DM810

Computer Game Programming II: AI

Lecture 5
3D Movement
Path Finding

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Resume

Kinematic Movement

- Seek
- Wandering

Steering Movement

- Variable Matching
- 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

Combined Steering

- Blending
- Priorities
- Cooperative Arbitration
- Steering Pipeline

Resume

- Predicting Physics
- Firing Solutions
- Jumping
- Coordinated Movement
- Motor Control

Outline

1. Movement in 3D

2. Pathfinding

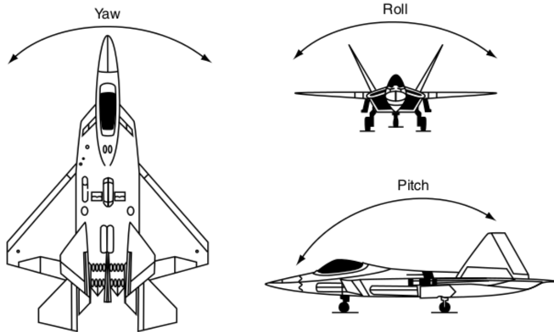
Outline

1. Movement in 3D

2. Pathfinding

Movement in 3D

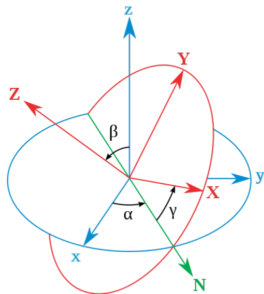
So far we had only orientation and rotation in the up vector.



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

Euler angles

Orientation and rotation in 3D have 3 degrees of freedom $\rightsquigarrow$ 3D vector.



Euler angles represent the spatial orientation of any coordinate system (X, Y, Z) as a composition of rotations from a coordinate system of reference (x, y, z).

- α between x-axis and line of nodes.
- β between z-axis and Z-axis.
- γ between the line of nodes and the X-axis.

Rotation matrix

Define unit vectors called basis.

The rotation is then fully described by specifying the coordinates (scalar components) of this basis in its current (rotated) position, in terms of the reference (non-rotated) coordinate axes.

The three unit vectors $\mathbf{u}, \mathbf{v}$ and $\mathbf{w}$ which form the rotated basis each consist of 3 coordinates, yielding a total of 9 parameters. These parameters can be written as elements of a 3×3 matrix $\mathbf{A}$, called rotation matrix.

$$\mathbf{A} = \begin{bmatrix} \mathbf{u}_x & \mathbf{v}_x & \mathbf{w}_x \\ \mathbf{u}_y & \mathbf{v}_y & \mathbf{w}_y \\ \mathbf{u}_z & \mathbf{v}_z & \mathbf{w}_z \end{bmatrix}$$

combining rotations:
$R = Z_\alpha X_\beta Z_\gamma$

conditions for $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$ to be a 3D orthonormal basis:

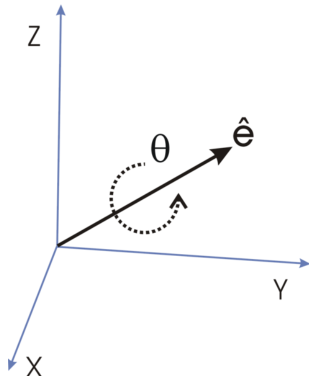
$$|\mathbf{u}| = |\mathbf{v}| = 1$$
$$\mathbf{u} \cdot \mathbf{v} = 0$$
$$\mathbf{u} \times \mathbf{v} = \mathbf{w}$$

6 conditions (cross product contains 3)
$\rightsquigarrow$ rotation matrix has 3 degrees of freedom

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.

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



Combining two successive rotations with this representation is not straightforward (in fact does not satisfy the law of vector addition)

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:

$$ij = k, \qquad ji = -k,$$
$$jk = i, \qquad kj = -i,$$
$$ki = j, \qquad ik = -j,$$

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

Expressing rotations in 3D as unit quaternions instead of matrices has some advantages:

- Extracting the angle and axis of rotation is simpler.
- Expression of the rotation matrix in terms of quaternion parameters involves no trigonometric functions
- Simple to combine two individual rotations represented as quaternions using a quaternion product
- More compact than the matrix representation and less susceptible to round-off errors
- Quaternion elements vary continuously over the unit sphere in $\mathbb{R}^4$, as orientation changes, avoiding discontinuous jumps
- Interpolation is more straightforward. See for example slerp.

