

# Einführung in die Spieleprogrammierung

Pfadplanung und Navigation



- Pfadplanung = Finden des optimalen Wegs in einer komplexen Umgebung
- Pfadplanung als Suchproblem
  - Uninformierte Suchstrategien
  - Informierte bzw. heuristische Suchstrategien
- Navigation = Bewegung, z.B. entlang eines geplanten Pfades oder Verfolgung eines anderen Objektes



- Kostenfaktoren:
  - Entfernung
  - Ressourcenverbrauch (Zeit, Brennstoff)
  - Taktische Aspekte (Deckung, Höhenvorteil, Bewegungsvorteil, Überraschung)
- Einflussfaktoren:
  - Größe des Agenten
  - Mögliche Fortbewegungsarten (Laufen, Springen, Klettern, Fliegen)
  - Dynamische Aspekte (Status von Türen und Fahrstühlen, Schadenswerte)

## Bewertungskriterien für Suchstrategien:

### 1. Vollständigkeit:

Werden vorhandene Lösungen garantiert gefunden?

### 2. Halteproblem:

Terminiert Algorithmus, wenn es keine Lösung gibt?

### 3. Effektivität:

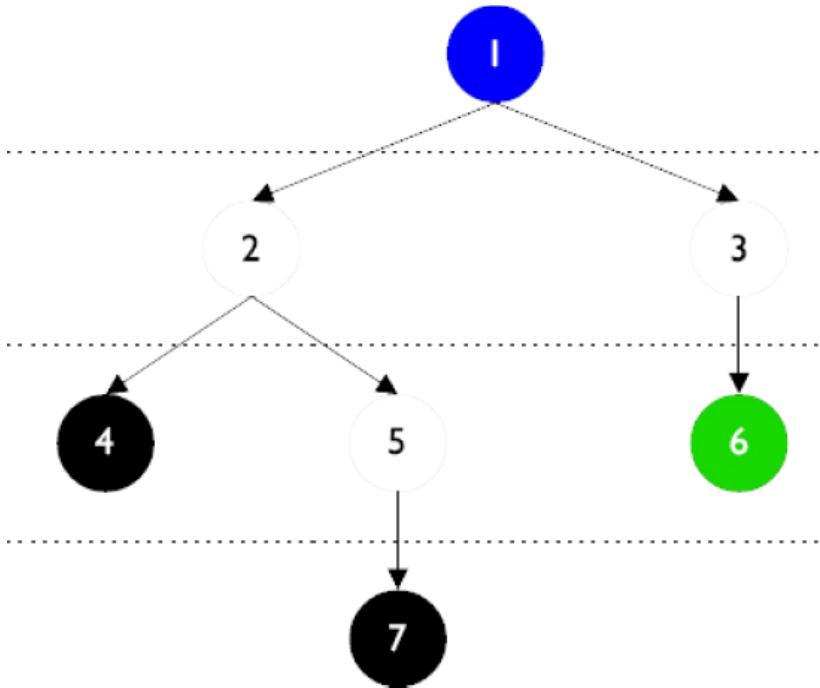
Wird die Lösung mit den geringsten Kosten gefunden?

### 4. Komplexität:

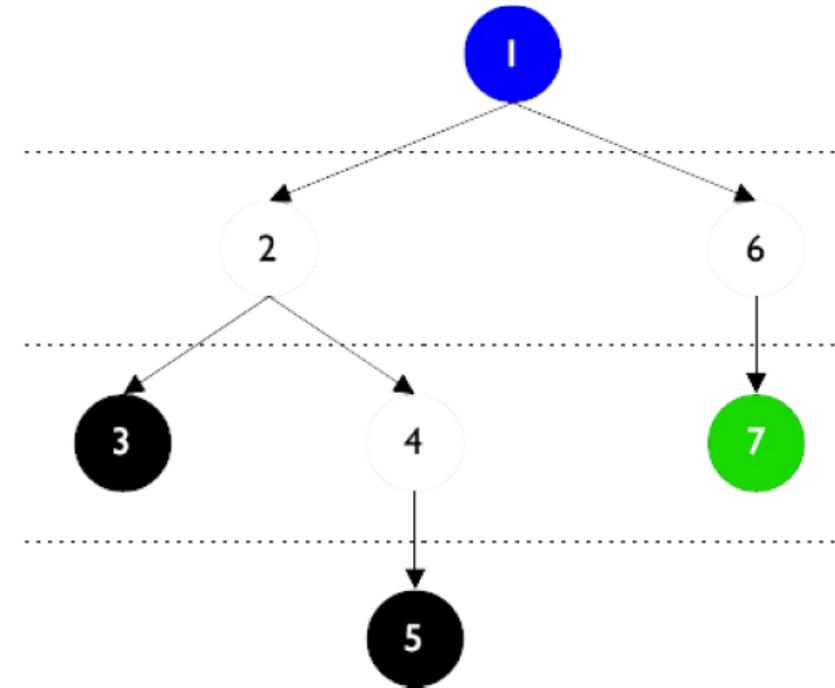
Wie viel Laufzeit und Speicherplatz benötigt der Algorithmus?

Repräsentation der Welt (= Suchraum) als Graph:

- Knoten: Raumunterteilung der Welt
  - Grundrisse
  - Einheitliche oder variable Gitter, z.B. basierend auf der Größe des kleinsten Agenten
  - Wegpunkte
  - Navigationsnetze
- Kanten: Erreichbarkeit, evtl. nötige Kosten
- Start- und Endpunkt



Breitensuche



Tiefensuche

```
Breadth-First(Node root)
```

```
{
```

```
    Queue open;  
    root.parent = NULL;  
    push(root, open);  
    loop
```

```
{
```

```
    if empty(open) return NO SOLUTION;
```

```
    Node current = pop(open);
```

```
    if IsGoal(current) return  
        constructPath(current);
```

```
    forall successor of current
```

```
{
```

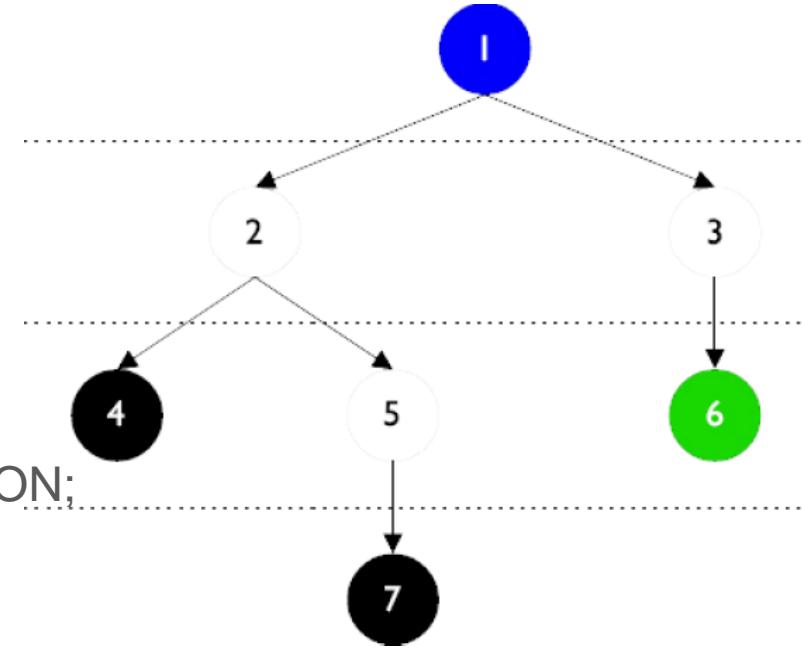
```
        successor.parent = current;
```

```
        append(successor, open);
```

```
}
```

```
}
```

```
}
```



```
Depth-First(Node root)
```

```
{
```

```
    Queue open;  
    root.parent = NULL;  
    push(root, open);
```

```
    loop
```

```
{
```

```
        if empty(open) return NO SOLUTION;
```

```
        Node current = pop(open);
```

```
        if IsGoal(current) return  
            constructPath(current);
```

```
        forall successor of current
```

```
{
```

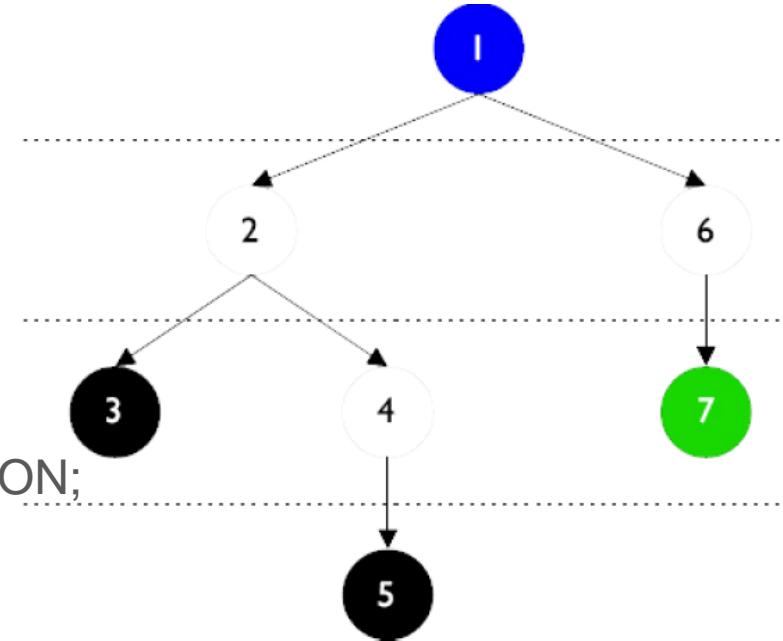
```
            successor.parent = current;
```

```
            push(successor, open);
```

```
}
```

```
}
```

```
}
```



