in der Praxis ansehen wollen laden Sie sich die Codebeispiele des AUM2 runter.

Soweit so gut. Die Entity bleibt nicht mehr an dem Hindernis stehen. Trotzdem sieht es noch sehr dürftig aus, wie sie versucht, sich an dem Hindernis vorbei zu schleichen. Auf die viel zu späte Hinderniserkennung (durch „Körperkontakt“ mit dem Hindernis) folgt eine zufällige Richtungswahl, die deutlich spürbar ist: Die Entity scheint eher zufällig auf den Zielpunkt zu stoßen als ihn wirklich zu finden! Das erste was wir also versuchen sollten, ist die Zufälligkeit aus der Bewegung der Entity herauszubekommen. Hierfür hat George Pirvu in der 4. Ausgabe des AUM eine wirklich gute Lösung gefunden: Er verleiht der Entity 2 Antennen, die sich links und rechts vor der Entity befinden. Sobald eine der Antennen in ein Hindernis eindringt, schlägt die Entity solange die Entgegengesetzte Richtung ein, bis sich die Antenne wieder außerhalb des Hindernisses befindet. Die Bewegung der Entity um ein Hindernis sieht um einiges eleganter und geplanter aus. Der Zufall wurde bis auf eine Ausnahme (siehe AUM 4) herausgenommen. Ein weiteres Problem, welches auch schon in Caveman’s Script auftaucht, bleibt aber erhalten: Die Entity ändert zwar vorübergehend ihre Richtung, doch sie ist auch während der Richtungsänderung nur auf ihren Zielpunkt fixiert und wendet sich nach einer Richtungsänderung diesem erneut zu. Auf den ersten Blick scheint dass ja der erwünschte Effekt zu sein und funktioniert bei einfachen Hindernissen auch sehr gut. Wenn wir unsere Entity mit diesem Script aber durch ein Labyrinth jagen, wird sie ihr Ziel wahrscheinlich nie erreichen.

Wieso das? Nun, auch in diesem Fall wird kein Weg zum Ziel gesucht und gefunden. Die Entity umgeht schlichtweg die Hindernisse, die ihr auf dem **direkten** Weg zum Ziel im Wege sind. Doch auch wenn sprichwörtlich alle Wege nach Rom führen, so ist der direkte Weg zum Ziel nicht immer der richtige und endet oft in einer Sackgasse (vor allem in Labyrinthen).
In unserem Beispiel reicht es sogar, ein sehr langes Hindernis einzusetzen:

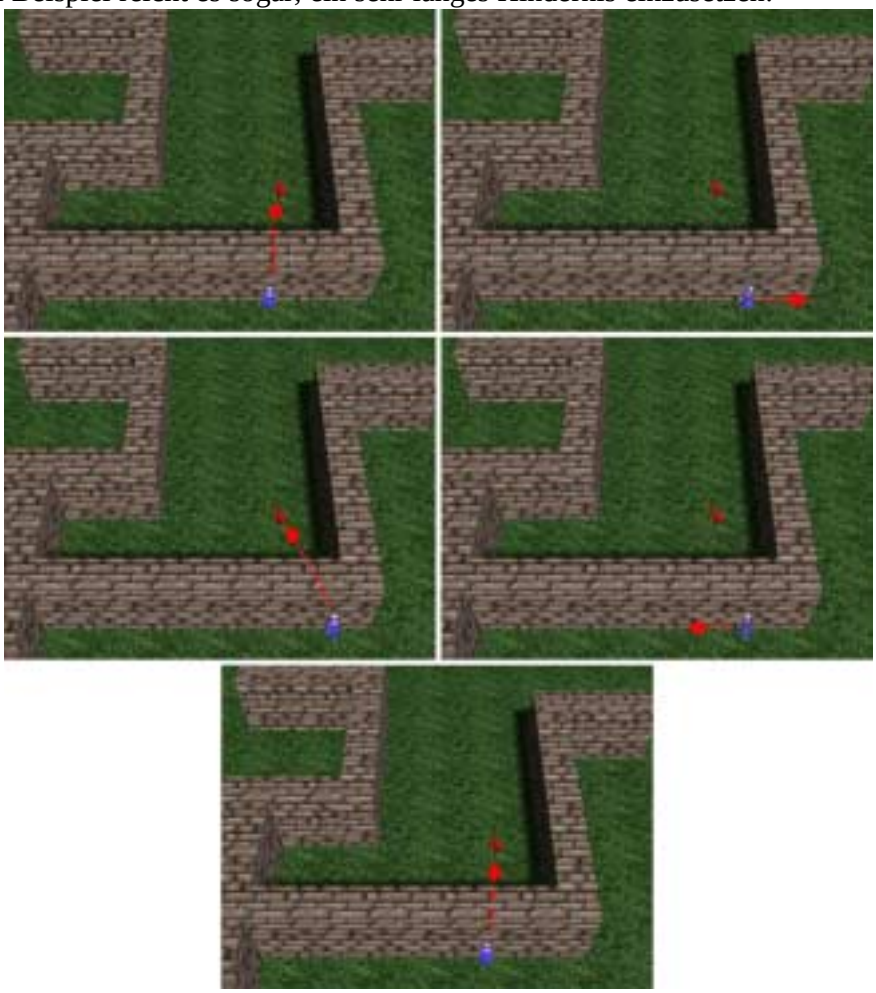

Abbildung 1: Verhalten einer Entity mit dem AI Demo-Script aus AUM4

Ungeachtet der restlichen Hindernisse scheitert unsere Entity sogar schon am ersten: Wie man sehen kann schlägt unsere Entity zunächst eine Richtung ein die sie durchaus zum Ziel führen könnte. Sie fixiert sich jedoch wieder auf ihren Zielpunkt, noch bevor sie das Hindernis passieren konnte und dreht sich dem Zielpunkt wieder zu. Dadurch dreht sie sich schließlich wieder in die Richtung, aus der sie ursprünglich gekommen ist. Wenn wir das Programm nicht abbrechen wird die Entity stundenlang hin und her rennen ohne dass sie den Zielpunkt jemals erreichen wird.

2.2 Künstliche Intelligenz

Wie können wir das ändern? Nun, die beste Möglichkeit eine KI zu entwickeln, ist von einer „echten“ Intelligenz abzukupfern. In den meisten Fällen reicht es aus, wenn man sich selber zu diesem Versuchsobjekt macht und versucht, sich in eine ähnliche Situation hineinzuversetzen und zu überlegen, wie man selber reagieren würde. Eine gute Beispielsituation wäre die Vorstellung, dass man sich in einer fremden Stadt ohne Stadtplan befindet und zu einer berühmten, vom aktuellen Standpunkt aus sichtbaren Sehenswürdigkeit wollte. Wir wissen zwar, in welcher Richtung unser Ziel liegt, aber damit ist noch lange nicht gesagt dass wir dort ankommen, wenn wir von unserem Standpunkt aus diese Richtung einschlagen. Nur weil eine Straße in diese Richtung führt kann man sich nicht sicher sein dass dies einige 100 Meter weiter noch immer so ist! Mit einem Stadtplan hätten wir schnell wieder den Überblick zurück: Wir könnten den Straßenverlauf verfolgen und erkennen, ob der so verlockend scheinende Weg sich nicht als eine Sackgasse entpuppt.

Was unsere Entity also braucht ist ein Stadtplan, den sie abfragen kann bevor sie loszieht. Theoretisch müsste man also das ganze Level noch mal in einer Art „Stadtplan“-Format abspeichern, und der Entity dann „beibringen“ diesen Stadtplan zu lesen. Diese Methode hat jedoch einige Nachteile: Zum einen ist es ein erheblicher Mehraufwand, die im WED erstelle Karte noch mal in einem anderen Format (z.B. 1 für ein Hindernis, 0 für keines pro Koordinatenpunkt) abzuspeichern oder ein Programm zu schreiben welches die Daten der WMP-Datei selber in ein solches Format bringen kann. Zum anderen ist eine Spielwelt nicht statisch. In den Leveln sollen sich unsere Entities, seien es Model oder WMB-Entities bewegen können. Die Welt ist einer ständigen Veränderung ausgesetzt. Dies würde gleichzeitig bedeuten, dass wir unseren Stadtplan ständig aktualisieren müssten. In einem Level mit vielen beweglichen Hindernissen nicht zu empfehlen! Aus diesen Gründen, aber auch weil es den Umfang dieses Tutorials meiner Meinung nach sprengen würde, gehe ich auf die Stadtplan-Methodik nicht weiter ein.

Doch was tun? Unsere Entity befindet sich in einer fremden Umgebung und hat keinen Stadtplan um zu sehen wie sie zu ihrem Ziel kommt. Stellen wir uns also noch einmal eine der häufigsten Fragen wenn es um das finden einer Lösung in Sachen KI geht: Wie würde **ich** mich in einer solchen Situation verhalten? Das erste, was mir einfallen würde, wäre jemanden mit Ortskenntnissen nach dem Weg zu fragen. Doch wie findet unsere Entity jemanden mit diesen Eigenschaften? Man könnte z.B. in der Levelmap selber Objekte platzieren, die man per C-Script miteinander „verknüpft“ so dass sie wissen ob ein Nachbarobjekt von diesem Punkt aus erreichbar ist oder nicht. Auf diese Weise könnte die Entity an den Objekten entlang zu ihrem Ziel navigiert werden. Als Objekte eignen sich hierfür vor allem Entities, aber auch Positionsmarken. Optimal wäre es wahrscheinlich, wenn letztere noch über ein paar Skills oder Flags verfügen würden, um detaillierte Informationen zu speichern. Wenn man diese benötigt, muss man auf Entities zurückgreifen, wobei man dann natürlich auf das Entity-Limit achten muss, welches durch den Nexus und MAX_ENTITIES vorgegeben wird. Und so praktisch und schnell die Abfrage zur Laufzeit des Spieles dieser gesetzten Wegpunkte auch sein mag, das setzen im WED ist schon eine aufwendige Prozedur, vor allem bei größeren Leveln mit komplizierten Hindernisstrukturen. Wäre es nicht praktischer, wenn unsere Entity diese Wegpunkte selber setzen könnte?

Genau dies wollen wir im folgenden Versuchen zu realisieren. Wobei es auch hierbei natürlich Nachteile gibt: Da unsere Entity die Umgebung noch nicht kennt muss ihr Script die Wegpunkte im Trial/Error verfahren setzen. Eine Definition der Wegpunkte durch den Leveldesigner kann natürlich viel geplanter erfolgen und er kommt auch mit viel weniger Wegpunkten im gesamten Level aus als unser Script. Alles in allem würde ein Script, welches die Wegpunkte eigenständige setzt also wahrscheinlich einen Tick länger brauchen als ein Script das sich vorhandener Wegpunkte bedient. Und dies ist hier durchaus wörtlich zu nehmen! In der Realität lässt sich der Unterschied wahrscheinlich kaum messen. Optimal wäre aber wahrscheinlich eine Kombination aus beiden Ideen. Dies würde zum einen weniger scripteigene als auch weniger im WED vorgefertigte Wegpunkte erfordern.

Das Prinzip der vorgefertigten Wegpunkte wird schon von 2 Scripts zur Verfügung gestellt: Zum einen MadJack's Pathfinder-DLL, zum anderen ein Pathfinding-Script von Wes Renfroe, welches mit einem Suchalgorithmus funktioniert, der auch bei der Router-Kommunikation Verwendung findet. Mehr dazu finden sie im 3DGS-Forum und natürlich auf den Homepages der Scripte (http://www.antares40.com/ / http://www.freewebz.com/eorfner/). Vielleicht gelingt es Ihnen ja mit Hilfe des Tutorials und den Beschreibungen der anderen Scripte eine Kombination aus den 2 unterschiedlichen Varianten des Wegpunkte-Setzens zu entwickeln.

2.3 Lösungsansätze umsetzen...

Wie schon gesagt möchte ich in diesem Tutorial beschreiben, wie man durch ein scriptbasiertes setzen von Wegpunkten eine Entity zu ihrem Ziel führen kann. Natürlich ist das mit einiger Arbeit verbunden und es wäre einfacher wenn man auf schon fertige Scripte, wie die eben beschriebenen zugreifen würde. Aber gerade die Tatsache, dass der andere Lösungsansatz schon realisiert wurde sollte Sie motivieren, sich anhand dieses Tutorials dem anderen zu widmen. Der erste Schritt hierbei ist natürlich, wie genau das Setzen neuer Wegpunkte per Script aussehen soll, bzw. wie die Wegpunkte selber aussehen sollen. Doch bevor wir damit richtig loslegen möchte ich Sie noch mit ein wenig Basiswissen zum besseren Verständnis versorgen.

Zunächst einmal wollen wir uns von dem Begriff „Wegpunkt/e“ trennen, so gut er auch die Objekte beschreiben mag, die uns zum Ziel führen. Im Fachjargon spricht man nicht von Wegpunkten, sondern von **Nodes**. Das liegt daran, dass beim Pathfinding wie schon angedeutet Suchalgorithmen zum Einsatz kommen. Das Script von Wes Renfroe verwendet z.B. einen Suchalgorithmus der bei Routern angewandt wird. Suchalgorithmen durchsuchen die ihnen übermittelten Daten sehr häufig in Form von Graphen, vor allem **Baumstrukturen** (siehe Abbildung 2). Die einzelnen Einträge in einer Baumstruktur werden Nodes (Knoten/Knotenpunkte) genannt, und so findet dieser Begriff auch hier Verwendung. In diesem Tutorial möchten wir uns einem Suchalgorithmus widmen, der nicht nur in Computerspielen erfolgreich eingesetzt wird: Der A* (A-Star).

3. Der A* – Ein Suchalgorithmus

3.1 Suchalgorithmen und Baumstrukturen

Bei dem A* handelt es sich um einen (graphischen) Suchalgorithmus, der aus einer Liste von möglichen Zuständen vom Ausgangszustand aus nach bestimmten Regeln den erwünschten Zielzustand sucht. Man nennt diese Art der Suche State-Space-Search und sie wird u.a. auch in Schachspielen verwendet. Hierzu ein simples Beispiel:

Nehmen wir an, wir haben eine Variable mit einem Ausgangswert von 1. Diese Variable soll nun den von uns bestimmten Zielwert 6 erreichen. Natürlich könnten wir in unserer Entity-Funktion für *temp=1;* einfach den Befehl *temp+=5;* geben. Aber wir möchten ja, dass unsere Entity zu einem gewissen Maße selber das Ziel findet. Dafür müssen wir noch einige Regeln definieren, die unsere Entity befolgen kann. In diesem Beispiel wären wahrscheinlich einfache Additionsregeln angebracht (z.B. rechne +2,+3,+4,+5 usw.). Zum besseren Verständnis hier eine Abbildung in Form einer Baumstruktur:

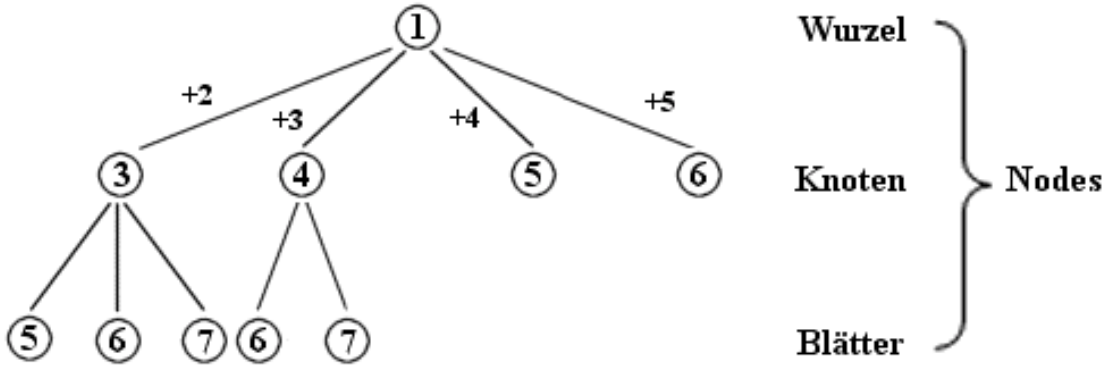

Abbildung 2: Eine Baumstruktur mit 3 Ebenen/Tiefen

Wie man an der Beschreibung rechts erkennen kann, beginnt man die Darstellung eines Baumes mit der **Wurzel**, die über **Kanten** mit **Knoten** oder **Blättern** verbunden sind. Knoten und Blätter unterschieden sich nur dadurch, dass auf ein Blatt keine weiteren Einträge (in die Tiefe des Baumes) mehr folgen. An einem Blatt endet also der „Ast“ eines Baumes. Alle Einträge in einem Baum, ob Wurzel, Knoten oder Blätter nennt man wie gesagt auch **Nodes**. Wenn man sich die Baumstruktur ansieht ist es recht gut Nachvollziehbar: Die einzelnen Einträge in der Baumstruktur könnte man durchaus als Knotenpunkte bezeichnen.

