

Informatik Seminar Spiele-KI

Ermittlung eines guten Wege- netzes für Navigationsalgorithmen

**Jens Remus
ii5604
Technische Informatik
SS 2007**

**Betreuung:
Prof. Dr. Sebastian Iwanowski**

<u>EINLEITUNG</u>	1
<u>MOTIVATION</u>	1
<u>BEGRIFFSDEFINITIONEN</u>	1
<u>Vertex</u>	1
<u>Polygon</u>	1
<u>konvexes Polygon</u>	1
<u>konkaves Polygon</u>	2
<u>Triangulation</u>	2
<u>REPRÄSENTATIONEN DER SPIELWELT FÜR NAVIGATIONSALGORITHMEN</u>	2
<u>Regular Grids (Reguläre Gitter)</u>	4
<u>Corner Graphs</u>	5
<u>Waypoint Graphs</u>	6
<u>Circle-Based Waypoint Graphs</u>	7
<u>Quadtrees</u>	8
<u>Space-Filling Volumes</u>	8
<u>Navigation Meshes</u>	9
<u>AUTOMATISCHE GENERIERUNG VON GUTEN NAVIGATION MESHES</u>	11
<u>HERTEL-MEHLHORN ALGORITHMUS</u>	14
<u>2 → 1 MERGE-OPERATION</u>	14
<u>3 → 2 MERGE-OPERATION</u>	15
<u>N → 1 MERGE-OPERATION</u>	15
<u>SUBDIVIDE-OPERATION</u>	16
<u>OPTIMIERUNG DER AUTOMATISCHEN GENERIERUNG VON NAVIGATION MESHES</u>	17
<u>NORMAL-POOLING</u>	17
<u>VERTEX-POOLING</u>	17
<u>OPTIMIERUNG DER DATENSTRUKTUR ZUR SCHNELLEN BESTIMMUNG DER DIREKTEN NACHBARSCHAFT VON</u> <u>POLYGONEN</u>	17
<u>OPTIMIERUNG DER DATENSTRUKTUR ZUR SCHNELLEN BESTIMMUNG DER UMLIEGENDEN NACHBARSCHAFT</u>	18
<u>VEREINEN VON DEGENERIERTEN VERTICES</u>	18
<u>AUSBLICK</u>	19
<u>AI GAME PROGRAMMING WISDOM 4, FEBRUAR 2008</u>	19
<u>LITERATURVERZEICHNIS</u>	20
<u>STICHWORTVERZEICHNIS</u>	22

Einleitung

Motivation

Die Performanz der Spiele-KI Navigation ist hauptsächlich von zwei Faktoren abhängig:

- Wahl und Optimierung des Suchalgorithmus.
- Wahl und Optimierung der Suchraumrepräsentation bzw. der Spielweltrepräsentation sowie des Suchraums selbst.¹

Mit Suchalgorithmen wie Depth First, Breadth First, Best First, Hill Climbing, Dijkstra² und A-Star (A*) steht eine Reihe an wohlbekannten und bereits bestens optimierten Algorithmen zur Verfügung.

Die Optimierung von Suchalgorithmen wurde bereits in der vorhergehenden Ausarbeitung „Anpassungen des A*-Algorithmus an reale Anwendungen“ von Jan Schliep am Beispiel des A* Algorithmus erläutert.

Diese Ausarbeitung konzentriert sich daher auf den zweiten Faktor, die Wahl und Optimierung der Suchraumrepräsentation – der Repräsentation der Spielwelt für Navigationsalgorithmen am Beispiel der *Navigation Meshes*.

Begriffsdefinitionen

Vertex

Der Vertex (latein: Punkt) beschreibt einen Punkt, Vektor oder n-Tupel im Raum. In der Geometrie definiert er eine Ecke eines Polygons.³

Polygon

Das Polygon (griechisch: „Vieleck“⁴) beschreibt das Gebiet einer Fläche, welches durch einen geschlossenen Streckenzug begrenzt ist.⁵

konvexes Polygon

Ein Polygon ist konvex, wenn es eine konvexe Menge beschreibt, d.h. wenn jede Verbindungsstrecke zwischen zwei beliebigen Punkten innerhalb des Polygons ebenfalls vollständig innerhalb dieses liegt.

Jedes einfache Polygon ist konvex, wenn alle seine inneren Winkel kleiner oder gleich 180° sind oder jede Verbindungsstrecke zwischen zwei seiner Vertices vollständig innerhalb des Polygons liegt.

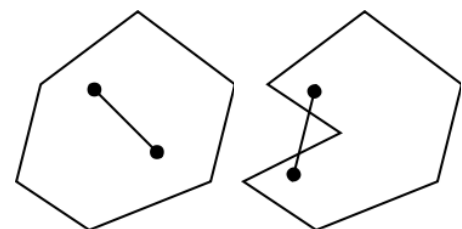


Bild 2: konvexes Polygon

Bild 1: konkaves Polygon

¹ Siehe [Tozour03], Seite 85.

² Benannt nach seinem Erfinder, Edsger Wybe Dijkstra, niederländischer Informatiker (1930-2002) [Wikipedia:Dijkstra].

³ Siehe [Wikipedia:Vertex].

⁴ Vgl. [Wikipedia:Polygon].

⁵ Siehe [O'Rourke98], Seite 1.

Dreiecke sind daher immer konvex, da die Summe ihrer inneren Winkel immer 180° beträgt und somit kein Winkel von mehr als 180° auftreten kann.

konkaves Polygon

Ein Polygon ist konkav, wenn es nicht konvex ist, d.h wenn es mindestens eine Verbindungsstrecke zwischen zwei beliebigen Punkten innerhalb des Polygons gibt, welche nicht vollständig innerhalb des Polygons liegt.

Jedes einfache Polygon ist konkav, wenn wenigstens einer seiner inneren Winkel größer als 180° ist oder eine Verbindungslinie zwischen zwei beliebigen seiner Vertices nicht vollständig innerhalb des Polygons liegt.

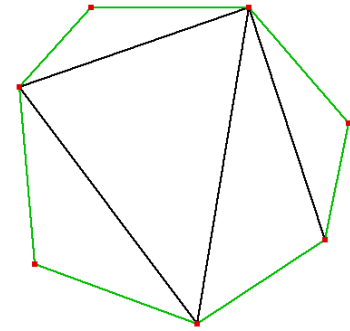


Bild 3:
Delaunay-Triangulation,
VoroGlide,
[http://www.geometrylab.de/
VoroGlide/](http://www.geometrylab.de/VoroGlide/)

Triangulation

Die Triangulation ist ein Verfahren, um aus einer Punktmenge ein Dreiecksnetz zu erstellen. Ein in der Computergraphik gebräuchliches Verfahren ist die Delaunay-Triangulation.⁶ Mit der Triangulation eines Polygons meint man daher die Zerlegung der Polygonfläche in Dreiecksflächen.

Repräsentationen der Spielwelt für Navigationsalgorithmen

Zwei wichtige Faktoren der unterschiedlichen Spielweltrepräsentationen sind die Performanz und der Speicherbedarf. Der Suchraum darf nur eine angemessene Menge an Speicher belegen und muss dennoch schnellstmögliche Suchen ermöglichen.⁷

Grundsätzlich sind alle Suchräume Graphen. Sie bestehen aus einer Anzahl von atomaren Navigationfeldern, den Ecken, und aus einer Anzahl von Verbindungen zwischen diesen, den Kanten.

Größere Graphen mit mehr Ecken und Kanten belegen meist mehr Speicher und resultieren normalerweise in langsameren Suchen.

Daraus folgt, dass man eine Minimierung des Suchraums / des Graphen anstreben muss, wobei eine zu starke Vereinfachung allerdings dazu führt, dass die Spielwelt nicht mehr ausreichend korrekt repräsentiert wird.⁸

Zwei weitere wichtige zu berücksichtigende Faktoren bei der Auswahl der Suchraumrepräsentation sind die Bewegungsfähigkeiten der Agenten und die Generierung des Suchraums.

⁶ Siehe [Wikipedia:Delaunay-Triangulation] und [Wikipedia:PolygonTriangulation].

⁷ Siehe [Tozour03], Seite 85.

⁸ Siehe [Tozour03], Seite 85f und [Tozour02], Seite 171.

Oft wird angenommen, dass sich unterschiedliche KI-Agenten auf die gleiche Art und Weise durch die Spielwelt bewegen. Es müssen jedoch eine Reihe an Eigenschaften der einzelnen KI-Agenten berücksichtigt werden, welche sich auf ihre Wegwahl auswirken:

- Bewegungsfähigkeiten, wie z.B. die Fähigkeit Türen, Leitern, Aufzüge, Teleporter oder „Jump Pads“ zu benutzen, die Fähigkeit zu springen, sich zu ducken oder die Fähigkeit zu schwimmen oder zu fliegen.
- Eigenschaften wie die Ausmaße und das Gewicht.
- Spezialfähigkeiten, wie z.B. die Fähigkeit, sich sicher durch Gas, Säure, Lava, etc. zu bewegen.

