DM842

Computer Game Programming: AI

Lecture 3
Movement Behaviors

Christian Kudahl

Department of Mathematics & Computer Science
University of Southern Denmark

Outline

Summery

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

Outline

Combined Steering

- First pathfinding then Seek
- in fact, due to collision avoidance, more complicated: need for combination of steering behaviors
- combining steering output and pipeline architectures
  1. blending: executes all the steering behaviors and combines their results using some set of weights or priorities.
     is the final movement feasible?
  2. arbitration: select one or more steering to have full control.

Outline

Weighted Blending

- crowd of rioting characters, want a mass movement where they stay by the others, while keeping a safe distance.
- blending: arriving at the center of mass of the group and separation from nearby characters.
- weighted linear sum of acceleration (weights do not need to sum to 1) — if above maximum, set to max Acceleration
- research on evolving weights using genetic algorithms or neural networks. Results not encouraging.

Weighted Blending

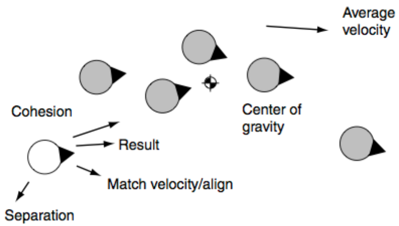
```
class BlendedSteering:
   behaviors # list of behavior and weight
   maxAcceleration
   maxRotation

   def getSteering():
      steering = new Steering()
      for behavior in behaviors:
         steering += behavior.weight * behavior.getSteering()
    if steering.linear.length()> maxAcceleration:
      steering.linear.normalize()
      steering.linear *= maxAcceleration
    if steering.angular > maxRotation:
      steering.angular.normalize()
      steering.angular *=maxRotation
      return steering
```

Flocking and Swarming

Flocking of boids (simulated birds) or herding of animals is obtained by weighted blend of (Craig Reynolds):

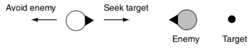
- separation, move away from boids that are too close
- cohesion, move towards the center of mass of the flock
- alignment and velocity matching, move in the same direction and at the same velocity as the flock

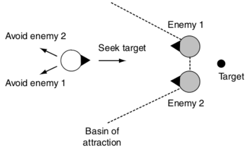
Equal weights but order of importance would be separation, cohesion, alignment. Also radius cut-off for only neighbors.



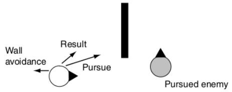
Problems with Blending

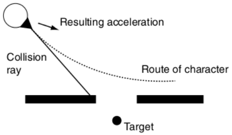
- blending works in sparse outdoor environments, in more constrained settings hard to debug
- conflicting behaviors: unstable and stable equilibrium



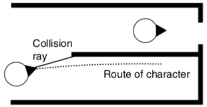


- obstacles and narrow passages





- nearsightedness, solved by pathfinding



Outline

Priorities

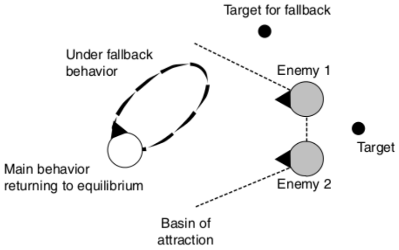
seek and evade always produce an acceleration
collision avoidance, separation, and arrive may suggest no acceleration. But when they do, it should not be ignored or diluted!

- priority-based system: behaviors are arranged in groups with regular blending weights. Groups are then placed in priority order.
- if the total result of a group is small ($\leq \epsilon$ parameter), then it is ignored and the next group is considered. Otherwise the acceleration is applied immediately and other groups ignored.
- Example: pursuing character with 3 groups: collision avoidance, separation and pursuit

```
class PrioritySteering:
   groups # list of BlendedSteering instances
   epsilon
   def getSteering():
      for group in groups:
         steering = group.getSteering()
         if steering.linear.length() > epsilon or abs(steering.angular) > epsilon:
            return steering
   return steering
```

Advantages

- adding a group (eg, wandering) can help to break unstable equilibria, but probably not stable ones