Der Ausgangszustand von 1 befindet sich in der Wurzel. Die Kanten stellen in diesem Fall die Regeln dar, wobei in der Abbildung einige Regeln, die nicht zum Ziel führen können, ausgelassen wurden. Wie solche Baumstrukturen nun genau in C-Script realisiert werden können wird zu einem späteren Zeitpunkt behandelt. Zunächst sollten Sie sich Anhand der Abbildungen das Prinzip einer Baumstruktur und deren Durchsuchung klarmachen.

3.2 Suche nach einem Suchalgorithmus 1 – Die Tiefensuche

Um die Lösung zu finden müssen wir unsere Entity nun mit einem Suchalgorithmus ausstatten, der diese Baumstruktur nach der Lösung durchsucht. Doch wie durchsucht man so einen Baum? Nun, die erste Idee die einem sicher kommt ist den Baum einfach von Links nach Rechts alle Knoten und Blätter zu durchsuchen. Man geht also zunächst „in die Tiefe“ bis zum ersten Blatt, dann zum zweiten usw. Dieses Suchverfahren nennt man Tiefensuche (Depth-First-Search).

In unserem Beispiel würde das heißen, dass wir von der 1 zuerst zur 3 gelangen, dann das Blatt mit dem Eintrag 5 besuchen, um schließlich zum Nachbarblatt 6 gelangen, womit wir den Zielzustand erreicht hätten und unsere Suche beendet wäre.

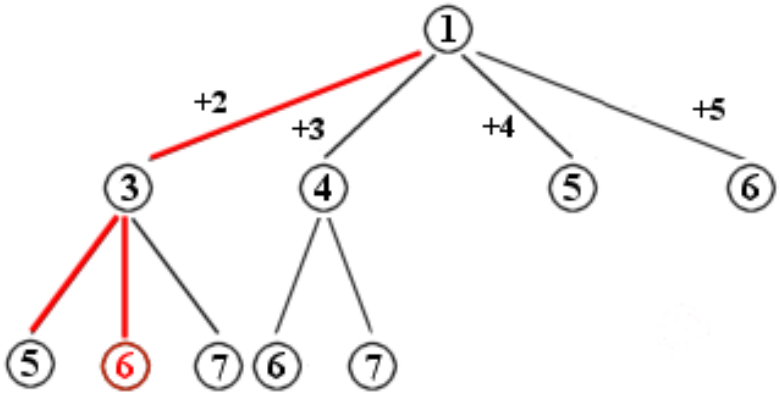

Abbildung 3: Die Tiefensuche (Depth-First-Search)

3.3 Suche nach einem Suchalgorithmus 2 – Die Breitensuche

Dies ist natürlich nicht die einzige Methode den Baum zu durchsuchen. Eine alternative wäre es, zunächst auf nur einer Ebene nach unserem Zielwert zu suchen, also nur die erste Knoten-Reihe. Dieses Suchverfahren nennt sich Breitensuche (Breadth-First-Search), da man hier zuerst „in die Breite“ sucht. In unseren Beispiel würden wir auch hier zunächst wieder den Knoten mit dem Eintrag 3 besuchen, dann jedoch den Nachbarknoten mit dem Wert 4, danach den Knoten mit dem Wert 5 und schließlich den Knoten mit dem Wert 6 aufsuchen, womit wir auch hier wieder unser Ziel erreicht hätten.

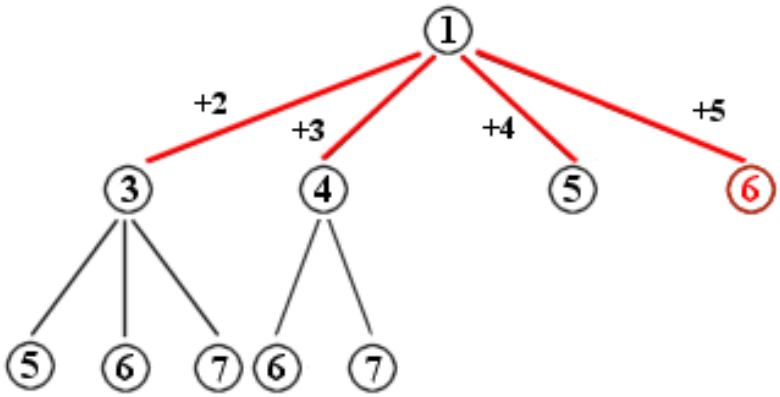

Abbildung 4: Die Breitensuche (Breadth-First-Search)

In unserem sehr einfachen Beispiel können wir schnell sehen, für welches Suchverfahren wir uns entscheiden sollten: Die Tiefensuche benötigt 3, die Breitensuche 4 Schritte, um zu einem Ergebnis zu gelangen. Grundsätzlich kann man sagen, dass die Tiefensuche vor allem dann eingesetzt wird, wenn es unwahrscheinlich ist dass die Baumstruktur zu viele Ebenen/Tiefen aufweist. Dementsprechend eignet sich die Breitensuche natürlich für Baumstrukturen, die nicht zu stark verzweigt sind. Hierbei kommt es normalerweise nicht auf Unterschiede von einem Schritt an, doch ich denke Sie können sich vorstellen dass die Unterschiede bei großen Baumstrukturen deutlich größer sein werden.

3.4 Heuristik!

Unsere Entity kann nun also eigenständig von 1 bis 6 addieren, in dem sie sich die nötigen Werte Schritt für Schritt zusammensucht. Wenn wir ehrlich sind ist dies noch ein äußerst dummes verhalten. Warum addiert unsere Entity nicht gleich 5, um auf 6 zu kommen? Depth-First-Search und Breadth-First-Search sind **blinde Suchverfahren**. Sie suchen anhand der Regeln denen man ihnen gegeben hat ganz einfach mit ihrer Methode durch die Baumstruktur bis sie den Zielzustand gefunden haben. Viel sinnvoller und effektiver wäre es jedoch, wenn unser Suchverfahren ungefähr abschätzen könnte, welche Kante des Baumes wahrscheinlich zum Ziel führt, und somit zuerst diesen Weg absucht. Dies nennt man ein **heuristisches Suchverfahren**.

Eine Heuristik ist eine Prognose, ein Wahrscheinlichkeitswert, wie günstig die nächste Node auf dem Weg zum Ziel liegt. Anhand dieser Prognose kann das Script entscheiden, welche Regel vom derzeitigen Zustand aus zum Ziel führen könnte. In unserem Fall wäre dies logischerweise der maximale Wert, der vom aktuellen Zustand aus addiert werden kann ohne den Zielwert zu überschreiten.

Doch jetzt mal ehrlich: Wer von uns braucht einen Code der dafür sorgt, dass unsere Entity so schnell wie möglich selber addieren kann? Anstatt diese Sache weiter zu vertiefen, die nur als Verständnisgrundlage dienen sollte, nähern wir uns jetzt lieber dem eigentlichen Problem an: Der Ausgangszustand für unser Pathfinding ist natürlich die aktuelle Position unserer Entity. Der Zielzustand der Ort, zu dem sich die Entity bewegen soll. Anhand von Regeln müssen wir nun also weitere Zustände erschaffen, die uns zum Ziel hinführen. Die weiteren Zustände wären also nichts anderes als Wegpunkte. Bei diesen ganzen unterschiedlichen Begriffen, die in unserem Fall sowieso das selbe bedeuten, ist es gut dass wir und schon im Vorhinein auf den allgemeinen Begriff **Node** geeinigt haben.

Gehen wir davon aus, dass sich die Entity in 8 verschiedene Richtungen bewegen kann. Sie muss sich von ihrer derzeitigen Position aus also für einen von 8 möglichen Nodes entscheiden. Unsere Regel wäre also: Bewege dich nach Norden, Nordost, Osten, Südost, Süden, Südwest, Westen oder Nordwest. In unserem Suchbaum würde dies folgendermaßen aussehen:

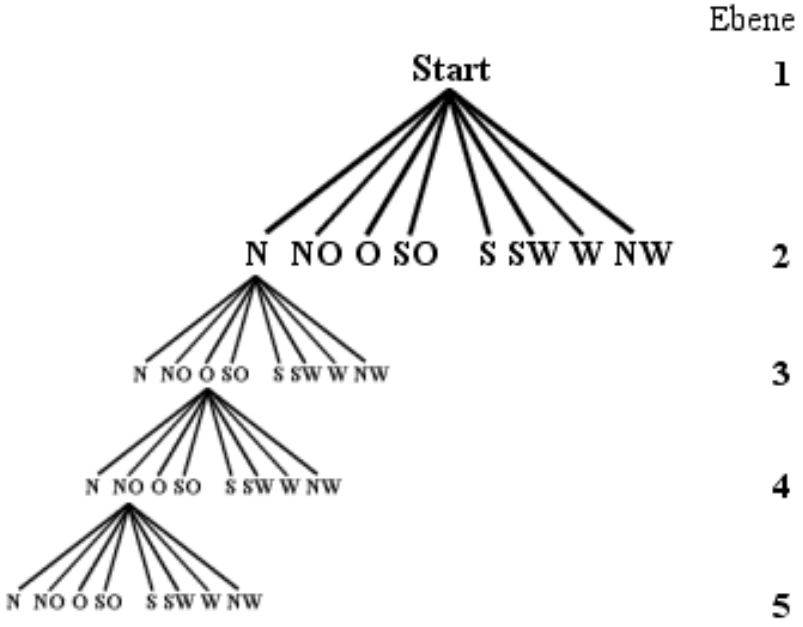

Abbildung 5: Darstellung einer Baumstruktur die in jeweils 8 Richtungen verzweigt. Es werden nur die „besten“ Nodes durchsucht.

Wie man auf der Abbildung erkennen kann entscheidet sich der Suchalgorithmus immer nur für eine der berechneten Richtungen (unsere Entity kann sich ja nicht teilen und mehrere Wege gleichzeitig einschlagen), von der die Suche fortsetzt und wieder 8 neue Richtungen untersucht werden. Über die Heuristik wird entschieden, welche der 8 möglichen Nodes am ehesten zum Ziel führt. Doch wie genau könnte eine Heuristik für unser Pathfinding aussehen? Wir erinnern uns: es muss ein Wert sein, der uns sagen kann wie günstig die nächste Node auf dem Weg zum Ziel liegt, vereinfacht ausgedrückt: Wie weit die derzeitige Node vom Ziel entfernt ist. In der Tat handelt es sich bei einer der einfachsten Heuristiken um nichts anderes als die Distanz vom aktuellen Wegpunkt zum Ziel. Ein Suchverfahren, was nach dieser Methode arbeitet nennt sich Best-First-Search

3.5 Suche nach einem Suchalgorithmus 3 - Best-First-Search

Wie der Name schon sagt wird aus den zur Auswahl stehenden Zuständen der vermeintlich „Beste“ gewählt und von diesem aus weitergesucht. In unserer Spielwelt heißt das nichts anderes als: Es wird sich für die Richtung mit der kleinsten Distanz zum Ziel entschieden und von dieser aus weitergesucht. Doch handelt es sich hierbei wirklich um den „besten“ Weg? Diese Methode wurde von TCS zu Testzwecken auch schon einmal in C-Script umgesetzt und zeigt uns auch sogleich wieder ein altbekanntes Problem: Der Kürzeste ist nun mal einfach nicht immer der beste Weg! Wie bei der Methode aus dem AUM 4 würde sich unsere Entity in einem komplexen Labyrinth hilflos verirren da sie sich immer wieder auf den kürzesten Weg zum Ziel fixieren würde.

Wie Sie sehen sind wir momentan wieder an unserer Anfangssituation angelangt: Unsere Entity „lässt“ zwar nach dem Weg fragen, doch die Informationen die sie daraus erhält lassen zu wünschen übrig. Und wie soll es jetzt weitergehen? Sie haben von mir nun einiges über Nodes, Baumstrukturen, Suchverfahren und Heuristik erzählt bekommen, und schlussendlich erzähle ich Ihnen dass sich an dem Verhalten der Entity im Grunde genommen nichts geändert hat. Nun, noch sind wir nicht am Ende unserer Weisheit. An dieser Stelle muss Ihnen eines klar sein: Nur weil ein Suchalgorithmus heuristisch arbeitet, bedeutet dass nicht dass er immer die beste Lösung findet. Nur eine gute Heuristik, angepasst auf die Bedürfnisse der Suche, wird Ihnen zu einem guten, wenn nicht vielleicht sogar zu einem optimalen Pathfinding verhelfen. Eine optimale Heuristik werden wir vorerst wohl kaum erstellen können, trotzdem sollten wir versuchen, unsere Heuristik zu verbessern, und sie nicht nur über die Distanz zum Ziel definieren.

3.6 Heursitik die Zweite

Damit unsere Entity nicht immer nur auf die kürzeste Distanz zum Ziel fixiert ist, brauchen wir einen Wert, der der Distanz zum Ziel genau entgegengesetzt ist. Wie wäre es mit einer Distanz zum Startpunkt? Dies würde nicht bedeuten, dass unsere Entity sich urplötzlich auf den Startpunkt fixieren würde und diesen anstelle des Zielpunktes versuchen würde zu erreichen. Mit dieser zusätzlichen Heuristik soll gewährleistet sein, dass die bisher zurückgelegte Weglänge vertretbar ist und der Algorithmus einen Weg der ihn nicht zum Ziel führt auch nicht weiter erforscht.

In der Regel wird hierfür nicht die echte Distanz vom Startpunkt zur aktuellen Position benutzt, sondern ein Wert, der sich **Wegkosten** nennt. Hierbei handelt es sich um einen Wert, der pro Ebene im Suchbaum um einen bestimmten Wert erhöht wird. Der Einfachheit halber erhöhen wir nur um 1, so dass die Wegkosten der aktuellen Ebene im Suchbaum entsprechen (siehe Abbildung 5). Sie werden es kaum glauben, aber mit der Verwendung von Wegkosten und der Distanz zum Ziel haben wir schon das Prinzip eines klassischen A*-Algorithmus erreicht.

4. Implementierung

4.1 Node-Definitionen

Im vorherigen Kapitel habe ich versucht, Ihnen das nötige Basiswissen über die Vorgehensweise von Suchalgorithmen zu vermitteln. Nun machen wir uns endlich daran, dass ganze in C-Script umzusetzen. Wie schon in 2.3 beschrieben, müssen wir uns nun erst einmal Gedanken machen, wie unsere Nodes in C-Script eigentlich aussehen sollen. Überlegen wir uns noch einmal wie genau die Nodes in unserem A* Verwendung finden: Unser Algorithmus kann sich von der aktuellen Entity-Position aus für eine von 8 Nodes entscheiden. Die Entscheidung wird anhand von Berechnungen (Heuristik) getroffen, wie gut die Node ein Finden des Zieles ermöglicht. Wenn er sich für eine der Nodes entschieden hat, kann er von dieser neuen Position aus wieder 8 neue Nodes wählen und für diese wieder Berechnungen anstellen. Das ganze geht so weiter bis eine der Nodes ihn schließlich direkt zum Ziel führt. Ab diesem Punkt muss der Algorithmus vom Ziel aus über die Nodes die ihn bis dorthin gebracht haben den Weg bis zum Startpunkt zurückverfolgen. Der Path muss auf diese Weise kreiert werden, da wir nur vom erreichten Zielpunkt aus sagen können, welche der vielen erforschten Nodes jetzt eigentlich wirklich am Finden des Zieles beteiligt waren. Die Heuristik konnte uns zwar wie gesagt als Wahrscheinlichkeitswert beim ermitteln der Nodes helfen, doch eine genaue Angabe, welche Nodes nun definitiv zum Ziel führen kann sie nicht bieten. Dies bedeutet auch, dass jede Node in C-Script gespeichert werden und jederzeit abrufbar sein muss, da jede Node eine potentielle Node zum Ziel sein kann. Als alternative können Sie auch einen Weg vom Ziel zum Start ermitteln lassen. Dadurch kann der erzeugte Path sofort von der Entity benutzt werden und muss nicht erst noch „umgedreht“ werden (hierzu später mehr).
Was muss eine Node in C-Script also alles speichern können?

- Natürlich die Koordinaten der Node, um die Distanz zum Ziel zu berechnen damit beim kreieren des Pathes die Entity auch weiß, wohin sie sich bewegen muss um eben diesem Path zu folgen.
- Die Heuristik, einer der wichtigsten Faktoren: Wie wahrscheinlich ist es, von dem derzeitigen Wegpunkt zum Ziel zu gelangen? Die Heuristik ist verantwortlich dafür, ob der Algorithmus so uneffizient wie ein Pathfinding ohne Suchalgorithmus wird, oder ob dadurch sogar der optimale Weg gefunden werden kann. Die Heuristik ist meistens aus unterschiedlichen Werten zusammengesetzt. In unserem Beispiel verwenden wir die Wegkosten und die Distanz zum Ziel als Heuristik.
- Ein Verweis auf die Vorgänger(Parent)-Node: Die Node muss wissen, von welcher Node aus sie gefunden wurde. Dies ist für das kreieren des Pathes am Schluß der Suche wichtig. Jede einzelne Node muss wissen, welche Node ihr Vorgänger war, von welcher Node aus sie ermittelt wurde. Nur so lässt sich ein Path bis hin zur Startnode zurückverfolgen.
- Eine ID: Damit jede Node später auch „wieder gefunden“ werden kann (als Parent-Node o.ä.), muss jeder Node eine individuelle Identifikationsnummer Besitzen, über die er angesprochen werden kann.