Es ist zu berücksichtigen, ob sich der Suchraum automatisch aus der Spielweltgeometrie erzeugen lässt oder von den Spielweltdesignern für jede Spielwelt manuell erzeugt werden muss.

Manuelles Plazieren der Wegpunkte durch die Designer birgt das Risiko, dass durch schlechtes Plazieren Fehler in der Navigation der KI-Agenten auftreten. Um dies zu vermeiden, muss die korrekte Navigation bei Änderungen immer wieder getestet werden. Bei vielen und großen Spielwelten führt dies zu einem hohen zusätzlichen Zeitaufwand für die Generierung und den Test des Suchraums. Werden später Änderungen an der Spielwelt vorgenommen, so muss der Suchraum unter Umständen neu erzeugt oder zumindest aufwändig überarbeitet werden.

Eine automatische Lösung erlaubt es, das Problem durch den Programmierer einmal zu lösen, anstatt es für jede Spielwelt einzeln lösen zu müssen. Zudem kann dies die Unterstützung von community-generierten Inhalten (Spielwelten) deutlich verbessern.⁹

Wichtig ist es weiterhin, den Unterschied zwischen einem Suchraum für Navigationsalgorithmen und „scripted sequences“ zu verstehen. Es ist oftmals notwendig bestimmte Abläufe, wie z.B. das Patrouillieren von Einheiten auf vorgegebenen Wegen zu realisieren. Die dabei angewendete Technik von statischen Wegpunkten ist nicht unbedingt gut für dynamisches Navigieren geeignet. Daher sollte man beide Techniken unabhängig voneinander implementieren.¹⁰

Im folgenden werden anhand der nebenstehenden Beispiel-Spielwelt der *Pathfinding Demo*¹¹ unterschiedliche Varianten präsentiert, die Spielwelt für Navigationsalgorithmen abzubilden. Es wird dabei im einzelnen auf die jeweiligen Vor- und Nachteile eingegangen.

Die jeweiligen Beispiel-Wege haben jeweils denselben Start- und Zielpunkt und wurden mit dem *A* Algorithmus* ermittelt. Sie sind jedoch nicht mittels Verfahren wie dem

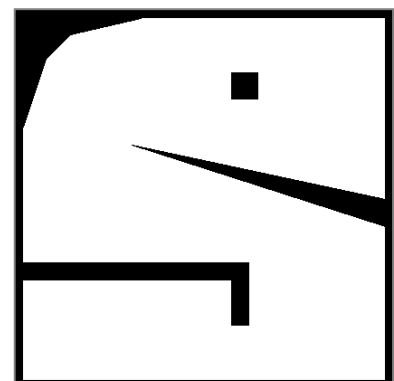


Bild 4: Beispiel-Spielwelt, *Pathfinding Demo*, Paul Tozour

⁹ Siehe [Booth04], Folie 49f.

¹⁰ Siehe [Tozour03], Seite 86f.

¹¹ Pathfinding Demo von Paul Tozour, <http://www.ai-blog.net/archives/000091.html>,
Kopie: PathfindingDemo.exe.

*String-Pulling*¹² optimiert und auch nicht mittels Verfahren wie den *Catmull-Rom Splines*¹³ geglättet.

Es ist jedoch zu beachten, dass die vorgestellten Varianten nur eine Auswahl der Möglichkeiten der Spielweltrepräsentationen für Navigationsalgorithmen darstellt.

Viele der Techniken sind sich vom Aufbau her sehr ähnlich oder basieren aufeinander. Es gibt jeweils unzählige Erweiterungs- oder Veränderungsmöglichkeiten. Weiterhin finden sich in der Literatur für ein und die selbe Technik unterschiedliche Bezeichnungen, da hierüber anscheinend kein Konsens unter den Entwicklern herrscht. Somit ist es schwierig die einzelnen Verfahren klar voneinander zu trennen und diese zu eindeutig zu benennen.

Die im folgenden benutzten Bezeichnungen und die meisten Informationen zu den vorgestellten Verfahren in den folgenden Unterkapiteln entstammen dem Artikel „Search Space Representations“¹⁴ von Paul Tozour¹⁵.

Einige der Verfahren, die *Regular Grids* (vornehmlich für 2D Umgebungen) und die *Waypoint Graphs* sowie die *Navigation Meshes* (insbesondere für 3D Umgebungen), haben sich besonders bewährt und sind daher weit verbreitet.

Regular Grids (Reguläre Gitter)

Quadratische, rechteckige, dreieckige oder hexagonale *Regular Grids* (im Deutschen *Reguläre Gitter*) sind die einfachste Form der Repräsentation der Spielwelt für Navigationsalgorithmen.

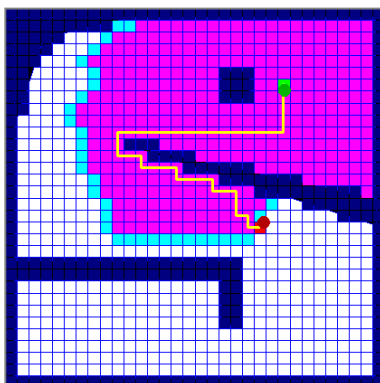


Bild 5: Regular Grid, Square Cells, 4-Way

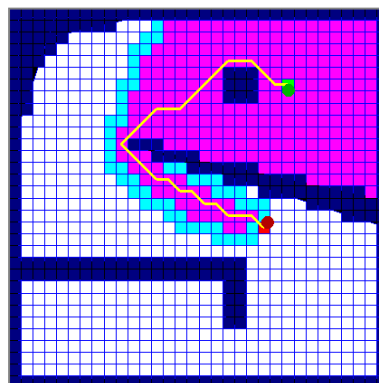


Bild 6: Regular Grid, Square Cells, 8-Way

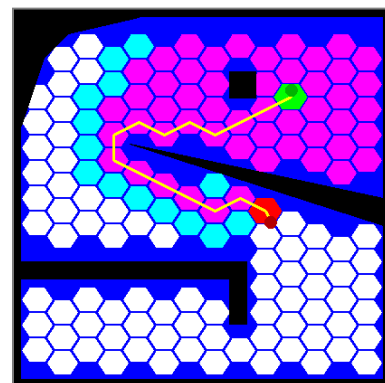


Bild 7: Regular Grid, Hexagonal Cells, 4-Way

Regular Grids eignen sich besonders gut für 2D Umgebungen wie zum Beispiel in Strategiespielen¹⁶ und drängen sich bei feldbasierten Spielen¹⁷ geradezu auf. In diesen Bereichen sind sie sehr weit verbreitet.

¹² Siehe [Tozour03], Seite 89.

¹³ Siehe [Tozour03], Seite 89.

¹⁴ Siehe [AIWisdom2].

¹⁵ Paul Tozour, renommierter Spieleentwickler mit Spezialisierung auf Spiele-KI, u.a. bei bekannten Spieleentwicklungsfirmen wie Ion Storm Austin und Retro Studios.

¹⁶ z.B. Battle Isle (Blue Byte), Sid Meier's Civilization (MicroProse), Warcraft (Blizzard).

¹⁷ z.B. Bomberman (Hudson Soft).

Die Größe der Felder wird proportional zum kleinsten KI-Agenten / Spielercharakter gewählt.

Vorteile der *Regular Grids* sind:

- Der Suchgraph ergibt sich direkt aus dem Gitter, sofern eine bidirektionale Verbindung zu angrenzenden Feldern besteht.
- Der Zugriff auf ein Navigationsfeld mittels einer Spielwelt-Koordinate ist in konstanter $O(1)$ Zeit möglich, da hierzu eine Umrechnungsformel verwendet werden kann. Keine der nachfolgenden Spielweltrepräsentationen bietet die Möglichkeit, zu einer gegebenen (X,Y) oder (X,Y,Z) Spielweltkoordinate den zugehörigen Navigationsknoten in konstanter Zeit zu ermitteln, da bei ihnen hierzu eine Graphen-Suche durchgeführt werden muss.
- Hindernisse, KI-Agenten und Spielercharaktere können einfach im Gitter markiert werden, was eine einfache Lösung für sowohl statisches als auch dynamisches Ausweichen ergibt.
- *Regular Grids* funktionieren sehr gut für feldbasierte 2D Spielwelten.

Nachteile der *Regular Grids* sind hingegen:

- Die meist recht hohe Anzahl an Feldern, welche zu einem hohen Speicherbedarf und einer langen Laufzeit der Suchalgorithmen führen kann.
- *Regular Grids* sind relativ ungeeignet für 3D Umgebungen, besonders bei übereinanderliegenden „begehbaren“ Flächen.
- Wege tendieren dazu, wie Züge auf einem Schachbrett auszusehen (ohne Glättungen durch den Einsatz von z.B. *Catmull-Rom Splines*).¹⁸

Corner Graphs

Corner Graphs bestehen aus Wegpunkten (Ecken), welche an den inneren „spitzen“ Ecken ($> 180^\circ$) der Spielwelt platziert werden und Verbindungen (Kanten) zwischen diesen.

