DM842

Computer Game Programming: AI

Lecture 4
Movement in 3D
Path Finding

Christian Kudahl

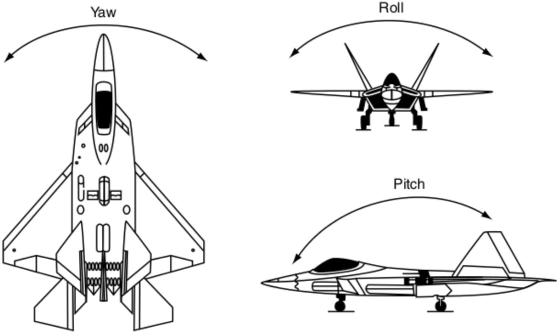
Department of Mathematics & Computer Science
University of Southern Denmark

Outline

1. Movement in 3D
2. Pathfinding

Movement in 3D

So far we had only orientation and rotation in the up vector.

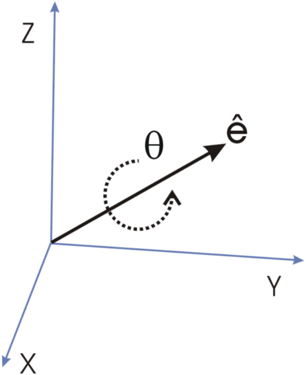


roll > pitch > yaw
⇝ we need to bring the third dimension in orientation and rotation.

Euler axis and angle

Any rotation can be expressed as a single rotation about some axis (Euler's rotation theorem). The axis can be represented as a 3D unit vector $\mathbf{e} = [e_x \;\; e_y \;\; e_z]^{\mathrm{T}}$, and the angle by a scalar θ.

$$\mathbf{r} = \theta \mathbf{e}$$



Quaternions

Quaternion: normalized 4D vector: $\hat{\mathbf{q}} = [q_1 \ q_2 \ q_3 \ q_4]^{\mathrm{T}}$

related to axis and angle:

$$q_1 = \cos(\theta/2)$$
$$q_2 = e_x \sin(\theta/2)$$
$$q_3 = e_y \sin(\theta/2)$$
$$q_4 = e_z \sin(\theta/2)$$

it follows:

$$q_1^2 + q_2^2 + q_3^2 + q_4^2 = 1$$

$a + bi + cj + dk$ with $\{a, b, c, d\} \in \mathbb{R}$ and where $\{1, i, j, k\}$ are the basis (hypercomplex numbers).
The following must hold for the basis

$$i^2 = j^2 = k^2 = ijk = -1$$

which determines all the possible products of i, j, and k:

$$\begin{aligned} ij &= k, & ji &= -k, \\ jk &= i, & kj &= -i, \\ ki &= j, & ik &= -j, \end{aligned}$$

A good 3D math library of the graphics engine will have the relevant code to carry out combinations rotations, ie, products of quaternions.

Summary

Kinematic Movement

- Seek
- Wandering

Steering Movement

- Seek and Flee
- Arrive
- Align
- Velocity Matching

Delegated Steering

- Pursue and Evade
- Face
- Looking Where You Are Going
- Wander
- Path Following
- Separation
- Collision Avoidance
- Obstacle and Wall Avoidance

Steering Behaviours in 3D

- Behaviours that do not change angles do not change: seek, flee, arrive, pursue, evade, velocity matching, path following, separation, collision avoidance, and obstacle avoidance

- Behaviours that change: align, face, look where you're going, and wander

Align

Input a target orientation
Output rotation match character's current orientation to target's.

$\hat{\mathbf{q}}$ quaternion that transforms current orientation $\hat{\mathbf{s}}$ into $\hat{\mathbf{t}}$ is given by:

$$\hat{\mathbf{q}} = \hat{\mathbf{s}}^{-1}\hat{\mathbf{t}}$$

$\hat{\mathbf{s}}^{-1} = \hat{\mathbf{s}}^*$ conjugate because unit quaternion (corresponds to rotate with opposite angle, $\theta^{-1} = -\theta$)

$$\hat{\mathbf{s}}^* = \begin{bmatrix} r \\ i \\ j \\ k \end{bmatrix}^{-1} = \begin{bmatrix} r \\ -i \\ -j \\ -k \end{bmatrix} \qquad \left(\hat{\mathbf{s}^*} = \begin{bmatrix} s_1 & = \cos\left(-\theta/2\right) \\ s_2 & = e_x \sin\left(-\theta/2\right) \\ s_3 & = e_y \sin\left(-\theta/2\right) \\ s_4 & = e_z \sin\left(-\theta/2\right) \end{bmatrix} \right)$$

