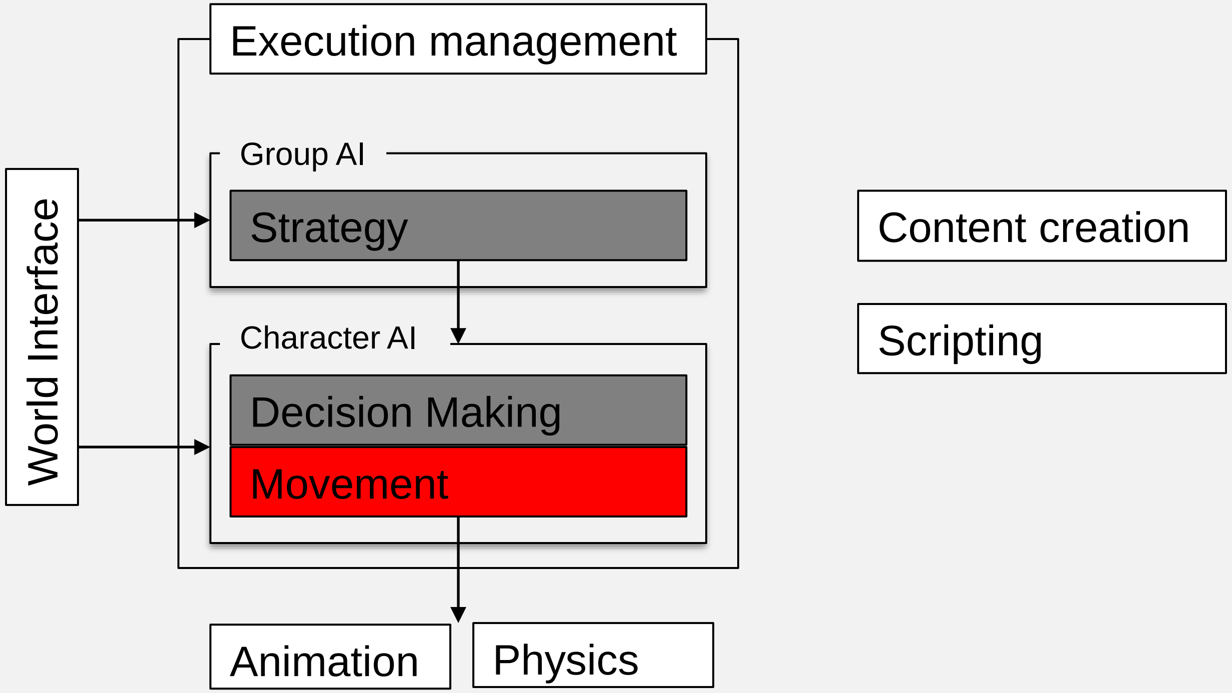
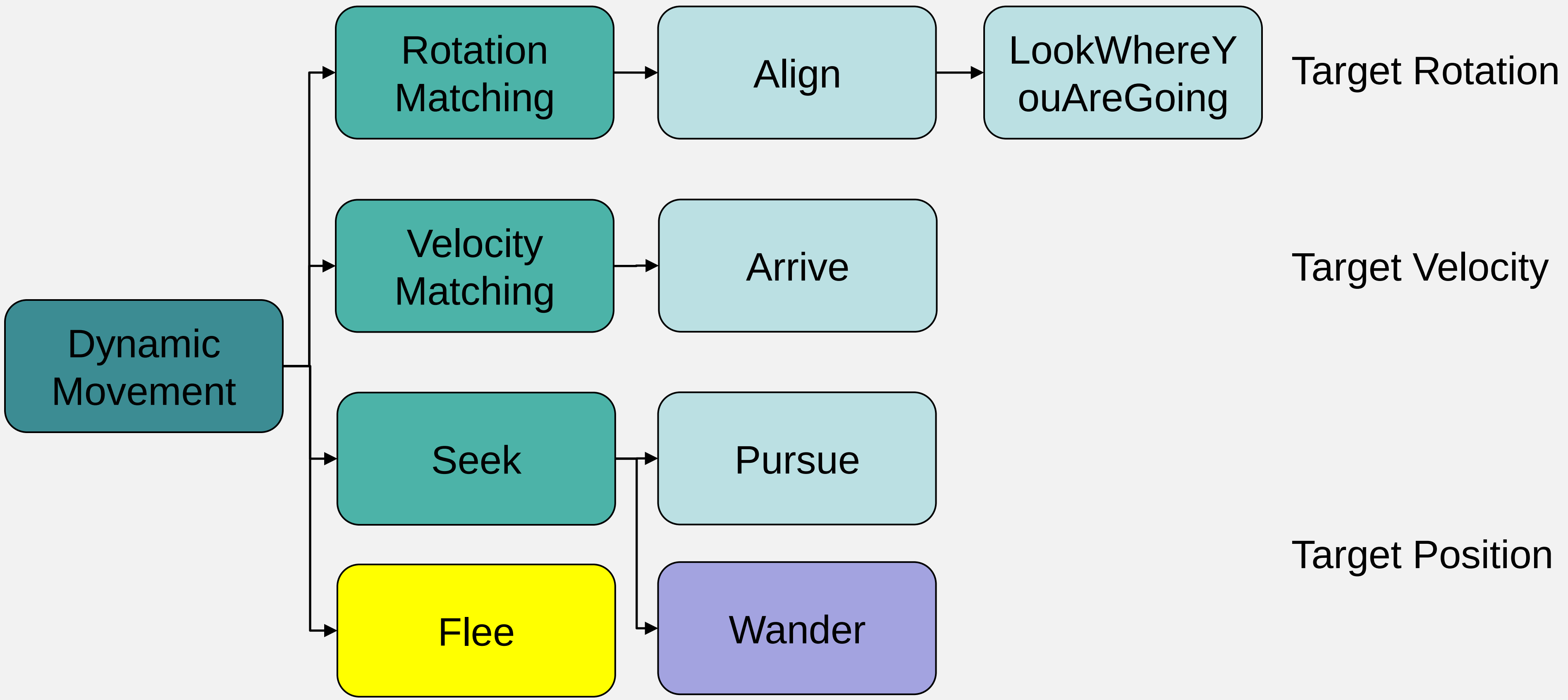
TÉCNICO
LISBOA

