1 Was ist Pathfinding

Pathfinding ist die Kunst, den kürzesten Weg von einem Startpunkt zu einem oder mehreren Endpunkten zu finden. Pathfinding findet Anwendung in Netzwerkfluss-Analysen, Routenplanern, Spielen usw.[http://de.wikipedia.org/wiki/Pathfinding]

Es wurden verschiedenste Algorithmen entwickelt, die alle ihre spezifischen Vor- und Nachteile haben. Der „A*-Algorithmus“ benutzt Heuristik. Hierbei wird der Weg zum Ziel geschätzt, um die Rechenzeit zu verringern. Der A* Algorithmus findet immer den optimalen Weg, falls einer existiert. [http://de.wikipedia.org/wiki/A*-Algorithmus]. Der „Dijkstra-Algorithmus“ unterscheidet sich vom A* Algorithmus dadurch, das er keine Heuristik benutzt. Er expandiert in alle Richtungen gleichermaßen. Zum Einsatz kommt dieser Algorithmus bei mehreren Zielen in unterschiedlichen Richtungen.

2 Das Programm

Ich unterteile den Algorithmus in 2 Klassen, in „TKarte“ und „Tpathfinding“. In der Klasse TKarte sind alle Informationen zur Umgebung gespeichert. TPathfinding führt die Berechnungen zum kürzesten Weg durch und speichert diesen. TPathfinding greift bei seinen Berechnungen auf TKarte zu. Der Zugriff wird beim Construktor festgelegt, indem die Speicheradresse von TKarte als Parameter angegeben wird. Man könnte die Umgebungsinformationen auch in TPathfinding speichern. Da es aber möglich ist, das man mehrere Ziele und Startpunkte hat, würde das bedeuten, das die Karteninformationen mehrfach gespeichert werden müssen. Wenn sich auf der Karte irgendetwas verändert (z.B. eine Brücke stürzt ein), müsste man in jeder Pathfinding Klasse die Änderung vornehmen. Ich habe die Unterteilung vorgenommen um Arbeitsspeicher zu sparen.

Die Klasse TPathfinding hat 2 Funktionen zur Berechnung des optimalen Weges. Zum einen wird der kürzeste Weg gesucht und zum anderen der, mit dem man zeitlich gesehen am schnellsten zu dem Ziel gelangt. Der Zeitaufwand wird ab sofort als Kosten bezeichnet. Das möchte ich an einem Beispiel verdeutlichen: Zwischen Startpunkt und Ziel liegt ein Gebirge. Das Gebirge lässt sich passieren. Es ist jedoch ziemlich kostenintensiv. Etwas von Start entfernt führt ein Weg durch das Gebirge. Dieser Weg ist nicht der kürzeste, ist aber mit dem geringsten kostenaufwand zu begehen [siehe Bild 1 und Bild 2]. Laut der Quelle soll man den Mehraufwand dazu addieren. Ich habe mich dagegen entschieden. Ich multipliziere den Mehraufwand. Wenn man den Mehraufwand addiert