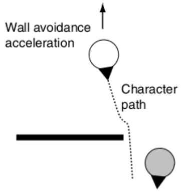


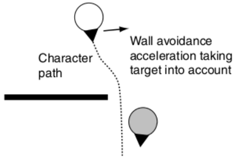
- Variable priorities:
  compute the steering of each group, sort the steering in decreasing order, select the first.
  (adds computation time)

Outline

Cooperative Arbitration

- Blending has stability problems
- Priorities may lead to abrupt changes





- Trend towards cooperation among different behaviors. That is, the response of one steering behavior becomes context aware
  $\rightsquigarrow$ adds complexity.

Cooperative Steering is handled with

- decision making techniques, ie, decision trees and state machines
- pipeline techniques

Outline

Steering Pipeline

Four stages in the pipeline:

- targeters work out where the movement goal is
  channels: positional target, orientation target, velocity target, and rotation target
  not "away from"
- decomposers provide sub-goals that lead to the main goal,
  like pathfinding, sequence of decomposers on increasing level of detail
- constraints limit the way a character can achieve a goal,
  represent moving or static obstacles
  gets the path from actuators
  determines sub-goals by finding the point of closest approach and projecting it out so that we miss the obstacle by far enough
  may require looping and deadlock resolution (call to planning or pathfinding)
- actuator limits the physical movement capabilities of a specific character.
  may decide which channels of subgoals take priority and which are eliminated

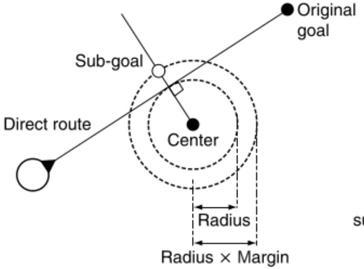
```
class SteeringPipeline:
   targeters
   decomposers
   constraints
   actuator
   constraintSteps
   deadlock
   kinematic # current kinematic data for the character

   def getSteering():
      goal # top level goal
      for targeter in targeters:
         goal.updateChannels(targeter.getGoal(kinematic))
      for decomposer in decomposers:
         goal = decomposer.decompose(kinematic, goal)
      validPath = false
      for i in 0..constraintSteps:
         path = actuator.getPath(kinematic, goal)
         for constraint in constraints:
            if constraint.isViolated(path):
               goal = constraint.suggest(path, kinematic, goal)
               break continue
         return actuator.output(path, kinematic, goal)
      return deadlock.getSteering()
```

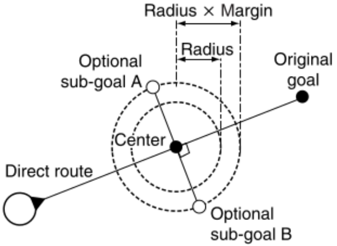
Compromise between pathfinding and more simple and fast movement behaviors.
If computationally costly needs to be spread through more than one frame.

Paths implementations:

- series of line segments, giving point-to-point movement information. Good for characters that can turn quickly.
- list of maneuvers, such as "accelerate" or "turn with constant radius." Suitable for complex steering requirements, including race car driving, harder for constraint checking

Obstacle Avoidance





Outline

Motor Control

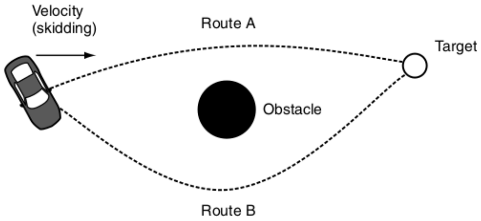
- increasingly, motion is being controlled by physics simulation: actuators
- Steering algorithms send movement requests to physics engine and actuators check feasibility
- eventually actuators must change the suggestion of the steering alg in order to match animation feats (eg, car turning)

Two ways to implement this:

- output filtering: simply remove all the components of the steering output that cannot be achieved.
  it does not work well where there is a small margin of error in the steering requests.
- capability-sensitive steering: actuators brought within steering (not with combined steering)