To convert $\hat{\mathbf{q}}$ back into an axis and angle:

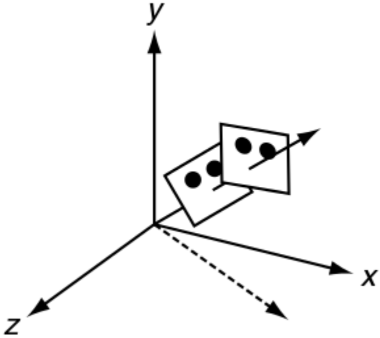
$$\theta = 2 \arccos q_1 \qquad \mathbf{e} = \frac{1}{2\sin(\theta/2)} \begin{bmatrix} q_2 \\ q_3 \\ q_4 \end{bmatrix}$$

Rotation speed: equivalent to 2D $\rightsquigarrow$ start at zero and reach θ and combine this with the axis $\mathbf{e}$.

Face and Look WYAG

Input a vector (from the current character position to a target, or the velocity vector).
Output a rotation to align the vector
That is: position the z-axis of the character in the input direction

In 2D we used $\theta = \arctan(v_x/v_z)$ knowing the two vectors. In 3D infinite possibilities



Align to a vector

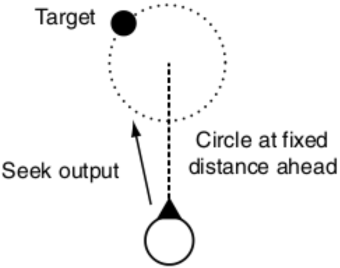
$\mathbf{v}_1$: unit vector pointing in direction we are currently looking.
$\mathbf{v}_2$: unit vector in direction we want to look.

$\mathbf{r} = \mathbf{v}_1 \times \mathbf{v}_2 = \sin(\theta)\mathbf{a}$
where θ is the angle between $\mathbf{v}_1$ and $\mathbf{v}_2$, and $\mathbf{a}$ is the axis we want to rotate around. Since $\mathbf{a}$ is a unit vector, we can find the angle. We now have axis and angle $\rightarrow$ put it into quaternion.

Special case: The cross product will be 0 if $\mathbf{v}_1$ and $\mathbf{v}_2$ are pointing in the same or opposite directions. If they point in same direction, no rotation needed. If they point in opposite directions, rotate $\mathbf{v}_1$ π radians around any axis (to make it point in the opposite direction).

Wandering

In 2D



keeps target in front of character and turning angles low

In 3D:

- 3D sphere on which the target is constrained,
- offset at a distance in front of the character.
- to represent location of target on the sphere, more than one angle. quaternion makes it difficult to change by a small random amount
- 3D vector of unit length. Update its position adding random amount $< \frac{1}{\sqrt{3}}$ to each component and normalize it again.

To simplify the math:

- wander offset (from char to center of sphere) is a vector with only a positive z coordinate, with 0 for x and y values.
- maximum acceleration is also a 3D vector with non-zero z value

Use Face to rotate and max acceleration toward target

Rotation in x–z plane more important than up and down (eg for flying objects) $\rightsquigarrow$ two radii

```
class Wander3D (Face3D):
  wanderOffset # 3D vector
  wanderRadiusXZ
  wanderRadiusY
  wanderRate # < 1/sqrt(3) = 0.577 to avoid ending up with a zero vector
  wanderVector # current wander offset orientation
  maxAcceleration # 3D vector
  # ... Other data is derived from the superclass ...
  def getSteering():
    # Update the wander direction
    wanderVector.x += (random(0,1)-random(0,1)) * wanderRate
    wanderVector.y += (random(0,1)-random(0,1)) * wanderRate
    wanderVector.z += (random(0,1)-random(0,1)) * wanderRate
    wanderVector.normalize()
    # Calculate the transformed target direction and scale it
    target = wanderVector * character.orientation
    target.x *= wanderRadiusXZ
    target.y *= wanderRadiusY
    target.z *= wanderRadiusXZ
    # Offset by the center of the wander circle
    target += character.position + wanderOffset * character.orientation
    steering = Face3D.getSteering(target)
    steering.linear = maxAcceleration * character.orientation
    return steering
```