They must sometimes be re-normalized due to rounding errors, but low computational cost.

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, $\theta^{-1} = -\theta$)

$$\hat{\mathbf{s}} = \begin{bmatrix} 1 \\ i \\ j \\ k \end{bmatrix}^{-1} = \begin{bmatrix} 1 \\ -i \\ -j \\ -k \end{bmatrix}$$

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

In 2D we used $\arctan$ knowing the two vectors. In 3D infinite possibilities: start with a "base" b orientation and find rotation $\mathbf{r}$ through the minimum angle possible so that its local z-axis ($\mathbf{z}_b$) points along the target vector $\mathbf{t}$.

$$\mathbf{r} = \mathbf{z}_b \times \mathbf{t} = (|\mathbf{z}_b||\mathbf{t}| \sin\theta)\mathbf{e}_r = \sin\theta \mathbf{e}_r$$

Since $|\mathbf{e}_r| = 1$ then $\theta = \arcsin|\mathbf{r}|$. Then divide divide $\mathbf{r}$ by θ to get the axis.

Target orientation $\hat{\mathbf{t}}$: turn axis and angle in a quaternion $\hat{\mathbf{r}}$, together with basis quaternion $\hat{\mathbf{b}}$ (commonly $[1\ 0\ 0\ 0]$) and compute:

$$\hat{\mathbf{t}} = \hat{\mathbf{b}}^{-1}\hat{\mathbf{r}} \qquad \text{if } \sin\theta = 0\text{: } \hat{\mathbf{t}} = \begin{cases} +\hat{\mathbf{b}} & \hat{\mathbf{z}}_b = \hat{\mathbf{z}}_t \\ -\hat{\mathbf{b}} & \text{otherwise} \end{cases}$$

Face in 3D


```
class Face3D (Align3D):
    baseOrientation
    target
    # ... Other data is derived from the superclass ...
    def calculateOrientation(vector):
        # Get the base vector by transforming the z-axis by base
        # orientation (this only needs to be done once for each base
        # orientation, so could be cached between calls).
        baseZVector = new Vector(0,0,1) * baseOrientation # rotate vector by quaternion
        if baseZVector == vector:
            return baseOrientation
        if baseZVector == -vector:
            return -baseOrientation

        # Otherwise find the minimum rotation from the base to the target
        change = crossproduct(baseZVector, vector)
        angle = arcsin(change.length())
        axis = change
        axis.normalize()
        return new Quaternion(cos(angle/2), sin(angle/2)*axis.x, sin(angle/2)*axis.y, sin
              (angle/2)*axis.z)

    def getSteering():
        direction = target.position - character.position # character.velocity.normalize()
        if direction.length() == 0: return target
        Align3D.target = explicitTarget
        Align3D.target.orientation = calculateOrientation(direction)
        return Align3D.getSteering()
```

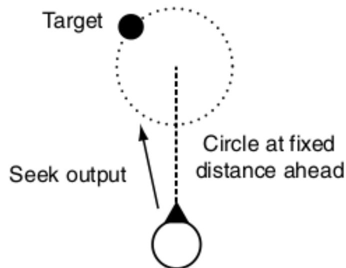
Products between quaternions:

$$\hat{\mathbf{v}} = \hat{\mathbf{q}}\hat{\mathbf{v}}\hat{\mathbf{q}}^* \qquad \hat{\mathbf{v}} = \begin{bmatrix} 0 \\ v_x \\ v_y \\ v_z \end{bmatrix}$$

$$\hat{\mathbf{p}}\hat{\mathbf{q}} = \begin{bmatrix} p_1 q_1 - p_i q_i - p_j q_j - p_k q_k \\ p_1 q_i + p_i q_1 + p_j q_k - p_k q_j \\ p_1 q_j + p_j q_1 - p_i q_k + p_k q_i \\ p_1 q_k + p_k q_1 + p_i q_j - p_j q_i \end{bmatrix}$$