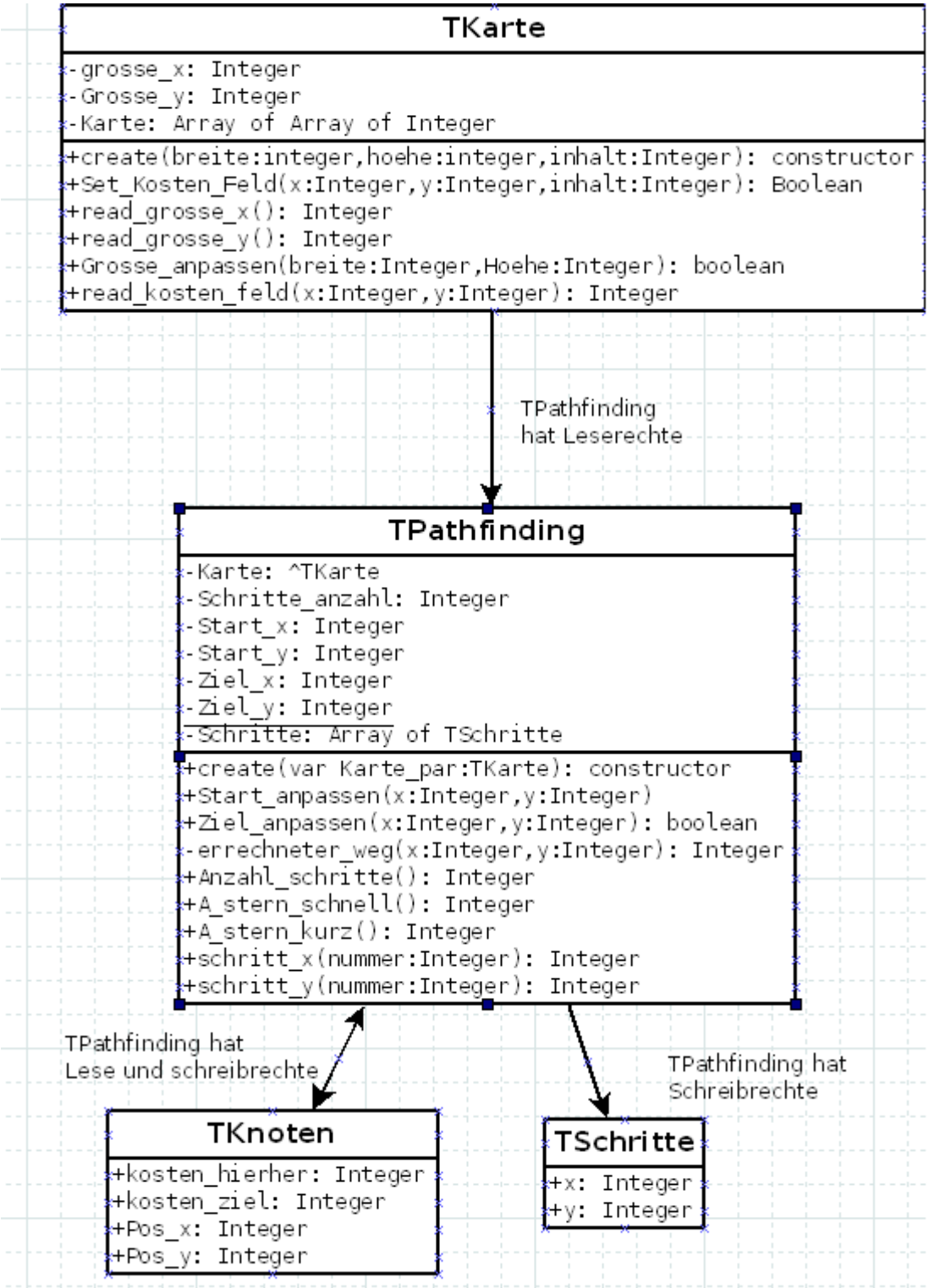
kommen fehlerhafte Karten zustande [Bild 4]. Richtig wäre [Bild 3].

Das Programm selber hat eine Karte die 40x30 Felder groß ist. Dieser Platz sollte ausreichen um den Algorithmus ordentlich zu testen. Bei der Suche nach dem kürzesten Weg benötigt man genau zwei Zustände, ein Hindernis und ein passierbares Objekt. Bei der Suche nach dem Weg mit den geringsten kosten habe ich mich für insgesamt fünf Objekte entschieden. Ein Hindernis, Gras, seichtes Gewässer, Sand und Gebirge. Die letzten vier sind alle passierbar. Um das Element Gras zu überqueren, benötigt man den einfachen Kostenaufwand. Für seichtes Wasser wird der Kostenaufwand zum Durchqueren verdoppelt, für den Sand verdreifacht und für das Gebirge vervierfacht.