## Fazit:

- Breitensuche findet optimale Lösung (= minimale Tiefe)
- Tiefensuche kann nicht-optimale Lösungen liefern
- Breitensuche ist vollständig
- Tiefensuche ist unvollständig, falls der Suchbaum unendlich groß ist

Verwendete Komplexitätsmaße: Verzweigungsfaktor  $b$ , Tiefe  $d$ ,  
Längster Pfad  $h$

Breitensuche: Laufzeit- & Speicherkomplexität:  $O(b^d)$

Tiefensuche:

Laufzeitkomplexität:  $O(b^d)$

Speicherkomplexität:  $O(h)$



## Tiefensuche mit iterativer Vertiefung:

- Suche bis in Tiefe i
- Falls in Tiefe i keine Lösung gefunden wurde, erhöhe Tiefenschranke

## Merkmale:

- Vollständige Suchstrategie
- Optimale Lösungen werden garantiert gefunden
- Laufzeitkomplexität:  $O(b^d)$
- Speicherkomplexität:  $O(bd)$



- Informierte bzw. Heuristische Suchstrategien berücksichtigen Kontextinformationen um die erfolgversprechendsten Knoten im Suchraum zu bevorzugen
- Mögliche Kontextinformationen über die Qualität eines Weges sind die bisherigen Kosten  $g$  und die geschätzten Kosten bis zum Ziel  $h$
- Beispiele:
  - Hill Climbing
  - Bestensuche
  - Dijkstra-Algorithmus
  - A\*-Algorithmus

- Bestimme alle Nachfolger des aktuellen Zustandes
- Bestimme Kosten  $h$  (geschätzte zukünftige) aller Nachfolger
- Wähle den kostengünstigsten Nachfolger aus
- Abbruch, falls aktueller Zustand kostengünstiger ist als alle Nachfolger
- Vorteile:
  - Leicht zu implementieren
  - lineare Laufzeit- und konstante Speicherkomplexität
- Problem:
  - Unvollständig: Ziel wird möglicherweise nicht erreicht
  - „Steckenbleiben“ in lokalen Minima

- Bestimme alle Nachfolger des aktuellen Zustand
- Bestimme Kosten  $g$  (seit Start) aller Nachfolger und füge sie zur Liste hinzu
- Wähle den kostengünstigsten Zustand aus der Liste aus (entferne aktuellen Zustand aus der Liste)
- Vorteile:
  - Leicht zu implementieren
  - lineare Laufzeit- und konstante Speicherkomplexität
- Problem:
  - Minimiert nicht die Gesamtkosten

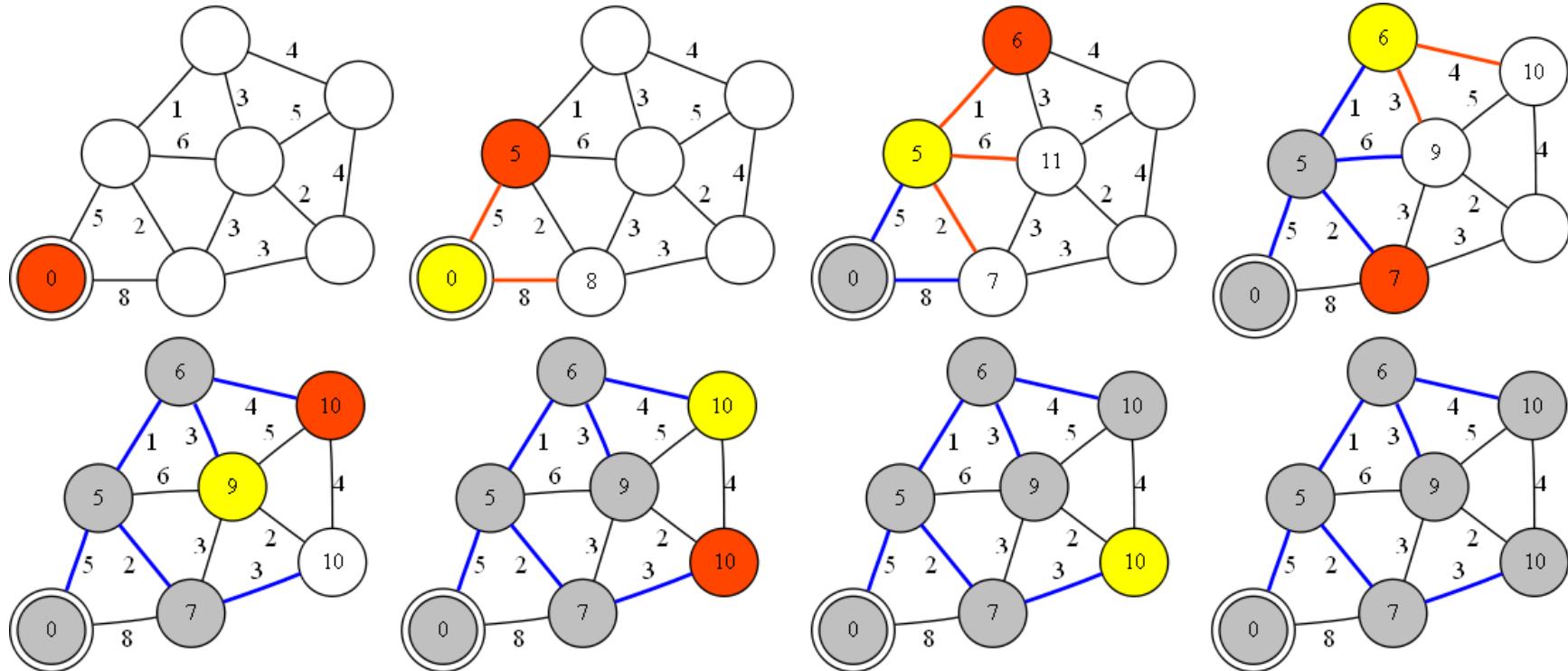
- Erstellt einen Shortest-Path-Tree (SPT)
- SPT enthält kürzeste Verbindung vom Startknoten zu jedem anderen Knoten
- Dijkstra liefert somit den optimalen Pfad zu allen potentiellen Zielknoten, iteriert hierzu aber in alle Richtungen
- Für den optimalen Pfad zu einem einzigen, bestimmten Zielknoten daher eher ungeeignet



# Dijkstra-Algorithmus

```
function Dijkstra(Graph, source):
    for each vertex v in Graph:                                // Initializations
        dist[v] := infinity                                     // Unknown distance function from source to v
        previous[v] := undefined                                // Previous node in optimal path from source
    dist[source] := 0                                         // Distance from source to source

    Q := the set of all nodes in Graph                         // All nodes in the graph are unoptimized - thus are in Q
    while Q is not empty:                                      // The main loop
        u := vertex in Q with smallest dist[]                  // where v has not yet been removed from Q
        if dist[u] = infinity: break                           // all remaining vertices are inaccessible from source
        remove u from Q
        for each neighbor v of u:
            alt := dist[u] + dist_between(u, v)
            if alt < dist[v]:                                    // Relax (u,v,a)
                dist[v] := alt
                previous[v] := u
    return dist[]
```