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 $< 1/\sqrt{3}$ 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
  wanderVector # current wander offset orientation
  maxAcceleration # 3D vector
  # ... Other data is derived from the superclass ...
  def getSteering():
    # Update the wander direction
    wanderVector.x += randomBinomial() * wanderRate
    wanderVector.y += randomBinomial() * wanderRate
    wanderVector.z += randomBinomial() * 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
```

Faking Rotation Axes

In aircraft, typically, rolling and pitching occur only with a turn

- bring 3D only into actuators, calculate the best way to trade off pitch, roll, and yaw based on the physical characteristics. Maybe costly.
- add a steering behavior that forces roll whenever there is a rotation
- blending approach with the following orientation values in steering:
  1. keep orientation θ around the up vector as the kinematic orientation.
  2. find the pitch ϕ by looking at the component of the vehicle's velocity in the up direction.
     The output orientation has an angle above the horizon given by:

     $$\phi = \arcsin \frac{\mathbf{v} \cdot \mathbf{u}}{|\mathbf{v}|} \qquad u \text{ is a unit vector in the up direction}$$

  3. find the roll ψ by looking at the vehicle's rotation speed around the up direction.

     $$\psi = \arctan \frac{r}{k} \qquad r \text{ is the rotation, } k \text{ is a constant}$$

```
def getFakeOrientation(kinematic, speedThreshold, rollScale):
  speed = kinematic.velocity.length()
  if speed < speedThreshold:
    if speed == 0:
      return kinematic.orientation
    else:
      fakeBlend = speed / speedThreshold
      kinematicBlend = 1.0 - kinematicBlend
  else:
    fakeBlend = 1.0
    kinematicBlend = 0.0
  yaw = kinematic.orientation # y-axis orientation
  pitch = asin(kinematic.velocity.y / speed) # tilt
  roll = atan2(kinematic.rotation, rollScale) # roll
  result = orientationInDirection(roll, Vector(0,0,1))
  result *= orientationInDirection(pitch, Vector(1,0,0))
  result *= orientationInDirection(yaw, Vector(0,1,0))
  return result