Capability-Sensitive Steering

if few actions try them all and choose the best
otherwise, heuristics



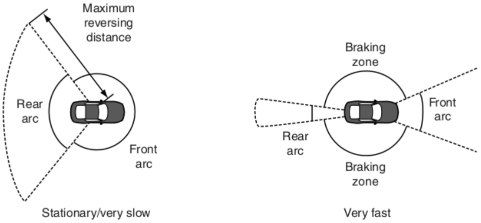
Heuristics

Human characters:

- If stationary or moving very slowly, and at a very small distance from its target, step there directly, even if this involves moving backward or sidestepping.
- If the target is farther away, the character will first turn on the spot to face its target and then move forward to reach it.
- If moving with some speed, and target is within a speed-dependent arc in front of it, then continue to move forward but add a rotational component (still using the straight line animation – hence some limit to how much rotation)
- If the target is outside its arc, then it will stop moving and change direction on the spot before setting off once more.

Heuristics

Cars and motorbikes



- If stationary, then accelerate.
- If moving and target lies between the two arcs, then brake while turning at the maximum rate that does not cause a skid.
- If target inside the forward arc, then continue moving forward and steer toward it. Move as fast as possible
- If target inside the rearward arc, then accelerate backward and steer toward it.

Outline

Predicting Physics

Needs for physics simulation:

- current position of a ball and move to intercept the ball
- character correctly calculating the best way to throw a ball so that it reaches a teammate who is running.
- where to stay to minimize chance of being hit by a grenade
- shoot accurately, and respond to incoming fire
- predicting trajectories

Outline

Firing Solution

Projectile trajectory

$$\boldsymbol{p}_t = \boldsymbol{p}_0 + \boldsymbol{u} s_m t + \frac{\boldsymbol{g} t^2}{2}$$

s_m muzzle velocity (speed at which the projectile left the weapon)
$\boldsymbol{u}$ is the direction the weapon was fired (normalized vector)
$g = -9.81 \mathrm{m/s^2}$ but in games about the double is used

Predicting a Landing Spot

$$t_i = \frac{-u_y s_m \pm \sqrt{u_y^2 s_m^2 - 2 g_y (p_{y0} - p_{yt})}}{g_y} \qquad \boldsymbol{p}_y = \begin{bmatrix} p_{x0} + u_x s_m t_i \\ p_{y0} \\ p_{z0} + u_z s_m t_i \end{bmatrix}$$

Firing Solution

Given a firing point $\boldsymbol{S}$ and s_m (may be varied too, eg, with grenades) and a target point $\boldsymbol{E}$, we want to know the firing direction $\boldsymbol{u}$, $|\boldsymbol{u}| = 1$.

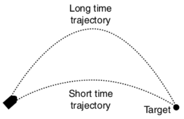
$$\begin{aligned} E_x &= S_x + u_x s_m t_i + \frac{1}{2} g_x t_i^2 \\ E_y &= S_y + u_y s_m t_i + \frac{1}{2} g_y t_i^2 \\ E_z &= S_z + u_z s_m t_i + \frac{1}{2} g_z t_i^2 \\ 1 &= u_x^2 + u_y^2 + u_z^2 \end{aligned}$$

four eq. in four unknowns, leads to:

$$|\boldsymbol{g}|^2 t_i^4 - 4(\boldsymbol{g} \cdot \boldsymbol{\Delta} + s_m^2) t_i^2 + 4|\boldsymbol{\Delta}|^2 = 0, \qquad \boldsymbol{\Delta} = \boldsymbol{E} - \boldsymbol{S}$$