## Idee:

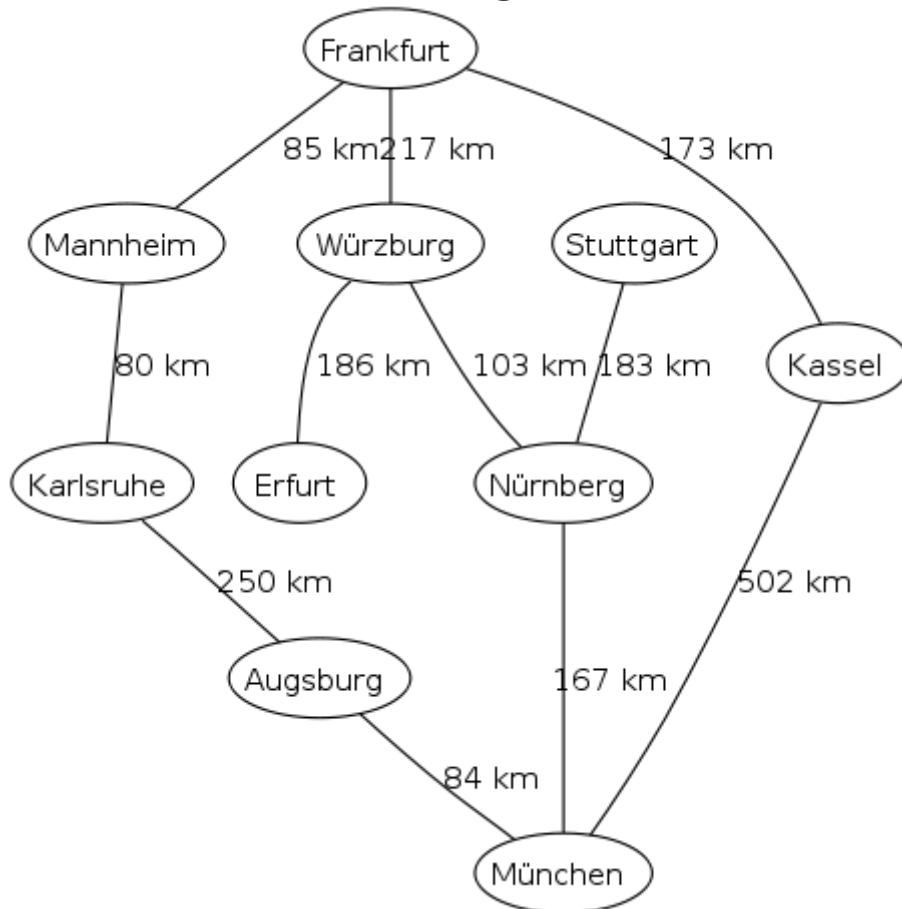
- Verwendung beider Heuristiken  $g$  und  $h$ 
  - $g$ : bisherige Kosten;  $h$ : zukünftige Kosten
- Verwaltung von Knoten in zwei separaten Datenstrukturen:
  - OPEN: Knoten, die bereits bewertet, aber noch nicht expandiert wurden
  - CLOSED: Knoten, die bewertet und expandiert wurden
- Reduzierung des Suchraums durch Ausschluss von Mehrfachwegen zu einem Knoten

- Start  $P_0$ , Ziel  $P_G$ , Wegpunkt  $P_i$
- Pfad  $P = P_0 \rightarrow P_i \rightarrow P_G$
- Voraussichtliche minimale Kosten für  $P$ :
  - $g(P_i^0)$ : Optimale Kosten für Abschnitt  $P_0 \rightarrow P_i$
  - $h(P_G^i)$ : Geschätzte minimale Kosten für Weg zum Ziel ( $P_i \rightarrow P_G$ )
  - $f(P_i)$ : Geschätzte minimale Gesamtkosten  $g(P_i^0) + h(P_G^i)$
- Speichere Kosten bekannter Wegstücke  $g(P_i^0)$
- Verfolge Wegstück mit den geringsten geschätzten Gesamtkosten

- Eine Kostenfunktion heißt *zulässig*, wenn sie niemals die voraussichtlichen Kosten  $h$  überschätzt.
- Ist die Kostenfunktion  $h$  zulässig, so findet der A\*-Algorithmus den optimalen Pfad.
- Emulation anderer Suchstrategien durch gewichtete Kostenfunktion:  
$$f(n) = g(n) + \omega h(n)$$
  - $0 \leq \omega < 1$ : Emuliert Breitensuche
  - $\omega \geq 1$ : Verstärkt den Einfluss der Heuristik

```
PriorityQueue open; Queue closed; Node N0;  
Assign f, g and h to N0;  
push(N0, open);  
loop  
{  
    if empty(open) then return NO_SOLUTION;  
    Node current = getBest(open);  
    if isGoal(current) then return path(current);  
    forall successor of successors(current)  
    {  
        Assign f, g and h to successor;  
        if successor ∈ open or successor ∈ closed  
            and successor is more efficient than the known solution  
            then update(open) or update(closed);  
            else insert(successor, open);  
    }  
    push(current, closed)  
}
```

## Kürzester Weg von Frankfurt nach München

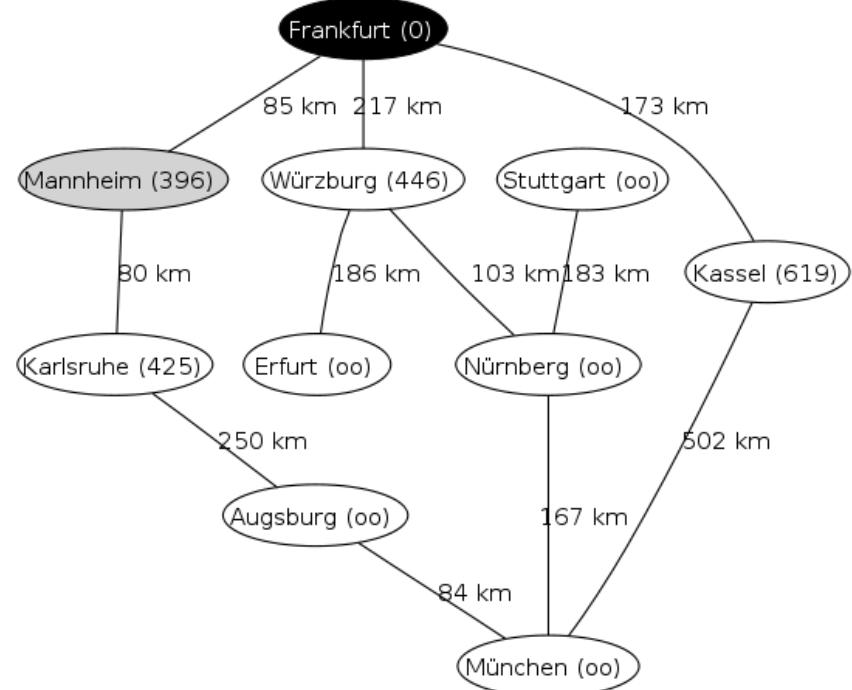
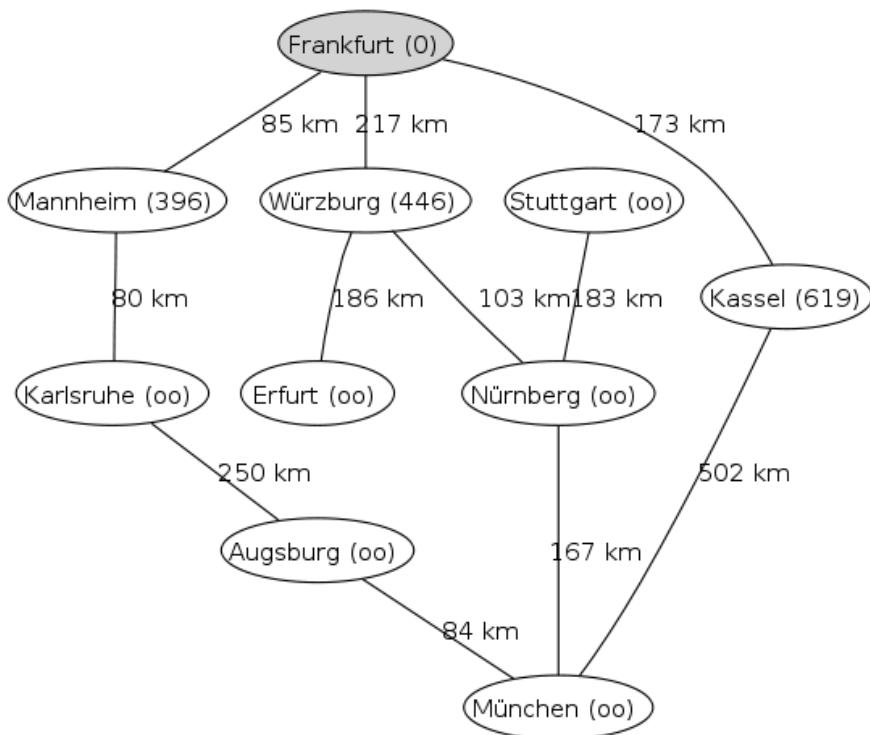


Augsburg <--> München: 43 km  
Erfurt <--> München : 342 km  
Frankfurt <--> München : 353 km  
Karlsruhe <--> München : 260 km  
Kassel <--> München : 446 km  
Mannheim <--> München : 311 km  
München <--> München : 0 km  
Nürnberg <--> München : 151 km  
Stuttgart <--> München : 199 km  
Würzburg <--> München : 229 km