Outline

1. Movement in 3D

2. Pathfinding

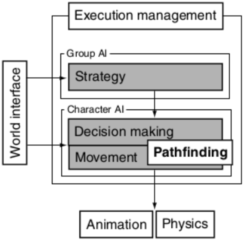
Motivation

For some characters, the route can be prefixed but more complex characters don't know in advance where they'll need to move.

- a unit in a real-time strategy game may be ordered to any point on the map by the player at any time
- a patrolling guard in a stealth game may need to move to its nearest alarm point to call for reinforcements,
- a platform game may require opponents to chase the player across a chasm using available platforms.

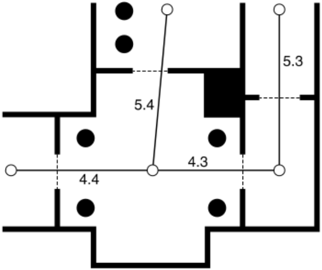
We'd like the route to be sensible and as short or rapid as possible

⇝ pathfinding (aka path planning) finds the way to a goal decided in decision making



Graph representation

Game level data simplified into directed non-negative weighted graph



node: region of the game level, such as a room, a section of corridor, a platform, or a small region of outdoor space

edge/arc: connections, they can be multiple

weight: time or distance between representative points or a combination thereof

Best first search

State Space Search
We assume:

- A start state
- A successor function
- A goal state or a goal test function

- Choose a metric of best
  Expand states in order from best to worst

- Requires:
  Sorted open list/priority queue
  closed list
  unvisited nodes

Best first search

Definitions

- Node is expanded/processed when taken off queue
- Node is generated/visited when put on queue
- g-cost is the cost from the start to the current node
- h-cost is a guess (heuristic) of the cost from the current node to the goal
- $c(a, b)$ is the edge cost between a and b

Algorithm Measures

- Complete
  Is it guaranteed to find a solution if one exists?
- Optimal
  Is it guaranteed to find the optimal solution?
- Time
- Space

Best-First Algorithms

Best-First Pseudo-Code

```
Put start on OPEN
While(OPEN is not empty)
  Pop best node n from OPEN # expand n
  if (n == goal) return path(n, goal)
  for each child of n: # generate
        children
    put/update value on OPEN/CLOSED
  put n in CLOSED
return NO PATH
```

Best-First child update

```
If child on OPEN, and new cost is less
  Update cost and parent pointer
If child on CLOSED, and new cost is less
  Update cost and parent pointer, move
       node to OPEN
Otherwise
  Add to OPEN list
```

Search Algorithms

Dijkstra's algorithm $\equiv$ **Uniform-Cost Search (UCS)**
$\rightsquigarrow$ Best-first with g-cost
Complete? Finite graphs yes, Infinite yes if $\exists$ finite cost path, eg, weights $> \epsilon$
Optimal? yes

Idea: reduce fill nodes: Heuristic: estimate of the cost from a given state to the goal

Pure Heuristic Search / **Greedy Best-first Search (GBFS)**
$\rightsquigarrow$ Best-first with h-cost
Complete? Only on finite graph
Optimal? No

A$*$
$\rightsquigarrow$ best-first with f-cost, $f = g + h$.
Optimal? depends on heuristic

Termination
When the node in the open list with the smallest cost-so-far has a cost-so-far value greater than the cost of the path we found to the goal, ie, at expansion. (like in Dijkstra)

Note: with any heuristic, when the goal node is the smallest estimated-total-cost node on the open list we are not done since a node that has the smallest estimated-total-cost value may later after being processed need its values revised.

In other terms: a node may need revision even if it is in the closed list ($\neq$ Dijkstra) because we may have been excessively optimistic in its evaluation (or too pessimistic with the others).