Auf den ersten Blick wäre es am einfachsten, die Nodes als Entities darzustellen. Über MY.X/Y/Z können jederzeit ihre Koordinaten abgefragt werden, zusätzliche Werte wie Heuristik und Verweis auf den Vorgänger können in Skills gespeichert werden. Und die A5 Engine vergibt an jede Entitiy automatisch eine Handle-ID, die man per *var=handle(object);* abfragen und in einer beliebigen Variable (oder Skill) speichern kann und per *ptr_for_handle(var);* ebenso leicht wieder in einen Pointer auf die Entitiy zurückverwandeln kann. Das klingt ja alles ganz schön und gut, aber leider hat die Verwendung von Entities einen großen Nachteil: Es handelt sich hierbei nun mal um Entities, deren Anzahl durch den Nexus beschränkt ist. Leider können wir bei unserem Pathfinding nicht genau sagen, wie

viele Nodes benötigt werden um das Ziel zu finden. Natürlich können wir dies beschränken, aber es wäre sinnvoll wenn unser Algorithmus so viele Nodes wie möglich verarbeiten könnte, um auch längere oder kompliziertere Wege zu finden. Da wir auch nur sehr wenige Werte und Eigenschaften von den Entities benutzen würden, wäre es eine arge Verschwendung von Entities.

Suchen wir also nach einer alternative. Das Hauptproblem hierbei ist die individuelle, einmalige ID pro Node. Zufallswerte scheiden hier natürlich aus. Doch bevor wir uns nun eine komplizierte Methode ausdenken, über die wir jeder Node eine ID zuweisen können, die einmalig ist und sozusagen einen Pointer der Node darstellt, wie es bei Entities der Fall ist, überlegen wir uns vorher noch mal ob so was nicht doch schon irgendwie in C-Script möglich ist.

Bei dieser alternative handelt es sich um Terrain. Jedes Terrain besteht aus Vertices, welche durchnummeriert sind, so dass jeder Vertex über eine eigene Nummer verfügt. Als besonderes Gimmick können wir sogar über die Funktion *vec_for_vertex(vector,terrain,number);* jederzeit den zur Vertex-Nummer gehörenden Vektor abfragen. Doch was ist, wenn Sie in ihrem Level/Spiel gar kein Terrain verwenden wollen? Nun, im späteren Verlauf des Tutorials werde ich natürlich auch erklären, wie das ganze ohne Terrain zu realisieren ist (bzw. wie man per C-Script ein „virutelles“ Vertex-Gitter erschaffen kann, was hier ja der eigentliche Grund für die Verwendung von Terrain ist). Wenn Sie erst einmal die Realisierung des A* per Terrain verstanden haben, ist der Transfer zu einer normalen Level-Oberfläche ganz einfach!

Kommen wir nun aber endlich zu unserer Node-Definition in C-Script. Es gilt, mehrere Werte abzuspeichern. Bei unserer Node muss es sich also um einen Array handeln. Dieser könnte in C-Script so aussehen:

```
var NODE[4];
...
NODE[ID]=vertex_number;
NODE[waycosts]=PARENT_NODE[waycosts]+1;
NODE[nodecosts]=NODE[waycosts]+vec_dist(current_pos,goal_pos);
NODE[parent]= 0;                          // Start-Node, kein Vorgänger
```

Auf den ersten Blick scheinen einige Werte zu fehlen: Was ist mit der Positionsangabe? Diese brauchen wir nicht extra zu speichern, da sie über die ID jederzeit herausgefunden werden kann. Und wo befinden sich die beiden Heuristik-Werte? Da die Wegkosten jedes Mal wenn 8 neue Nodes erstellt werden um 1 erhöht wird, muss der alte Wert der Wegkosten immer zugänglich sein, in diesem Beispiel über *PARENT_NODE[waycosts]*. Damit wäre der erste Teil der Heuristik abgedeckt. Der zweite Teil der Heuristik, die Distanz zum Ziel, wird wie Sie vielleicht schon erkannt haben im Eintrag *[nodecosts]* auf die Wegkosten aufaddiert. In dem Vektor *current_pos* befindet sich der Vektor der aktuellen Node, der vorher im Script über *vec_for_vertex(current_pos,terrain,NODE[ID]);* abgefragt wird. Doch hierbei sollte es sich ja erstmal um eine C-Script-Definition der Node handeln, um das eigentliche Script kümmern wir uns später. Der zweite Teil der Heuristik ist also auch schon enthalten. Aber warum habe ich die beiden Einträge dann nicht einfach Heuristik_1 und Heuristik_2 genannt, sondern waycosts und nodecosts?

Die Summe aus mehreren heuristischen Werten wird normalerweise unter dem Begriff **Nodekosten** zusammengefasst. Und da wir im Verlauf des Tutorials noch oft auf die Wegkosten, aber auch auf die Gesamtkosten zu sprechen kommen, ist es einfacher von den einzelnen Wegkosten zu sprechen und den gesamten Nodekosten als von der Wegkosten-Heuristik und der Summe aller heuristischer Werte.

Aber leider können wir die Node-Definition wie wir sie soeben aufgestellt haben nicht belassen: Von diesen Nodes werden ja nun je nach Abstand vom Start zum Ziel und dem Abstand der einzelnen Nodes zueinander recht viele gebraucht. Da wir wie schon gesagt nicht wissen wie viele Nodes dabei entstehen können, ist es auch nicht ratsam, prophylaktisch hunderte von Arrays in Form von *var NODE1[4]; var NODE2[4];* usw. zu definieren. Sinnvollerweise müssen alle Nodes also zusammen in einem Array abgespeichert werden. Eine einzelne Node stellt aber schon einen Array dar. Wie sollen wir einen Array also in einem Array darstellen?

4.2 Multidimensionale Arrays

Wie im C-Script-Manual erklärt ist es uns mit einem Trick möglich, multidimensionale Arrays zu kreieren. In unserem Fall reicht ein 2-Dimensionaler Array, der *Anzahl der Nodes**4 (da jede Node 4 Einträge benötigt) umfasst. Nehmen wir an wir möchten Platz für 1000 Nodes haben, dann müssen wir laut C-Script-Manual einen normalen Array erzeugen mit 4000 möglichen Einträgen. Abrufen könnten wir ihn dann folgendermaßen:

```
var NODE_LIST[4000]     // 1000*4
...
NODE_LIST[j*4+i]=temp;        // j=0..3999  i=0..4
```

Schön und gut, wir wissen jetzt wie das ganze funktioniert. Doch ich denke man sollte sich noch einmal die Zeit nehmen zu erklären, wie man auf diese Codezeilen kommt.

Nehmen wir an, eine Variable wäre nichts anderes als eine Schublade. Diese Schublade hat einen Namen, nämlich den der Variablen (z.B. *temp*). In der Schublade befindet sich ein Wert, welcher der Variablen *temp* gerade zugewiesen ist (z.B. 1.4). Ein Array ist nun nichts anderes als ein „Regal“ voller Schubladen. Der Name des Arrays ist in diesem Falle der Name des Regals und die Schubladen sind über eine Index-Zahl ansprechbar (bei einem Vektor, dessen Array-Länge 3 ist z.B. [0],[1],[2]. Nicht vergessen: Bei einem Array fängt man immer bei 0 mit dem Zählen an!). Jede dieser Schubladen beinhaltet einen eigenen Wert.

Bei einem Multidimensionalen Array wiederum fassen wir einige dieser Schubladen noch einmal zu größeren Schubladen zusammen. Die Schubladen in den Regalen sind also noch einmal in Fächer aufgeteilt. Nehmen wir an, wir möchten einen 2-Dimensionalen Array erstellen, in dem 3 Nodes gespeichert werden sollen, die jeweils 4 Werte besitzen. Wir benötigen also einen Array der Länge 3x4=12. Die erste Node ist in den Array-Schubladen [0],[1],[2] und [3] gespeichert. Man könnte diese also abrufen in dem man z.B. *NODE_LIST[2]* abfragt. So bekommt man den dritten Eintrag der ersten Node. Doch was wenn man auf den dritten Eintrag der zweiten Node zugreifen will? Dieser befindet sich nämlich in der Schublade [6]. Abbildung 6 verrät schon die Lösung: Um auf den dritten Eintrag der zweiten Node zugreifen

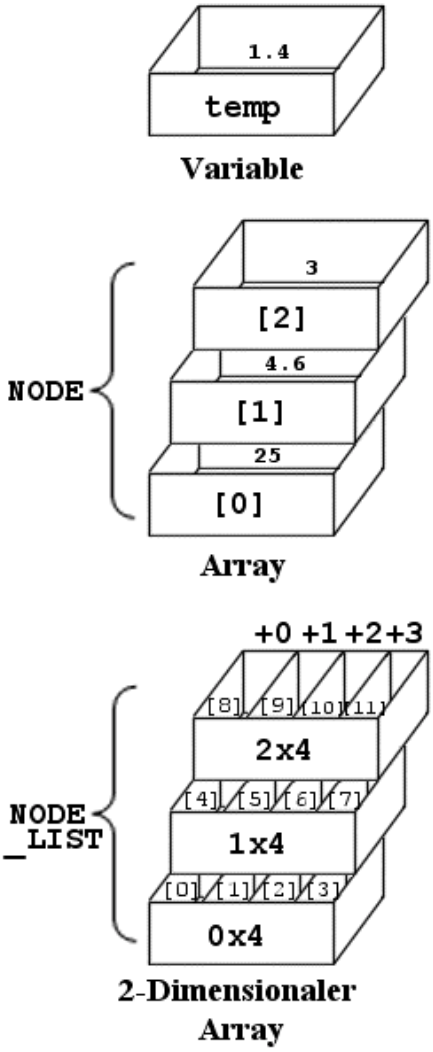

Abbildung 6: Grafische Darstellung eines Arrays

zu können, müssen die ersten 4 Schubladen übersprungen werden (1x4, damit landet man direkt im fünften Array-Eintrag), und dann noch so viele Schubladen passiert werden, bis man zum gewünschten Eintrag vorgestoßen ist (+2 um den dritten Eintrag der zweiten Node abzurufen). Eine Schublade entspricht hierbei also einer Node, und die Fächer in den Schubladen beinhalten die einzelnen Node-Einträge. Bei der Variablen *j* handelt es sich also um den Zugriff auf eine Node, bei *i* um die Bestimmung des gewünschten Node-Eintrages. Wenn wir dies also auf unsere Node-Struktur anwenden, würden sich durch den Befehl `temp=NODE_LIST[3*4+2];` in *temp* die Nodekosten der vierten Node befinden, die wir auf unserer Suche nach günstigen Nodes zum Ziel im *NODE_LIST* - Array gespeichert haben.

Fassen wir zusammen: Unser Algorithmus soll von der aktuellen Entity-Position aus 8 neue Positionen bestimmen und für diese jeweils die Nodekosten (Wegkosten + Heuristik) berechnen und sie schließlich als Nodes in einem Array abspeichern, so dass diese Nodes jederzeit abrufbar sind. Anhand der Nodekosten muss sich der Algorithmus jetzt für eine der Nodes entscheiden, an der es mit dem Pathfinding fort fährt. Es muss also die Node mit den geringsten Nodekosten aus dem Array herausgesucht werden. Allerdings wäre es doch recht ineffektiv, den Algorithmus, der (dank Heuristik) nur die Nodes absucht und speichert, die ihn zum Ziel führen könnten, noch einmal in seinen schon gespeicherten Nodes nach der aktuell besten Node suchen zu lassen. Das käme einer doppelten Suche gleich. Wäre es nicht viel luxuriöser, wenn unser Array mit den gespeicherten Nodes schon vorsortiert wäre, so dass sich im ersten Array-Eintrag immer die momentan günstigste Node befindet? Unter den vielen Sortieralgorithmen, die es gibt, eignet sich hierfür der **Heapsort** am besten.

4.3 Heapsort

Der Heapsort ist ein Sortierverfahren das eine spezielle Datenstruktur benutzt: den **Heap** (engl. Haufen). Bei einem Heap handelt es sich um nichts anderes als einen **binären Baum** (eine Baumstruktur, deren Knoten immer nur 2 Kanten aufweisen), dessen Wurzel/Knoten immer einen kleineren (oder größeren, je nach **Heap-Eigenschaft**) Wert haben als ihre direkten Blätter/Nachfolgerknoten. Die Heap-Eigenschaft wird durch die **Heap-Ordnung** bestimmt. Wenn Wurzel/Knoten einen kleinerer Wert als ihre direkten Nachfolger im Heap haben, ist dies eine aufsteigenden Heap-Ordnung (denn die Werte werden größer je tiefer man den binären Baum hinunter geht). Ein Heap mit dieser Heap-Eigenschaft könnte so aussehen:

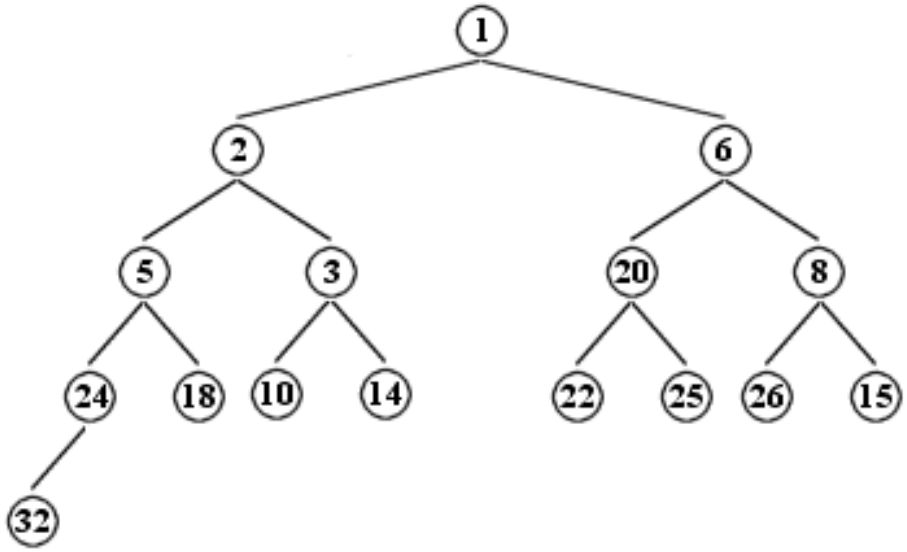

Abbildung 7: Ein Heap mit aufsteigender Heap-Ordnung

Wie man sehen kann befindet sich bei einem Heap mit aufsteigender Ordnung der kleinste Wert dieser Datenstruktur immer in der Wurzel (bei einer absteigenden Ordnung wäre es der größte Wert). Dies ist für uns natürlich unheimlich praktisch, denn wenn wir unseren Array mit den Nodeeinträgen auf diese Weise nach den Nodekosten sortieren könnten, würde sich in der Wurzel immer die momentan günstigste Node (die mit den geringsten Nodekosten) befinden.

Doch wie sollen wir unseren Array als Baumstruktur, bzw. sogar als Heap darstellen? Nun, glücklicherweise ist dies sehr leicht zu realisieren: Anstatt die erste Node in unserem *NODE_LIST* - Array an Position [0] abzuspeichern, legen wir diese an Position [1] ab. Die Nachfolger einer Node in *NODE_LIST[i]* (i sei eine beliebige Position im Array) werden an den Positionen *NODE_LIST[2*i]* und *NODE_LIST[2*i+1]* abgespeichert. In Array-Form sähe ein Heap dann also so aus:

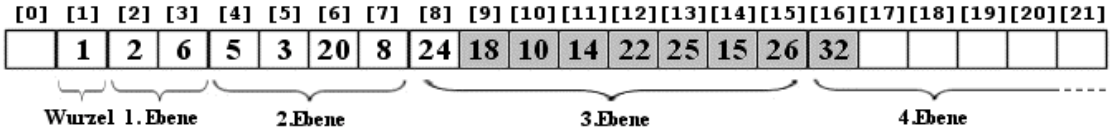

Abbildung 8: Darstellung des Heaps in einem Array. Blätter des Heaps sind grau unterlegt.

Auf dem selben Weg kann man einen so angelegten Array natürlich auch abfragen: Wenn wir also wissen möchten, welche Werte die Nachfolgeknoten/Blätter der vierten Node haben (Wert 5 siehe Abbildung 8), greifen wir folgendermaßen auf die Nachfolger zu:

```
temp.X=ARRAY[2*4];
temp.Y=ARRAY[2*4+1];
```

Anhand der obigen Abbildung kann man nachvollziehen, welche Werte sich jetzt in *temp.X* und *temp.Y* befinden: Die Werte 24 und 18. Ein Blick auf Abbildung 7 bestätigt uns auch das Ergebnis. Das ganze funktioniert natürlich nur, wenn es sich bei der Node um kein Blatt handelt, denn ein Blatt hat keine Nachfolger mehr. Mit der Rechung n/2+1 (wobei n die Anzahl aller Nodes im Array ist, auf der Abbildung also 16) kann man herausfinden, ab welcher Position im Array die Blätter abgespeichert sind. Dieser Rechenschritt ist der Grund, warum wir an der Array-Position [1] mit den Eintragungen beginnen. Die Überprüfung der Blätter lässt sich so einfacher umsetzen. Aber was machen wir, wenn wir anstatt der Nachfolger (**Child-Nodes**) einer Node ihren Vorgänger (**Parent-Node**) abfragen wollen. Dies ist auch kein Problem: Wo wir vorher multipliziert haben, wird nun dividiert: Die Parent-Node lässt sich immer finden, in dem man die Position der aktuellen Node durch 2 teilt: *NODE_LIST[i/2]*. Zum besseren Verständnis empfehle ich, dass Sie die angegebenen Rechenschritte anhand der Abbildung 7 und 8 ausprobieren und überprüfen.