Collision Avoidance

Game AI Engine

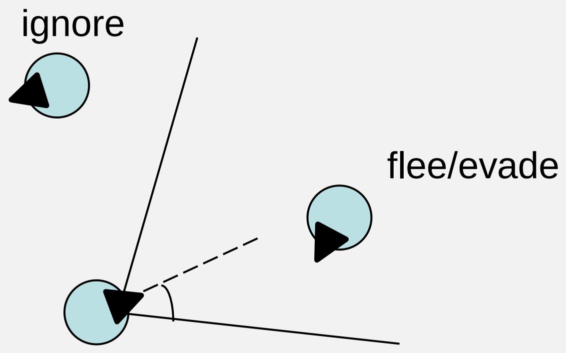


Dynamic Movement Examples



Detecting and avoiding collisions

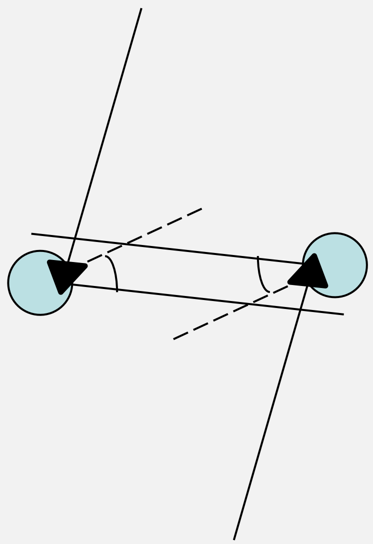
- Simplest approach
  - Use a cone to detect the presence of another character
  - When a character is detected use flee/evade