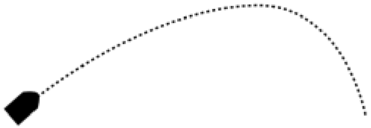
solve in t, and get two solutions

$$\boldsymbol{u} = \frac{2\boldsymbol{\Delta} - \boldsymbol{g} t_i^2}{2 s_m t_i}$$



typically choose the lower one

Drag: Air Resistance



The path is not anymore a parabola

Highly simplified: the drag force can be described as: $D = -kv - cv^2$, v velocity of projectile and k, c are parameters. Equation of motion is non linear differential equation

$$\boldsymbol{p}''_t = \boldsymbol{g} - k\boldsymbol{p}'_t - c\boldsymbol{p}'_t|\boldsymbol{p}'_t|$$

iterative method via simulation, alternatively, removing final term we can solve

$$\boldsymbol{p}_t = \frac{\boldsymbol{g}t - \boldsymbol{A}e^{-kt}}{k} + \boldsymbol{B}, \quad \boldsymbol{A} = s_m\boldsymbol{u} - \frac{\boldsymbol{g}}{k}, \quad \boldsymbol{B} = \boldsymbol{p}_0 - \frac{\boldsymbol{A}}{k}$$

Iterative Targeting Technique

We wish to solve the firing solution controlling its accuracy to make sure we can hit small or large objects correctly.

- start with a tentative direction
- simulate real projectile motion by a physics system
- continue guessing until within a radius from target

To guess one can use the equations without drag or the one with drag simplified.

Binary search: find a tentative upper or lower bound, then the opposite bound and continue by binary search.

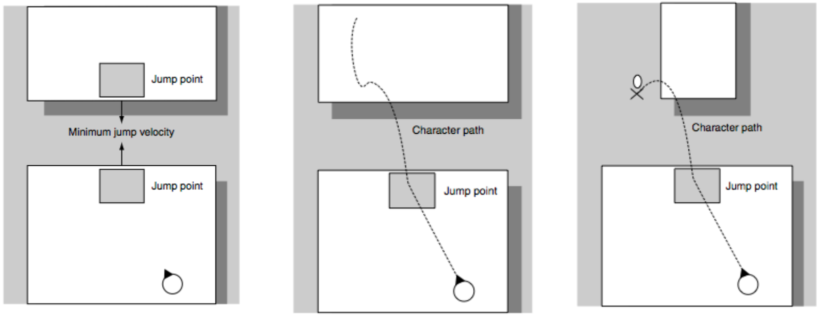
Only possible when the physics engine can easily set up isolated simulations (ie, different from the current game world) and it is fast enough

Outline

Jumping

- Jumping between platforms...
  steering controller needs to check that the character is moving at i) correct speed ii) correct direction iii) jump action is executed at the right moment. $\rightsquigarrow$ Rather complex!

- Simpler support leaves to the designer the choice of jump points and minimal component velocity in the right direction



To carry out the jump the character undergoes the following steps:

1. decide to make a jump by the pathfinding system or a simple steering behavior
2. recognize which jump by pathfinding system or by steering behaviour with lookahead.
3. once found the jump point to use: velocity matching steering behaviour to bring the character into the jump point with correct velocity and direction.
4. once on the jump point, launch a jump action, the game engine will do the rest.

Problem resolutions:

- designer incorporates more information into the jump point data, ie, restrictions on approach velocities (bug prone)
- designer puts jump points such that the AI cannot fail
- incorporate in pathfinding
- landing pads + characters use trajectory prediction to calculate the velocity required to jump from jump point to landing pad + velocity matching

  v_y is upwards velocity of jump and it is given, we wish to find v_x, v_z. Three equations in three unknowns