Doch wie genau funktioniert jetzt der Heapsort? Hierzu eines noch mal vorweg: Für den A* benötigen wir nicht den kompletten Heapsort, der eine schon gefüllte Datenstruktur je nach Bedürfnis sortiert und gegebenenfalls erst noch in einen Heap umwandelt. Wir benutzen den Heap von vornherein als Datenstruktur (keine Umwandlung), dieser ist zu Beginn jedoch noch leer. In unserem Fall muss der Heapsort nach der Eingabe eines neuen Wertes diesen direkt in den Heap einsortieren. Das Ändern und Löschen eines Eintrages (und die damit verbundene Neusortierung) muss vom Heapsort auch bewältigt werden. Einen Heap, den man mit einer solchen Art von Heapsort benutzt, nennt man Prioritätenliste (**Priority Queue**). Die Priorität liegt natürlich auf den Nodekosten, die möglichst gering sein sollen.

Der Heapsort kann entweder von Oben nach Unten (Von der Wurzel bis zum letzten Blatt) oder von Unten nach Oben (von einem bestimmten Punkt im Heap bis zur Wurzel) beginnen. Da durch unsere Nodesuche immer wieder neue Nodes an das Ende des Arrays als Blätter hinzugefügt werden, empfiehlt es sich, mit dem Sortieren von Unten nach Oben zu beginnen (**Bottom-Up**-Heapsort): Die letzte Node im Array (oder diejenige, an der etwas verändert wurde) wird dahingehend überprüft, ob sie einen kleineren oder größeren Wert als ihre Parent-Node hat. Tritt der erste Fall ein werden Parent-Node und Child-Node vertauscht, denn laut unserer Heap-Ordnung muss die Parent-Node immer einen kleineren Wert als ihre Child-Nodes haben. Diese Prozedur wird solange wiederholt, bis die Wurzel erreicht ist oder die Parent-Node einen niedrigeren Wert als die Child-Node hat. Diese Prozedur nennt sich Upheap:

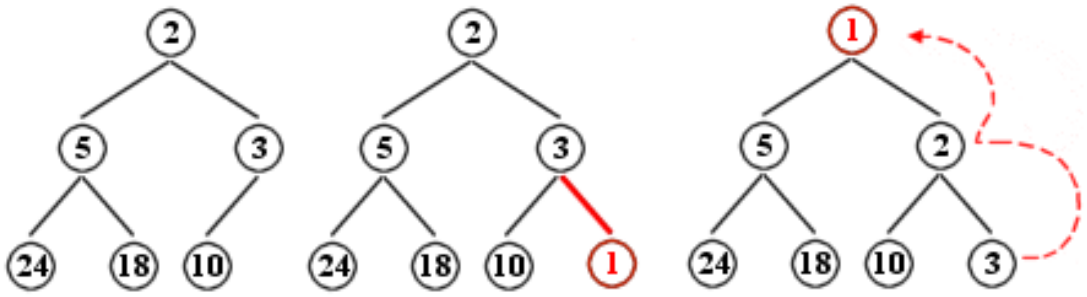

Abbildung 9: Die Upheap-Prozedur des Heapsorts

Nachdem es evtl. nötig gewesen ist, einen Eintrag nach oben zu verschieben, um die Heap-Struktur wieder herzustellen, kann eben genau dieses hochschieben dafür gesorgt haben, dass die Heap-Struktur zwar an einer Stelle wieder stimmt, an anderen aber wieder verändert wurde. Weiterhin kann es ja sein, dass der Wert einer Node verändert wurde und nicht kleiner sondern größer geworden ist (und somit der Upheap gar nicht Verwendung findet). Eventuell ist der Wert der Node sogar größer, als die Werte seiner Child-Nodes, wodurch die von uns definierte Heap-Ordnung wieder außer Kraft gesetzt wurde (Parent-Nodes müssen einen kleineren Wert als ihre Parent-Node besitzen). Darum müssen wir am Ende eines Upheaps (egal ob er jetzt an der Wurzel oder einem anderen Knoten endet) noch mal von Oben nach Unten sortieren. Die Child-Node der aktuellen Node mit dem geringsten Wert wird dahingehend überprüft, ob diese nicht evtl. plötzlich einen kleineren Wert als die aktuelle, ihre Parent-Node enthält. Während der Upheap Nodes (maximal) bis zur Wurzel „hochgepushed“ hat, tut der Downheap ebendies jetzt nach unten, maximal bis er ein Blatt erreicht hat. Blätter haben keine Child-Nodes, und müssen vom Downheap nicht weiter überprüft werden. Das Ankommen an einem Blatt muss beim Downheap also die Abbruchbedingung sein.

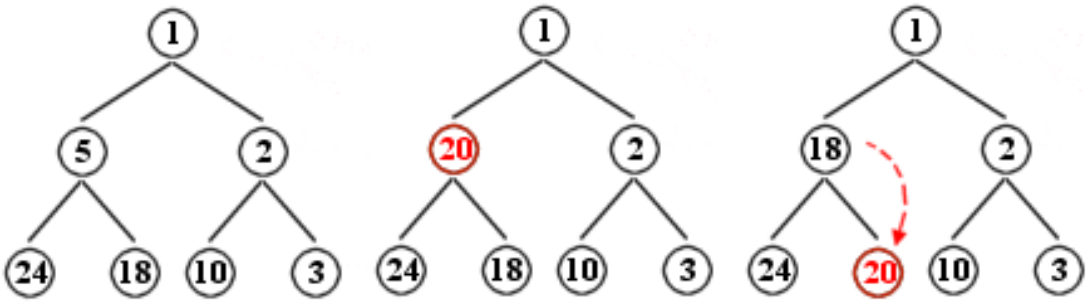

Abbildung 10: Die Downheap-Prozedur des Heapsorts

In C-Script sähe unsere Implementierung des Heapsort schließlich so aus. Hierbei speichern wir die Anzahl aller vorhandenen Nodes in der freien Position [0] des Arrays:

```
function heapsort(node);
function exchange(x,y);
function delete(node);

var ARRAY[21];    // Beliebige Größe des Array, hier 21 (20 Nodes möglich).

DEFINE n,ARRAY[0];

function heapsort(node) // node = Array-Position der zu untersuchenden Node
{
      var parent;
      var child;
// Upheap
      while(node>1)                 // Node ist keine Wurzel
      {
            parent=node/2;
            if(ARRAY[parent]>=ARRAY[node])
            {
                  exchange(parent,node);
            }
            else
            {
                  node=parent;      // Node für Downheap auf Parent setzen.
                  break;            // Wenn kein exchange, dann abbrechen.
            }
            node=parent;
      }
// Downheap
      while( node<INT(n/2+1) )      // Node ist kein Blatt
            {
            child=node*2;
            if(child+1<n)           // existiert 2.Nachfolger?
            {
                  if(ARRAY[child+1]<ARRAY[child]) { child+=1; }
            }
            if(ARRAY[node]>=ARRAY[child])
            {
                  exchange(node,child);
            }
            else { break; }         // Wenn kein exchange, dann abbrechen
            node=child;
      }
}

function exchange(x,y)
{
      i=0;
      while (i<4)
      {
            temp=NODE_LIST[x*4+i];
            NODE_LIST[x*4+i]=NODE_LIST[y*4+i];
            NODE_LIST[y*4+i]=temp;
            i+=1;
      }
}

function delete(node)
{
      OPEN[index]-=1;
      j=OPEN[index];
      exchange(node,j);
      vec_set(OPEN[j*4],NULLVECTOR);
      OPEN[j*4+3]=0;
      heapsort(1);
}
```

Der Code bietet nicht viel neues. Die Parent- und Child-Nodes werden anhand der schon oben beschriebenen Berechnungen ermittelt. Es wird überprüft ob die Heap-Ordnung noch vorhanden ist, wenn nicht werden die aktuelle und ihre Parent/Child-Node ausgetauscht und die Parent-/Child-Node für den nächsten Durchlauf als aktuelle Node definiert. Wenn die Heap-Ordnung noch steht wird die jeweilige Prozedur (Upheap oder Downheap) abgebrochen. Durch die Abbruchbedingung der while-Schleife finden die beiden Prozeduren auch ein Ende wenn der Heapsort die Wurzel erreicht hat (Upheap) oder ein Blatt erreicht wurde (Downheap). Beim Downheap ist noch zu erwähnen dass hier vor der Überprüfung der aktuellen Node mit ihrer Child-Node abgefragt wird, ob die Node über 2 Child-Nodes verfügt und welche von ihnen die mit den geringeren Nodekosten ist, so dass die aktuelle Node nur mit ihrer günstigsten Child-Node verglichen wird. Die *exchange*- und *delete*-Funktionen dürften sich von selber erklären.

Das einzige Problem dass wir jetzt noch haben ist die Tatsache, dass wir nicht einen einfachen, sondern einen 2-Dimensionalen Array als Heap darstellen müssen. Mit ein wenig Denkarbeit ist aber auch dieses Problem sofort gelöst: Wir möchten den Heap nach den Nodekosten sortieren. Laut unserer Node-Definition (s. Seiten 12-14) werden diese im 2-Dimensionalen Array an der Position *NODE_LIST[j*4+2]* gespeichert. *j* ist hierbei wie schon erwähnt die Position der Node in *NODE_LIST* von der die Nodekosten ausgelesen werden sollen. Die Variablen *parent*, *child* und der Parameter *node* aus dem Heapsort sind auch nichts anderes als Positionsangaben. Wir müssen also nur die Positionsangabe *j* im Heapsort durch die *parent*/*child* (und *node*) -Variable ersetzen, und schon wird auf die Kosten der Nodes in unserem 2-Dimensionalen Array zugegriffen. Aus NODE*_LIST[j*4+2]* wird *NODE_LIST[parent*4+2]* bzw. *NODE_LIST[child*4+2]*. Im Sourcecode selber müssen wir dann alle *parent* und *child* - Angaben durch die eben genannten ersetzen:

```
...
parent=node/2;
if(NODE_LIST[parent*4+2]>=NODE_LIST[node*4+2])
...
child=node*2;
if(child+1<n)
{
      if(NODE_LIST[(child+1)*4+2]<NODE_LIST[child*4+2]) { child+=1; }
}
if(NODE_LIST[node*4+2]>=NODE_LIST[child*4+2])
…
```

Mehr ist nicht nötig, um den Heapsort auf unsere spezielle Datenstruktur anzupassen.

4.4 Das eigentliche Script

Wenn Sie bis zu diesem Punkt gekommen sind: Gratulation! Bis jetzt verlief dieses Tutorial ja sehr trocken mit nur wenigen Zeilen Code. Aber nun ist es soweit: Nachdem ich Ihnen durch die vorangegangenen Kapitel hoffentlich ein ausreichendes Basiswissen vermittelt habe, fangen wir nun mit dem Scripten des so begehrten Pathfinding-Algorithmus an.

Begeben wir uns wieder zu der Ausgangssituation: Wir haben eine Start- und Zielposition. Das erste was wir tun müssen, ist diese Startposition als Node zu definieren. Da alle nachfolgenden Nodes immer von ihrem Vorgänger abhängig sind müssen wir eine Anfangs-Node haben, von der aus der gesamte Algorithmus startet. Hier kommen wir schon zu unserem ersten Problem: Wir können zwar über die ID (Vertex-Nummer) der Node einen Vektor ermitteln (*vec_for_vertex*), aber wie soll man aus der Startposition der Entity auf die dazugehörige Vertex-Nummer schließen? Hierfür müssen wir vor dem Start des A* nochmals weitere Berechnungen anstellen:

```
entitiy* sel_terrain;

var path_area[5];
var vecdist1[3];
var vecdist2[3];
var dist_mod[3];

DEFINE row_no,4;
DEFINE col_no,5;

function init_astar(&start)
{
      cleanup();  // Lösche alle Einträge in vom A* verwendeten Variablen
      path_area.X=sel_terrain.X+sel_terrain.MIN_X ;
      path_area.Y=sel_terrain.Y+sel_terrain.MAX_Y;
      path_area.Z=start[2];         // Z-Wert der Startposition
      path_area[row_no]=sel_terrain.skill1;
      path_area[col_no]=sel_terrain.skill2;

      vec_set(vecdist1,path_area);
      vec_set(vecdist2,path_area);
      vecdist2.X=sel_terrain.X+sel_terrain.MAX_X;
      dist_mod.X=INT(vec_dist(vecdist1,vecdist2))/path_area[row_no];
      vec_set(vecdist1,path_area);
      vec_set(vecdist2,path_area);
      vecdist1.Y=sel_terrain.Y+sel_terrain.MIN_Y;
      dist_mod.Y=INT(vec_dist(vecdist1,vecdist2))/path_area[col_no];
}
```

Über die *cleanup*-Funktion werden sämtliche Variablen die für den A* benötigt werden gelöscht, bzw. auf 0 zurückgesetzt. Vor allem der *WAYPOINT*-Array muss für einen neuen Durchlauf des A* komplett leer sein. Ein Beispiel für die *cleanup*-Funktion finden Sie im mitgelieferten C-Script. Ich halte es nicht für nötig, eine solch einfache Funktion in diesem Tutorial weiter zu beschreiben. Weiterhin gehe ich in der *init_astar*-Funktion davon aus, dass der Pointer *sel_terrain* über eine andere *action* oder *function* schon dem zu verwendendem Terrain zugewiesen wurde und auch die Startposition für das Pathfinding in einem Vektor abgespeichert wurde, der an den Parameter *start* übergeben wird. Der Vektor *path_area* wird auf die obere linke Ecke des Terrains gesetzt, um die Koordinaten für Vertex-Nummer 1 herauszubekommen. Da Terrain nicht rotiert werden kann befindet sich Vertex-Nummer 1 immer in der linken oberen Ecke des Terrains. Die äußeren Grenzen des Terrains lassen sich mit den *MIN_X/Y/Z* und *MAX_X/Y/Z* Werten abrufen, die Höhe der *path_area* wird auf die Startposition gesetzt. Weiterhin müssen wir die Vertexanzahl pro Zeile (X-Achse) und Spalte (Y-Achse) des Terrains wissen. Diese beiden Werte werden beim erstellen einer Heightmap festgelegt und müssen von Ihnen noch im WED in die ersten beiden Skills des Terrains eingetragen werden.

In der Berechnung selber messen wir zunächst die Entfernung von der linken oberen zur rechten oberen Ecke des Terrains (X-Achse). Nun dividieren wir diesen Wert durch die Vertexanzahl einer Terrain-Zeile. Der Wert den wir in *dist_mod.X* herausbekommen gibt an, wie weit ein Vertex von dem anderen in einer Zeile in Quants entfernt ist. Einen solchen Wert benötigen wir auch für die Spalten des Terrains. Deswegen führen wir die selbe Prozedur erneut aus, nur diesmal mit der Entfernung von der linken oberen Ecke zur linken unteren Ecke (Y-Achse). Wir dividieren durch die Vertexanzahl einer Terrain-Spalte und kennen nun auch den Abstand der Vertices einer Terrain-Spalte, der in *dist_mod.Y* gespeichert wird. Wie kann uns dieser Wert nun nützlich sein? Wenn wir nun die Entfernung von der aktuellen Node-Position zum Vektor der Vertex-Nummer 1 berechnen, und diesen durch den in *dist_mod.X/Y* ermittelten Abstand zweier Vertices auf X- bzw. Y-Achse dividieren, erhalten wir jeweils 2 Werte: Die Spalten- und Zeilenposition des Vertex:

```
function get_id(vector)
{
      vec_set(vecdist1,path_area);
      vec_set(vecdist2,vector);
      vecdist1.Y=vecdist2.Y;
      vecdist1.Z=vecdist2.Z;
      dist.X=INT(vec_dist(vecdist1,vecdist2)/dist_mod.X); //Spaltenposition
      vec_set(vecdist1,path_area);
      vecdist1.X=vecdist2.X;
      dist.Y=INT(vec_dist(vecdist1,vecdist2)/dist_mod.Y); //Zeilenposition
      return ((dist.X+1)+path_area[row_no]*dist.Y));
}
```

Der Vektor der aktuellen Node wird über den Parameter *vector* übergeben. Die schon beschriebene Entfernungsmessung wird zuerst nur für die X-Achse ausgeführt (Y- und Z-Werte werden gleichgesetzt). Nach der Berechung der Spaltenposition des Vertex in *dist.X* wird auch noch die Y-Entfernung gemessen und daraus auch die Zeilenposition des Vertex ermittelt und in *dist.Y* gespeichert. Doch wie bekommen wir aus diesen beiden Werten die Vertex-Nummer heraus? Ganz einfach: Um von einer Vertex-Nummer auf den Vertex zu kommen, der genau eine Zeile unter dem Ausgangsvertex liegt, müssen wir einfach nur die Anzahl der Vertices pro Zeile auf den Wert addieren: Nehmen wir an wir haben ein Terrain mit 15x15 Vertices, also mit 15 Vertices pro Zeile und Spalte. Der Vertex, der sich direkt eine Zeile unter dem Vertex mit der Nummer 8 befindet wäre Vertex 23, denn 8+15=23. Der Vertex 2 Zeilen unter 8 wäre 38, denn 8+15x2 (wir müssen 2 Zeilen überspringen, also 2x15 Vertices)=38. Wir bedienen uns der selben Formel in unserem Script und geben den ermittelten Wert zurück:

```
      return ((dist.X+1)+path_area[row_no]*dist.Y));
```

Doch was bedeutet dieses *dist.X+1*? Warum addieren wir zur Spaltenposition noch Eins dazu? Das liegt daran, dass bei der Messung der Entfernung 0 rauskommen könnte, wenn es sich bei der aktuellen Node um Vertex-Nummer 1 handelt. Wenn man 0 mit einem beliebigen Wert dividiert kommt trotzdem 0 dabei raus, und nach unserer Berechnung hätte der Vertex 1 dann die Nummer 0. Durch das addieren mit 1 gleichen wir das Problem mit der Entfernung wieder aus. Dies ist für die Zeilenposition *dist.Y* nicht notwendig, denn wenn hier bei der Entfernung 0 rauskommt befinden wir und in der ersten Zeile, die nur durch *dist.X* angegeben wird. Wir müssen die Verticeanzahl pro Zeile nicht auf die Spaltenposition aufaddieren. In unserem Anfangsbeispiel würde Vertex-Nummer 8 also so ermittelt werden: 8+15x0=8.