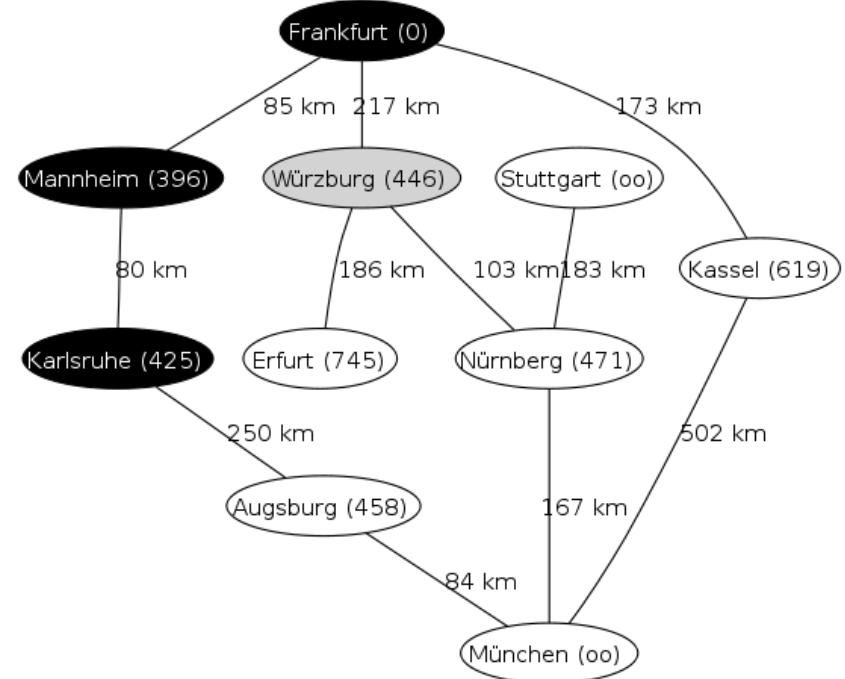
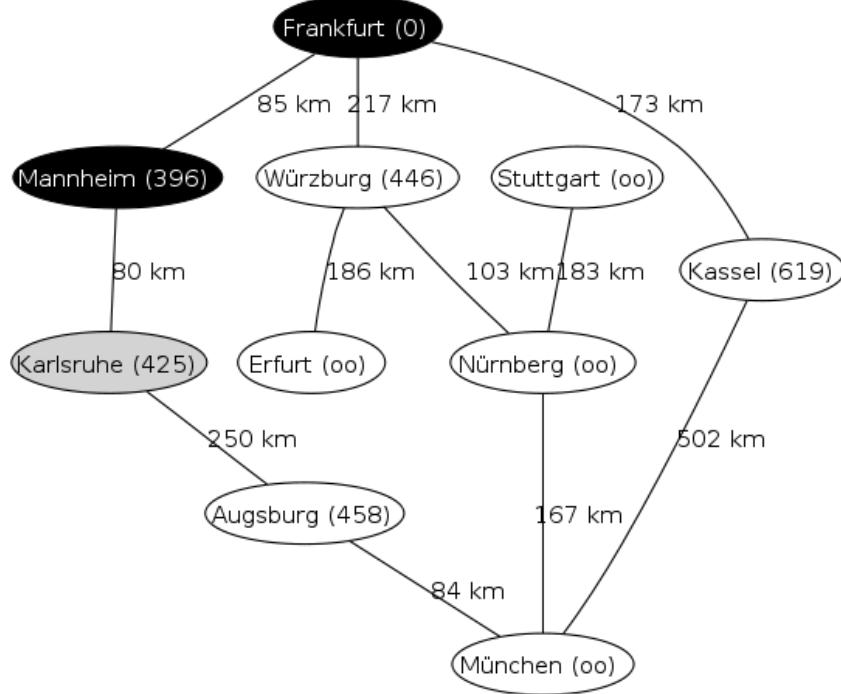
Luftlinien als Heuristik  $h$

Straßenkarte mit Distanzen g

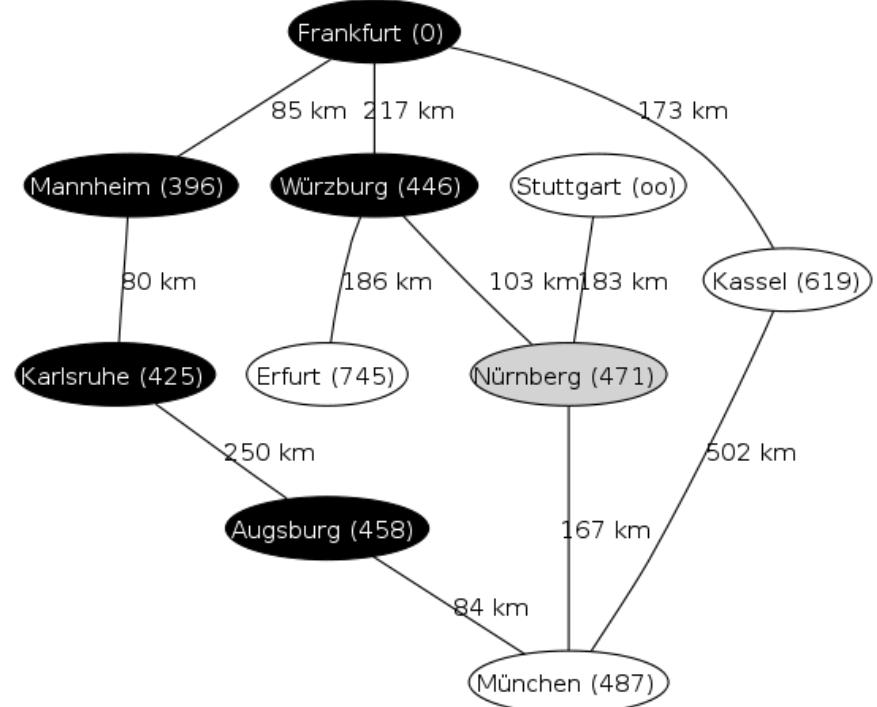
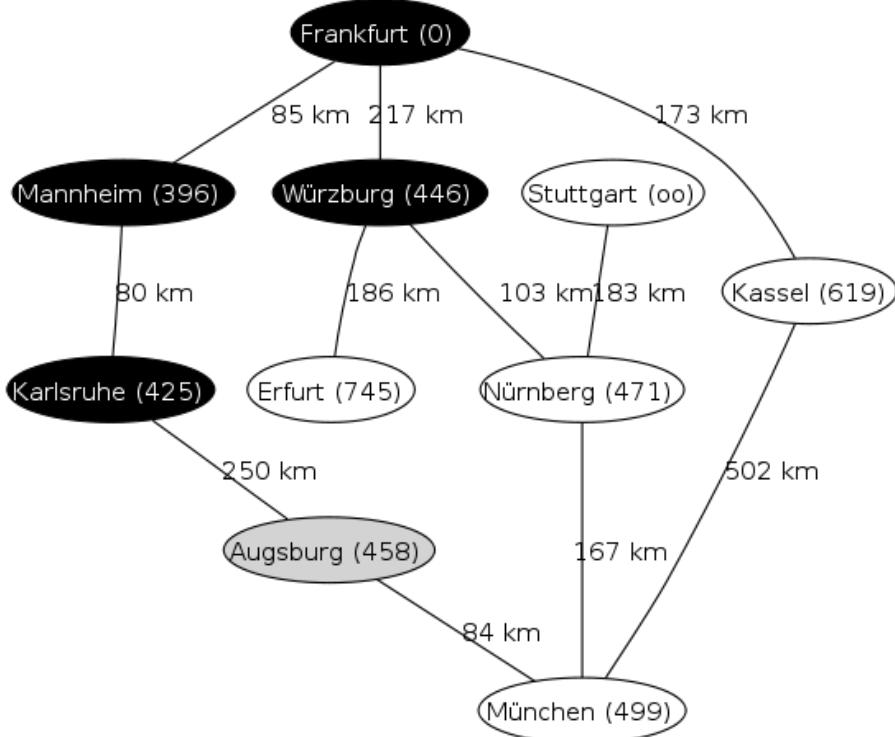
## Kürzester Weg von Frankfurt nach München



## Kürzester Weg von Frankfurt nach München



## Kürzester Weg von Frankfurt nach München



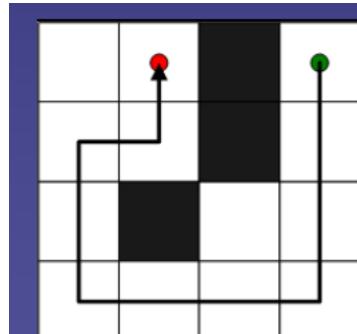
=> Kürzester Weg über Würzburg und Nürnberg, 487 km



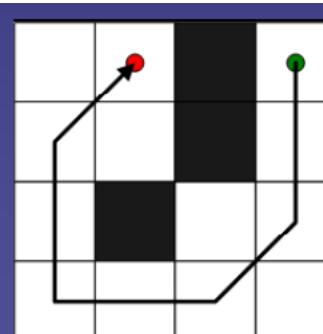
- Repräsentationsmöglichkeiten:
  - Einheitliche Unterteilung
  - Wegpunkte
  - Navigationsnetze
- Beurteilungskriterien:
  - Performanz
  - Speicherverbrauch
  - Visuelle Qualität der Pfade