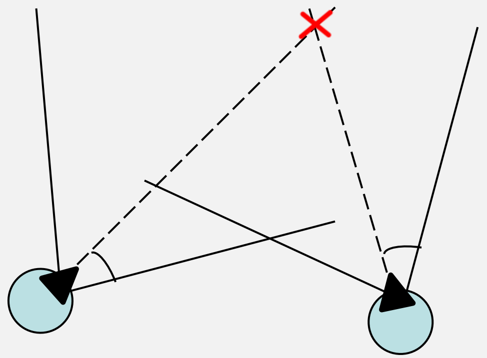


Collision Avoidance

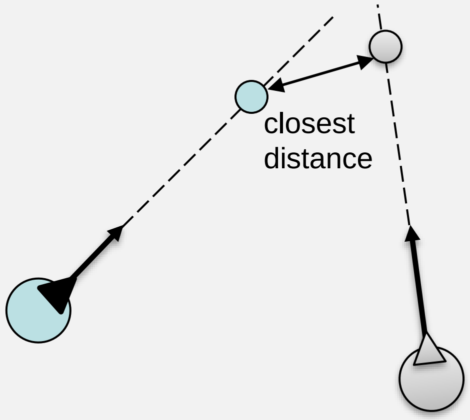
- Problem with the described approach
  - Doesn’t work well with a high number of characters
  - Evade behaviour triggered in situations where it shouldn’t
  - Not activated in some situations where it should

Problems with Naive Approach



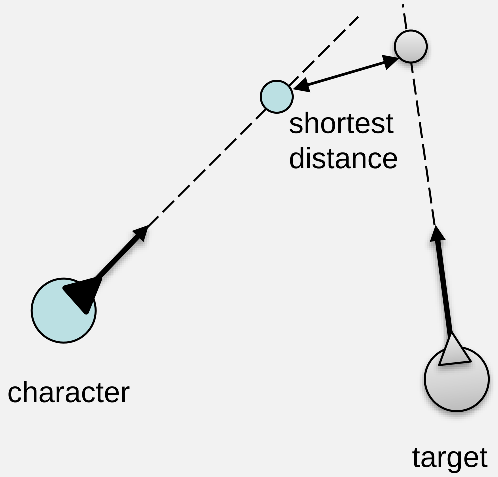


Avoiding Collisions



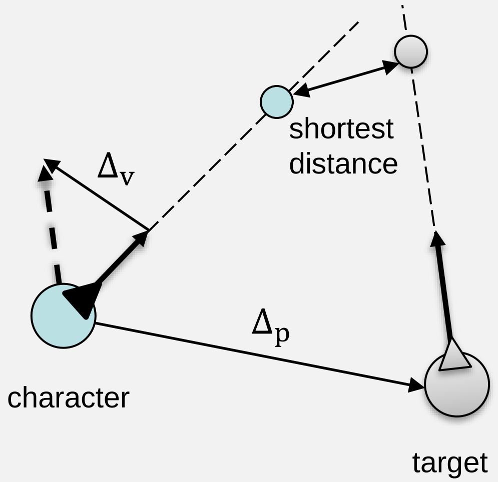
- $t_{closest}$ = ?
- $t_{closest} = \frac{distance?}{velocity?}$

Avoiding Collisions



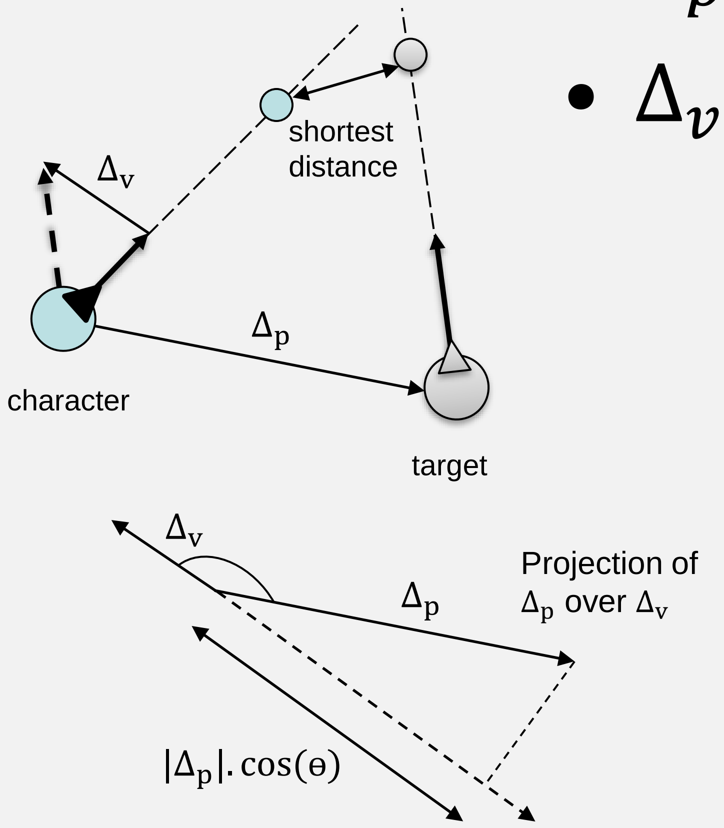
- $\Delta_p = P_{target} - P_{character}$
- $\Delta_v = V_{target} - V_{character}$