Die Karte wird mit der Maus gestaltet. Hierfür klickt man das Objekt an, welches man auf die Karte setzten möchte und klickt auf die Karte. Delphi stellt für die Arbeit mit der Maus drei Funktionen bereit. Diese werden auch alle gebraucht. Die Erste wird aktiv, wenn eine Maustaste gedrückt wird. Sie liefert als Parameter unter anderem die Maustaste die gedrückt wird und Positionsangaben. In dieser Funktion überprüfe ich, ob die linke Maustaste gedrückt wird. Ist dies der Fall, wird die globale Variable „Maus_gedrueckt“ auf den Wert TRUE gesetzt. Wenn man nur diese Funktion benutzt, müsste man für jedes Feld, das man ändern will einmal klicken. Für die Benutzung einer dauerhaft gedrückten Maustaste benötigt man noch 2 andere Funktionen. Die erste der 2 Funktionen wird aktiviert wenn die Maustaste losgelassen wird. Hier wird überprüft, welche Taste losgelassen wurde. Ist es die linke, wird die Variable Maus gedrueckt auf FALSE gesetzt. Die Letzte Funktion, die man benötigt wird aktiv, wenn die Maus bewegt wird. Solange Maus gedrueck TRUE ist werden hier die Objekte eingezeichnet und an TKarte übergeben.

2.1 Das Klassendiagramm

UML unterstützt keine Zeiger Variablen. Jedoch benötige ich in TPathfinding eine Zeiger Variable um einen Verweis auf TKarte zu haben. Ich habe die Zeiger Variable so dargestellt, wie sie in Delphi dargestellt werden würde, mit einem ^ vor dem Name. UML unterstützt auch keine Referenzen. Auch diese habe ich wie in Delphi dargestellt.



2.2 Der A*Stern Algorithmus

Der A* Algorithmus ist ein Hilfsmittel zur Lösung von Problemen in der Graphentheorie. „In der Graphentheorie ist ein Graph eine Menge von Punkten (man nennt diese dann Knoten oder auch Ecken), die eventuell durch Linien (sog. Kanten bzw. Bögen) miteinander verbunden sind. Die Form der Punkte und Linien spielt in der Graphentheorie keine Rolle.”

[http://de.wikipedia.org/wiki/Graphentheorie]. Jede Linie hat eine Bestimmte Länge. Um vom Start zum Ziel zu gelangen muss man n Knoten Passieren. Ein paar dieser Knoten sind nicht passierbar und somit aus der Berechnung auszuschließen. In meinen Beispiel hat jeder Knoten genau acht Knoten die er erreichen kann, unabhängig davon ob sie passierbar sind oder nicht. Der Abstand zwischen den Knoten die er erreichen kann ist konstant. Jeder Knoten besitzt Informationen zu seiner Position, zu dem Kosten, die er vom Start verursacht, zu den geschätzten Kosten zum Ziel und der Summe aus geschätzten Kosten zum Ziel und Kosten bis zu diesem Knoten.

Mein Algorithmus bewegt sich in insgesamt acht Richtungen, die die acht Knoten repräsentieren: Nach oben, unten, links, rechts und Diagonal. Für jede Diagonale werden die „Kosten_hierher“ um 3 erhöht. Für horizontale und vertikale Bewegungen werden die Kosten_hierher um 2 erhöht. Dies hat folgenden Hintergrund: Die Diagonale ist die Hypotenuse eines rechtwinkligen Dreiecks. Die Katheten sind einmal die Bewegung in der Horizontalen und die Bewegung in der Vertikalen. Laut dem Satz des Pythagoras ($Hypotenuse^2 = Kathete^2 + Kathete^2$) ist die Wurzel aus 2^2+2^2 gleich 2,82843. Die Quelle arbeitet mit 10, für horizontale und vertikale Bewegungen, und 14 für diagonale Bewegungen. Die Unterschiede zwischen beiden Varianten sind minimal. Meine Werte sind etwas ungenauer. Jedoch haben meine Werte trotzdem nur 0,18 Kosten_hierher Ungenauigkeit. Die Vorgegebene Methode hat 0,14Kosten_hierher Ungenauigkeit. Ich habe mich für diese Methode entschieden, weil die Karte so knapp fünfmal so groß werden kann. Ich habe mich für ganzzahlige Werte entschieden, da man mit ihnen schneller rechnen kann. [vgl Quelle]. Das Beispielprogramm zeichnet Vierecke statt Achtecke. Bei ihm ist noch jede Ecke des Vierecks eine Seite des Achtecks. Dadurch kommt es, dass der Algorithmus Wege findet, die augenscheinlich verschlossen sind. [Bild „Durch Wand“].