Vorteil der *Corner Graphs* ist:

- Der *Corner Graph* kann automatisch aus der Spielweltgeometrie erzeugt werden, indem die konvexen Ecken ermittelt werden und dann getestet wird, ob sich ein KI-Agent auf einer geraden Linie zwischen diesen bewegen kann.

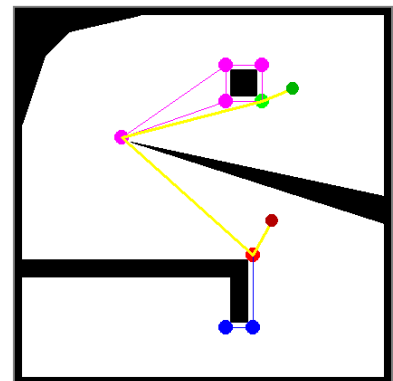


Bild 8: Corner Graph

Nachteile der *Corner Graphs* sind hingegen:

- *Corner Graphs* erzeugen oftmals suboptimale Wege und limitieren die KI-Agenten, sich an den Wänden der Spielwelt entlang zu bewegen.
- Die KI-Agenten scheinen „auf Schienen“ zu sein, während sie durch die Spielwelt navigieren.

¹⁸ Siehe [Tozour03], Seite 88ff und [Rabin00], Seite 273.

- Die Komplexität von $O(n^2)$ beim automatischen Generieren der Verbindungen (Kanten) des *Corner Graphs*, da es notwendig ist, alle Ecken gegen alle anderen Ecken zu testen. Ein worst-case hierfür wäre beispielsweise ein Raum mit vielen Säulen.¹⁹

Waypoint Graphs

Waypoint Graphs sind den *Corner Graphs* sehr ähnlich, mit der Ausnahme, dass die Wegpunkte üblicherweise in der Mitte der Räume und Gänge platziert werden und nicht entlang den Wänden und Ecken. Dies vermeidet das Problem der *Corner Graphs*, bei dem die KI-Agenten an den Wänden entlang laufen.

Waypoint Graphs und ihre unzähligen Varianten (*Corner Graphs*, *Circle-Based Waypoint Graphs*, *Nodemaps*²⁰, ...) sind neben den *Navigation Meshes* (auch *Space-Filling Volumes*) eine der verbreitetsten Techniken zur Suchraumrepräsentation für Navigationsalgorithmen.

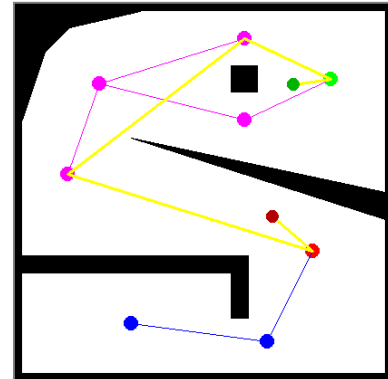


Bild 9: Waypoint Graph

Vorteile der *Waypoint Graphs* sind:

- *Waypoint Graphs* sind eine in 3D Spielen sehr verbreitete Technik, da sie in der Lage sind, 3D Umgebungen relativ einfach abzubilden.
- Wegpunkt-basierte Suchraumrepräsentationen funktionieren relativ gut bei Gebäuden und besonders gut bei engen Gängen und anderen Umgebungseinschränkungen, welche den KI-Agenten dazu zwingen sich in geraden Linien zu bewegen.

Nachteile der *Waypoint Graphs* sind hingegen:

- Wegpunkt-basierte Suchraumrepräsentationen funktionieren nur sehr eingeschränkt bei großen Räumen oder offenem Feld, da sie hier künstliche Einschränkungen einführen, die in der eigentlichen Spielwelt nicht existieren. Dadurch erzeugen *Waypoint Graphs* wie auch schon die *Corner Graphs* oftmals suboptimale Wege.
- Die bei den *Corner Graphs* bemängelte Komplexität von $O(n^2)$ beim automatischen Generieren der Verbindungen (Kanten).
- Die möglicherweise explodierende Anzahl an Ecken und Kanten im Suchgraph. Ein worst-case hierfür wäre ein großer Raum, welcher den Designer zwingt, eine hohe Anzahl an Wegpunkten zu platzieren, wodurch die Anzahl der Kanten exponentiell steigt.
- Die KI-Agenten scheinen „auf Schienen“ zu sein, während sie durch die Spielwelt navigieren.
- Der *Waypoint Graph* muss üblicherweise manuell vom Spielweltdesigner erzeugt werden, was zusätzlichen Aufwand bedeutet. Hierbei werden alle möglichen Wege der KI-Agenten durch den Designer festgelegt, wodurch die KI-Agenten in ihrer Bewegungsfreiheit eingeschränkt werden.²¹

¹⁹ Siehe [Tozour03], Seite 90ff.

²⁰ Siehe [Hancock02], Seite 194.

²¹ Siehe [Tozour03], Seite 92ff.

Circle-Based Waypoint Graphs

Circle-Based Waypoint Graphs sind eine Variante der *Waypoint Graphs*, welche jeden Wegpunkt um einen Radius-Parameter erweitern, welcher die ungefähre freie Umgebung um den Wegpunkt herum angibt.

Im Gegensatz zu *Waypoint Graphs* werden nur Verbindungen (Kanten) zwischen sich überlappenden Kreisen eingefügt.

Durch Hinzufügen eines Höhen-Parameters, welcher die ungefähre Höhe eines jeden Kreises angibt, lassen sich die *Circle-Based Waypoint Graphs* zu *Cylinder-Based Waypoint Graphs* erweitern.

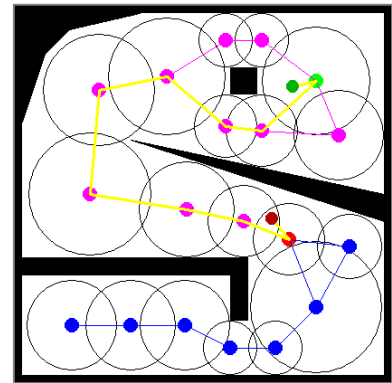


Bild 10: Circle-Based Waypoint Graph

Vorteile der *Circle-Based Waypoint Graphs* sind:

- Wesentlich geringere Komplexität des automatischen Generierens der Verbindungen (Kanten) gegenüber den *Waypoint Graphs*, da nicht mehr getestet werden muss, ob eine gerade Verbindung zwischen allen Wegpunkten (Ecken) vorhanden ist.
- Geringere Anzahl an Verbindungen (Kanten) in den meisten Umgebungen (besonders im offenen Feld) im Vergleich zu den *Waypoint Graphs*, wodurch die Größe des Suchgraphen reduziert wird.

Nachteile der *Circle-Based Waypoint Graphs* sind hingegen:

- Die KI-Agenten scheinen „auf Schienen“ zu sein, während sie durch die Spielwelt navigieren, sofern die Kreisfläche nicht zur Optimierung der Wege hinzugezogen wird.
- *Circle-Based Waypoint Graphs* erzeugen kein optimales Ergebnis in engen Räumen, wo auf Grund der geringen Überlappungen viele kleine Kreise eingefügt werden müssen, wodurch die Wegpunktanzahl gegenüber den *Waypoint Graphs* steigt. Hier kann es sinnvoll sein, eine hybride Lösung aus *Waypoint Graphs* für Gebäude und *Circle-Based Waypoint Graphs* für offenes Feld zu verwenden.²²

²² Siehe [Tozour03], Seite 94.

Quadrees

Quadrees sind eine Mischung der *Regular Grids* und einer Variante der *Space-Filling Volumes*. Sie unterteilen die Spielwelt wie die *Regular Grids* in eine gitterartige Struktur, wobei die quadratischen Gitterfelder jeweils eine variable Größe haben, welche sich durch rekursive Viertelung der Spielweltgröße ergibt.

Gegenüber den *Space-Filling Volumes* besteht aber immer noch die Möglichkeit mittels einer Zuordnungstabelle zu einer gegebenen Spielwelt-Koordinate die Kachel zu ermitteln. Hierzu berechnet man mit Hilfe der kleinsten Kachelgröße den Kachel-Index eines „virtuellen“ *Regulären Gitters* und ermittelt mit Hilfe der Zuordnungstabelle den Index der *Quadtree*-Kachel.