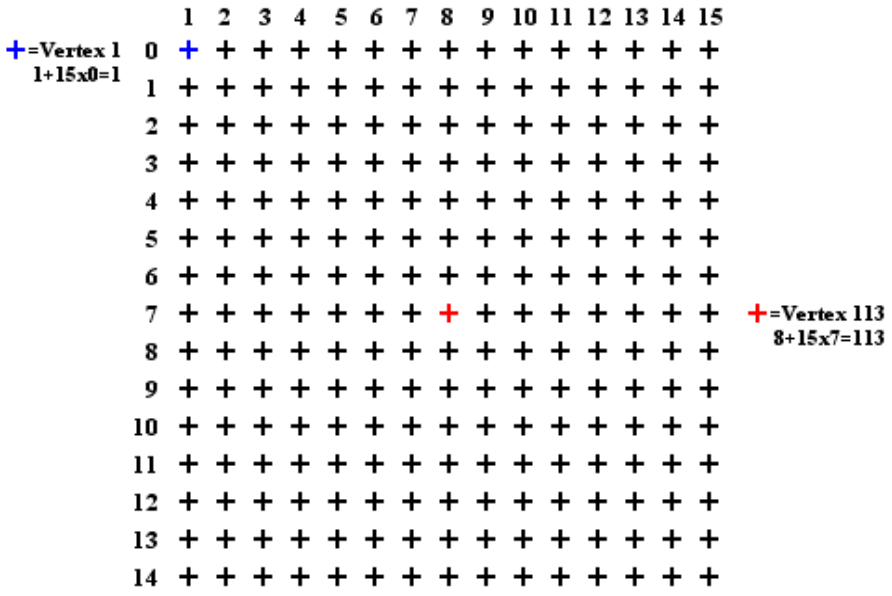

Abbildung 11: Ein Terrain mit 15x15=225 Vertices

Nachdem wir nun also jederzeit über einen Vektor einen Vertex für das Terrain ermitteln können (vorausgesetzt die Node befindet sich auf dem Terrain, wovon ich in diesem Script ausgehe, ansonsten müssten noch mal Begrenzungen in der Funktion *get_id()* gemacht werden) fangen wir endlich mit dem A* selber an. Beginnen wir mit der ersten Node, die als erstes definiert werden muss.

```
DEFINE index,0;         // Der Node-INDEX ist in Array-Eintrag 0 abgelegt

function astar(&start,&goal); // Der Funktion müssen Start- und Zielposition
                              // übermittelt werden.

var NODE_LIST[4000]     // Reicht für 1000 Nodes
var j;
var cost_value=1;

function astar(&start,&goal)
{
      init_astar(start);
      NODE_LIST[index]=1; // Setzt den Node-Zähler auf 1
      j=NODE_LIST[index]; // j wird auf 1.Eintrag gesetzt
      NODE_LIST[j*4+0]=get_id(start);     // Berechnung der 1.Node-ID
      NODE_LIST[j*4+1]=cost_value;        // Wegkosten
      NODE_LIST[j*4+2]=NODE_LIST[j*4+1]+vec_dist(start,goal); // Nodekosten
      NODE_LIST[j*4+3]=0;           // Hat keine Parent-Node,ist Start-Node

      NODE_LIST[index]+=1;                // Node Zähler=2, Platz für 2.Node
}
```

Nach der Initialisierung setzen wir *NODE_LIST[index]* auf den ersten Eintrag an dem die erste Node gespeichert werden kann. Da *NODE_LIST[index]* als Positionsangabe viel zu lang und umständlich einzutippen ist, bedienen wir uns der schon vorgestellten Positionsvariablen *j*, die mit *NODE_LIST[index]* gleichgesetzt wird. Mit *get_id(&vector)* ermitteln wir aus der Startposition heraus die erste Node-ID, die Wegkosten werden auf 1 gesetzt. Dadurch, das wir für die Wegkosten eine Variable verwenden (*cost_value*), lassen sich diese jederzeit bequem ändern und somit kann das Pathfinding individuell auf eigene Bedürfnisse angepasst werden. Es ist sowieso zu empfehlen, dass Sie sich Gedanken darüber machen, was ihr Pathfinding alles können muss und Sie somit eventuell eigene heuristische Werte zusätzlich einführen. Wie die Nodekosten zustande kommen wurde ja schon bei der Definition der Node erklärt, die Parent-Node dieser ersten Node wird auf 0 gesetzt. Diese Vertex-Nummer gibt es nicht und somit ist klar, dass diese Node keinen Parent besitzt und die Start-Node ist.

Und jetzt geht es richtig los: Der A* soll so lange laufen, bis er einen Weg zum Ziel gefunden hat. Natürlich gibt es immer Situationen in denen dies nicht möglich ist oder bei denen zu viele Nodes gebraucht werden. Mehr noch, als eigentlich für den Array vorgesehen. In solchen Fällen sollte der Algorithmus stoppen, bevor der *NODE_LIST*-Array überläuft und die A5-Engine crashed. Dies ist die wichtigste Abbruchbedingung für einen stabilen Algorithmus und ist somit auch die Bedingung, die wir für die *while*-Schleife setzen, die den Algorithmus eben bis zu diesem Punkt wiederholt. Die nächste Abbruchbedingung ist wie erwähnt das Finden eines Weges vom Start zum Ziel. Dies wird in der Schleife nun mit einem *if*-Satz gleich zu Beginn realisiert. Wenn sich die Node in einem bestimmten Abstand zum Ziel befindet (z.B. weniger als der doppelte Abstand zwischen 2 Vertices), wird eine neue Funktion gestartet in der die Koordinaten aller Nodes, die auf dem Weg zum Ziel liegen, in einem weiteren Array gespeichert werden. Dies wird über das Zurückverfolgen der Parant-Nodes bis zum Startpunkt realisiert (hierzu später mehr). Der A*-Algorithmus wäre an dieser Stelle beendet, die Koordinaten können nun aber von einer Entity benutzt werden, um an ihnen entlang zum Ziel zu finden.

```
...
var WAYPOINTS[3000];    // Reicht für 1000 Wegpunkte/Vektoren
var i;
var current_vec[3];
...
function astar(&start,&goal)
{
      ...
      NODE_LIST[index]+=1;
      while(NODE_LIST[index]*4<NODE_LIST.length)
      {           // Solange Anzahl Nodes nicht Array-Größe überschreitet
            vec_for_vertex(current_vec,sel_terrain,NODE_LIST[1*4]);
            if(vec_dist(current_vec,goal)<=dist.X*2)
            {     // Befindet sich die Node unmittelbar vor dem Zielpunkt?
                  vec_set(WAYPOINTS[0],goal);
                  i=3;
                  get_path(NODE_LIST[1*4]);
                  // Konstruiere einen Path von der Goal-Node über die
                  // Parent-Nodes zum Startpunkt zurück
                  return;     // Beende den Algorithmus
            }
            define_children(NODE_LIST[1*4]);
      }
}
```

Wenn das Ziel noch nicht erreicht ist muss der Algorithmus nach neuen Child-Nodes suchen. Wie schon erwähnt kann unsere Entity von ihrer Position aus 8 verschiedene Richtungen wählen: Norden, Nordost, Osten, Südost, Süden, Südwest, Westen, Nordwest. Diese 8 Richtungen werden ganz einfach ermittelt, in dem wir von der aktuellen Node-Position in unserem Vertex-Gitter einfach eine Zeile/Spalte weiter bzw. zurück gehen. Dies führen wir in der Funktion *define_children()* aus:

```
...
var node_child[8];      // 8 Neue Nodes

function define_children(parent_id)
{
  if(parent_id<path_area[row_no])
  {   node_child[0]=0;  }
  else
  {
    node_child[0]=parent_id-path_area[row_no]*vertex_dist;
  }
  if(parent_id<path_area[row_no] || frc((parent_id)/path_area[row_no])==0)
  {   node_child[1]=0;  }
  else
  {
    node_child[1]=parent_id-path_area[row_no]*vertex_dist+1*vertex_dist;
  }
  if(frc((parent_id)/path_area[row_no])==0)
  {   node_child[2]=0;  }
  else
  {
    node_child[2]=parent_id+1*vertex_dist;
  }
  if(parent_id>path_area[row_no]*(path_area[col_no]-1) ||
               frc((parent_id)/path_area[row_no])==0)
  {   node_child[3]=0;  }
  else
  {
    node_child[3]=parent_id+path_area[row_no]*vertex_dist+1*vertex_dist;
  }
```

```
if(parent_id>path_area[row_no]*(path_area[col_no]-1))
{   node_child[4]=0;  }
else
{
  node_child[4]=parent_id+path_area[row_no]*vertex_dist;
}
if(parent_id>path_area[row_no]*(path_area[col_no]-1) ||
            frc((parent_id-1)/path_area[row_no])==0)
{   node_child[5]=0;  }
else
{
  node_child[5]=parent_id+path_area[row_no]*vertex_dist-1*vertex_dist;
}
if(frc((parent_id-1)/path_area[row_no])==0)
{   node_child[6]=0;  }
else
{
  node_child[6]=parent_id-1*vertex_dist;
}
if(parent_id<path_area[row_no] ||frc((parent_id-1)/path_area[row_no])==0)
{   node_child[7]=0;  }
else
{
  node_child[7]=parent_id-path_area[row_no]*vertex_dist-1*vertex_dist;
}
```

Ignorieren wir vorerst die vielen *if*-Bedingungen und schauen uns die Anweisungen in den *else*-Klammern an:

```
node_child[0]=parent_id-path_area[row_no]*vertex_dist;
```

Dies ist nichts anderes als das schon auf S.21 beschriebene ermitteln eines Vertex, der sich eine Zeile unter (bzw. in diesem Fall über) dem aktuellen Vertex befindet. *path_area[row_no]* gibt die Anzahl der Vertices pro Zeile an, die von der aktuellen Node (die nun zur Parent- der zukünftigen Child-Nodes geworden ist, deswegen wurde der Parameter *parent_id* genannt) subtrahiert wird um die Vertex-Nummer in der nächst höheren Zeile herauszubekommen. Die Variable *vertex_dist* ist eigentlich überflüssig für diese Berechnung, stellt aber ein willkommenes Feature dar: Wenn man für sein Pathfinding nicht jeden Vertex benutzen muss (z.B. weil man ein Terrain mit vielen Vertices und kleinem Scale hat), kann man über diese Variable angeben ob und wenn wie viele Vertices übersprungen werden können. Bei einem Wert von 1 wird jeder Vertex für das Pathfinding benutzt, bei einem Wert von 2 nur jeder zweite, so dass sich in unserem Fall die erste Child-Node nicht ein, sondern 2 Zeilen über der aktuellen Node befinden würde. Durch *vertex_dist* lässt sich das Pathfinding von der Anzahl der Vertices auf dem Terrain unabhängiger machen und die Anzahl der Nodes vom Weg zum Ziel evtl. verkürzen. Allerdings kann das Ignorieren von Vertices auch dazu führen, dass einige Wege auf dem Terrain nicht mehr gefunden werden. Hier sollte man vorher testen, mit welcher *vertex_dist* das Pathfinding auf Ihrem Terrain am besten läuft.

Die nächste Child-Node befindet sich im Nordosten von der aktuellen Node. Also nach wie vor eine (oder mehrere bei *vertex_dist*>1) Zeile(n) über der aktuellen, jedoch auch noch eine Spalte weiter. Eine Spalte bei einer Vertex-Anordnung wie beim Terrain ist nichts anderes als eine Einheit: Der Vertex, der sich eine Spalte weiter von Vertex-Nummer 1 befindet, ist die Nummer 2. Es muss also nur noch zusätzlich mit 1 addiert werden, um eine Spalte weiterzukommen. Aber auch hier lässt sich die Anzahl der Spalten noch einmal mit *vertex_dist* modifizieren. Die ganze Berechnung sieht dann so aus:

```
node_child[1]=parent_id-path_area[row_no]*vertex_dist+1*vertex_dist;
```

Die restlichen Child-Nodes werden nun auf die selbe Weise definiert, nur das je nach Richtung ein Vorzeichenwechsel stattfindet. Anhand des Codes lässt sich dies sicher leicht nachvollziehen. Beschäftigen wir uns nun mit den etwas komplizierteren *if*-Bedingungen:

Die ganzen *if*-Bedingungen sind dazu da das Erstellen neuer Nodes an den Grenzen des Terrains zu verhindern. Die Grenzen sind hierbei die erste und die letzte Zeile sowie die erste und die letzte Spalte. Beim Erreichen dieser Grenzen sind bestimmte Richtungen nicht mehr begehbar (da für sie kein Vertex mehr existiert). Für jede der 8 Richtungen wird also vorher abgefragt, ob eine für sie relevante Grenze erreicht wurde. Hierbei handelt es sich immer um eine Kombination aus den Abfragen der 4 Grenzbereiche, so dass ich auch nur das Ermitteln dieser 4 Grenzbereiche erläutere und nicht alle 8 Abfragen für jede Richtung.

Ob sich eine Vertex-Nummer in der ersten Zeile befindet und somit an der oberen Grenze, lässt sich recht leicht herausfinden: Da sich in der ersten Zeile die Vertex-Nummern von 1 bis Anzahl der Vertices pro Zeile befinden (z.B. 15), sind alle Vertex-Nummern die kleiner als die Anzahl der Vertices pro Zeile sind (<15) nicht begehbar. Ein weiterer Schritt nach Norden würde eine Node-ID außerhalb des Terrains erzeugen, was eine Fehlermeldung zur Ursache hätte.

```
if(parent_id<path_area[row_no])
```

Solange die Vertex-Nummer also kleiner als die Anzahl der Vertices pro Zeile ist, wird die *node_child* für die Richtung Norden nicht erstellt, ihr Wert auf 0 gesetzt. Dies kann keine Vertex-Nummer sein und wird somit später zur Abfrage benutzt, ob sich in *node_child* eine Vertex-Nummer befindet.