(Some implementations may stop already when the goal is first visited, or expanded, but then not optimal)

However if the heuristic has some properties then we can stop earlier:

Theorem

If the heuristic is:

- admissible $h(n) \leq h^*(n)$ where $h^*(n)$ is the *true* cost from n to goal ($h(n) \geq 0$, so $h(G) = 0$ for any goal G)
- consistent

$$h(n) \leq c(n, n') + h(n') \qquad n' \text{ sucessor of } n$$

(triangular inequality holds)

then when A* selects a node for expansion (smallest estimated-total-cost), the optimal path to that node has been found.

E.g., $h_{\mathrm{SLD}}(n)$ (straight line distance) never overestimates the actual road distance

Note:

- consistent $\Rightarrow$ admissible

Heuristic Examples.

E.g., for the 8-puzzle:

$h_1(n)$ = number of misplaced tiles
$h_2(n)$ = total Manhattan distance
(i.e., no. of squares from desired location of each tile)

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

$h_1(S) = 6$
$h_2(S) = 4+0+3+3+1+0+2+1 = 14$

To Reopen or not in A*?

An admissable heuristic in enough to allow us to terminate in A* after expanding the goal.

If we require our heuristic to be consistent, we will never have to open any closed node, since each one will be expanded only when the shortest path to it has been found $\rightarrow$ better running time.
(Similar to Dijkstra)

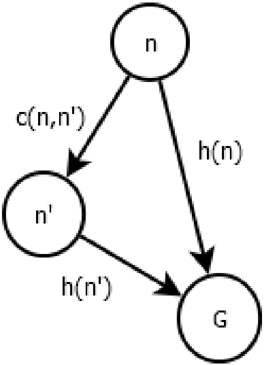
Optimality of A*

A heuristic is consistent if

$$h(n) \leq c(n, n') + h(n')$$

If h is consistent, we have

$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n, n') + h(n') \\ &\geq g(n) + h(n) \\ &= f(n) \end{aligned}$$



This gives us **Observation 1**: $f(n)$ is nondecreasing along any path. (note that g is the cost of getting to the current node in this specific path)

Optimality of A*

Observation 2: When A^* selects a node, n, for expansion, the optimal path to that node has been found.

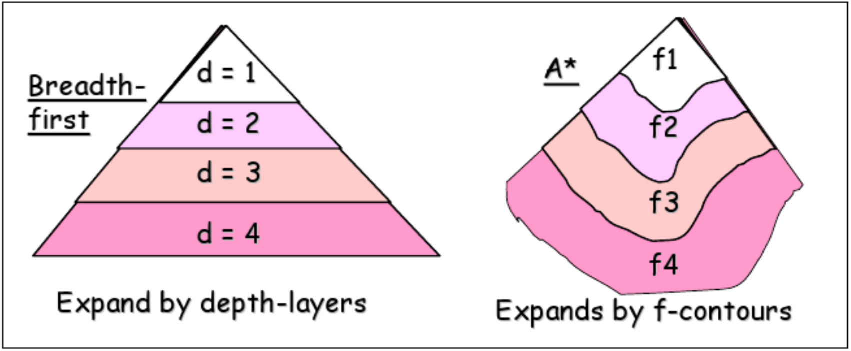
Using **Observation 1**, we see that if a shorter unexplored path to n existed, we would instead expand a node on that path, since it has smaller f value.

Optimality of A*

Observation 3: When the goal state is expanded, we have found the shortest path to it.

Follows from **Observation 2**.

A* vs. Breadth First Search



Properties of A*

Complete? Yes
Optimal? Yes—cannot expand f_{i+1} until f_i is finished
A* expands all nodes with $f(n) < C^*$
A* expands some nodes with $f(n) = C^*$
A* expands no nodes with $f(n) > C^*$

Time: $O(|E| + V \log V + V \cdot h)$, where E is the number of edges and V is the number of nodes (vertices), and h is the time needed to calculate the heuristic. As long as the time for h is $O(\log V)$, the last term can be omitted. Note, that for V and E, we only need to count those nodes (and their edges), whose f-value is less than or equal to that of the goal since other nodes are never processed. Using heuristic 0 gives Dijkstra, but a better heuristic greatly improves running time.

Space: $O(V)$ Keeps all nodes in memory