# quaternion for rotation by a given angle around a fixed axis.
def orientationInDirection(angle, axis):
  result = new Quaternion()
  result.r = cos(angle*0.5)
  sinAngle = sin(angle*0.5)
  result.i = axis.x * sinAngle
  result.j = axis.y * sinAngle
  result.k = axis.z * sinAngle
  return result
```

Outline

1. Movement in 3D

2. Pathfinding

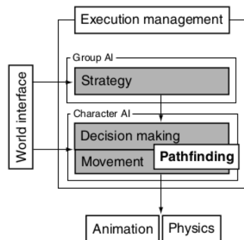
Motivation

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.

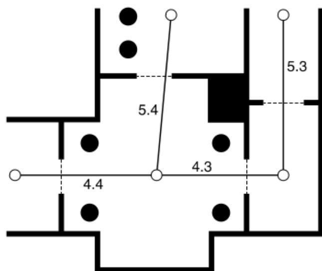
We'd like the route to be sensible and as short or rapid as possible

⇝ 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

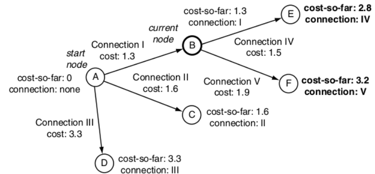
edge/arc: connections, they can be multiple

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

Dijkstra – Uniform cost search

shortest path algorithm from start point to all other points
Invariants:

Processing current node



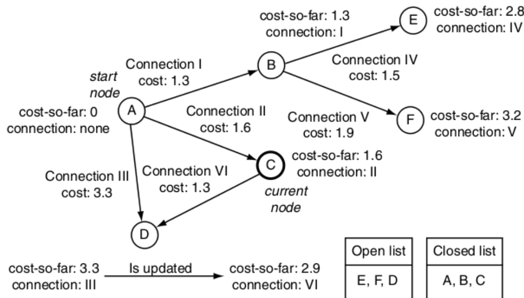
Lists of Nodes
Each node belongs to one of three categories:

- in closed list, having been processed in its own iteration;
- in open list, having been visited from another node, but not yet processed in its own right;
- or it is unvisited.

At each iteration, the algorithm chooses the node from the open list that has the smallest cost-so-far. After processing the node is moved from the open to the closed list.

What if we arrive at a node that instead than unvisited is open or closed?

- if it is higher than the recorded value, don't update the node and don't change what list it is on.
- if the new cost-so-far value is smaller than the node's current cost-so-far, update it with the better value. Move node to the open list. If it was on the closed list (will never be the case), remove it from there.



The algorithm terminates when the open list is empty. Enough when the goal node is the smallest node on the open list (note: not when it is first found).

The path is found by going backward from goal to start

Data structures:

list used to accumulate the final path: not crucial, basic linked list

graph : not critical: adjacency list, best if arcs are stored in contiguous memory, in order to reduce the chance of cache misses when scanning

open and closed lists: critical!

1. push
2. remove
3. extract min
4. find an entry

http://stegua.github.com/blog/2012/09/19/dijkstra/

- $O(nm)$ in space and memory (if $O(1)$ data structures).
- solution includes shortest path to everywhere (wasteful) many fill nodes.

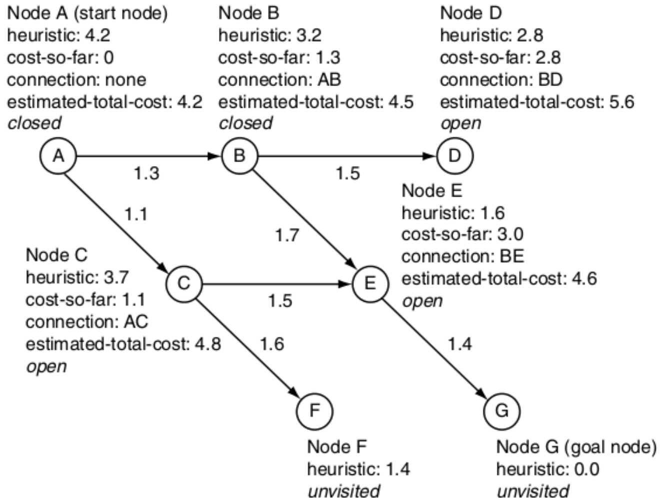
A* search

shortest path algorithm from start point to goal point

Idea: avoid expanding paths that are already expensive,
instead of considering the open node with the lowest cost-so-far value, choose the node that is heuristically *most likely* to lead to the shortest overall path.

Evaluation function $f(n) = g(n) + h(n)$

$g(n)$ = cost-so-far to reach n
$h(n)$ = estimated cost to goal from n
$f(n)$ = estimated total cost of path through n to goal


Node A (start node)
heuristic: 4.2
cost-so-far: 0
connection: none
estimated-total-cost: 4.2
closed
Node B
heuristic: 3.2
cost-so-far: 1.3
connection: AB
estimated-total-cost: 4.5
closed
Node D
heuristic: 2.8
cost-so-far: 2.8
connection: BD
estimated-total-cost: 5.6
open
A
B
D
1.3
1.5
1.1
1.7
Node E
heuristic: 1.6
cost-so-far: 3.0
connection: BE
estimated-total-cost: 4.6
open
Node C
heuristic: 3.7
cost-so-far: 1.1
connection: AC
estimated-total-cost: 4.8
open
C
E
1.5
1.6
1.4
F
G
Node F
heuristic: 1.4
unvisited
Node G (goal node)
heuristic: 0.0
unvisited

Termination

When the node in the open list with the smallest cost-so-far (not) has a cost-so-far value greater than the cost of the path we found to the goal. (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:

If the heuristic is:

- admissible, i.e., $h(n) \leq h^*(n)$ where $h^*(n)$ is the *true* cost from n ($h(n) \geq 0$, so $h(G) = 0$ for any goal G)
- consistent (triangular inequality holds, see later)

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

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

Note:

- consistent $\Rightarrow$ admissible
- if the graph is a tree, then admissible is enough.