Der A* Algorithmus ist in zwei Phasen unterteilt. In der ersten Phase sucht er einen Weg zum Ziel. In dieser Phase untersucht er, ob es einen Weg gibt. In der zweiten Phase wird der Weg wird der Beste weg bestimmt und gespeichert.

Bevor man mit Phase 1 startet sollte man verschiedene Szenarien herausfiltern, z.B. das der Startknoten nicht Passierbar ist, oder der Start und Zielknoten die Selbe Position haben.

Für den Algorithmus benutzte ich insgesamt 2 Listen.

1. Die Offene Liste, in der alle Knoten stehen, die noch nicht relevant waren für die suche nach dem kürzesten Weg
2. Die abgearbeitete Liste, in der alle Knoten stehen, die schon den geringsten Gesamtkosten

Wert hatten.

Die zwei Listen sind dynamisch sodass weniger Arbeitsspeicher benötigt wird. Eine zweite Möglichkeit wären ein Statisches Zweidimensionales Array des Typs TKnoten. Die TKnoten würde in diesem Fall noch eine zusätzliche Variable bekommen, die besagt, ob es in der offenen, der abgearbeiteten oder in noch keiner Liste steht. Der zweite Nachteil ist, das man für die suche nach dem Knoten, der in der offenen Liste steht, mit den geringsten Gesamtkosten, das gesamte Array durchsuchen muss. So wird die abgearbeitete Liste und Knoten die in noch keiner Liste stehen durchsucht

2.2.1 Phase 1

Zu Beginn der ersten Phase wird ein Knoten erstellt, der auf den Angaben zur Startposition liegt. Diesen Knoten werden „Kosten_hierher“ 0 zugeteilt, damit bei der Rückverfolgung klar wird, wann der Start erreicht wurde. Dieser Knoten ist das Erste Element der Offenen Liste.

Ab jetzt werden folgende Arbeitsanweisung wiederholt, solange bis keine Elemente mehr in der offenen Liste sind oder ein Knoten auf dem Ziel liegt. Im ersten Fall ist kein Weg vorhanden. Im zweiten Fall beginnt Phase 2.
Zuerst wird in der offenen Liste nach dem Knoten gesucht, welcher die geringsten Gesamtkosten hat. Das gefundene Element heißt „Aktueller Knoten“. Die 8 Knoten um den aktuellen Knoten herum werden überprüft, ob sie Passierbar sind. Ist dies der Fall wird überprüft, ob sie schon in der offenen oder abgearbeiteten Liste existieren.
Ist dieser Knoten schon in einer Liste vorhanden, kommt es zu einer Überprüfung ob die „Kosten hierher“ bei den aktuellen Knoten niedriger sind als bei dem überprüften Knoten aus der Liste. Ist dies der Fall ersetzt der Kosten hierher Wert des Aktuellen Knoten den aus der Liste. Diese Arbeitsanweisung ist notwendig, da der Algorithmus sonst nicht ordnungsgemäß arbeitet. Falls der Knoten in keiner Liste steht wird seine Position bestimmt. Der Knoten heißt ab jetzt „neuer Knoten“. Danach werden die neuen Kosten hierher bestimmt. Der Wert vom aktuellen Knoten wird mit den Schrittkosten addiert. Es gibt zwei Schrittkosten die dazu addiert werden können. Für Oben, Unten, Links, Rechts ist der Wert zwei, für die diagonalen Richtungen ist der Wert drei.

Nachdem die Kosten hierher bestimmt wurden, wird ermittelt, wie weit es noch zum Ziel ist. Dies wird in der Variable „Kosten Ziel“ gespeichert. Die Variable Kosten Ziel ist ein Schätzwert. Berechnen lässt er sich durch Folgende Funktion:
Kosten_Ziel = | Ziel_x – Start_x | * 2 + | Ziel_y – Start_y | * 2

Diese Methode wird Manhattan Methode genannt [vgl Quelle].