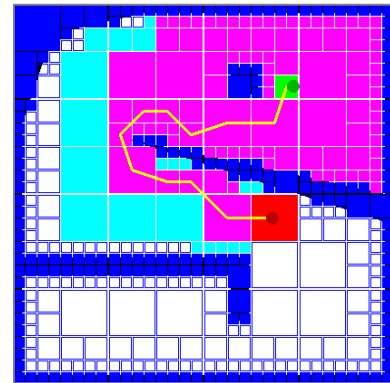


Bild 11: Quadtree

Vorteile der *Quadrees* sind:

- *Quadrees* können vollständig automatisch generiert werden, indem die Spielwelt mit einem Quadrat überdeckt wird, welches iterativ jeweils geviertelt wird, sofern seine Grundfläche von Hindernissen überlagert wird.

Nachteile der *Quadrees* sind hingegen:

- Die sehr hohe Anzahl an Feldern (Ecken) führt zu einer Explosion des Suchgraphen, wenn im Anschluss an die Generierung keine Felder miteinander verbunden werden.

Space-Filling Volumes

Space-Filling Volumes ähneln den *Circle-Based Waypoint Graphs*, wobei sie Rechtecke oder 3D Boxen anstelle von Kreisen oder Zylindern verwenden.

Allerdings werden *Space-Filling Volumes* teilweise auch mit *Navigation Meshes* gleichgesetzt. Eine weitere Bezeichnung für die *Space-Filling Volumes* ist auch „*Navigation Areas*“.²³

Vorteile der *Space-Filling Volumes* sind:

- *Space-Filling Volumes* können automatisch generiert werden, indem eine Reihe an Rechtecken oder Boxen gleichmäßig über die Spielwelt verteilt werden und anschließend inkrementell vergrößert werden, bis sie an Hindernisse oder ihresgleichen stoßen. Alternativ kann auch ein *Regular Grids* über die Spielwelt gelegt werden, dessen Zellen anschließend inkrementell miteinander verbunden werden.
- *Space-Filling Volumes* funktionieren in rechtwinkligen Umgebungen meist besser als *Circle-Based Waypoint Graphs*.

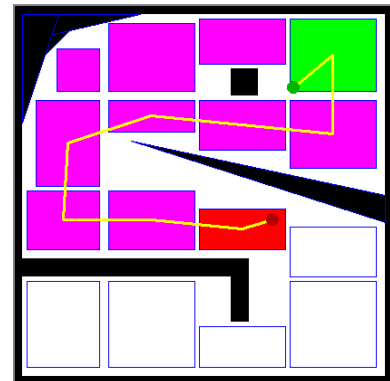


Bild 12: Space-Filling Volumes

²³ Siehe [Booth04], Folie 8ff.

Nachteil der *Space-Filling Volumes* ist hingegen:

- *Space-Filling Volumes* füllen nicht zwangsläufig die gesamte Spielwelt aus, vor allem in nicht rechteckigen Umgebungen, wodurch Teile der Spielwelt unzugänglich bleiben.²⁴

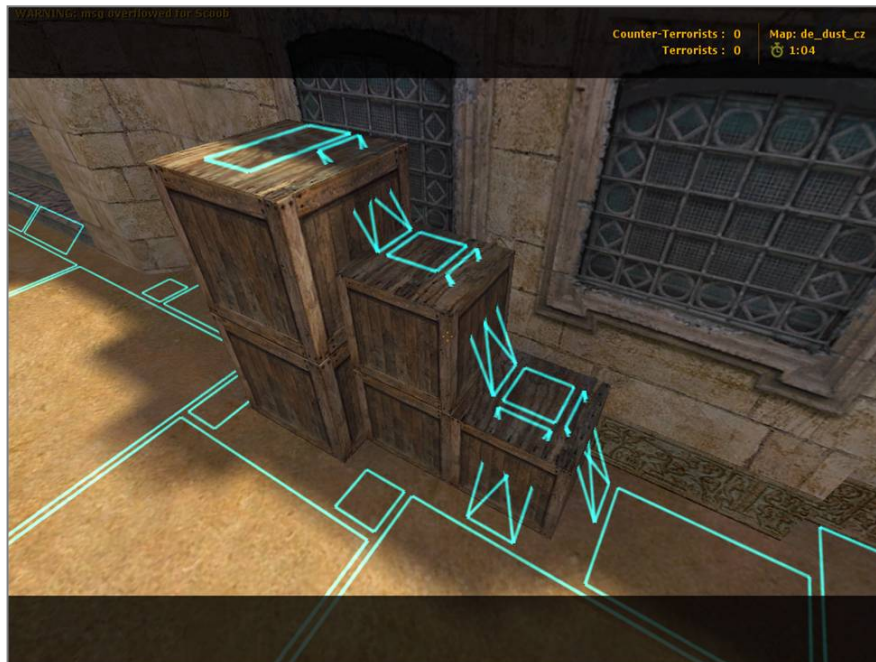


Bild 13: *Navigation Areas* in „Dust“, Counter-Strike: Source (Valve) aus [Booth04], Folie 13.

Navigation Meshes

„Ein *Navigation Mesh (NavMesh)* ist eine Menge von konvexen Polygonen, welche die ‚begehbare‘ Fläche einer 3D Umgebung beschreiben.“²⁵

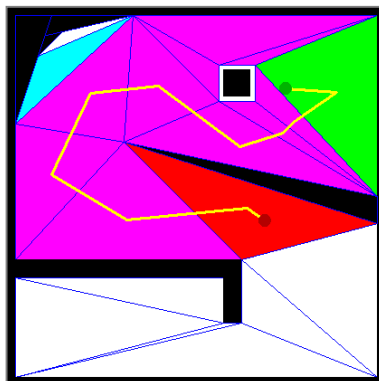


Bild 14: *Navigation Mesh (NavMesh)*, Triangle-Based

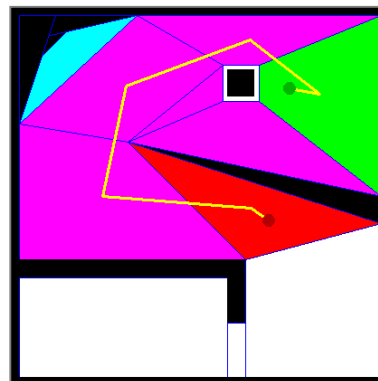


Bild 15: *Navigation Mesh (NavMesh)*, N-Sided-Poly-Based

Dabei ist die Konvexität der Polygone essentiell, da sie garantiert, dass sich ein KI-Agent innerhalb eines Polygons von einem Punkt zu einem anderen Punkt in einer geraden Linie bewegen kann. Daher ist jedes Polygon eine Ecke des Suchgraphen und jede gemeinsame Kante zwischen zwei Polygonen eine Kante des Graphen.

²⁴ Siehe [Tozour03], Seite 95.

²⁵ Vgl. [Tozour02], Seite 171.

Es gibt zwei grundlegende Typen von *Navigation Meshes*, die *Triangle-Based Navigation Meshes*, welche sich ausschließlich aus Dreiecken zusammensetzen und die *N-Sided-Poly-Based Navigation Meshes*, welche sich aus Polygonen mit beliebiger Eckenzahl zusammensetzen, solange diese konvex sind.

N-seitige Polygone können die Spielwelt normalerweise einfacher für Navigationsalgorithmen repräsentieren als dies mit Dreiecken möglich wäre, wodurch der Speicherbedarf sinkt und die Suchgeschwindigkeit steigt.

Navigation Meshes sind derzeit „State of the Art“, wenn es um die Repräsentation von 3D Umgebungen für Navigationsalgorithmen geht.

Vorteile der *Navigation Meshes* sind:

- *Navigation Meshes* erlauben es, den optimalen Weg für KI-Agenten mit unterschiedlichen Bewegungsfähigkeiten zu finden.
- Sie können sowohl Gebäude als auch große freie Flächen gleichgut repräsentieren.
- *Navigation Meshes* lassen sich automatisch aus der Spielweltgeometrie generieren, wenn dies auch nicht trivial ist. Es ist generell nicht möglich, die Spielweltgeometrie selbst als *NavMesh* zur Navigation zu verwenden.

Nachteile der *Navigation Meshes* sind hingegen:

- Der wohl wichtigste Nachteil der *Navigation Meshes* ist, dass möglicherweise eine große Anzahl an Polygonen gespeichert werden muss, vor allem in komplexen geometrischen Umgebungen.
- Die automatische Generierung des *Navigation Meshes* ist nicht trivial, besonders wenn auch die Anzahl der verwendeten Polygone minimiert werden soll.²⁶

²⁶ Siehe [Tozour03], Seite 95ff.

Automatische Generierung von guten Navigation Meshes

Im folgenden wird die letztgenannte Technologie zur Repräsentation der Spielwelt mit einem Fokus auf die automatische Generierung des Suchraums ausführlich erklärt, da die *Navigation Meshes* derzeit „State of the Art“ bezüglich der Repräsentation von 3D Umgebungen sind.