- Einheitliche Unterteilung in Zellen, i.d.R. in Quadrate oder Waben (Hex-Felder)
- Meist bei (Echtzeit-)Strategiespielen eingesetzt, wenn Karte aus gleichmäßigen Zellen besteht
- Zentren der Zellen werden zu Knoten
- Benachbarte Knoten werden durch Kanten verbunden
- Gittergröße: Fest vorgegeben oder z.B. Größe der kleinsten Einheit

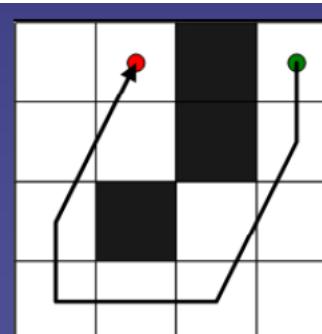
- Vorteil: Leicht automatisch generierbar
- Nachteile:
  - Großer Speicherbedarf
  - Suchraum wird schnell sehr groß
  - Unnatürliche Pfade, können aber durch Nachbearbeitung geglättet werden



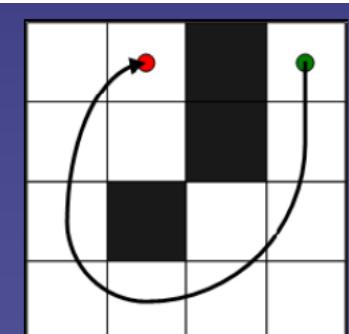
4er-Nachbarschaft



8er-Nachbarschaft

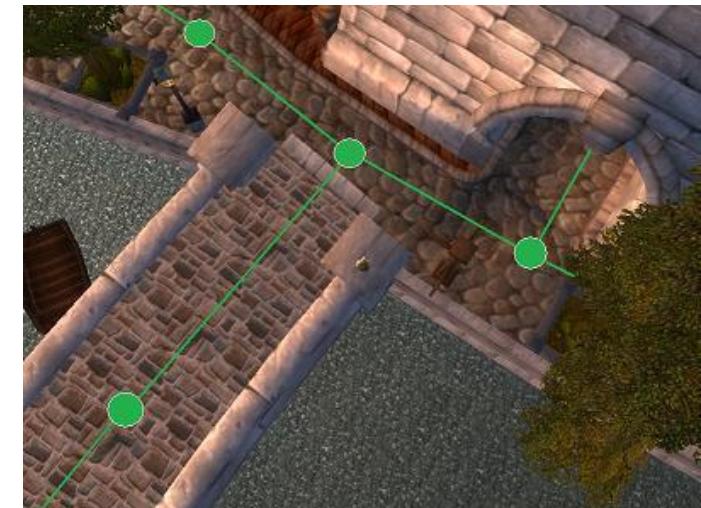
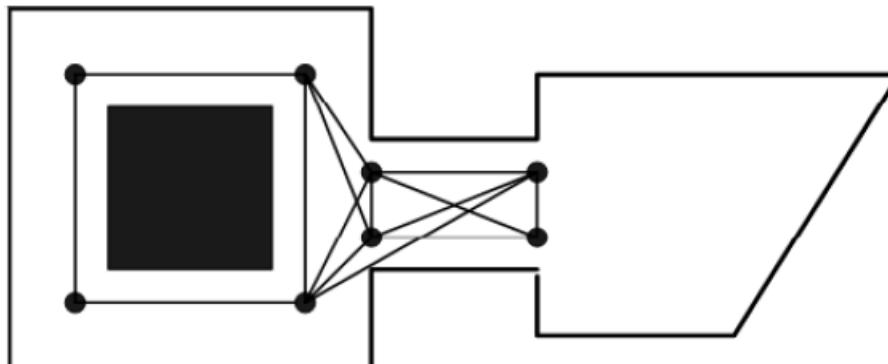


String Pulling



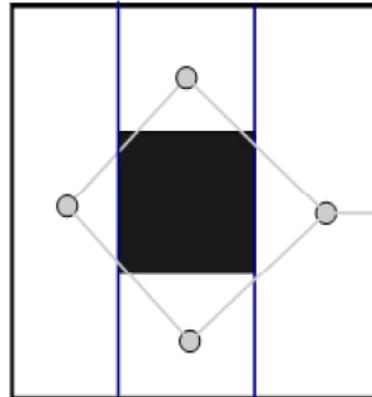
Splines

- Halbautomatisches oder manuelles Verteilen von Wegpunkten in der Welt
- Erstellen einer Navigationstabelle: Welche anderen Punkte sind von einem bestimmten Punkt aus sicht- bzw. erreichbar?
- Pfadplanung anhand der Tabelle



- Vorteile:
  - Relativ kleiner Suchraum
  - Berücksichtigung von Hindernissen möglich
  - Resultierende Pfade führen auf direktem Weg zum Ziel
  - Gut geeignet für Innenräume
- Nachteile:
  - Manueller Aufbau von Graphen aufwendig, automatischer Aufbau u.U. schwierig
  - Keine Repräsentation von dynamischen Objekten
  - Welten mit krummen Wänden können zu unnötig komplexen Graphen führen
  - Eher schlecht für offene Landschaften geeignet

- Polygonales Netz, beschreibt frei begehbar konvexe Areale
- Integrierter Ansatz für Wegsuche und Kollisionsvermeidung
- Vermeidet Kollisionen mit statischen Hindernissen
- Portal: Kante, die zwei Polygone verbindet



Sommersemester 2019

Einführung in die Spieleprogrammierung

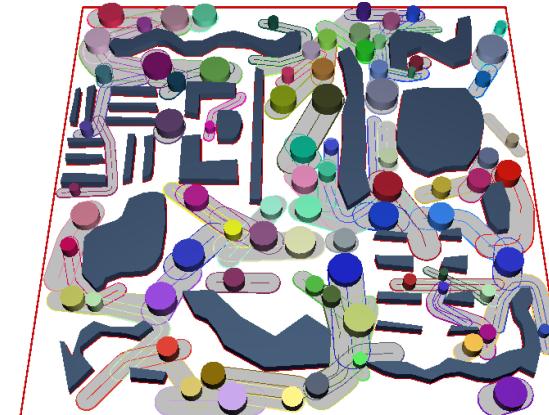
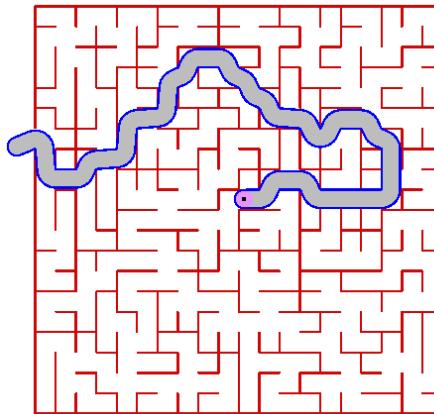


- Offline Erstellung der Navigationstabelle für ein Navigationsnetz aus  $N$  Polygonen:
  - Tabelle besitzt  $N \times N$  Einträge
  - Für jedes Paar( $i, j$ ) ( $1 \leq i, j \leq N$ ):
    - Bestimme kürzesten Pfad zwischen Mittelpunkten der Polygone
    - Bestimme und speichere Portal
- Pfadplanung:
  - Ähnlich Wegpunkte-Ansatz
  - Aber: Ausweichmöglichkeit und Pfadoptimierung innerhalb der Polygone