Das Abfragen nach der letzten Zeile und somit der unteren Grenze ist fast genauso leicht: Dies lässt sich ganz einfach realisieren wenn wir die letzte Vertex-Nummer der vorletzten Zeile ermitteln. Alle Nummern, die größer als die ermittelte Vertex-Nummer sind, gehören zur letzten Zeile. Hierzu wieder unser Beispiel mit dem 15x15-Vertices-Terrain: 15 Vertices x 14 Zeilen (1 weniger als Maximalanzahl an Zeilen)=210. Vertices größer als 210 werden durch die *if*-Bedingung keine „südlichen“ Child-Nodes erzeugen.

```
if(parent_id>path_area[row_no]*(path_area[col_no]-1))
```

Als nächstes widmen wir uns der letzten Spalte: Hier ist uns mal wieder wie so oft die Anzahl der Vertices pro Zeile behilflich. Wenn wir eine der Vertex-Nummern in der letzten Spalte durch diese Anzahl dividieren, muss das Ergebnis der Zeile entsprechen, in der sich der Vertex befindet. Dies ist keine Zauberei sondern einfachste Mathematik: Gehen wir wieder von unserem 15x15-Vertices-Terrain aus. Schon an der Bezeichnung lässt sich die Lösung erkennen: Wir haben ein Terrain mit 15 Zeilen, in denen sich jeweils 15 Vertices befinden. 15x15=225, welches dem letzten Vertex entspricht, der sich somit auch in der letzten Spalte befindet. 225:15=15, denn er befindet sich in der 15.Zeile. Wenn es sich nun aber um einen Vertex handelt, der sich nicht in der letzten Spalte befindet, kommt bei einer Division durch die Anzahl der Vertices pro Zeile kein gerader Wert raus (z.B. 16:15=1,06). Also prüfen wir mit der Engine-internen *frc*-Funktion, ob wir einen ungeraden oder geraden Wert nach der Division herausbekommen haben, in letzterem Falle wird *node_child* auf 0 gesetzt, um diese Richtung als nicht begehbar zu kennzeichnen:

```
if(frc((parent_id)/path_area[row_no])==0)
```

Schlussendlich brauchen wir noch eine Abfrage, um herauszufinden ob sich ein Vertex in der ersten Vertex-Spalte des Terrains befindet. Hierfür gehen wir ähnlich vor: Denn wenn sich die Vertex-Nummer in der ersten Spalte befindet, befindet sich Vertex-Nummer *-1* in der letzten Spalte! Also benutzen wir im Grunde genommen wieder die selbe Abfrage, nur dass wir diesmal von der aktuellen Vertex-Nummer 1 abziehen.

```
if(frc((parent_id-1)/path_area[row_no])==0)
```

Diese if-Bedingungen werden je nach Richtung miteinander kombiniert. Sind die angegebenen Bedingungen nicht erfüllt, befindet sich die aktuelle Parent-Node also nicht an einer der Grenzen des Terrains, wird durch die Anweisungen in den *else*-Klammern die Node-ID einer neuen Child-Node berechnet.

Wenn man für jedes *node_child* eine Entity per *ent_create()* erstellen würde, sähe das ganze in der Spielwelt dann ungefähr so aus:

Abbildung 12: Darstellung der ersten 8 Nachbarnodes

Nachdem wir nun die neuen IDs für die 8 Child-Nodes kennen, müssen wir noch die restlichen Werte für diese Nodes berechnen. Dies können wir wieder bequem mit einer *while*-Schleife lösen, die wir für jede Child-Node einmal durchlaufen lassen, also 8 mal. Doch bevor wir das machen, speichern wir die derzeit günstigste Node in die Variable *current_node*. In der folgenden Schleife bei der die neuen Child-Nodes der *NODE_LIST* hinzugefügt werden, wird für jede neue Node der Heapsort ausgeführt. Dabei ist es sehr wahrscheinlich, dass es eine neue günstigste Node gibt. Bevor die Schleife aber noch nicht beendet ist und somit noch nicht alle Child-Nodes erstellt wurden, muss die vormals günstigste Node - die aktuelle Parent- der Child-Nodes - nach wie vor abrufbar sein.

```
…
var current_node[4];
…
define_children(parent_id);
vec_set(current_node,NODE_LIST[1*4]);
current_node[3]=NODE_LIST[1*4+3];
i=0;
while(i<8)        // i< als 8 Nodes
{
  if(node_child[i]>0)
  {
    j=NODE_LIST[index];       // j= Position für neue Node
    vec_for_vertex(current_vec,sel_terrain,current_node[0]);
    vec_set(vecdist1,current_vec);
    vec_set(vecdist2,NULLVECTOR);
    vec_for_vertex(current_vec,sel_terrain,node_child[i]);
    vecdist2.Z=current_vec.Z;
    vecdist1.X=0;
    vecdist1.Y=0;
    if(vecdist(vecidst1,vecdist2)<dist.X) // Höhendistanz<
      {     // Vertex nicht zu hoch gelegen
        NODE_LIST[j*4+0]=node_child[i];
        NODE_LIST[j*4+1]=current_node[1]+cost_value;
        NODE_LIST[j*4+2]=NODE_LIST[j*4+1]+vec_dist(current_vec,goal);
        NODE_LIST[j*4+3]=current_node[0];
        heapsort(j);
        NODE_LIST[index]+=1;
      }
  }   // Ende if(node_child[i]>=0)
  i+=1;     // Zugriff auf nächsten Vektor im Array im nächsten Durchlauf
}     // Ende while(i<8)
}     // Ende while(NODE_LIST[index]*4<NODE_LIST.length)
}     // Ende astar-Function
```

Dieser letzte Teil dürfte recht einfach zu verstehen sein: Wenn sich in node_child[i] eine Node befindet (der Wert also nicht 0 ist) kann die Node erstellt werden. Hierfür wird aber erst noch einmal abgefragt, ob sich die Node nicht „in einem Hindernis“ befindet. Auf unserem Terrain definieren wir einfach, das ein Hindernis vorliegt sobald die Child-Node eine größere Höhendistanz (Distanz auf der Z-Achse) zu ihrer Parent-Node hat als der Abstand zwischen 2 Nodes (in diesem Fall dist.X). Nicht die beste Hinderniserkennung, aber für unsere Zwecke reicht es erstmal.

Vom Prinzip her war es das auch schon. Die neue Child-Node wurde in *NODE_LIST* aufgenommen und per Heapsort einsortiert. Nun, das ganze dürfte noch nicht so funktionieren wie wir uns das vorstellen, dazu fehlt noch der Feinschliff. Aber vorerst wollen wir uns einmal ansehen, wie anhand der Nodes ein Path für die Entity erstellt wird. Wie man schon im Code sehen konnte, wird wenn sich eine Node nahe dem Ziel befindet die Funktion *get_path* aufgerufen und die dem Ziele nahe gelegenen Node übergeben. Nun wird von der übergebenen Node anhand der Parent-Einträge der Weg bis zur Start-Node zurückverfolgt:

```
function get_path(parent_id)
{
      vec_for_vertex(current_vec,sel_terrain,parent_id);
      temp=INT(i/3);
      if(temp<NODE_LIST[index]){vec_set(WAYPOINTS[i],current_vec); }
}
```

Die Koordinaten der ID werden per *vec_for_vertex()* in den Vektor *current_vec* abgelegt. Dieser wird dann in den *WAYPOINTS*-Array kopiert. Nun kennen wir also die Koordinaten der Parent-Node von der Goal-Node. Doch wie gelangen wir jetzt zu der nächsten Parent-Node? Wir müssten nun auf die Einträge der aktuellen Parent-Node zugreifen können, um deren Parent-Node zu ermitteln. Doch wie finden wir die Position der aktuellen Parent-Node im *NODE_LIST*-Array heraus? Am einfachsten wäre es wohl, alle Einträge der *NODE_LIST* zu durchsuchen bis wir die ID der aktuellen Parent-Node gefunden haben. In diesem Beispiel hat die *NODE_LIST* auch nur 4000 Einträge, die sind schnell durchsucht. Doch was wenn Sie mehr als 4000 Nodes für ihr Pathfinding speichern müssen? Bei langen oder komplizierten Wegen sind 4000 Nodes nicht sehr viel. Ihre *NODE_LIST* wird wahrscheinlich um einiges größer sein als in diesem Beispiel. Und je größer die *NODE_LIST*, desto länger dauert ein Durchsuchen des Arrays. Eine Suche nach dem Eintrag ist also nicht zu empfehlen. Wie wäre es dann mit einer erneuten Sortierung der *NODE_LIST*, diesmal nicht nach den Nodekosten, sondern nach den Parent-Nodes? Nun, nur weil die Parent-Node geographisch „vor“ der aktuellen Node liegt, heißt das nicht dass die ID der Parent-Node einen kleineren Wert hat als die der aktuellen Node. Die Parent-Nodes lassen sich in der *NODE_LIST* in keine brauchbare Ordnung sortieren.

Am schnellsten würde das Ermitteln der Position einer Parent-Node gehen, indem wir die Position einfach kennen würden. So seltsam sich das jetzt anhören mag, so einfach ist die Umsetzung: Wir müssen einfach jedes Mal wenn eine Node in der *NODE_LIST* erstellt wird, deren aktuelle Position in der *NODE_LIST* in einem separaten Array speichern. Hierbei würde ein Array-Eintrag einer Node-ID entsprechen. Der Array muss also so viele Einträge fassen, wie es mögliche Nodes gibt. In diesem Beispiel wären das 70x70=4900 +1 Einträge, denn die Nummerierung bei Terrain beginnt mit 1. Und da es Übersichtlicher und einfacher ist, werden wir auch hier wie in *NODE_LIST* den 0.Eintrag des Arrays leer lassen, und mit dem ersten Eintrag beginnen. Mit der Deklaration *var ID_LIST[4901];* können wir ganz einfach die aktuelle *NODE_LIST*-Position der Node mit der ID 4900 erfahren, indem wir diese mit dem Befehl *node_id=ID_LIST[4900]* abfragen. Sollte hierbei 0 das Ergebnis sein, wurde die Node bisher noch nicht erforscht, denn auch in der *NODE_LIST* gibt es keine 0.Position für Nodes, da an dieser Position die Anzahl aller Nodes abgespeichert wird.

Vorraussetzung hierfür ist natürlich, das wir in *ID_LIST* immer die aktuelle Position der Node in *NODE_LIST* abspeichern. Jedes Mal, wenn sich die Position einer Node in der *NODE_LIST* ändert, z.B. durch den Heapsort, muss auch der *ID_LIST* diese neue Positionsangabe übergeben werden. Dies lässt sich mit nur wenigen Zeilen Code in der *exchange*-Funktion realisieren:

```
function exchange(x,y)
{
      i=0;
      while (i<4)
      {
            temp=NODE_LIST[x*4+i];
            NODE_LIST[x*4+i2=NODE_LIST[y*4+i];
            NODE_LIST[y*4+i]=temp;
            i+=1;
      }
      ID_LIST[NODE_LIST[x*4+0]]=x;
      ID_LIST[NODE_LIST[y*4+0]]=y;
}
```

In unserer *get_path*-Funktion können wir nun also ganz einfach die *NODE_LIST*-Position der aktuellen Parent-Node erfahren. Mit dieser Positionsangabe greifen wir wieder auf die *NODE_LIST* zu, und überprüfen ob die aktuelle Parent-Node auch eine Parent-Node besitzt. Wenn dies der Fall ist, wird *get_path()* einfach noch einmal ausgeführt, diesmal aber mit einer neuen Parent-Node. Eine Funktion die sich selber aufruft nennt man rekursiv. Sie wird sich so lange selber aufrufen, bis es keine Parent-Node mehr gibt (der Start-Node haben wir ganz am Anfang die Parent-Node 0 verpasst, die es nicht geben kann). Damit befinden sich zum Schluß im *WAYPOINTS*-Array (der übrigens auch Multidimensional ist) alle Nodes die einen zum Ziel führen, mit der Zielposition als ersten Eintrag, und der Startposition als letzten. Wenn man diesem Weg also folgen möchte, muss man vom letzten erstellten Eintrag bis 0 runterzählen, um die Koordinaten in der richtigen Reihenfolge ausgeben zu können.

```
function get_path(parent_id)
{
      vec_for_vertex(current_vec,sel_terrain,parent_id);
      temp=INT(i/3);
      if(temp<NODE_LIST[index]){vec_set(WAYPOINTS[i],current_vec); }
      j=ID_LIST[parent_id];         // NODE_LIST-Position der Parent-Node
      if(NODE_LIST[j*4+3]!=0)
      {
            i+=3;
            get_path(NODE_LIST[j*4+3]);
      }
}
```

Das Kreieren des Pathes ist aber nicht die einzige Möglichkeit, bei der wir auf unsere jederzeit verfügbare Positionsangabe der Nodes auf der *NODE_LIST* zurückgreifen können: Wir können nun jederzeit einen Vergleich zwischen schon vorhanden und gerade neu erforschten Nodes anstellen und somit schon vor dem Erstellen einer Node in der *NODE_LIST* überprüfen, ob es überhaupt sinnvoll ist, diese Node abzuspeichern. Damit beginnen wir den Feinschliff des Algorithmus: Es ist uns nun möglich, die von der Heuristik als gut befundene Nodes noch einmal unter anderen Gesichtspunkten zu betrachten, und können den Algorithmus somit effizienter machen, bisherige Problemstellen beheben und auf eigene Wünsche anpassen. Fangen wir am besten noch einmal von vorne an:

```
var NODE_LIST[4000];          // 4*1000
var ID_LIST[4901];            // 70*70+1
var WAYPOINTS[3000]           // 3*1000
```

```
var j;
var i;
var i2;
var cost_value=1;
var node_child[8];
var current_vec[3];
var current_node[4];

DEFINE index,0;

function init_astar(&start);
function astar(&start,&goal);
function get_id(&vector);
function define_children(&vector);
function heapsort(v);
function exchange(x,y);
function get_path(parent_id);
```

Auch wenn wir alle relevanten Funktionen hiermit deklariert haben, beschäftige ich mich im folgenden nur noch einmal mit dem Hauptalgorithmus, die anderen Funktionen wurden schon ausführlich besprochen, bzw. werden nicht benötigt (wie z.B. *delete()* , s.U.). Vieles dürfte Ihnen ja schon bekannt sein, Veränderungen gegenüber dem zuvor besprochenen Sourcecode sind Orange dargestellt:

```
function astar(&start,&goal)
{
  init_astar(start);
  NODE_LIST[index]=1;
  j=NODE_LIST[index];
  NODE_LIST[j*4+0]=get_id(start);
  NODE_LIST[j*4+1]=cost_value;
  NODE_LIST[j*4+2]=NODE_LIST[j*4+1]+vec_dist(start,goal);
  NODE_LIST[j*4+3]=NULL;
  ID_LIST[NODE_LIST[j*4+0]]=j;      // Start-Node-Position gespeichert
  NODE_LIST[index]+=1;
  while (NODE_LIST[index]*4<NODE_LIST.length)
  {
    vec_for_vertex(current_vec,sel_terrain,NODE_LIST[1*4]);
    if(vec_dist(current_vec,goal)<=dist.X*2)
    {       // Ist Node=Zielpunkt?
      vec_set(WAYPOINTS[0],goal);
      i=3;
      get_path(NODE_LIST[1*4+3]);
      return(1);
    }
    if(NODE_LIST[1*4]==current_node[0])
    {
      NODE_LIST[1*4+2]=100000;
      heapsort(1);   //Starte einen Heapsort von der Wurzel aus
    }
    if(NODE_LIST[1*4+2]==100000) {return (0);}
    define_children(NODE_LIST[1*4]);
    vec_set(current_node,NODE_LIST[1*4]);
    current_node[3]=NODE_LIST[1*4+3];
    while(i<8)          // i< als 8 Nodes
    {
      if(node_child[i]>0)
      {
        j=NODE_LIST[index];         // j= Position für neue Node
        vec_for_vertex(current_vec,sel_terrain,current_node[0]);
        vec_set(vecdist1,current_vec);
        vec_set(vecdist2,NULLVECTOR);
        vec_for_vertex(current_vec,sel_terrain,node_child[i]);
```

```
        node_costs=current_node[1]+cost_value+vec_dist(current_vec,goal);
            // Kosten=Kosten der Parent-Node+1+Abstand zum Ziel
        if(j<=0 || j*4>=NODE_LIST.length){break;}
        if(ID_LIST[node_child[i]]!=0)
        {         // Existiert diese Node bereits?
          j=ID_LIST[node_child[i]];
          if(NODE_LIST[j*4+2]==100000){i+=1; continue;} // Sackgassen-Node?
          if(NODE_LIST[j*4+1]==current_node[1]-cost_value){i+=1; continue;}
              // Hinter der aktuellen Node liegende Nodes (einen Wegkosten-
              // Wert weniger) müssen nicht mehr untersucht werden.
          if(node_costs>NODE_LIST[j*4+2]) {i+=1; continue;}
              // Aktuelle Berechnung höher als vorherige
        }
        vecdist2.Z=current_vec.Z;
        vecdist1.X=0;
        vecdist1.Y=0;
        if(vecdist(vecidst1,vecdist2)<dist.X)
        {
          NODE_LIST[j*4+0]=node_child[i];
          NODE_LIST[j*4+1]=current_node[1]+cost_value;
          NODE_LIST[j*4+2]=node_costs;
          NODE_LIST[j*4+3]=current_node[0];
          ID_LIST[node_child[i]]=j;      // Speicher Position der neuen Node
          NODE_LIST[heap_index]=NODE_LIST[index];
          if(j!=NODE_LIST[index]){NODE_LIST[heap_index]-=1;}
          heapsort(j);
          if(j==NODE_LIST[index])
          {
            NODE_LIST[index]+=1;
          }
        }   // Ende if(vecdist(vedist1,vecdist2)<start.Z)
      }     // Ende if(node_child[i]>=0)
      i+=1;
    }       // Ende while(i<8)
  }         // Ende while (NODE_LIST[index]*4<NODE_LIST.length)
  return (0);
}           // Ende astar();
```

Befindet sich in dem Sourcecode mehr Orange als sie erwartet haben? Keine Angst, das ganze ist einfacher als es aussieht: Gerade bei komplizierten Hindernisstrukturen kann es vorkommen, dass der A* in eine Sackgasse gerät, aus der er trotz Wegkosten/Heuristik nicht mehr rauskommt. Im Klartext heißt das: Es wird keine bessere Node mehr gefunden als die soeben besuchte. Unser Algorithmus würde also ein und die selbe Node immer wieder besuchen ohne weiter voran zu kommen. Um dies zu verhindern überprüfen wir vor der Suche nach neuen Nodes, ob die derzeit beste Node die selbe ist, die im letzten Durchlauf der Schleife besucht wurde. Der erste Gedanke, der einem kommt, wäre wahrscheinlich diese Node einfach zu löschen. Das Problem daran ist, dass dadurch die Node zu einem späteren Zeitpunkt wieder besucht werden könnte und der Algorithmus dadurch immer wieder in die Sackgasse laufen würde. Die Sackgassen-Node muss erhalten bleiben und als solche „markiert“ werden, damit der Algorithmus diese ab sofort nicht mehr besucht. Dies tun wir indem wir die Nodekosten auf einen Wert setzen, dessen natürliches auftauchen sehr unwahrscheinlich ist (wir werden ihn zu einem späteren Zeitpunkt noch einmal abfragen müssen, es handelt sich hierbei um die „Markierung“). Wenn dieser Wert gleichzeitig noch sehr hoch ist (in diesem Fall 100000), wird diese Node durch den darauf folgenden Heapsort sehr weit nach unten befördert, und an erster Stelle in der *NODE_LIST* steht eine neue beste Node, mit der wir weiter arbeiten können. Zur Sicherheit fragen wir diese neue Node gleich ab, ob sie auch eine Sackgassen-Node ist (Nodekosten = 100000). Ist dies der Fall, wird der komplette Algorithmus abgebrochen. Dies ist eine Vorsichtsmaßnahme um an bestimmten Problemstellen keine Endlosschleife hervorzurufen. Dieser Fall dürfte jedoch in den seltensten Fällen auftreten (z.B. wenn die Start-Position von Hindernissen umgeben ist und

somit keine Ausweichmöglichkeiten vorhanden sind). Durch diese Methode wird die delete-Funktion nicht benötigt, im Anhang werden Sie erfahren, warum ich sie beim Heapsort trotzdem aufgeführt habe.