$$\begin{aligned} E_x &= S_x + v_x t \\ E_y &= S_y + v_y t + \frac{1}{2} g_y t^2 \\ E_z &= S_z + v_z t \end{aligned} \qquad \begin{aligned} t &= \frac{-v_y \pm \sqrt{2g(E_y - S_y) + v_y^2}}{g} \\ v_x &= \frac{E_x - S_x}{t} \\ v_z &= \frac{E_z - S_z}{t} \end{aligned}$$

```
class Jump (VelocityMatch):
  jumpPoint
  canAchieve = False
  maxSpeed
  maxYVelocity
  def getSteering():
    if not target:
      target = calculateTarget()
    if not canAchieve:
      # hence no steering towards target
      return new SteeringOutput()
    if character.position.near(target.
        position) and
      character.velocity.near(target.
          velocity):
       # we jump hence no steeering
       scheduleJumpAction()
       return new SteeringOutput()
    return VelocityMatch.getSteering()
```

```
def calculateTarget():
  target = new Kinematic()
  target.position = jumpPoint.
        jumpLocation
  sqrtTerm = sqrt(2*gravity.y*jumpPoint
        .deltaPosition.y +
                 maxYVelocity*
                       maxVelocity)
  time = (maxYVelocity - sqrtTerm) /
        gravity.y # 1st
  if not checkJumpTime(time):
     time = (maxYVelocity + sqrtTerm) /
           gravity.y # 2nd
  checkJumpTime(time)

def checkJumpTime(time):
  vx = jumpPoint.deltaPosition.x / time
  vz = jumpPoint.deltaPosition.z / time
  speedSq = vx*vx + vz*vz
  if speedSq < maxSpeed*maxSpeed:
    target.velocity.x = vx
    target.velocity.z = vz
    canAchieve = true
  return canAchieve
```

Hole Fillers

Another approach:

- jump area detector
- character leads towards them with a steering opposite to wall avoidance: move towards them at full speed
- when the character enters in the area it executes the jump
- more flexibility in jumping point
- no control on the landing point

Outline

Coordinate Movement

Individuals can

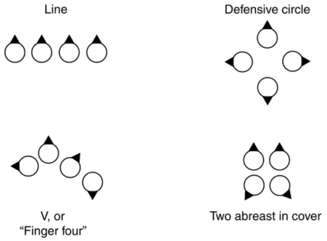
1. make decisions as a whole and move in a prescribed, coordinated group (top down) or
2. make decisions that complement each other (bottom up)

Coordinate Movement

Top down approach:

Formation: a set of locations where a character can be positioned. One location is the leader position.

Formation motion is the movement of a group of characters retaining group organization



Fixed Formations

- The leader moves independently from formation
- the others follow with no need for kinematics or steering:

$$\begin{array}{ll} \vec{p}'_s = \vec{p}_l + \vec{\Theta}_l \vec{p}_s & \text{position,}\Theta\text{orientation matrix} \\ \theta_s = \theta_l + \theta_s & \text{orientation} \end{array}$$

- but leader needs to take care of the size of the formation when moving

Scalable Formations: slots computed by a size-dependent function

Emergent Formations

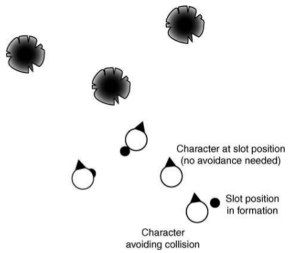
- each character has its own steering system using the arrive behavior.
- each agent selects as target one of the others agents in the formation (eg, V formation)
- the formation emerges from the individual rules of each character, like in flocking
- characters can react individually
- it may be hard to design rules for the desired shape

Combined Fixed and Emergent

Two-level formation steering:

- First level: fixed formation (with a leader that moves it)
- Second level: characters move autonomously avoiding collisions and targetting locations with an arrive behaviour

- actually no need for a leader, the formation moves alone around an anchor point (eg, center of mass of slots)
- steering of anchor points must look at formation, speed moderated if agents not in their slots.



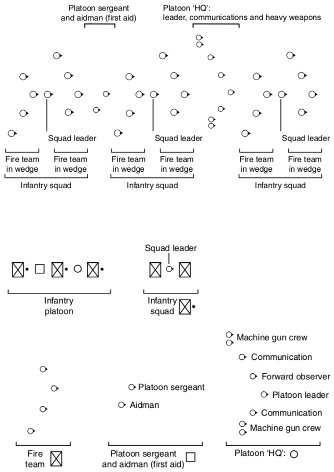
offset to move a small distance ahead of the center of mass

$\boldsymbol{p}_a = \boldsymbol{p}_c + k_{\text{offset}} \boldsymbol{v}_c$

$\boldsymbol{p}_c$ position of center of mass of chars.
$\boldsymbol{v}_c$ velocity of venter of mass