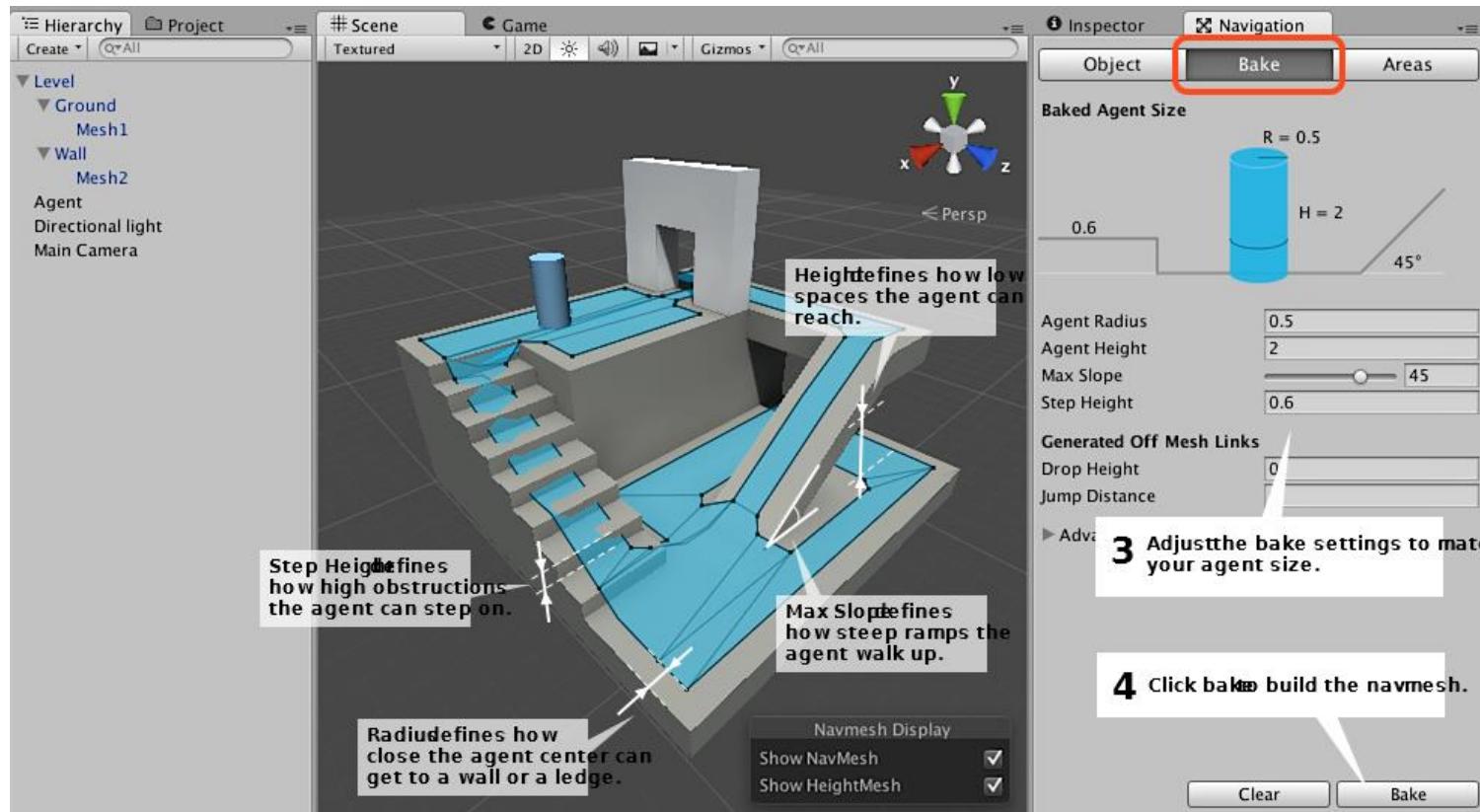
- Vorteile:
  - Ausnutzen vorhandener Geometrie
  - Erstellung kann automatisiert werden
  - Kleinerer Suchraum
- Nachteile:
  - Eventuell Maschinelles Ausdünnen notwendig
  - Dynamische Änderungen am Netz zur Laufzeit u.U. schwierig

## Kallmann: „Shortest Paths with Arbitrary Clearance from Navigation Meshes“ (2010)



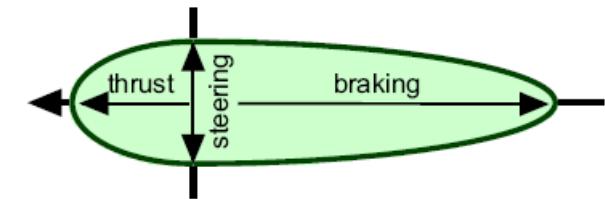
<http://graphics.ucmerced.edu/projects/10-sca-tripath/>

## Nav Meshes in Unity



- Keine einzelne „beste“ Methode:
  - Auch A\* hat Schwächen, meist Nachbearbeitung notwendig
  - Auswahl der Datenstruktur ebenso wichtig wie Suchalgorithmus
- Qualität der Lösung bedingt durch:
  - Genre
  - Layout der Spielwelt
  - Design der KI
  - Speicher und Performance der Zielplattform

- „Steering Behaviors“, basierend auf „Flocks, Herds, and Schools: A Distributed Behavioral Model“ (Reynolds, 1987) und „Steering Behaviors for Autonomous Characters“ (Reynolds, 1999)
- Eingesetzt z.B. in Dungeon Keeper 2
- „Fahrzeug“modell:
  - Punktmasse mit Geschwindigkeit, Beschleunigung, Richtungsvektor und Wahrnehmungsbereich
- Vorteile:
  - Einfache Implementierung
  - Effiziente Berechnungen
  - Für die meisten Anwendungen ausreichend
- Nachteile:
  - Entspricht nicht der Realität
  - Kein Schleudern, kein Anschneiden von Bewegungen
  - Erlaubt Drehungen im Stand





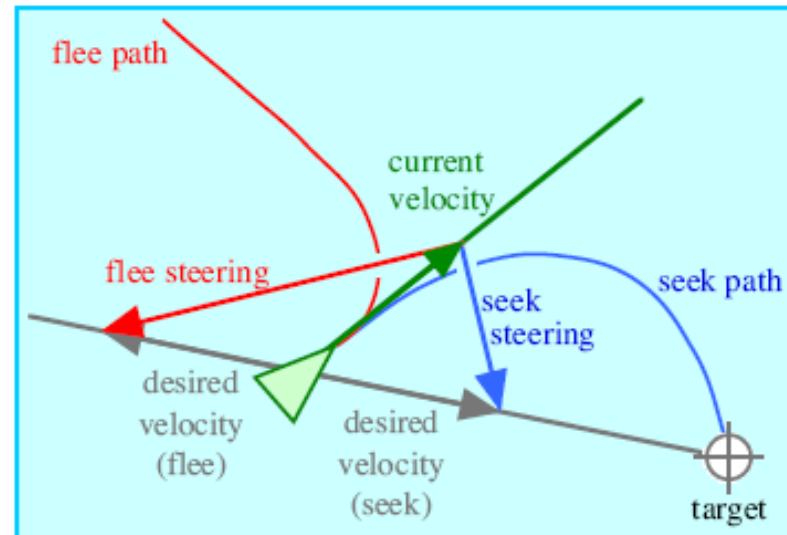
## Verschiedene Verhaltensmuster, u.a.:

- Einzelverhalten:
  - Seek, Flee
  - Pursue, Evade
  - Wander
  - Arrival
  - Obstacle Avoidance
  - Path Following
- Gruppenverhalten („Flocking“):
  - Separation
  - Cohesion
  - Alignment
  - Leader Following

Vorgehensweise (pro Fahrzeug, pro Frame):

- **Eingabe:**
  - Ein oder mehrere Verhaltensmuster (Kombination möglich)
  - Position
  - Zielpunkt
  - Hindernisse
  - Andere Fahrzeuge
  - ...
- **Ausgabe:**
  - Richtungsvektor
  - Beschleunigung entlang dieses Vektors

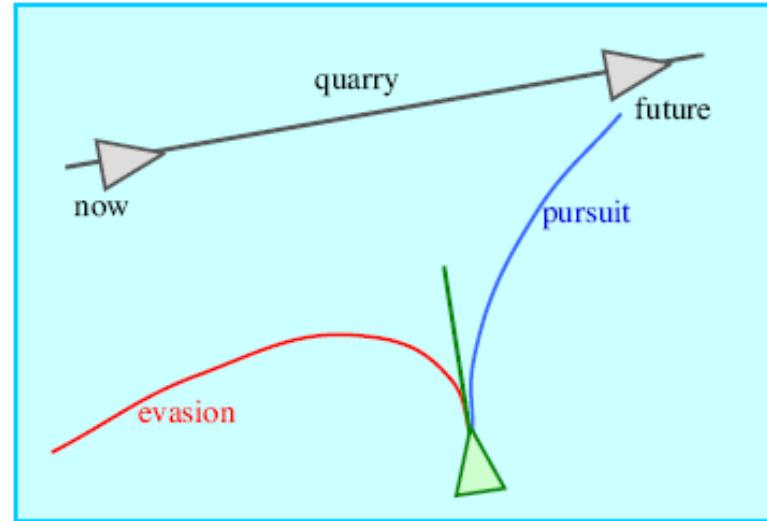
- Seek: Steuert Fahrzeug auf einen bestimmten Zielpunkt zu
- Flee: Steuert Fahrzeug von einem bestimmten Zielpunkt weg



- Berechnung:

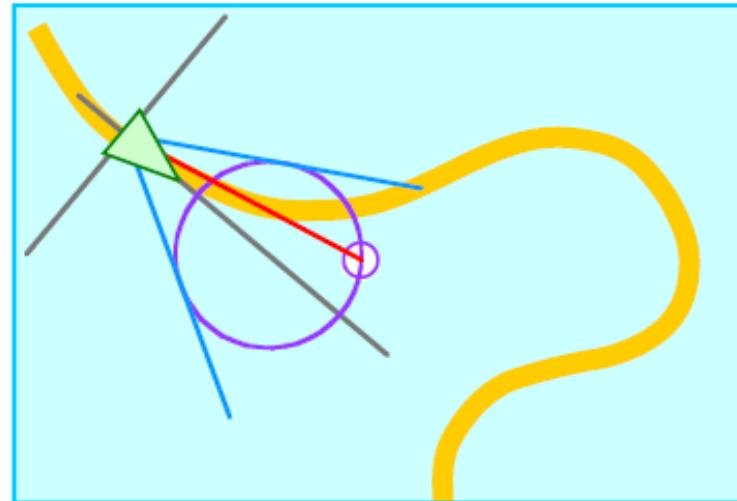
```
desired_velocity = normalize(position - target) * max_speed  
steering = desired_velocity - current_velocity
```