Avoiding Collisions



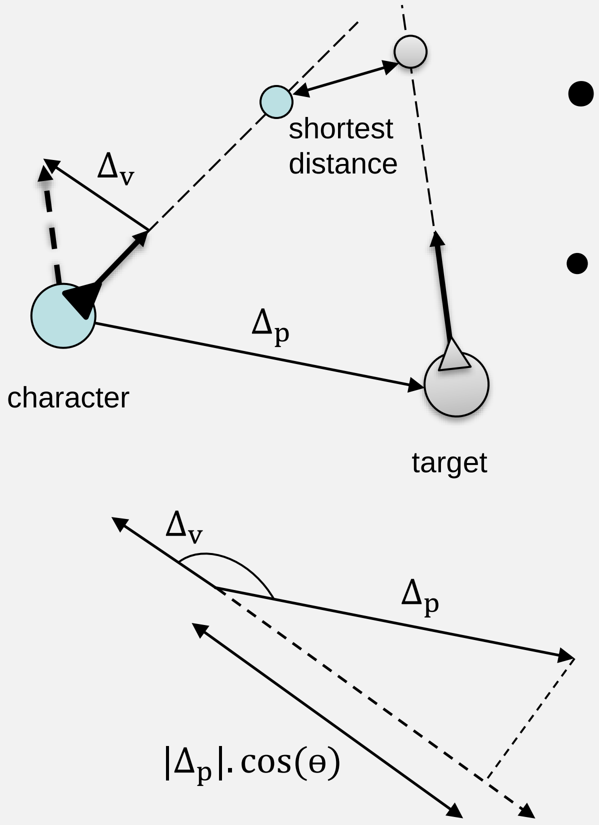
- $\Delta_p = P_{target} - P_{character}$
- $\Delta_v = V_{target} - V_{character}$

Avoiding Collisions



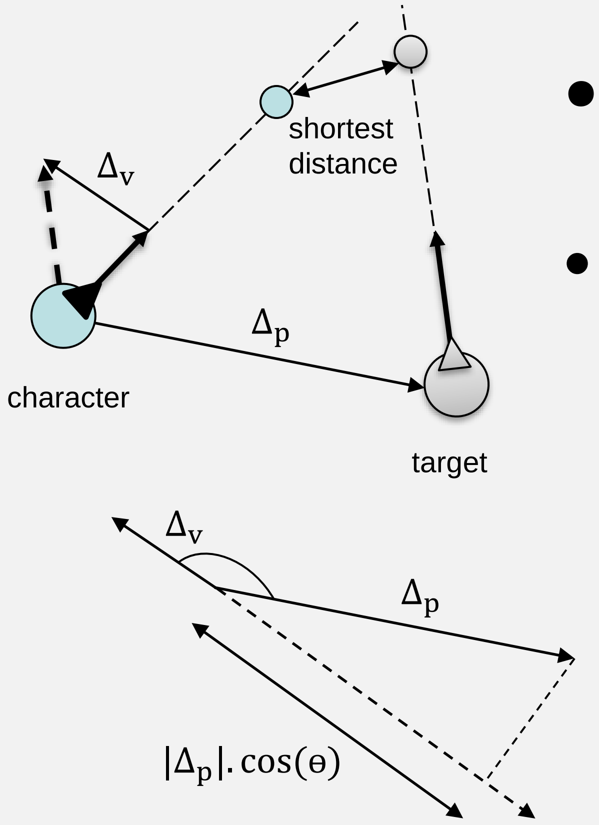
- $\Delta_p = P_{target} - P_{character}$
- $\Delta_v = V_{target} - V_{character}$