Die nächste Überprüfung, ob es sich bei der aktuellen Node um eine Sackgassen-Node handelt, wird beim Erforschen der Child-Nodes vorgenommen. An dieser Stelle werden die Child-Nodes auf bereits vorhandene Nodes mit der selben ID überprüft, um zu entscheiden ob die bereits vorhandene Node in der *NODE_LIST* durch die neue Child-Node ersetzt werden soll. Hierfür wird abgefragt, ob es für die aktuelle Node schon einen Eintrag in *ID_LIST* gibt. Ist dies der Fall, wird geprüft ob es sich um eine bekannte Sackgassen-Node handelt. Wenn auch dies zutrifft, wird durch den *continue*-Befehl der komplette Rest dieses Schleifendurchlaufes übersprungen und mit der Überprüfung der nächsten Child-Node Fortgefahren. Dadurch wird die als Sackgassen-Node erkannte Child-Node nicht erneut in der NODE_LIST eingetragen. Das selbe geschieht auch mit Child-Nodes, die schon vom Parent der aktuellen Node untersucht wurden (die Wegkosten also nur um genau 1 kleiner sind als bei der aktuellen Node). Dies würde einen Schritt zurück bedeuten, der auch dazu führen kann, dass sich der Algorithmus irgendwann im Kreis dreht:

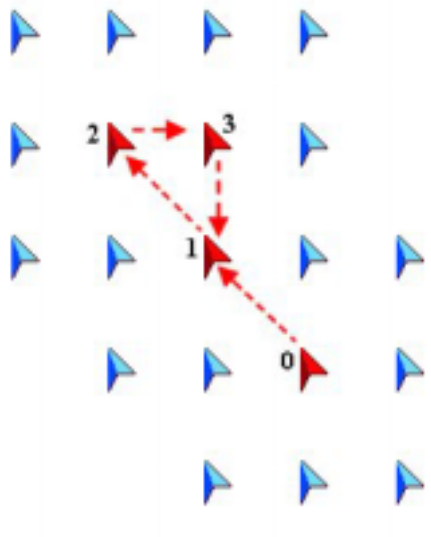

Abbildung 13: Darstellung eines A*, der im Kreis läuft

Eine Child-Node wird weiterhin verworfen, wenn ihre Nodekosten größer sind als die der bereits vorhandenen Node. Sollte sie jedoch kleinere Nodekosten aufweisen (was nicht mehr extra abgefragt werden muss), so werden die bisher eingetragenen Kosten dieser Node durch die neuen ersetzt. Hierfür wurde schon zuvor *j* auf die Position der bereits vorhandenen Node gesetzt. Dies ist sinnvoll, wenn der Algorithmus in eine Sackgasse gelaufen ist und nun einen neuen Weg berechnet. Schon zuvor gefundene Nodes könnten auf einem neuen Weg weniger Kosten verursachen und können durch diese Abfragen verändert werden. Da Nodes mit der selben ID (nur diese werden hier ja miteinander verglichen) auch die selben Koordinaten und somit die selbe Distanz zum Ziel haben kann es sich bei den kleineren Nodekosten nur um einen Unterschied der Wegkosten handeln, was trotzdem ausschlaggebend für einen guten Weg sein kann.

Wenn die aktuelle Child-Node noch keinen Eintrag in der *ID_LIST* hat, so wird sie ganz normal an der nächsten freien Stelle auf der *NODE_LIST* (*j=NODE_LIST[index]*) eingetragen. In diesem Fall wird auch der NODE_LIST[index] später erhöht. Wenn es sich aber um ein Ersetzen einer vorhandenen Node handelt, ist die Anzahl der Nodes gleich geblieben. Dadurch ist die Anzahl der Nodes in *NODE_LIST[index]* um eins zu hoch. Der Heapsort muss aber für einige Rechenschritte die richtige Anzahl wissen, so dass wir vor dem

Heapsort noch einmal Abfragen ob *j* und *NODE_LIST[index]* übereinstimmt (wenn nicht wurde keine neue Node hinzugefügt). Wenn keine neue Node hinzugefügt wurde wird nicht von *NODE_LIST*, sondern von dem neuen Wert *NODE_LIST[heap_index]* 1 subtrahiert, so dass der Heapsort wieder mit der richtigen Anzahl Nodes sortieren kann. Hierfür müssen allerdings alle Stellen im Heapsort, an denen *NODE_LIST[index]* vorkam, durch *NODE_LIST[heap_index]* ersetzt werden. *NODE_LIST[heap_index]* ist nichts anderes als *NODE_LIST[1]*. Da die erste Node erst in *NODE_LIST[4]* beginnt, haben wir die ersten 4 Einträge zur freien Verfügung. Nach dem Heapsort wird dann noch mal abgefragt ob eine neue Node hinzugefügt wurde (nach fast der der selben Methode) und daraufhin entschieden, ob *NODE_LIST[index]* noch erhöht werden muss oder nicht.

Damit haben wir es geschafft. Wir haben einen funktionierenden Pathfinding-Algorithmus in A* realisiert. Das einzige was noch fehlt sind Scripte zum benutzerfreundlichen definieren von Start- und Zielpunkt, als auch eine Möglichkeit die Wegpunkte in *WAYPOINTS* zum Bewegen einer Entity auszulesen. Hierzu finden Sie Beispiele in dem mitgelieferten C-Script. Im Tutorial möchte ich mich nun mit einer möglichen Pathfinding-Lösung ohne Terrain auseinandersetzen.

4.5 Ohne Terrain...

Wie schon vorher angekündigt möchte ich nun eine Pathfinding-Lösung ohne Terrain vorstellen. Natürlich bedienen wir uns hierbei wieder dem A*, es sind noch nicht mal viele Änderungen am schon vorhandenen Algorithmus nötig, um ihn auf unsere neuen Bedürfnisse anzupassen. Erinnern wir uns zurück: Eines der Hauptprobleme war, jeder Node eine individuelle ID zu verpassen und von dieser ID wieder auf die Welt-Koordinaten schließen zu können. Dieses Problem ließ sich leicht umgehen, da ein Terrain aus einem Vertex-Gitter mit eindeutiger Nummerierung besteht, und uns die A5-Engine mit *vec_for_vertex* eine Funktion zur Verfügung stellt, mit der wir jederzeit von der Vertex-Nummer auf die Welt-Koordinaten schließen konnten. Doch dieses Vertex-Gitter steht uns ohne Terrain nicht mehr zur Verfügung. Was also tun? Die Lösung ist mal wieder absolut simpel: Wir kreieren einfach ein eigenes, sozusagen „virtuelles“ Vertex-Gitter. Das einzige, was wir hierfür brauchen, sind die Koordinaten des ersten Vertex, um mit deren Hilfe die Vertex-Nummer der Startposition zu ermitteln. Alle anderen Vertices werden über *define_children* abhängig von dem Startvertex definiert. Somit ist es egal ob wir ein „echtes“ oder „virtuelles“ Vertex-Gitter vorliegen haben. Wichtig ist nur, das wir einen Ersatz für *vec_for_vertex* finden, denn diese Funktion ist natürlich nur bei echten Vertices gültig. Fangen wir aber zunächst mit der Initialisierung an, und somit bei der Definition der ersten Vertex-Nummer auf unserem virtuellen Vertex-Gitter:

Hierfür müssen wir wissen, wie viele Zeilen und Spalten das Vertex-Gitter haben soll und wie groß der Abstand zwischen den einzelnen „virtuellen Vertices“ ist. Da diese ja nichts anderes als unsere Node-ID’s darstellen, verwende ich für sie auch ab sofort wieder den Begriff Nodes. Den Abstand und die Zeilen-/Spaltenanzahl können wir komplett selber definieren, wenn wir einige Dinge von vornherein beachten: Mehr als 1000 Nodes pro Achse sind nicht möglich, den 1000x1000=1000000 und damit sind wir an der Grenze der Variablengröße in C-Script angekommen. Derart viele Nodes sind auch nicht zu empfehlen, da die *ID_LIST* diese alle verwalten muss. Das Problem dabei ist das Leeren der *ID_LIST* in der cleanup-Funktion. Je größer die *ID_LIST*, desto länger dauert die Säuberung. Der A* selber benötigt bei normalen Hindernisstrukturen nicht mehr als 5ms (getestet auf P3, 500MHz, GF2). Bei einer *ID_LIST[1000000]* (was bei 1000 Nodes pro Achse nötig wäre) braucht er aber schon mehr als 15ms, was zu ersten Einbrüchen in der FPS führt. In unserem Beispiel bedienen wir uns 200 Nodes pro Achse. Achtung: Dieses Beispiel geht von einem quadratischen Vertex-Gitter aus und benötigt somit nur einen Wert für Nodes pro Achse und den Abstand zwischen den Nodes. Wenn Sie dies ändern wollen benötigen sie einen X- und einen Y-Wert für die im Script definierten Variablen *nodes_per_axis* und *node_dist*. Mit *node_dist*, dem Abstand

zwischen den Nodes können wir die Genauigkeit des Pathfindings, aber auch die Größe des Bereiches festlegen. Je kleiner der Wert desto genauer erfolgt das Pathfinding, desto kleiner wird aber auch der Raum der abgefragt werden kann. In diesem Beispiel wählen wir einen Wert von 50, dadurch ist der Bereich der für das Pathfinding abgefragt wird 10000 Quants groß (50x200), sowohl auf X- als auch auf Y-Achse. Beginnen wir erstmal mit der Initialisierung:

```
var nodes_per_axis=200;
var node_dist=50;

function init_astar(&start)
{
      cleanup();
      path_area.X=start[0]-(nodes_per_axis*node_dist)/2;
      path_area.Y=start[1]+(nodes_per_axis*node_dist)/2;
      path_area.Z=start[2];
      path_area[row_no]=nodes_per_axis;
      path_area[col_no]=nodes_per_axis;
      dist_mod.X=node_dist;
      dist_mod.Y=node_dist;
}
```

Die erste Node in unserem Vertex/Node-Gitter wird diesmal nicht an den Ausmaßen des Terrains festgemacht, sondern an der Startposition des A*. Die Koordinaten für die erste Node der *path_area* werden genau um die Hälfte des möglichen Bereiches (in unserem Fall 5000 Quants) entfernt von der Startposition gesetzt. Dadurch befindet sich die Startposition immer in der Mitte des Pathfinding-Bereiches und kann in jede Richtung die selbe Entfernung zurücklegen. Zeilen- und Spalten-Anzahl werden natürlich auch in den *path_area*-Array eingetragen. Durch diese neue *path_area* hat sich so gut wie nix am eigentlichen Script geändert. Was noch fehlt, ist der schon angesprochene Ersatz für *vec_for_vertex*:

```
function vec_for_id(id)
{
      temp=id/nodes_per_axis;
      if(frc(temp)==0) { temp-=1; }
      dist.Y=int(temp);
      dist.X=abs((id-1)-dist.Y*nodes_per_axis);

      current_vec.X=path_area.X+dist.X*node_dist;
      current_vec.Y=path_area.Y-dist.Y*node_dist;
      current_vec.Z=path_area.Z;
}
```

Wie wir wissen kann man die Zeilenposition einer ID ermitteln, indem wir sie durch die Anzahl der Nodes pro Achse dividieren. Wir wissen auch, dass wenn hierbei ein gerader Wert herauskommt, sich die Node in der letzten Spalte der *path_area* befindet. Die Node-ID 200, die sich in der letzten Spalte der ersten Zeile befindet (bei 200 Nodes pro Zeile/Achse) würde also den Wert 1 zurückliefern. Für unsere Berechnung muss die Zählung der Zeilen jedoch mit 0 beginnen (siehe Abbildung 11 auf S.21). Wir müssen bei geraden Werten also noch mal 1 abziehen, damit die IDs in der letzten Spalte auch in die richtige Zeile eingeordnet werden. Ungerade Werte liefern bei dieser Berechnung von vornherein die richtige Zeilenposition (Bsp.: 4/200=0.02=0). Nun benötigen wir noch die Spaltenposition, die wir herausbekommen in dem wir von der aktuellen ID aus so oft die Anzahl der Nodes pro Achse abziehen, bis sie sich in der ersten (der nullten) Zeile befindet. In dieser entspricht die Node-ID der Spaltenposition (siehe Abbildung 11). Der Betrag der angibt, wie oft wir die Anzahl der Nodes pro Achse von der aktuellen ID abziehen müssen, entspricht natürlich der Zeilenposition die wir in *dist.Y* schon ermittelt haben. Im Grunde genommen benutzen wir die selbe Rechenmethode wie auf S.21, nur das wir eben nicht die Node-ID in der nächst höheren

Zeile suchen, sondern in der ersten Zeile. Wozu jetzt noch die *-1* beim ID-Wert bei er Berechnung von *dist.X*? Wie wir schon wissen beginnt unser ID-Bereich mit der Node-ID 1. Diese ist logischerweise 0 Quants von dem Ursprung des Bereiches, den *path_area.X/Y/Z*-Werten entfernt. Sie entspricht also genau diesen Koordinaten. Bei der Berechnung vom Vektor der Node-ID 1 in *current_vec* müssen *dist.X* und *dist.Y* also 0 entsprechen, damit *current_vec path_area* entspricht. Die Zeilenposition (*dist.Y*) ist bei der Node-ID 1 ohnehin 0, muss also nur noch die Spaltenposition auf 0 gesetzt werden, was wir erreichen indem wir von der aktuellen Node-ID 1 subtrahieren. Bei der Berechnung des Vektors tun wir also so, als ob wir mit unserer Node-Zählung bei 0 beginnen. Zum besseren Verständnis hier noch mal ein Beispiel:

In unserem 200x200 ID's großen Pathfinding-Bereich möchten wir den Vektor der Node-ID 800 herausfinden. 800/200=4. Da sich Node-ID 800 jedoch in Zeile 3 befindet, müssen wir noch mal 1 subtrahieren. *dist.Y* entspricht also dem Wert 3. Um die Spaltenposition herauszubekommen müssen wir jetzt 3 Zeilen nach oben in unserem Node-Gitter gehen, um in Zeile 0 zu gelangen, wo die Node-ID der Spaltenposition entspricht. Wir müssen also 3x200=600 Nodes vom aktuellen Wert subtrahieren. Dieser muss aber aus oben genannten Gründen selber noch mal um 1 erniedrigt werden. 799-600=199. Zu *path_area.X* werden also 199x50 Quants addiert und von *path_area.Y* 3x50 Quants subtrahiert, um in *current_vec* die Koordinaten der Node-ID 800 zu erlangen.

