DM810

Computer Game Programming II: AI

Lecture 3
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

Outline

1. Delegated Steering
    - Pursue and Evade
    - Face
    - Looking Where You Are Going
    - Wander
    - Path Following
    - Separation
    - Collision Avoidance
    - Obstacle and Wall Avoidance
2. Combined Steering
    - Blending
    - Priorities
    - Cooperative Arbitration
    - Steering Pipeline

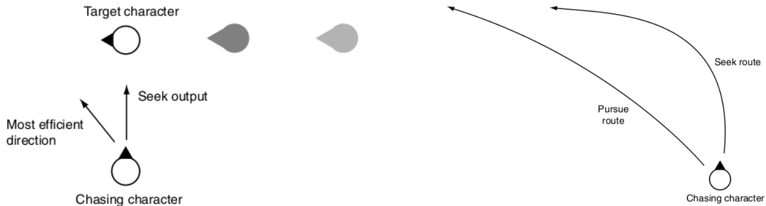
Outline

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

2. Combined Steering
   - Blending
   - Priorities
   - Cooperative Arbitration
   - Steering Pipeline

Pursue and Evade

So far we chased based on position, but if target is far away it would look awkward:



- need to predict where it will be at some time in the future.
- Craig Reynolds's original approach is simple: we assume the target will continue moving with the same velocity it currently has.
- new position used for std seek behavior
- use max time parameter to limit the prediction

Pursue and Evade

```
class Pursue (Seek): # inherited from Seek
    maxPrediction # time limit
    target
    # ... Other data is derived from the superclass ...
    def getSteering():
        direction = target.position - character.position
        distance = direction.length()
        speed = character.velocity.length()
        if speed <= distance / maxPrediction:
            prediction = maxPrediction
        else:
            prediction = distance / speed
        Seek.target = explicitTarget
        Seek.target.position += target.velocity * prediction
        return Seek.getSteering()
```

for evade just call `Flee.getSteering()`
if overshooting, then call `Arrive`

Face

Look at target.
Calculates the target orientation first and delegate to Align the rotation

```
class Face (Align):
    target
    # ... Other data is derived from the superclass ...
    def getSteering():
        direction = target.position - character.position
        if direction.length() == 0: return target
        Align.target = explicitTarget
        Align.target.orientation = atan2(-direction.x, direction.z)
        return Align.getSteering()
```

Looking Where You're Going

- We would like the character to face in the direction it is moving
- In the kinematic movement algorithms we set it directly.
- In steering, we can give the character angular acceleration
- similar to Face

```
class LookWhereYoureGoing (Align):
    # ... Other data is derived from the superclass ...
    def getSteering():
        if character.velocity.length() == 0: return
        target.orientation = atan2(-character.velocity.x, character.velocity.z)
        return Align.getSteering()
```

Wander

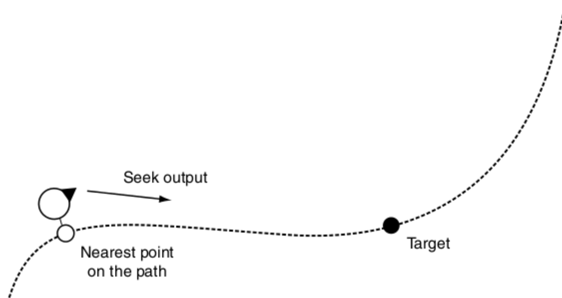
- Move aimlessly around
- In kinematic wander behavior, we perturbed the direction by a random amount. This makes the rotation of the character erratic and twitching.
- add an extra layer, making the orientation of the character indirectly reliant on the random number generator.
- circle around the character on which the target is constrained + Seek
- or circle around the target + face
- or target + look where you're going
- target will twitch on the circle, but the character's orientation will change smoothly.

Wander

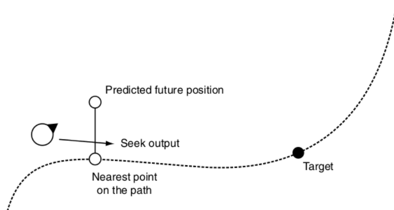
```
class Wander (Face):
   wanderOffset # forward offset of the wander
   wanderRadius
   wanderRate # max rate of change of the orientation
   wanderOrientation # current orientation
   maxAcceleration
   # ... Other data is derived from the superclass ...
   def getSteering():
      wanderOrientation += randomBinomial() * wanderRate
      targetOrientation = wanderOrientation + character.orientation
      target = character.position + wanderOffset * character.orientation.asVector() #
            center of the wander circle
      target += wanderRadius * targetOrientation.asVector()
      steering = Face.getSteering()
      steering.linear = maxAcceleration * character.orientation.asVector() # full
            acceleration towards
      return steering
```

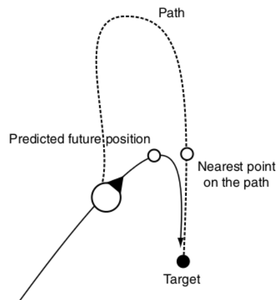
Path Following

- Takes a whole path (line segment or curve splines) as target (eg, a patrol rute). Resulting behavior: move along the path in one direction
- Delegated:
  1. find nearest point along the path. (may be complex)
  2. select a target at a fixed distance along the path.
  3. Seek