Avoiding Collisions

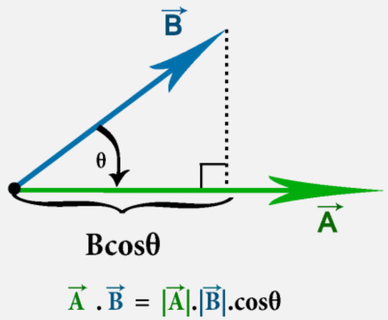


- $\Delta_p = P_{target} - P_{character}$
- $\Delta_v = V_{target} - V_{character}$
- $t_{closest} = -\dfrac{|\Delta_p|.\cos(\theta)}{|\Delta v|}$

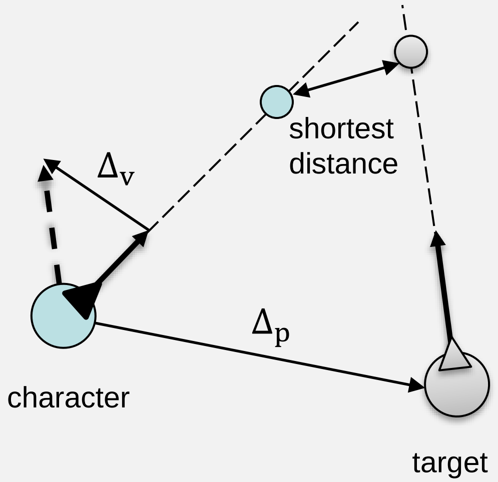
Avoiding Collisions



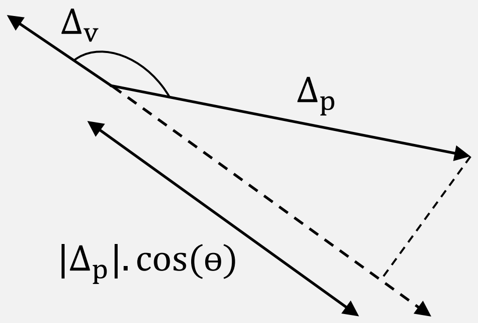
- $\Delta_p = P_{target} - P_{character}$
- $\Delta_v = V_{target} - V_{character}$
- $t_{closest} = -\dfrac{|\Delta_p|.\cos(\theta)}{|\Delta v|}$

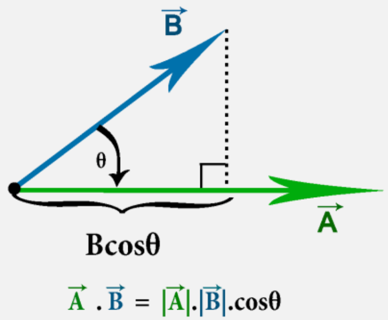


Avoiding Collisions



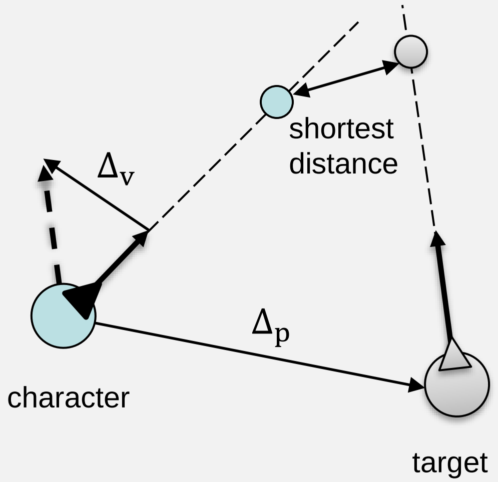
- $\Delta_p = P_{target} - P_{character}$
- $\Delta_v = V_{target} - V_{character}$
- $t_{closest} = - \dfrac{|\Delta_p|.|\Delta v|.\cos(\theta)}{|\Delta v||\Delta v|}$