Formations of formations

Anchor point of one formation tries to stay in a slot position of another formation
wedge (V) formation + column formation



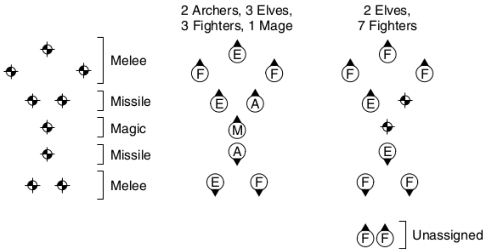
Slot Roles and Assignments

Problems:

- slots may have roles that cannot be occupied by whatever character, eg, leader slots (hard roles)
- there may be more than one agent for each role
- each character may have one or more roles that it can fulfill

May end up in an infeasible situation in which characters are left stranded with nowhere to go.

Simplification: use soft slots with a slot cost for each character



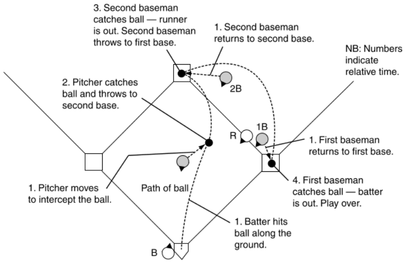
Slot Assignment

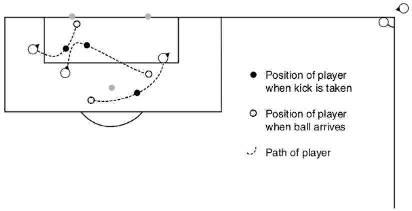
- Brute force, ie, all slot assignments, is not practicable
- assignment problem by Hungarian method in $O(n^3)$ but generalized assignment problem is NP-hard
- heuristic:
  1. sort characters highly constrained first and flexible characters last, ie in increasing order of $\sum_{j\in A(i)} 1/(1+c_{ij})$, c_{ij} is slot cost for agent i, $A(i)$ is feasible slots for i.
     c_{ij} can include distance.
  2. assign the agents considered in the formed order to the best free slot.

  Even this can be too slow and must be split over several frames.

Dynamic Slots and Plays

- Formations that change shape over time, eg, in sport games


3. Second baseman catches ball — runner is out. Second baseman throws to first base.
1. Second baseman returns to second base.
NB: Numbers indicate relative time.
2. Pitcher catches ball and throws to second base.
2B
1B
R
1. First baseman returns to first base.
4. First baseman catches ball — batter is out. Play over.
1. Pitcher moves to intercept the ball.
Path of ball
1. Batter hits ball along the ground.
B

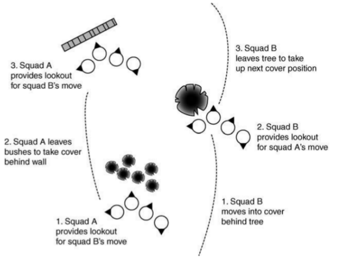

● Position of player when kick is taken
○ Position of player when ball arrives
Path of player

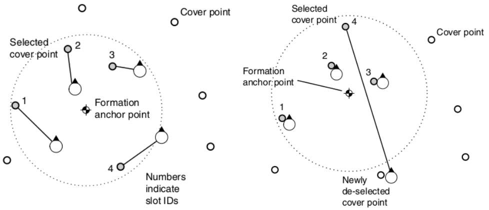
- changes of patterns can be jumps (arrive behaviour of characters will take care) or smooth

- typically no need for more than one level

Tactical Movement

Another application of dynamic formation: approximation of bounding overwatch. Formation moves in a predictable sequence between whatever cover is near to the characters.


3. Squad A provides lookout for squad B's move
2. Squad A leaves bushes to take cover behind wall
1. Squad A provides lookout for squad B's move
3. Squad B leaves tree to take up next cover position
2. Squad B provides lookout for squad A's move
1. Squad B moves into cover behind tree



cover points are in the environment rather than geometrically determined.

Summary

- Predicting Physics
- Firing Solutions
- Jumping
- Coordinated Movement