Eigentlich könnte es das schon gewesen sein, bis auf eine Kleinigkeit: Die Hinderniserkennung! Auf dem Terrain hatten die Vertices unterschiedliche Höhen und somit konnten wir bei einer zu großen Höhendistanz davon ausgehen, dass hier ein Hindernis vorliegt. Unsere virtuellen Vertices liegen alle auf einer Ebene, und normale Hindernisstrukturen wie Blocks oder Map-Entities würden vollkommen ignoriert werden, was ja nun absolut nicht Sinn der Sache ist. Um jedoch die Möglichkeit eines Pathfindings auf verschiedenen Höhenebenen offen zu lassen behalten wir die Berechnung der Höhendistanz bei, erweitern jedoch um die einfachste Art der Hinderniserkennung, den *trace*:

```
function astar(&start,&goal)
{
  …
  while (NODE_LIST[index]*4<NODE_LIST.length)
  {
    vec_for_id(NODE_LIST[1*4]);
    if(vec_dist(current_vec,goal)<=node_dist*2)
    …
    while(i<8)          // i< als 8 Nodes
    {
      if(node_child[i]>0)
      {
        j=NODE_LIST[index];         // j= Position für neue Node
        vec_for_id(current_node[0]);
        vec_set(vecdist1,current_vec);
        vec_set(vecdist2,NULLVECTOR);
        vec_for_id(node_child[i]);
        node_costs=current_node[1]+cost_value+vec_dist(current_vec,goal);
            // Kosten=Kosten der Parent-Node+1+Abstand zum Ziel
        if(j<=0 || j*4>=NODE_LIST.length){break;}
        if(ID_LIST[node_child[i]]!=0)
        {         // Existiert diese Node bereits?
          j=ID_LIST[node_child[i]];
          if(NODE_LIST[j*4+2]==100000){i+=1; continue;} // Sackgassen-Node?
          if(NODE_LIST[j*4+1]==current_node[1]-cost_value){i+=1; continue;}
              // Hinter der aktuellen Node liegende Nodes (einen Wegkosten-
              // Wert weniger) müssen nicht mehr untersucht werden.
          if(node_costs>NODE_LIST[j*4+2]) {i+=1; continue;}
```

```
            // Aktuelle Berechnung höher als vorherige
        }
        ME=sel_entity;
        trace_mode=IGNORE_ME+IGNORE_PASSABLE+IGNORE_PASSENTS+IGNORE_MODELS
                   +IGNORE_SPRITES+USE_BOX;
        result=trace(vecdist1,current_vec);
        vecdist2.Z=current_vec.Z;
        vecdist1.X=0;
        vecdist1.Y=0;
        if(result==0 && vec_dist(vecdist1,vecdist2)<node_dist)
        {
          NODE_LIST[j*4+0]=node_child[i];
          ...
        }         // if(result==0 && vec_dist(vecdist1,vecdist2)<50)
      }           // Ende if (node_child[i]>=0)
      i+=1;
    }             // Ende while(i<8)
  }               // Ende while(index<array)
  return (0);
}                 // Ende astar();
```

Bevor die Child-Nodes der derzeit günstigsten Node also in die *NODE_LIST* aufgenommen werden, wird jede Child-Node von der aktuellen Node aus angetraced. Dadurch kann festgestellt werden, ob die Child-Node nicht in einem Hindernis stecken könnte. Auch wenn die *trace*-Funktion als sehr rechenintensiv gilt, macht sie unseren Algorithmus nicht wirklich langsamer: Wie schon erwähnt benötigt der Algorithmus bei normaler Anwendung weniger als 5ms. Natürlich gibt es im Beispiellevel auch Situationen in der der Algorithmus 15ms und mehr benötigt. Doch wenn man *cost_value* und *node_dist* richtig einstellt sollte man immer auf ein zufrieden stellendes Ergebnis kommen.

Damit wären wir am Ende des A* angekommen. Trotz der vielen Dinge, die es zu beachten gab haben Sie nun jedoch kein endgültiges Ergebnis vor sich liegen. Wenn sie eines bei diesem Tutorial gelernt haben sollten, dann dass es beim Pathfinding am wichtigsten ist, den Algorithmus auf ihre Bedürfnisse so weit wie möglich zu optimieren. Hierfür können sie die Werte *nodes_per_axis*, *node_dist* und *cost_value* variieren, oder aber auch eine vollkommen neue Heuristik schaffen. Wahrscheinlich haben Sie auch den Wunsch, den A* zu erweitern, so dass man ihn ohne Terrain auch auf unterschiedlichen Höhenebenen anwenden kann. Mit der vorliegenden Version des A* und ein wenig Arbeit ist dies alles möglich. Um Sie beim Optimieren und Erweitern nicht ganz alleine dastehen zu lassen, habe ich diesem Tutorial noch einen Anhang hinzugefügt, der Ihnen vielleicht dabei helfen wird, den richtigen Weg für die Umsetzung solcher Erweiterungen zu finden. Es handelt sich hierbei aber wirklich nur um Denkanstöße. Den Code für die vorgestellten Möglichkeiten müssen sie schon selber schreiben...

Anhang: Optionen, optimieren, erweitern...

I. OPEN und CLOSED?

Diejenigen, die sich in den Bereich Pathfinding schon eingearbeitet haben vermissen evtl. die für den A* so bekannten OPEN und CLOSED-Arrays. Für diejenigen, die nicht wissen worum es sich dabei handelt hier noch einmal eine kurze Erklärung: Die klassische Strategie beim Ermitteln der Parent-Node und der Überprüfung nach vorhandenen Nodes ist in der Tat, den gesamten Array zu durchsuchen. In C++ ist ein solches Durchsuchen auch viel schneller möglich als in C-Script, weiterhin gibt es nicht die Abfrage nach einer Endlosschleife die in C-Script zu einem Abbruch des Algorithmus bei einer zu langen Suche führen würde.

Um dieses Durchsuchen des „*NODE_LIST*“-Arrays zu Beschleunigen, hat man ihn einfach in 2 Teile aufgeteilt: Die Nodes, von denen schon die Child-Nodes bekannt sind (die also schon einmal als vorläufig günstigste Node erkannt wurden), also alle Parent-Nodes, werden in den CLOSED-Array gespeichert. Alle anderen Nodes werden im OPEN-Array gespeichert. Da viele Nodes jedoch erst im Verlauf des Algorithmus zu Parent-Nodes werden, befinden sich viele von Ihnen auch im OPEN-Array. Aus diesem werden sie bei einer Übertragung in den CLOSED-Array gelöscht (dafür habe ich im Heapsort auch die delete-Funktion angegeben). Zum bessere Verständnis hier noch mal eine Grafik:

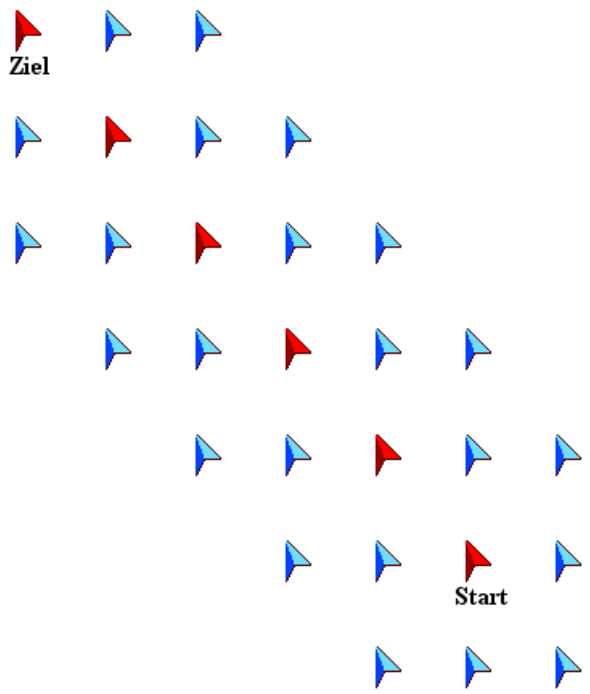

Abbildung 14: Beispielverlauf eines A*. Blau=Nodes in OPEN, Rot=Nodes im CLOSED

Bei einem Durchsuchen des OPEN-Arrays würden also die schon bekannten Parent-Nodes nicht stören, bei einem Durchsuchen des CLOSED-Arrays die nicht weiter erforschten Child-Nodes. Der Vorteil liegt hier vor allem beim kreieren des Pathes, wofür nur der CLOSED-Heap benutzt werden muss. Bei der Überprüfung nach schon vorhandenen Nodes müssten allerdings wieder beide Arrays abgefragt werden. Letztendlich ist die von mir vorgestellte Lösung mit nur einem OPEN-Array - der *NODE_LIST* - um einiges schneller. Der Vollständigkeit halber liegt diesem Tutorial auch eine A* Umsetzung mit OPEN- und CLOSED-Array bei. Damit diese Variante auch in C-Script funktioniert, benutze ich wieder die Speicherung der Node-Positionen in einer *ID_LIST*, bzw. in zweien, denn wir brauchen

jetzt jeweils eine *ID_LIST* für OPEN und für CLOSED. Würde ich OPEN und CLOSED einfach so durchsuchen lassen, wie beim klassischen A*, würde die A5 wahrscheinlich schon sehr bald einen Endless Loop melden. Der in der Datei *astar_2a.wdl* zu findende Sourcecode ist also nur als stark vereinfachtes Beispiel anzusehen, wie der A* traditionell in höheren Programmiersprachen wie C++ und Delphi Anwendung aussehen könnte.

II. Postprocessing

Wenn wir unseren A* nun in Aktion treten lassen, können wir mit dem Ergebnis das er liefert eigentlich zufrieden sein: Je nachdem wie gut wir ihn auf unsere Bedürfnisse angepasst haben findet er einen zufrieden stellenden Weg. Na ja, wenn wir ehrlich sind ist der Path den der A* gefunden hat wirklich mit wenigen Wegkosten verbunden (unabhängig davon wie diese Wegkosten in ihrem Algorithmus zustande kommen). Aber als unbedingt schön ist dieser Path noch nicht anzusehen. An einigen Stellen kann es vorkommen, dass der Path in Zickzack-Form verläuft. Dies hat damit zu tun dass wir eben nur 8 Richtungen festgelegt haben, in die sich unsere Entity bewegen kann. Die Anzahl der Richtungen zu erhöhen würde aber mit zu großen Performanceeinbußen verbunden sein. Statt dessen können wir den vom A* gefundenen Path noch einmal nachbearbeiten, bevor wir unsere Entity auf den Weg schicken. Dies könnte so aussehen, dass überprüft wird ob der A* nicht einige überflüssige Waypoints gesetzt hat. Eine Gerade innerhalb des Pathes könnte man z.B. auch mit nur 2 Punkten darstellen. Mehr Punkte sind eigentlich nicht nötig. Auch bei einem Zickzack-Verlauf ist es sehr wahrscheinlich, dass man diesen in eine Gerade umwandeln kann und somit nicht nur Waypoints spart, sondern auch einen schöner anzusehenden Path herausbekommt. Um ein solches Path-Smoothing zu erreichen, müsste man vom Startpunkt (oder bei einem Zickverlauf von dem ersten betroffen Waypoint) aus die nachfolgenden Waypoints antracen, um zu sehen ob man einige der Waypoints nicht auslassen kann, da einer der nächsten Waypoints von der aktuellen Position aus genauso gut (wenn nicht schneller) zu erreichen ist als wenn man dem eigentlichen Path folgt. Damit die Performance nicht so sehr unter dem Postprocessing leidet könnten Sie diesen Schritt auch erst einen Frame nach dem eigentlichen A* beginnen. Denn ob die Entity nun im selben Frame wie der eigentliche A* losgeht oder einen Frame später wird der Spieler wohl kaum merken.

III. Unterschiedliche Höhenebenen

Wie schon erwähnt wird das virtuelle Vertex-Gitter beim A* ohne Terrain auf einer Höhenebene angelegt: Die Nodes liegen alle auf dem Z-Wert der Startkoordinaten. Doch wenn wir schon eine 3D-Engine benutzen, sollen die Möglichkeiten die sich dadurch ergeben auch genutzt werden: Unsere Entity soll z.B. auch Treppen hochgehen können. Aber wie könnte man dies erreichen? Man könnte beim Starten des Algorithmus den Abstand vom Startpunkt zum Boden messen. Dieser Abstand muss nun immer eingehalten werden. Sobald sich der Boden erheben würde, würde also auch die Node ihre Höhe ändern. Um dies zu erreichen müsste aber für jede mögliche Node noch mal ein *trace* nach unten ausgeführt werden. Dadurch würde sich die Anzahl der *traces* verdoppeln, was der Performance sicher nicht gut tun würde. Damit der Algorithmus bei einem Ziel auf einer anderen Höhenebene eben dieses auch anstrebt, könnte man die Z-Distanz der aktuellen Node zur Goal-Node auf die Heuristik aufschlagen. Dadurch würde der Algorithmus versuchen, so schnell wie möglich auf die gleiche Höhenebene zu kommen wie der Zielpunkt. Damit er dabei keine Hindernisse „überspringt“, haben wir ja schon im vorhandenen Sourcecode die Begrenzung eingebaut, dass Nodes die höher als *node_dist* liegen ignoriert werden. Trotzdem taucht ein alt bekanntes Problem wieder auf: Der Algorithmus ist darauf fixiert, so schnell wie möglich auf die Höhenebene des Ziels zu gelangen. Wie wir aber mittlerweile wissen ist dies nicht unbedingt der beste Weg um am Ziel anzukommen. Die Wahrscheinlichkeit dass der Algorithmus an komplexen Hindernissen in der 3.Dimension (z.B. verschiedene Treppen die zu unterschiedlichen Ebenen führen) scheitert ist sehr hoch. Weiterhin ist diese Variante wie schon erwähnt mit Performanceeinbußen verbunden.

Viel versprechender könnte die Nutzung von Hilfsentities sein, die angeben an welchen Stellen ein Höhenwechsel möglich ist und auf welche Höhenebene dieser Weg führt. Diese müssten jedoch vom Level-Designer wieder manuell gesetzt werden. An dieser Stelle müssten die beiden unterschiedlichen Möglichkeiten zur Erstellung von Nodes also kombiniert werden, wie schon am Anfang des Tutorials vorgeschlagen. Wenn festgestellt wird, dass sich das Ziel auf einer anderen Höhenebene befindet, könnte man per *ent_next()* die Entities suchen, die sich auf der selben Höhenebene befinden und von diesen aus einen Weg auf die Höhenebene des Startpunktes berechnen. Der erste Teil vom Algorithmus würde also genau entgegengesetzt zum üblichen verlaufen. Ist ein Weg auf die Höhenebene des Ziels gefunden, kann der eigentliche A* beginnen und einen Weg zu der ersten Hilfsentity suchen, die sie dann schließlich auf die Höhenebene des Zieles bringt. Auf dieser Ebene angelangt müsste dann noch mal ein A* ausgeführt werden, um zu Ziel zu gelangen (dieser Schritt könnte natürlich auch schon beim wählen der Hilfsentity auf der Höhenebene des Ziels erfolgt sein). Anstelle von Hilfsentities könnte man natürlich auch Pathes verwenden.

IV. Bei Verwendung mehrerer Terrains...

Wenn Sie das Terrain-Pathfinding verwenden, funktioniert dieses ja nur auf einem Terrain. Was aber, wenn die Entity eine Strecke zurücklegen muss, die über mehrere Terrains führt? Nun, dies ist kein großes Problem: In der Funktion *define_children* haben Sie ja schon eine Möglichkeit kennen gelernt, wie man herausfindet, ob sich eine Node am Ende eines Terrains befindet. Bei dieser Abfrage müssen sie einfach überprüfen, ob der Zielpunkt überhaupt auf dem aktuellen Terrain liegt. Ist dies nicht der Fall, müssen Sie einfach einen neuen *trace* von der aktuellen Node (evtl. um den *node_dist* Betrag verschoben) auf den Boden vornehmen. Befindet sich an dieser Stelle ein weiteres Terrain, können sie den Pointer *sel_terrain* auf das neue Terrain setzen. Der bisherige A* sollte beendet werden, die gefundenen Waypoints in einem anderen Array gesichert werden. Die Koordinaten der letzten Node (wie gesagt evtl. modifiziert durch *node_dist*) wird nun als neuer Startpunkt benutzt, der Zielpunkt bleibt gleich und ein neuer A* wird gestartet. Wenn der Zielpunkt erreicht ist werden die nun gefundenen Waypoints auch in den zweiten Waypoints-Array gespeichert werden. Dieser muss also groß genug sein, um die Waypoints mehrerer Terrains fassen zu können. Das endgültige Abschreiten des gefundenen Pathes wird dann über den neuen Waypoints-Array abgehandelt.

V. Weitere Quellen

Wie schon im Vorwort erwähnt ist ein Austausch untereinander sehr wichtig. Gerade im Bereich KI gibt es so viele verschiedene Umsetzungsmöglichkeiten, dass man ruhig mal gucken sollte was andere „Kollegen“ bisher für Ideen hatten. Und nicht nur die Umsetzung, auch die unterschiedlichen Anwendungsmöglichkeiten können inspirieren. Vor allem die Wahl der Heuristik ist sehr interessant und Sie sollten sich unbedingt mal anschauen, was man so alles in die Heuristik hineinpacken kann. Für weiteres stöbern kann ich folgende, teilweise recht bekannte Seiten empfehlen:

http://www.gameai.com – Gute Linkliste, nicht nur zum Pathfinding
http://www.gamasutra.com – Zu empfehlen vor allem die Artikel von
Bryan Stout (http://www.gamasutra.com/features/19990212/sm_01.htm),
Macro Pinter (http://www.gamasutra.com/features/20010314/pinter_01.htm),
und Patrick Smith (http://www.gamasutra.com/features/20020405/smith_01.htm)

Für die Gamasutra-Artikel benötigen Sie einen User-Account auf gamasutra.com