Das im folgenden aufgezeigte Verfahren zur automatischen Generierung von *Navigation Meshes* wird im Artikel „Building a Near-Optimal Navigation Mesh“²⁷ von Paul Touzour beschrieben. Zusätzlich sind einige Erweiterungen des Artikels „Improving on Near-Optimality: More Techniques for Building Navigation Meshes“²⁸ von Fredrik Farnstrom²⁹ eingeflossen.

Das Verfahren ist für fast jede Art von 3D Spielwelt gleich gut geeignet, jedoch ist generell zu beachten, dass ein statisch generierter Suchraum keine Behandlung von dynamischen Hindernissen beinhaltet.

Die erzeugten *Navigation Meshes* erheben den Anspruch, „gut“ oder auch „nahezu optimal“ zu sein. Hierzu ist zunächst zu klären, was ein optimales *Navigation Mesh* ausmacht. Nach Paul Tozour müssen folgende Kriterien³⁰ erfüllt werden:

- **Vollständigkeit.** Das *NavMesh* sollte die komplette sinnvoll von KI-Agenten begehbare Oberfläche der Spielwelt abdecken.
- **Einfachheit.** Das *NavMesh* sollte die Spielwelt mit möglichst wenigen Polygonen abdecken.
- **Konsistenz.** Der *NavMesh*-Generator sollte für dieselbe Eingabe immer dieselbe Ausgabe erzeugen.
- **Exzellente Laufzeit.** Die erzeugte Datenstruktur sollte in der Lage sein, Anfragen von den Suchalgorithmen schnell zu beantworten und dabei viel schneller sein als die direkte Verwendung der Spielweltgeometrie.
- **Vollständige Automatisierung.** Der *NavMesh*-Generator sollte das *NavMesh* vollständig automatisiert aus der Spielweltgeometrie erzeugen, so dass der Spielweltdesigner nicht manuell eingreifen muss.
- **Zumutbare Generationszeit.** Die Erzeugung des *NavMesh* sollte nicht mehr als ein paar Minuten benötigen, da die Spielweltdesigner das *NavMesh* öfters während der Entwicklung neu erzeugen müssen.
- **Robust im Bezug auf Degenerationen.** Der *NavMesh*-Generator sollte in der Lage sein, mit degenerierten Eingaben umzugehen. Dazu zählen z.B. Polygone mit weniger als drei Vertices, Polygone ohne Ausdehnung, Polygone mit einer Kante der Länge 0 oder Polygone, welche benachbart scheinen, deren gemeinsame Kanten-Vertices aber nicht identisch sind.
- **Robust in der Handhabung von zusätzlicher einschneidender Geometrie.** Der *NavMesh*-Generator sollte in der Lage sein, mit zusätzlicher Geometrie umzugehen, welche die existierende Spielweltgeometrie einschneidet. Hierzu zählen beispiels-

²⁷ Siehe [AIWisdom].

²⁸ Siehe [AIWisdom3].

²⁹ Fredrick Farnstrom, Spieleentwickler bei Rockstar San Diego.

³⁰ Siehe [Tozour02], Seite 172f.

weise Gegenstände oder Säulen, welche im Raum stehen und von der Raumgeometrie „subtrahiert“ werden müssen.³¹

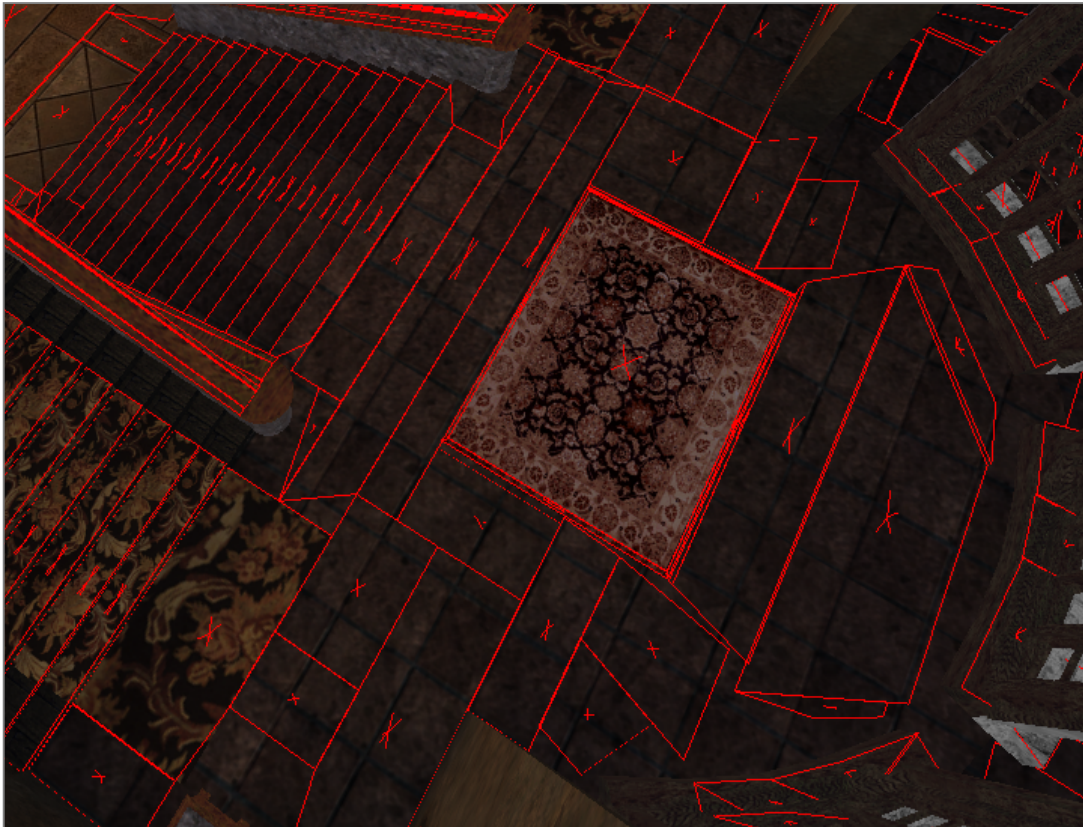


Bild 16: Ein Teil eines Navigation Meshes. Jedes konvexe Polygon ist ein Knoten des NavMesh. „Color Plate 4“ aus [Tozour02] (enthalten auf der dem Buch beiliegenden CD-ROM).

Um das *Navigation Mesh* für eine gegebene Spielwelt zu generieren, muss sie demzufolge in eine möglichst kleine Anzahl von konvexen Polygonen zerlegt werden, welche nur die von den KI-Agenten begehbbare Fläche bedecken.

Das Problem, eine beliebige Fläche mit einer minimalen Anzahl an konvexen Polygonen abzudecken, wird in der Geometrie als *Optimal Convex Partition Problem* bezeichnet. Der schnellste derzeit bekannte Algorithmus von Keil hat allerdings eine Laufzeit von $O(n^3 \log n)$.³² Es gibt jedoch einen relativ einfachen Algorithmus von Hertel³³ und Mehlhorn³⁴ mit linearer Laufzeit, der zwar nicht die minimale Anzahl an konvexen Polygonen ermittelt, dafür allerdings garantiert, maximal viermal so schlecht wie das Optimum zu sein.³⁵

³¹ Vgl. [Tozour02], Seite 172f.

³² Siehe [O'Rourke98], Seite 61.

³³ Stefan Hertel, außer Publikationen nichts im Internet auffindbar.

³⁴ Kurt Mehlhorn, deutscher Informatiker und Direktor des Max-Planck-Instituts für Informatik in Saarbrücken [Wikipedia:Mehlhorn].

³⁵ Siehe [O'Rourke98], Seite 60 und [Tozour02], Seite 175.

Die vollständig automatische Generierung des *Navigation Meshes* aus der Spielweltgeometrie läuft folgendermaßen ab:

1. Hinzufügen aller Spielweltgeometrie-Polygone, die zur Oberfläche der Spielwelt gehören und eine Normale besitzen, die annähernd senkrecht ist, sofern sich die KI-Agenten nur auf dem Boden und nicht auf den Wänden entlang bewegen können. Hierzu bestimmt man den Winkel zwischen der Normalen des Polygons und einer senkrechten Geraden und vergleicht diesen mit einer Schranke. Liegt der Winkel über dieser, so ist die Fläche zu steil für die Agenten. Die Normale wird mit dem Polygon gespeichert, da sie für den weiteren Generierungsprozess benötigt wird und auch während der Navigation nützlich sein kann, um die Steilheit des *NavMesh*-Knotens mit in die Wegwahl einzubeziehen. Meist handelt es sich bei den rohen Spielweltgeometrie-Polygonen um Dreiecke, die laut Definition bereits konvex sind. Nichtkonvexe (also konkave) Polygone müssen vor dem Hinzufügen trianguliert werden.
2. Anwendung des Hertel-Mehlhorn Algorithmus bzw. wiederholte Anwendung der $2 \rightarrow 1$ Merge-Operation (siehe Kapitel „[Hertel-Mehlhorn Algorithmus](#)“ und „[2 \$\rightarrow\$ 1 Merge-Operation](#)“).
3. Wiederholte Anwendung der $3 \rightarrow 2$ Merge-Operation (siehe Kapitel „[3 \$\rightarrow\$ 2 Merge-Operation](#)“).
4. Wiederholte Anwendung der $N \rightarrow 1$ Merge-Operation (siehe Kapitel „[N \$\rightarrow\$ 1 Merge-Operation](#)“).
5. Wiederholung der Schritte 2, 3 und 4, bis keine weitere Optimierung mehr möglich ist.
6. Entfernung „unnützer“ *NavMesh*-Knoten. Das sind Knoten, deren Oberfläche zu klein ist, um von den KI-Agenten sinnvoll genutzt zu werden. Hierzu iteriert man über alle Knoten des *NavMesh*, bestimmt ihre Oberfläche und entfernt sie, wenn sie einen bestimmten Schwellenwert unterschreiten. Dies reduziert drastisch den Speicherbedarf des *Navigation Meshes*, denn jedes noch so kleine Polygon belegt mehrer Bytes zur Speicherung seiner Vertices und seiner Normalen, sowie weitere Bytes in anderen Datenstrukturen, welche im Kapitel „[Optimierung der automatischen Generierung von Navigation Meshes](#)“ erläutert werden. Weiterhin reduziert dies auch den späteren Suchaufwand für die Navigationsalgorithmen durch die Reduzierung des Suchgraphen um die unnötigen Knoten.
7. Subtrahierung der statischen Hindernis-Geometrie. Hierzu werden die von statischen Hindernissen (teilweise) bedeckten *NavMesh*-Knoten ermittelt. Ist ein solches Polygon vollständig durch ein Hindernis bedeckt, so wird es aus dem *NavMesh* entfernt. Ist es nur teilweise bedeckt, so wird es durch rekursives Anwenden der Subdivide-Operation (siehe Kapitel „[Subdivide-Operation](#)“) in kleinere konvexe Polygone zerlegt.
8. Wiederholung von Schritt 7, bis keine *NavMesh*-Knoten mehr (teilweise) von Hindernissen bedeckt sind oder eine minimale Polygongröße erreicht wurde. Diese Schranke verhindert eine unendliche Zerteilung und kann mit der Schranke aus Schritt 6 identisch sein. Sie repräsentiert weiterhin eine Grenze der Annäherung des *NavMesh* an die statischen Hindernisse in der Spielwelt.

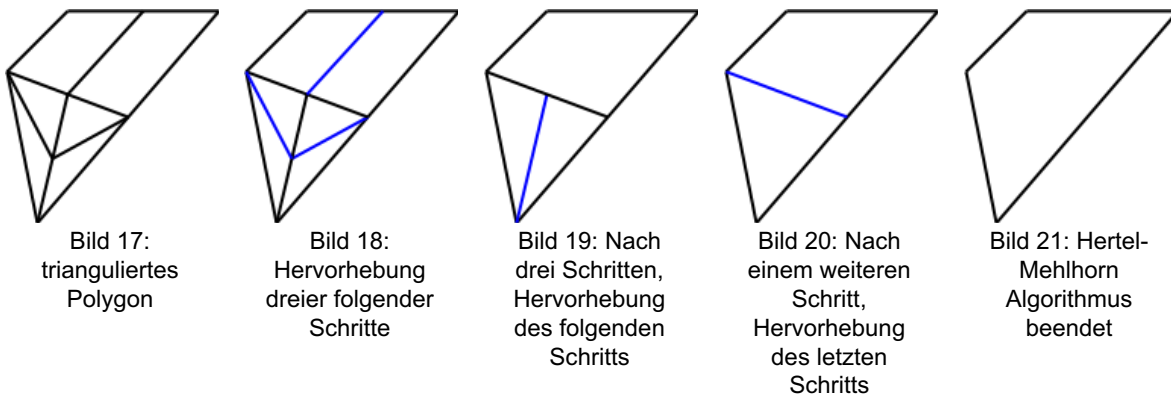
9. Wiederholung der Schritte 2, 3 und 4, bis keine weitere Optimierung mehr möglich ist.
10. Erneutes Anwenden von Schritt 6.
11. Speichern des *Navigation Meshes* zur späteren Verwendung durch das Spiel.

Hertel-Mehlhorn Algorithmus

Der Hertel-Mehlhorn Algorithmus funktioniert folgendermaßen:

1. Triangulation der Fläche (bzw. Zerlegung in ausschließlich konvexe Polygone).
2. Entfernung einer „unwichtigen“ Kante zwischen zwei konvexen Polygonen. Eine „unwichtige“ Kante ist eine Kante, durch deren Entfernung ein neues konvexes Polygon entsteht.
3. Wiederholung von Schritt 2 bis keine „unwichtige“ Kante mehr existiert.

Folgende Bilderserie³⁶ verdeutlicht die Schritte 2 und 3 des Hertel-Mehlhorn Algorithmus:

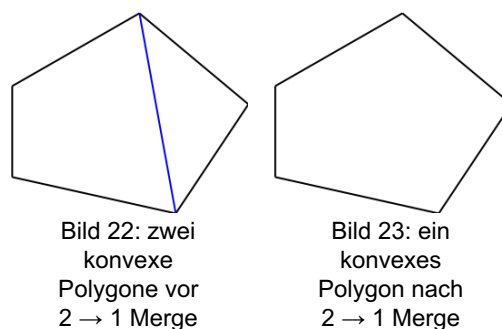


2 → 1 Merge-Operation

Die 2 → 1 Merge³⁷-Operation entspricht dem Schritt 2 des Hertel-Mehlhorn Algorithmus. Sie fasst zwei benachbarte konvexe Polygone zu einem einzigen konvexen Polygon zusammen, sofern beide durch eine „unwichtige“ Kante verbunden sind.

Ein Verfahren zur Bestimmung der direkten Nachbarschaft von Polygonen wird im Kapitel „[Optimierung der Datenstruktur zur schnellen Bestimmung der direkten Nachbarschaft von Polygonen](#)“ beschrieben, mit dessen Hilfe gemeinsame Kanten der Polygone im Navigation Mesh ermittelt werden können.

Einzelheiten zum exakten Verfahrensablauf finden sich im Artikel von Paul Tozour.



³⁶ Bilderserie aus [Tozour02], Seite 176.

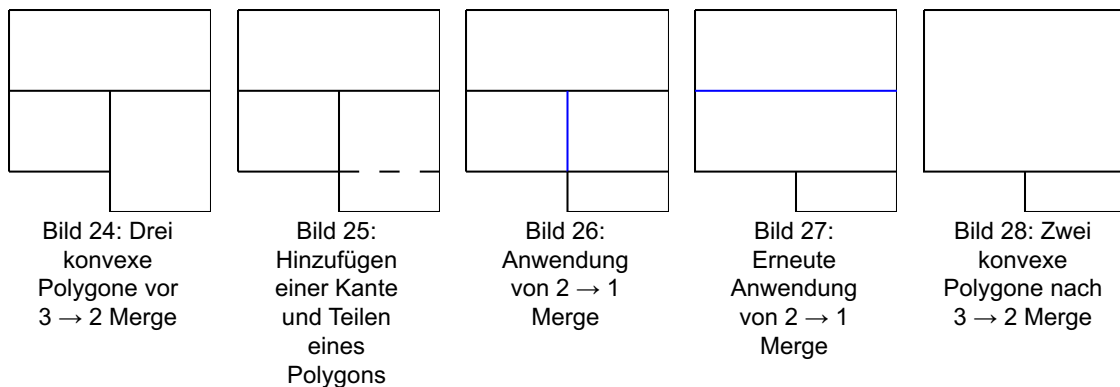
³⁷ Siehe [Tozour02], Seite 176.

3 → 2 Merge-Operation

Die 3 → 2 Merge³⁸-Operation fasst drei benachbarte konvexe Polygone zu zwei neuen konvexen Polygonen zusammen. Sie ist wesentlich komplizierter als die 2 → 1 Merge-Operation, da die Auswahl der für die Operation in Frage kommenden Polygone weitaus aufwendiger ist und diese nicht direkt zusammengefasst werden können, sondern vorher erst eines der drei Polygone in zwei neue zerteilt werden muss.

Die genauen Kriterien, die die drei Polygone erfüllen müssen und das exakte Verfahren sind im Artikel von Paul Tozour nachzulesen.

Folgende Bilderserie³⁹ zeigt den Ablauf der 3 → 2 Merge-Operation:



N → 1 Merge-Operation

Die N → 1 Merge⁴⁰-Operation ist eine Erweiterung der 2 → 1 Merge-Operation. Sie fasst N benachbarte konvexe Polygone mit einem gemeinsamen Vertex zu einem einzelnen konvexen Polygon zusammen, sofern durch Entfernung des gemeinsamen Vertex wieder ein konvexes Polygon entsteht.