- Pursue: Steuert auf ein bestimmtes bewegliches Ziel zu
- Evade: Steuert von einem bestimmten beweglichen Ziel weg



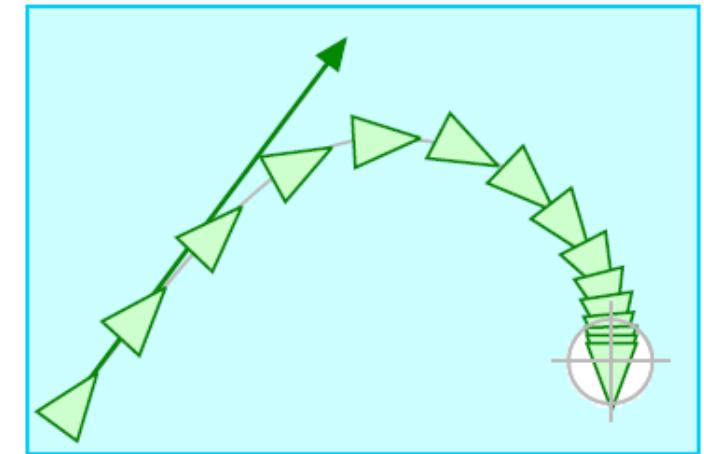
- Vorgehen: Bestimmte zukünftige Position des Ziels, wende Seek bzw. Flee auf diese Position an

- Zufälliges Erkunden der Welt, dabei aber keine zu abrupten Richtungsänderungen



- Idee: Grobe Beibehaltung der Bewegungsrichtung
- Umsetzung: Spitze des Richtungsvektors muss auf einem vor dem Fahrzeug platzierten Kreis sitzen

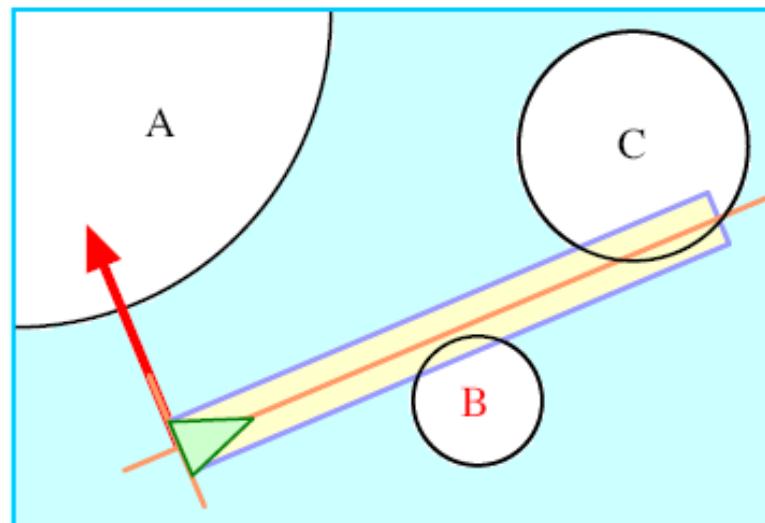
- Im Gegensatz zu Seek (das Fahrzeug bewegt sich mit voller Geschwindigkeit auf das Ziel zu und schießt womöglich darüber hinaus) bremst Arrival das Fahrzeug kontrolliert ab



- Berechnung:

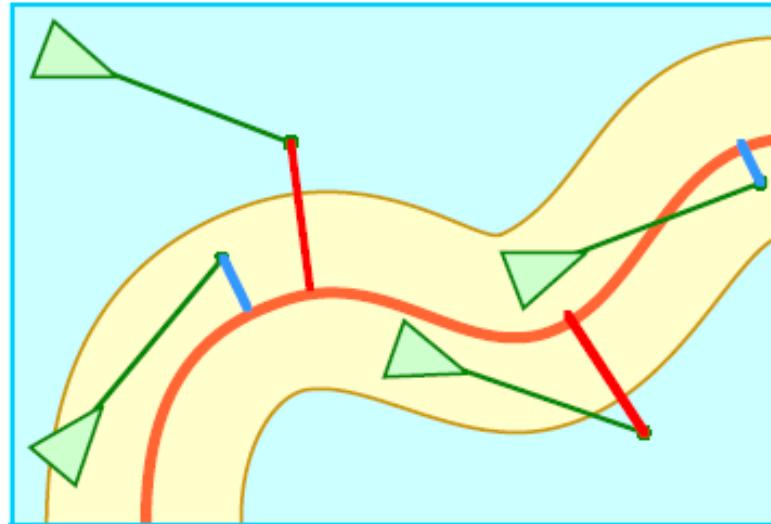
```
target_offset = target - position
distance = length(target_offset)
ramped_speed = max_speed * (distance / slowing_distance)
clipped_speed = minimum(ramped_speed, max_speed)
desired_velocity = (clipped_speed / distance) *
target_offset
steering = desired_velocity - velocity
```

- Fahrzeug sollte nicht blind auf Hindernisse zusteuern  
⇒ vorausschauendes Verhalten
- Im Gegensatz zu Flee wird Richtung nur dann korrigiert, wenn sich Hindernisse in der Nähe des Fahrzeuges befinden



- Idee:
  - Hindernisse werden durch Kreise angenähert.
  - Man versucht durch Gegensteuern ein imaginäres Rechteck vor dem Fahrzeug frei von Hindernissen zu halten

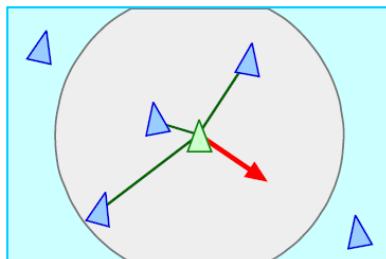
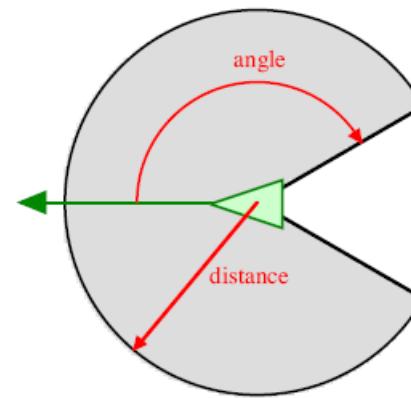
- Steuerung entlang eines vorgegebenen Pfades, aber nicht „wie auf Schienen“, sondern mit gewissem Spielraum



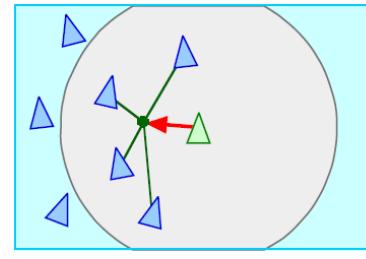
- Idee:
  - Repräsentiere Pfad durch Liniensegmente mit Radius
  - Bewege Fahrzeug innerhalb des entstehenden Korridors
  - Überprüfung, ob zukünftige Position im Korridor liegt, falls nein, Seek-Verhalten auf einen Punkt im Korridor

- Ziel: Simulation von koordinierten Bewegungen großer Gruppen (Schwärme), z.B. Menschenmengen, Tierherden, Verkehrsströme
- Problem: Globale Koordinierung ist aufwändig und fehleranfällig
- Idee: Verteiltes Modell, durch einfache lokale Regeln für jedes einzelne Schwarm-Mitglied („Boid“) entsteht komplexes globales Verhalten („Emergenz“):
  - Vermeide Kollision mit statischen Hindernissen
  - Vermeide Kollision innerhalb des Schwarms
  - Suche Nähe zum Schwarm
  - Gleiche eigene Bewegungen denen des Schwarms an
  - (folge dem Anführer)

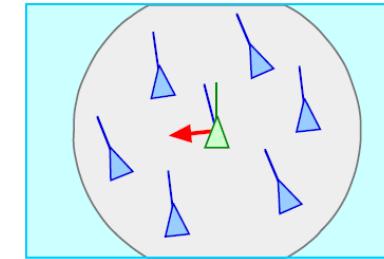
Für jeden Boid: Repräsentation des Sichtfelds und der lokalen Nachbarschaft, in jedem Frame Anwendung der lokalen Regeln