Das letzte Attribut des neuen Knotens ist „Kosten gesamt“. Kosten gesamt ist die Summe aus Kosten hierher und Kosten ziel.

Der neue Knoten wird in die offene Liste geschoben. Sind alle 8 Knoten um den Aktuellen Knoten herum überprüft, wird der Aktuelle Knoten in die abgearbeitete Liste geschoben. Das alles geschieht in der ersten Phase.

2.2.2 Phase 2

Diese Phase wird nur durchlaufen, falls ein Aktueller Knoten in der ersten Phase auf den Zielkoordinaten lag. In der zweiten Phase wird der Weg gespeichert. Hierfür wird ausgehend vom Ziel Knoten, den Aktuellen Knoten, der Weg rückläufig verfolgt. Die acht Knoten um den Aktuellen werden überprüft, welcher die geringsten kosten hierher hat. Seine Koordinaten werden gespeichert. Danach wird er selbst zum Aktuellen Knoten. Der Start ist erreicht, und somit der Weg vollständig gespeichert, falls die Aktuellen Kosten null betragen.

2.3 Struktogramm

Das Struktogramm liegt auf der beigelegten CD. Es ist zu umfangreich um es leserlich auszudrucken.

2. 4 Funktionsbeschreibung

2.4.1 Funktionen der TKarte Klasse

TKarte.create(breite, hoehe, inhalt: Integer)

- Erstellt eine Karte mit der Gegebenen Höhe und Breite
- Jedes Feld erhält den gegebenen Inhalt

TKarte.Grosse_anpassen(breite, hoehe: Integer)

- Ändert die Größe der Karte in die gegeben Höhe und Breite

- Liefert True zurück falls es geklappt hat

TKarte.Set_Kosten_Feld(x, y, feld: Integer)

- Setzt den Inhalt des Feldes auf den gegebenen Inhalt
- Liefert True zurück falls es geklappt hat

TKarte.read_kosten_feld(x, y: Integer)

- gibt den Aufwand zurück den man betreiben muss um das Feld zu passieren

TKarte.read_grosse_x

- Ließt die Höhe der Karte aus

TKarte.read_grosse_y

- Ließt die Breite der Karte aus

2.4.2 Funktionen der TPathfinding Klasse

Tpathfinding.schritt_x(nummer: Integer)

- Liefert die Position in x richtung des gegebenen schrittes zurück
- Sie liefert -1 zurück falls dieser Schritt noch nicht vorhanden ist

Tpathfinding.schritt_y(nummer: Integer)

- Liefert die Position in y richtung des gegebenen schrittes zurück
- Sie liefert -1 zurück falls dieser Schritt noch nicht vorhanden ist

TPathfinding.read_Ziel_x

- Diese Funktion gibt den X wert des Zieles zurück

TPathfinding.read_Ziel_y

- Diese Funktion gibt den y wert des Zieles zurück

TPathfinding.read_Start_x

- Diese Funktion gibt den X wert des Zieles zurück

TPathfinding.read_Start_y

- Diese Funktion gibt den Y wert des Zieles zurück

TPathfinding.AStern_kurz

- Diese Funktion beschreibt den kürzesten Weg vom Start zum Ziel
- sie hat folgende Rückgabe werte:
    0: alles hat funktioniert
    1: kein Weg gefunden
    2: ziel nicht definiert
    3: ziel liegt außerhalb des Intervalls
    4: Start nicht definiert
    5: Start liegt außerhalb des Intervalls
    6: Start = Zeile
    7: Start ist Hindernis
    8: Ziel ist Hindernis

TPathfinding.AStern_schnell

- Diese Funktion beschreibt den zeitlich Schnellsten Weg vom Start zum Ziel
- sie hat folgende Rückgabe werte:
    0: alles hat funktioniert
    1: kein Weg gefunden
    2: ziel nicht definiert
    3: ziel liegt außerhalb des Intervalls
    4: Start nicht definiert
    5: Start liegt außerhalb des Intervalls
    6: Start = Zeile
    7: Start ist Hindernis
    8: Ziel ist Hindernis