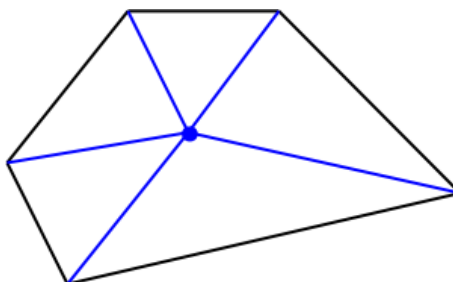


Bild 29: Benachbarte konvexe Polygone mit gemeinsamen Vertex vor N → 1 Merge

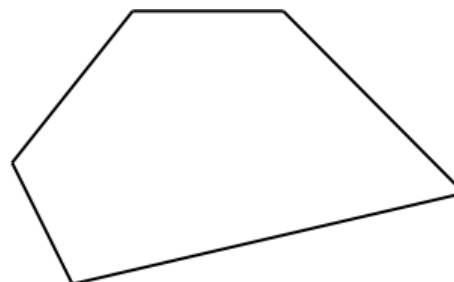


Bild 30: Resultierendes konvexes Polygon nach N → 1 Merge

³⁹ Bilderserie aus [Tozour02], Seite 177.

⁴⁰ Siehe [Farnstrom06], Seite 124.

³⁸ Siehe [Tozour02], Seite 177ff.

Subdivide-Operation

Die Subdivide⁴¹-Operation zerteilt ein beliebiges konvexes Polygon mit N Vertices in N neue konvexe Polygone. Die Operation wird zum Entfernen von Hindernis-Geometrie benötigt, um teilweise durch Hindernisse bedeckte Polygone nicht komplett aus dem Navigation Mesh entfernen zu müssen.

Zur Zerteilung eines konvexen Polygons werden sein Zentrum durch Mittelwertbildung der einzelnen Komponenten seiner Vertices sowie die einzelnen Mittelpunkte seiner Kanten bestimmt. Anschließend werden die neuen Polygone zum Navigation Mesh hinzugefügt, welche aus jeweils dem Zentrum, zwei Mittelpunkten und einem originalen Vertex des alten Polygons bestehen. Die Vertex-Liste eines jeden neuen Polygons wird folgendermaßen aus den ermittelten Daten aufgebaut: Zentrum, erster Kanten-Mittelpunkt, originaler Vertex, zweiter Mittelpunkt.

Folgende Bilderserie⁴² zeigt den Ablauf der Subdivide-Operation für jeweils ein Viereck und ein Dreieck:

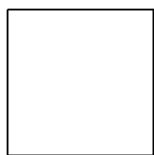


Bild 31:
Vor
Subdivide

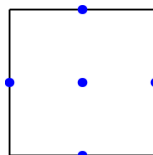


Bild 32:
Wahl der
Vertices

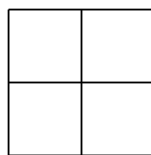


Bild 33:
Nach
Subdivide

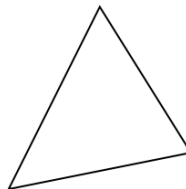


Bild 34: Vor
Subdivide

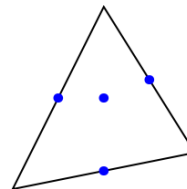


Bild 35: Wahl
der Vertices

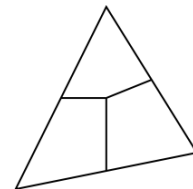


Bild 36:
Nach
Subdivide

⁴¹ Siehe [Tozour02], Seite 181.

⁴² Bilderserie aus [Tozour02], Seite 181.

Optimierung der automatischen Generierung von Navigation Meshes

Im folgenden werden einige Techniken zur Optimierung der automatischen Generierung von *Navigation Meshes* vorgestellt.

Normal-Pooling

Jeder Normalenvektor belegt mindestens 12 Bytes (drei 4-Byte Gleitkommazahlen) an Hauptspeicher. Bei 20'000 Knoten (Polygonen) im Navigation Mesh entspricht dies bereits ~240 KiB⁴³.

Es existieren zwei einfache Möglichkeiten, den Speicherbedarf zu optimieren:

1. Anstelle des Normalenvektors wird ein Zeiger auf einen Normalenvektor gespeichert, wobei der Zeiger den Wert NULL haben soll, wenn die Normale vertikal ist. Diese Lösung ist besonders gut in Gebäuden, wo 99% der Polygone eine senkrechte Normale besitzen. Dadurch wird der benötigte Hauptspeicher auf ~4 Byte pro Knoten gesenkt.
2. Alle in der Spielwelt vorkommenden unterschiedlichen Normalenvektoren werden in einem Normal-Pool gespeichert. Jeder Knoten speichert nur noch einen 2-Byte Index auf seinen Normalenvektor.⁴⁴

Vertex-Pooling

Wie auch schon beim Normal-Pooling können alle Vertices in einem einzelnen Vertex-Pool gespeichert werden. Jeder Knoten (Polygon) speichert nur noch einen 2- oder 4-Byte Index für jeden seiner Vertices. Der Speicherbedarf kann durch diese Maßnahme jedoch nur dann reduziert werden, wenn Vertices von vielen Polygonen gemeinsam genutzt werden.⁴⁵

Optimierung der Datenstruktur zur schnellen Bestimmung der direkten Nachbarschaft von Polygonen

Mehrere der Schritte zur automatischen Generierung des *Navigation Mesh* aus einer gegebenen Spielweltgeometrie benötigen die Information, welche Polygone direkt benachbart sind. Müsste man jedes einzelne Polygon gegen jedes andere testen, um die Nachbarschaft zu bestimmen, so würde die Generierungszeit ins Unermessliche steigen.

Eine Lösung für dieses Problem ist der Einsatz einer erweiterten Variante des Vertex-Pooling. Dabei wird zu jedem Vertex eine Liste von Zeigern zu allen Polygonen gespeichert, die den Vertex in ihrer Vertex-Liste beinhalten. Das angewandte Prinzip ähnelt dem Reference-Counting, wobei die Polygone verfolgt werden, die den Vertex verwenden.

Werden zu einem gegebenen Polygon seine angrenzenden Polygone gesucht, so können diese nun sehr schnell über die Polygon-Listen seiner Vertices ermittelt werden.

⁴³ Siehe [Wikipedia:KiBiByte].

⁴⁴ Siehe [Tozour02], Seite 182f.

⁴⁵ Siehe [Tozour02], Seite 183.

Weiterhin kann man auch nach beliebigen benachbarten Polygonen suchen, indem man den Vertex-Pool durchläuft.

Problematisch ist allerdings das ständige Entfernen und Hinzufügen von Vertices zum Vertex-Pool beim Erzeugen des *Navigation Mesh*. Daher dürfen die Vertices in diesem nicht in einer Liste mit linearen Sucheigenschaften gespeichert werden, sondern besser in einer Hash Map.⁴⁶

Optimierung der Datenstruktur zur schnellen Bestimmung der umliegenden Nachbarschaft

Weitere in dieser Ausarbeitung nicht behandelte Schritte zur automatischen Generierung des *Navigation Mesh* aus einer gegebenen Spielweltgeometrie benötigen die Information, welche Polygone sich in der umliegenden Nachbarschaft befinden, also nicht direkt an ein gegebenes Polygon grenzen. So müssen beispielsweise Verbindungen zwischen dicht beieinanderliegenden *NavMesh*-Knoten (z.B. Bordsteinkante, Treppe) eingefügt werden, deren Lücke für die KI-Agenten kein Hindernis darstellt.

Wie auch schon bei der „[Optimierung der Datenstruktur zur schnellen Bestimmung der direkten Nachbarschaft von Polygonen](#)“ ist es nicht sinnvoll, jedes Polygon gegen jedes andere zu testen.

Eine Lösung für dieses Problem ist der Einsatz einer Spatial Hash Map zur Speicherung der Polygone. Hierzu wird ein Reguläres Gitter über die Spielwelt gelegt. Für jedes Feld wird gespeichert, welche Polygone ganz oder teilweise darin enthalten sind. Hierzu werden die Feld-Indizes eines jeden Polygons als Schlüssel und das Polygon selbst als Wert in einer Hash Map gespeichert. Über die Feld-Indizes eines Polygons können nun seine umliegenden Nachbar-Polygone einfach aus der Hash Map ermittelt werden. Auch können alle benachbarten Polygone mit einem bestimmten Maximalabstand besser bestimmt werden, indem nur noch die Polygone von benachbarten Gitterfeldern gegeneinander getestet werden.⁴⁷

Vereinen von degenerierten Vertices

Die Spielweltgeometrien enthalten oft degenerierte Vertices. Das sind Paare von Vertices, die an derselben Position zu sein scheinen, jedoch einen minimalen Abstand zueinander besitzen, so dass sie z.B. von der Hash-Funktion des Vertex-Pools nicht als identisch erkannt werden.