- Predictive path following
- smoother behavior but may short-cut the path





Path Following


```
class FollowPath (Seek):
   path # Holds the path to follow
   pathOffset # distance along the path
   currentParam # current position on path

   # ... Other data from superclass ...
   def getSteering():


      currentParam = path.getParam(
           character.position, currentPos)
      targetParam = currentParam +
           pathOffset
      target.position = path.getPosition(
           targetParam)
      return Seek.getSteering()
```

```
class FollowPath (Seek):
   path # Holds the path to follow
   pathOffset # distance along the path
   currentParam # current position on path
   predictTime = 0.1 # prediction time
   # ... Other data from superclass ...
   def getSteering():
      futurePos = character.position +
           character.velocity * predictTime
      currentParam = path.getParam(
           futurePos, currentPos)
      targetParam = currentParam +
           pathOffset
      target.position = path.getPosition(
           targetParam)
      return Seek.getSteering()
```

Separation

- keep the characters from getting too close and being crowded.
- if the behavior detects another character closer than some threshold then evade with strength depending on distance else zero.

linear:

```
strength = maxAcceleration * (threshold - distance) / threshold
```

inverse square:

```
strength = min(decayCoefficient / (distance * distance), maxAcceleration) # k is a constant
```

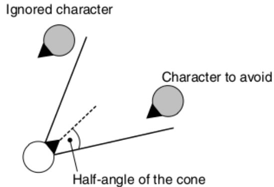
Separation

```
class Separation:
   character # kinematic data
   targets # list of potential targets
   threshold
   decayCoefficient
   maxAcceleration
   def getSteering():
      steering = new Steering
      for target in targets:
         direction = target.position - character.position
         distance = direction.length()
         if distance < threshold:
            strength = min(decayCoefficient / (distance * distance), maxAcceleration)
            direction.normalize()
            steering.linear += strength * direction
      return steering
```

Speed up by spatial data structures: Multi-resolution maps, quad- or octrees, and binary space partition (BSP) trees

Collision Avoidance

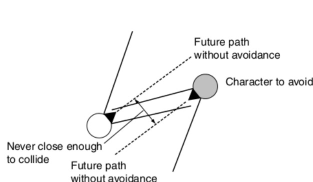
- with large numbers of characters moving around: only engage if the target is within a cone in front of the character.
- average position and speed of all characters in the cone and evade that target. Alternatively, closest character in the cone.

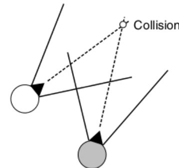


cone checked by dot product

```
if orientation.asVector() . direction >
      coneThreshold:
# do the evasion
else:
# return no steering
```

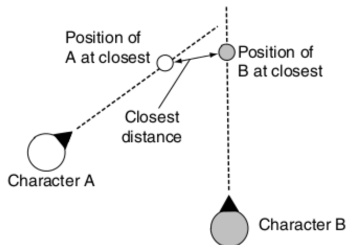
Two problematic situations:





Collision Avoidance

Closest approach: work out the closest predicted distance objects will have on the basis of current speed and compare against some threshold radius.



$$\boldsymbol{p} = \boldsymbol{p}_t - \boldsymbol{p}_c$$
$$\boldsymbol{v} = \boldsymbol{v}_t - \boldsymbol{v}_c$$

$$t = -\frac{\boldsymbol{p} \cdot \boldsymbol{v}}{|\boldsymbol{v}|^2}$$

position at time of closest approach:

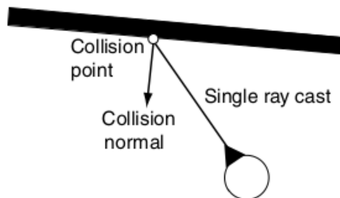
$$\boldsymbol{p}'_c = \boldsymbol{p}_c - \boldsymbol{v}_c t$$
$$\boldsymbol{p}'_t = \boldsymbol{p}_t - \boldsymbol{v}_t t$$

With group of chars: search for the character whose closest approach will occur first and to react to this character only.

```
class CollisionAvoidance:
  character, targets
  maxAcceleration
  radius # collision threshold
  def getSteering():
    shortestTime = infinity
    firstTarget = None # target that will collide first
    firstMinSeparation, firstDistance, firstRelativePos, firstRelativeVel
    for target in targets:
      relativePos = target.position - character.position
      relativeVel = target.velocity - character.velocity
      relativeSpeed = relativeVel.length()
      timeToCollision = (relativePos . relativeVel) / (relativeSpeed * relativeSpeed)
      distance = relativePos.length()
      minSeparation = distance-relativeSpeed*shortestTime
      if minSeparation > 2*radius: continue
      if timeToCollision > 0 and timeToCollision < shortestTime:
        shortestTime = timeToCollision
        firstTarget = target
        firstMinSeparation = minSeparation
        firstDistance = distance
        firstRelativePos = relativePos
        firstRelativeVel = relativeVel
    if not firstTarget: return None
    if firstMinSeparation <= 0 or distance < 2*radius: # colliding
      relativePos = firstTarget.position - character.position
    else:
      relativePos = firstRelativePos + firstRelativeVel * shortestTime
    relativePos.normalize()
    steering.linear = relativePos * maxAcceleration
    return steering
```

