DM810

Computer Game Programming II: AI

Lecture 4
Movement

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

Outline

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

Firing Solution

Projectile trajectory

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

s_m muzzle velocity (speed at which the projectile left the weapon)
u_t is the direction the weapon was fired
$g = -9.81\mathrm{ms}^{-1}$ but in games about the double is used

Predicting a Landing Spot

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

Firing Solution

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

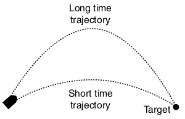
$$\begin{aligned} E_x &= S_x + u_x s_m t_i + \frac{1}{2} g_x t_i^2 \\ E_y &= S_y + u_y s_m t_i + \frac{1}{2} g_y t_i^2 \\ E_z &= S_z + u_z s_m t_i + \frac{1}{2} g_z t_i^2 \\ 1 &= u_x^2 + u_y^2 + u_z^2 \end{aligned}$$

four eq. in four unknowns, leads to:

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

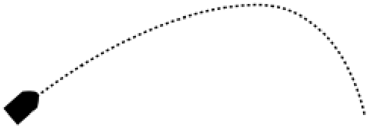
solve in t, and get two solutions

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



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

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

iterative method via simulation, alternatively, removing second 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 that can easily set up isolated simulations (ie, different from the current game world) and it is fast enough

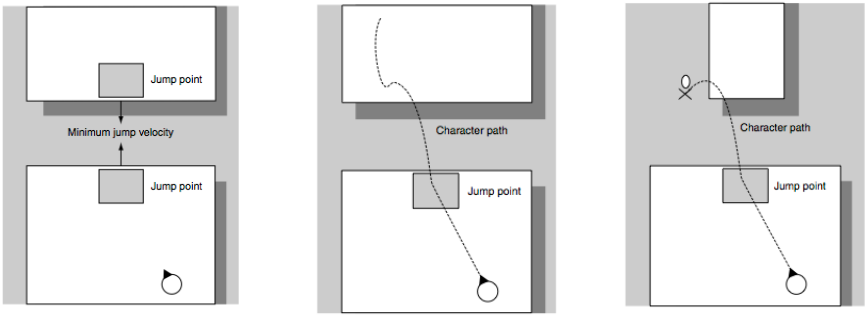
Moving characters: simplifying assumption constant velocity and direction

Outline

Jumping

- Jumping between a 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

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

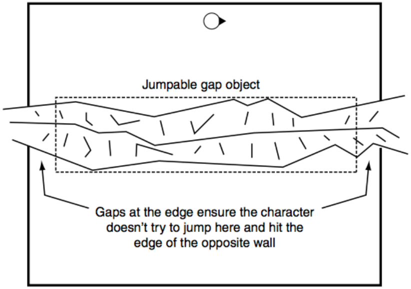
```
class Jump (VelocityMatch):
  jumpPoint
  canAchieve = False
  maxSpeed
  maxYVelocity
  def getSteering():
    if not target:
      target = calculateTarget()
    if not canAchieve:
      # hence no steering towards target
      return new SteeringOutput()
    if character.position.near(target.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

- jump detector area
- character leads towards them with a mechanism opposite to well avoidance
- when the character enters in the area it jumps
- more flexibility in jumping point
- no control on the landing point


Jumpable gap object
Gaps at the edge ensure the character doesn't try to jump here and hit the edge of the opposite wall

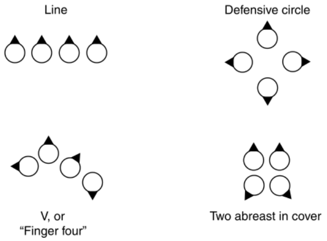
Outline

Coordinate Movement

Individuals can

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

Formation motion is the movement of a group of characters retaining group organization (under 2)
Formation: a set of locations where a character can be positioned. One location is the leader position.



Fixed Formations

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

$$p_s = p_l + s$$
$$\omega_s = \omega_l + \omega_s$$

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

Scalable Formations

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

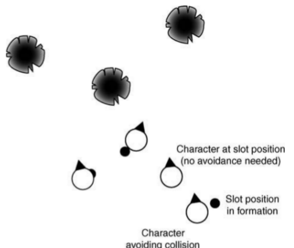
Two-level formation steering:

First level:

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

Second level:

- actually no need for a leader, the formation moves alone around an anchor point
- steering of anchor points can be simplified, only important obstacles to consider, but speed moderated if agents not in their slots.



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

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

$\boldsymbol{p}_c$ center of mass of chars

$\boldsymbol{p}'_{s_i} = \boldsymbol{p}_{s_i} - \boldsymbol{p}_c$

(similarly for velocity and rotation)

```
def updateSlots():
  anchor = getAnchorPoint()
  orientationMatrix = anchor.orientation.asMatrix()
  for i in 0..slotAssignments.length():
    relativeLoc = pattern.getSlotLocation(slotAssignments[i].slotNumber)
    location = new Static()
    location.position = relativeLoc.position * orientationMatrix + anchor.position
    location.orientation = anchor.orientation + relativeLoc.orientation
    location.position -= driftOffset.position
    location.orientation -= driftOffset.orientation
    slotAssignments[i].character.setTarget(location)
```

Example

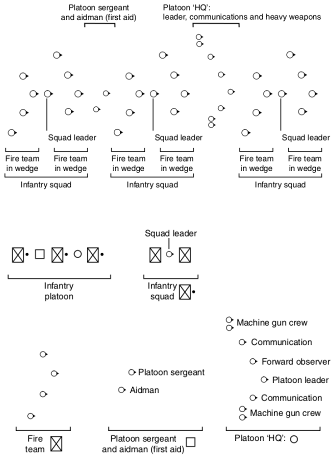
```
class DefensiveCirclePattern:
  characterRadius
  def calculateNumberOfSlots(assignments):
    filledSlots = 0
    for assignment in assignments:
      if assignment.slotNumber >=
           maxSlotNumber:
        filledSlots = assignment.slotNumber
    numberOfSlots = filledSlots + 1
    return numberOfSlots

  def getDriftOffset(assignments):
    center = new Static() # center of mass
    for assignment in assignments:
      location = getSlotLocation(assignment
           .slotNumber)
      center.position += location.position
      center.orientation += location.
           orientation
    numberOfAssignments = assignments.
         length()
    center.position /= numberOfAssignments
    center.orientation /=
         numberOfAssignments
    return center
```