Eine mögliche Lösung für dieses Problem besteht darin, alle Vertices in einer Liste nach ihren Koordinaten zu sortieren und anschließend iterativ nah beieinander liegende Vertices zu einem einzelnen Vertex zu vereinen.⁴⁸

⁴⁶ Siehe [Tozour02], Seite 183f.

⁴⁷ Siehe [Farnstrom06], Seite 124ff.

⁴⁸ Siehe [Tozour02], Seite 184.

Ausblick

Es wurde ein Verfahren zur automatischen Generierung eines Navigation Mesh aus der Spielweltgeometrie beleuchtet, welches hauptsächlich im Artikel „Building a Near-Optimal Navigation Mesh“⁴⁹ von Paul Tozour beschrieben wird.

Im Artikel „Improving on Near-Optimality: More Techniques for Building Navigation Meshes“⁵⁰ von Fredrik Farnstrom werden noch einige weitere hier nicht erwähnte Verbesserungen erläutert.

Generell ist es beispielsweise noch notwendig, die einzelnen Knoten und Verbindungen der Suchraumdarstellung mit weiteren Informationen (sogenannten „tags“ oder „flags“) zu markieren.

Für Knoten könnte zum Beispiel die Unterscheidung zwischen befestigtem Boden (Weg, Asphalt, ...), unbefestigtem Boden (Sand, Kies, ...), Flüssigkeit (Wasser, Lava, Säure, ...), Leiter, Teleporter, Aufzug, etc. getroffen werden. Weiterhin auch Eigenschaften wie Versteck, Konfrontationspunkt (mit Gegnern), Hinterhalt,

Für die Verbindungen kann entsprechend zum Beispiel zwischen der Fortbewegungsart (kriechen, springen, fallen, klettern, ...) oder Hindernis-Navigations-Informationen (Tür mit Schalter „X“, Aufzug mit Schalter „X“, ...) unterschieden werden.

Einige dieser Informationen lassen sich zum größten Teil auch automatisch generieren, wobei einige Anpassungen und Erweiterungen an den *NavMesh*-Generator notwendig sind. Andere können erst während des Spielgeschehens ermittelt werden, zum Beispiel durch zählen der in einem *NavMesh*-Knoten gefallenen Einheiten der KI und des Spielers.

AI Game Programming Wisdom 4, Februar 2008

Steve Rabin präsentiert auf seiner Webseite bereits die von ihm akzeptierten Artikel seines kommenden Buchs der *AI Game Programming Wisdom* Reihe, welches im Februar 2008 wieder bei Charles River Media erscheinen soll.⁵¹

Das Kapitel Pathfinding wird unter anderem folgende Artikel zum Thema *Navigation Meshes* enthalten:

- Fast Pathfinding Based on Triangulation Abstraction (Michael Buro)
- Real-Time Dynamic *NavMesh* Generation (Paul Marden, Forrest Smith)
- Automatic Generation of Path Nodes for a General Purpose 3D Environment (John Ratcliff)
- Intrinsic Detail in Navigation Mesh Generation (James Stewart, Colt McAnlis)
- *NavMesh* Generation: An Empirical Approach (David Hamm)

Annähernd 50% der Artikel des Kapitels „Pathfinding“ handeln von *Navigation Meshes*.

⁴⁹ Siehe [AIWisdom].

⁵⁰ Siehe [AIWisdom3].

⁵¹ Siehe [AIWisdom4].

Literaturverzeichnis

- [AIWisdom]** Rabin, Steve: „AI Game Programming Wisdom“, Charles River Media, 2002.
http://www.aiwisdom.com/byresource_aiwisdom.html
- [AIWisdom2]** Rabin, Steve: „AI Game Programming Wisdom 2“, Charles River Media, 2003.
http://www.aiwisdom.com/byresource_aiwisdom2.html
- [AIWisdom3]** Rabin, Steve: „AI Game Programming Wisdom 3“, Charles River Media, 2006.
http://www.introgameDEV.com/resource_aiwisdom3.html
- [AIWisdom4]** Rabin, Steve: „Call For Proposals: AI Game Programming Wisdom 4“, IntroGameDev.com.
<http://www.introgameDEV.com/cfp.html>
Kopie: IntroGameDev.com - Call For Proposals - AI Game Programming Wisdom 4 2007-05-03.pdf
- [Booth04]** Booth, Michael: „The Making of the Official Counter-Strike Bot“, Turtle Rock Studios, Game Developers Conference 2004.
<http://www.gdconf.com/conference/2004.htm>
- [Farnstrom06]** Farnstrom, Fredrik: „Improving on Near-Optimality: More Techniques for Building Navigation Meshes“, AI Game Programming Wisdom 3, Charles River Media, 2006, Kapitel 2.2, S. 113-128.
- [Hancock02]** Hancock, John: „Navigating Doors, Elevators, Ledges, and Other Obstacles“, AI Game Programming Wisdom, Charles River Media, 2002, Kapitel 4.5, S. 193-201.
- [O'Rourke98]** O'Rourke, Joseph: „Computational Geometry in C“, Cambridge University Press, 1998.
- [Rabin00]** Rabin, Steve: „A* Speed Optimizations“, Game Programming Gems, Charles River Media, 2000, Kapitel 3.5, S. 272-287.
- [Tozour02]** Tozour, Paul: „Building a Near-Optimal Navigation Mesh“, AI Game Programming Wisdom, Charles River Media, 2002, Kapitel 4.3, S. 171-185.
- [Tozour03]** Tozour, Paul: „Search Space Representations“, AI Game Programming Wisdom 2, Charles River Media, 2003, Kapitel 2.1, S. 85-102.
- [Wikipedia:Delaunay-Triangulation]** Wikipedia: „Delaunay-Triangulation“.
<http://de.wikipedia.org/wiki/Delaunay-Triangulation>
Kopie: Wikipedia - Delaunay-Triangulation 2007-06-04.pdf
- [Wikipedia:Dijkstra]** Wikipedia: „Edsger Wybe Dijkstra“.
http://de.wikipedia.org/wiki/Edsger_Wybe_Dijkstra
Kopie: Wikipedia - Edsger Wybe Dijkstra 2007-05-11.pdf
- [Wikipedia:Kibibyte]** Wikipedia: „Byte“.
<http://de.wikipedia.org/wiki/Kibibyte>
Kopie: Wikipedia - Byte 2007-06-04.pdf
- [Wikipedia:Mehlhorn]** Wikipedia: „Kurt Mehlhorn“.
http://de.wikipedia.org/wiki/Kurt_Mehlhorn
Kopie: Wikipedia - Kurt Mehlhorn 2007-06-04.pdf

[Wikipedia:Polygon] Wikipedia: „Polygon“.

<http://de.wikipedia.org/wiki/Polygon>

Kopie: Wikipedia - Polygon 2007-04-25.pdf

[Wikipedia:PolygonTriangulation] Wikipedia: „Polygon Triangulation“.

http://en.wikipedia.org/wiki/Polygon_triangulation

Kopie: Wikipedia - Polygon triangulation 2007-06-04.pdf

[Wikipedia:Vertex] Wikipedia: „Vertex“.

<http://en.wikipedia.org/wiki/Vertex>

Kopie: Wikipedia - Vertex 2007-06-04.pdf

Stichwortverzeichnis

Catmull-Rom Splines	4f
Circle-Based Waypoint Graphs	6ff
Corner Graphs	5f
Cylinder-Based Waypoint Graphs	7
Graph	2
KI-Agent	5f, 9
N-Sided-Poly-Based Navigation Meshes	10
Navigation Meshes	1, 4, 6, 8ff, 13f, 17, 19
NavMesh	9
Polygon	
- konkav	2
- konvex	1f, 9f
Polygon	1f, 9f
Quadtrees	8
Regular Grids	4f, 8
Reguläre Gitter	4
Space-Filling Volumes	6, 8f
Spielweltrepräsentation	
- Circle-Based Waypoint Graphs	6ff
- Corner Graphs	5f
- Cylinder-Based Waypoint Graphs	7
- N-Sided-Poly-Based Navigation Meshes	10
- Navigation Meshes	1, 4, 6, 8ff, 13f, 17, 19
- NavMesh	9
- Quadtrees	8
- Regular Grids	4f, 8
- Reguläre Gitter	4
- Space-Filling Volumes	6, 8f
- Triangle-Based Navigation Meshes	10
- Waypoint Graphs	4, 6f
Spielweltrepräsentation	1
String-Pulling	4
Suchalgorithmus	
- A* Algorithmus	3
- A*	1
- A-Star	1
- Best First	1
- Breadth First	1
- Depth First	1
- Dijkstra	1
- Hill Climbing	1
Suchalgorithmus	1
Suchraum	1
Suchraumrepräsentation	
- Spielweltrepräsentation	1
Triangle-Based Navigation Meshes	10
Vertex	1f
Waypoint Graphs	4, 6f