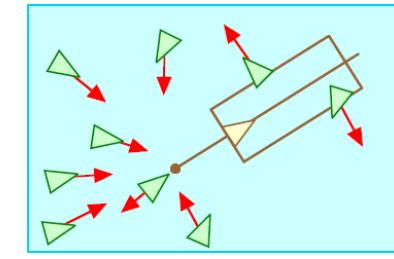
Separation



Cohesion

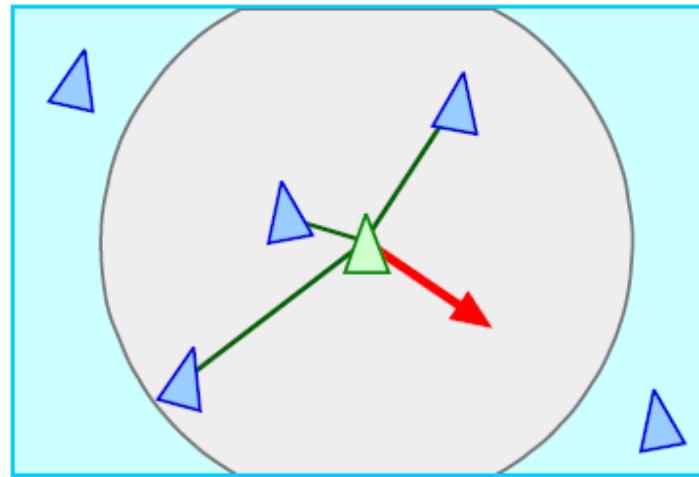


Alignment



Leader Following

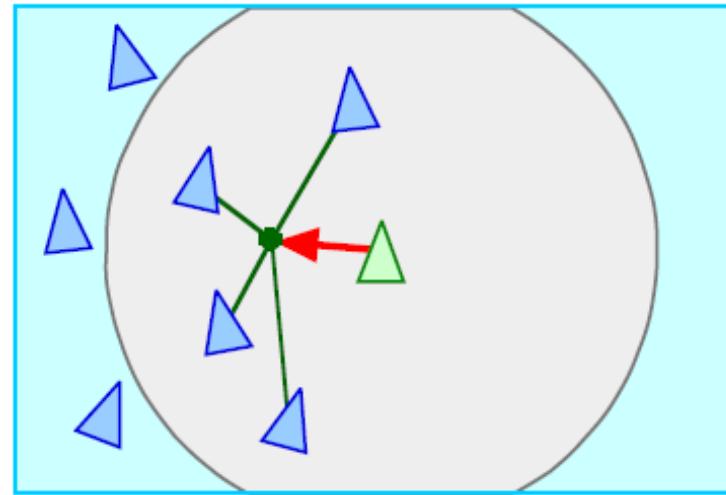
- Ziel: Abstand zu anderen Fahrzeugen halten



- Prinzip: Bestimmung einer Abstoßkraft
- Berechnung:

```
repulsive_force_n = normalise(pos_n - position) / dist_n;  
steering_force = truncate(sum(repulsive_force_n),max_force);
```

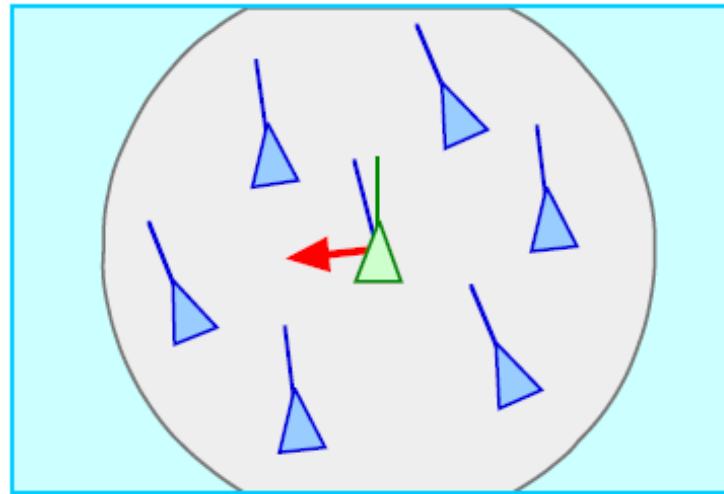
- Ziel: Anschluss an andere Fahrzeuge



- Prinzip: Die durchschnittliche Position aller Nachbarn ansteuern
- Berechnung:

```
average_position = sum(position_n) / n;  
steering_force = average_position - position;
```

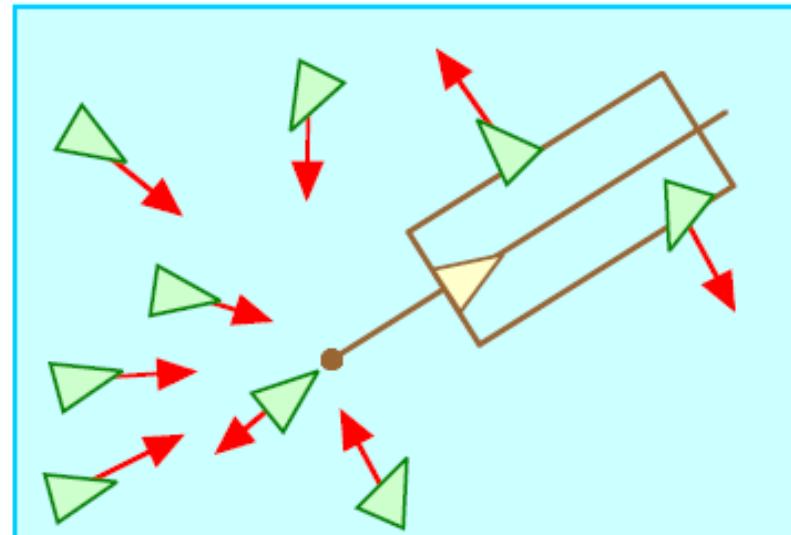
- Ziel: Ausrichtung an andere Fahrzeuge anpassen



- Prinzip: Die durchschnittliche Ausrichtung aller Nachbarn ansteuern
- Berechnung:

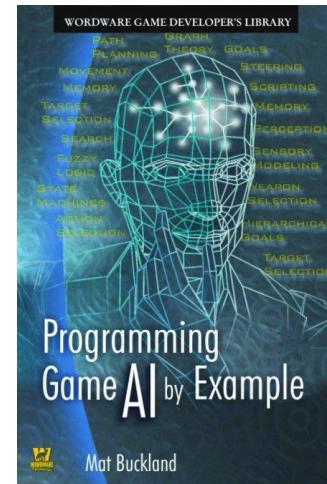
```
average_heading = sum(heading_n) / n;  
steering_force = average_heading - heading;
```

- Ziel: Einem Anführer folgen



- Prinzip: Kombination der Verhaltensweisen Separation, Cohesion, Alignment und Arrival (mit Zielpunkt hinter dem Anführer)

# Steering Behaviors



<http://www.red3d.com/cwr/steer/>



# Kombination von Verhaltensmustern

- Addition der Richtungsvektoren
- Gewichtete Addition der Richtungsvektoren
- Gewichtete Addition der Richtungsvektoren mit Priorisierung
- Wahrscheinlichkeitsbasierte Aktivierung gemäß Priorisierung

- Pinguine und Feldermäuse in „Batman Returns“
- Dinosaurierherde in „Jurassic Park“
- Tierherden in „König der Löwen“
- Fischschwärme in „Findet Nemo“
- Ork-Armeen und Vogelschwärme in „Der Herr der Ringe“
- Schlachtszenen in „Troja“
- ...



Assassin's Creed (2007)



Dead Rising (2006)



Hitman (2016)

Shao & Terzopoulos: „Autonomous Pedestrians“ (2005)



<http://www.cs.ucla.edu/~dt/videos/pedestrians/sca05-2.mp4>

<http://www.cs.ucla.edu/~dt/papers/sca05/sca05.pdf>

Yeh et al.: „Composite Agents“ (2008)



<http://gamma.cs.unc.edu/CompAgent/CompAgent.avi>

<http://gamma.cs.unc.edu/CompAgent/egMain.pdf>

Shopf, Oat & Barczak: „GPU Crowd Simulation“ (2008)



<http://download-developer.amd.com/GPU/videos/GPUCrowdSimulation.mov>

[http://developer.amd.com/gpu\\_assets/GPUCrowdSimulation.pdf](http://developer.amd.com/gpu_assets/GPUCrowdSimulation.pdf)

# Pathfinding: Horizon Zero Dawn





## Beyond Killzone: Creating AI Systems for Horizon Zero Dawn

- Waypoints -> Detour/Recast (Navmesh)
- Dynamic Navmesh in Bubble, threaded, per NPC-Klasse
- Kosten per Polygon, Gefahr, zerstörbare Hindernisse
- Velocity Obstacles; 5 Gefahren
- Menschen: Machine Learning Model: natürliches Verhalten
- Bezier; Smoothing/Clothoids Euler Spirals(Kreis vgl. Boids, skalierung ~ Geschwindigkeit), funnel

<http://www.gdcvault.com/play/1024912/Beyond-Killzone-Creating-New-AI>



# Links

- Open Steer: Open Source Steering Behaviors und Visualisierungen: <http://opensteer.sourceforge.net/>
- Programming Game AI by Example, Sourcen und Binaries: <http://www.jbpub.com/catalog/9781556220784/samples/>
- <http://aigamedev.com/open/articles/simulating-crowd-flow-dynamics/>
- <http://crowdsimulation.blogspot.com/>