Obstacle and Wall Avoidance

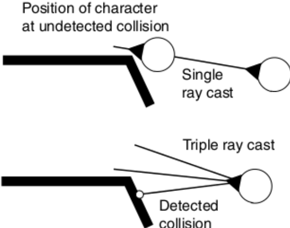
- So far targets are spherical and center of mass
- More complex obstacles, eg, walls, cannot be easily represented in this way.
- cast one or more rays out in the direction of the motion.
- If these rays collide with an obstacle,
  then create a target to avoid the collision, and do seek on this target.
- rays extend to a short distance ahead corresponding to a few seconds of movement.

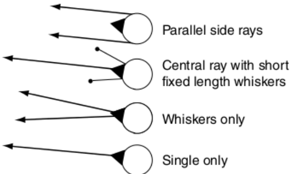


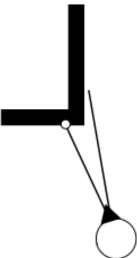
```
class ObstacleAvoidance (Seek):
   collisionDetector
   avoidDistance
   lookahead
   # ... Other data from superclass ...
   def getSteering():
      rayVector = character.velocity
      rayVector.normalize()
      rayVector *= lookahead
      collision = collisionDetector.getCollision(character.position, rayVector)
      if not collision: return None
      target = collision.position + collision.normal * avoidDistance
      return Seek.getSteering()
```

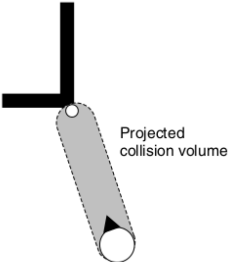
`getCollision` implemented by casting a ray from `position` to `position + moveAmount` and checking for intersections with walls or other obstacles.

Problems and Work Around









Outline

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

2. Combined Steering
    - Blending
    - Priorities
    - Cooperative Arbitration
    - Steering Pipeline

Combined Steering

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

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

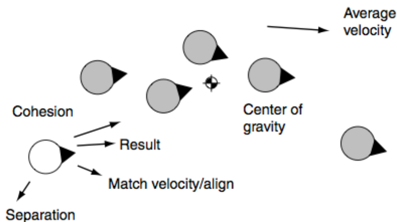
```
class BlendedSteering:
   behaviors # list of behavior and weight
   maxAcceleration
   maxRotation

   def getSteering():
      steering = new Steering()
      for behavior in behaviors:
         steering += behavior.weight * behavior.behavior.getSteering()
      steering.linear = max(steering.linear, maxAcceleration)
      steering.angular = max(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):

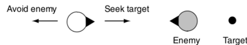
- separation, move away from boids that are too close
- alignment and velocity matching, move in the same direction and at the same velocity as the flock
- cohesion, move toward the center of mass of the flock

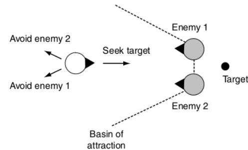
Equal weights but order of importance would be separation, cohesion, alignment. Also radius cut-off for only neighbors.



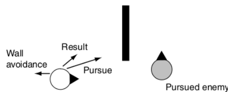
Problems with Blending

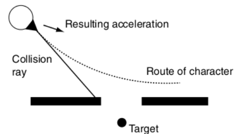
- blending works in sparse outdoor environments, in more constrained settings hard to debug
- conflicting behaviors: unstable and stable equilibrium



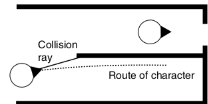


- obstacles and narrow passages





- nearsightedness, solved by pathfinding



Priorities

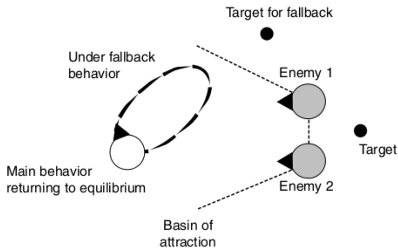
seek and evade always produce an acceleration
collision avoidance, separation, and arrive may suggest no acceleration. But when they do, it should not be ignored or diluted!

- 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
```

Problems

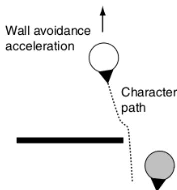
- adding a group (eg, wandering) can help to break unstable equilibria

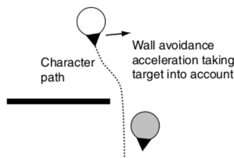


- Variable priorities:
  compute the steering of each group, sort the steering in decreasing order, select the first.
  (adds computation time)

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

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 decided 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:

- series of line segments, giving point-to-point movement information.
- 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