```
  def getSlotLocation(slotNumber):
    angleAroundCircle = slotNumber /
         numberOfSlots * PI * 2
    # The radius depends on the radius of
         the character,
    # and the number of characters in the
         circle:
    # we want there to be no gap between
         character's shoulders.
    radius = characterRadius / sin(PI /
         numberOfSlots)
    # Create a location, and fill its
         components based
    # on the angle around circle.
    location = new Static()
    location.position.x = radius * cos(
         angleAroundCircle)
    location.position.z = radius * sin(
         angleAroundCircle)
    # The characters should be facing out
    location.orientation =
         angleAroundCircle
    return location
```

Formations of formations

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



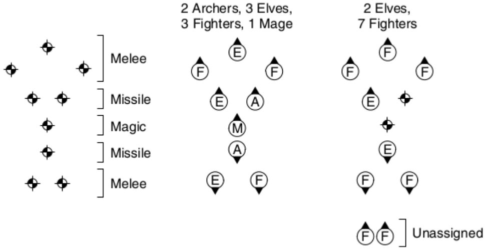
Slot Roles and Assignments

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



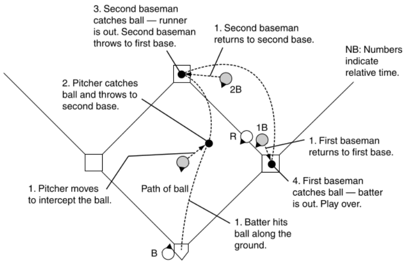
Slot Assignment

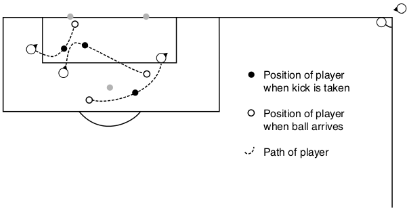
- Brute force, ie, all slot assignments, is not practicable
- assignment problem by Hungarian method in $O(n^3)$ but generalized assignment problem is NP-hard
- heuristic:
  1. sort characters highly constrained first and flexible characters last, ie in increasing order of $\sum_{j \in A(i)} 1/(1 + c_{ij})$, 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

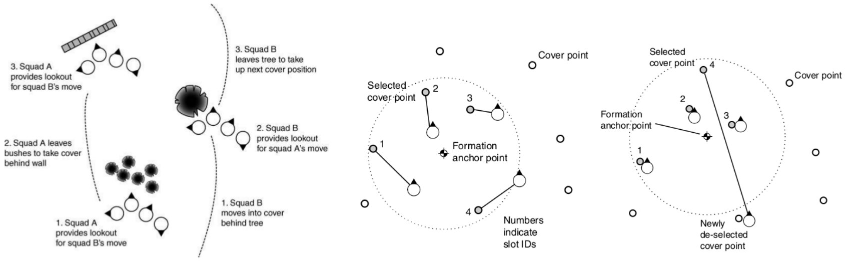

3. Second baseman catches ball — runner is out. Second baseman throws to first base.
1. Second baseman returns to second base.
NB: Numbers indicate relative time.
2. Pitcher catches ball and throws to second base.
2B
1B
R
1. First baseman returns to first base.
1. Pitcher moves to intercept the ball.
Path of ball
4. First baseman catches ball — batter is out. Play over.
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 (hence no need for drift)

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
3. Squad B leaves tree to take up next cover position
2. Squad B provides lookout for squad A's move
2. Squad A leaves bushes to take cover behind wall
1. Squad B moves into cover behind tree
1. Squad A provides lookout for squad B's move
Cover point
Selected cover point
1
2
3
4
Formation anchor point
Numbers indicate slot IDs
Selected cover point
Cover point
Formation anchor point
Newly de-selected cover point

cover points are in the environment rather than geometrically determined.

Outline

Motor Control

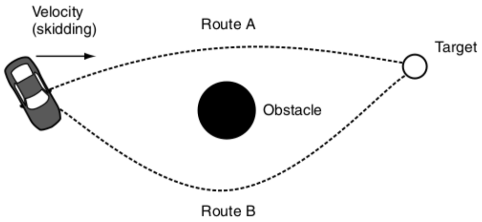
- increasingly, motion is being controlled by physics simulation: actuators
- Steering algorithms send movement requests to physics engine and actuators check feasibility
- eventually actuators must change the suggestion of the steering alg in order to match animation feats (eg, car turning)

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



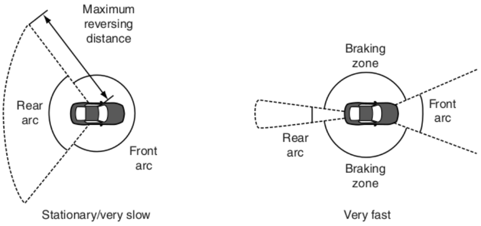
Heuristics

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.

$\rightsquigarrow$ hard to parametrize