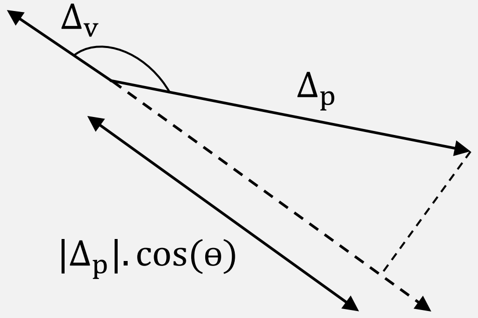


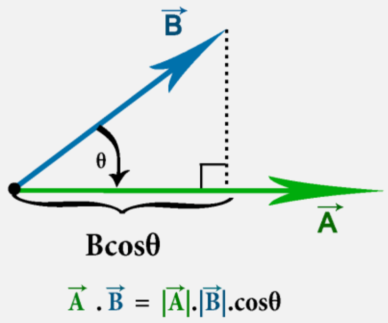


Avoiding Collisions

- $\Delta_p = P_{target} - P_{character}$
- $\Delta_v = V_{target} - V_{character}$
- $t_{closest} = - \frac{\Delta p . \Delta v}{|\Delta v|^2}$







Avoiding Collisions

- Why use the dot product instead of the original formula?

$$t_{closest} = - \frac{|\Delta_p|.\cos(\theta)}{|\Delta v|}$$

Avoiding Collisions

- Why use the dot product instead of the original formula?

$$t_{closest} = - \frac{|\Delta_p|.\cos(\theta)}{|\Delta v|}$$

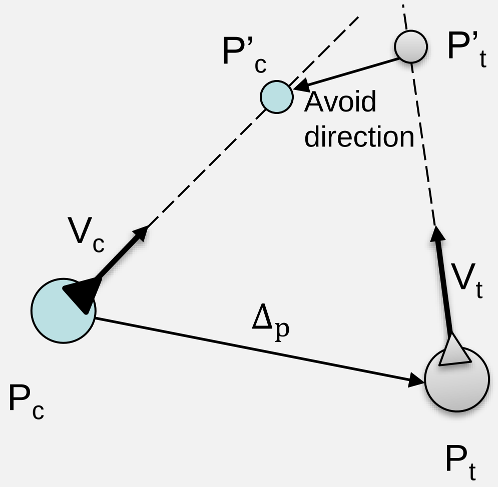
- Avoid calculating 2 norms and a cosine

$$t_{closest} = - \frac{\Delta p . \Delta v}{|\Delta v|^2}$$

→ very efficient

→ faster than $|\Delta v|$

Avoiding Collisions



- $t_{closest} = -\dfrac{\Delta p . \Delta v}{|\Delta v|^2}$

- $P'_t = P_t + V_t . t_{closest}$
- $P'_c = P_c + V_c . t_{closest}$

- $V_{avoid} = P'_c - P'_t$

Character Avoidance

```
Class CharacterAvoidance
    character, target, maxAcceleration, collisionRadius, maxTimeLookAhead

def getMovement ()

    deltaPos = target.position – character.position
    deltaVel = target.velocity – character.velocity
    deltaSqrSpeed = deltaVel.sqrMagnitude()

    if(deltaSqrSpeed==0) return empty movement output

    timeToClosest = -Vector3.Dot(deltaPos,deltaVel)/deltaSqrSpeed

    if(timeToClosest > MaxTimeLookAhead) return empty movement output

    //for efficiency reasons I use the deltas instead of character and target
    futureDeltaPos = deltaPos + deltaVel * timeToClosest
    futureDistance = futureDeltaPos.magnitude()

    if(futureDistance > 2*collisionRadius) return empty movement output

    if(futureDistance <= 0 or deltaPos.magnitude() < 2 * collisionRadius)
        //deals with exact or immediate collisions
        movementOutput.linear = character.position – target.position
    else
        movementOutput.linear = futureDeltaPos*-1

    movementOutput.linear = movementOutput.linear.normalized()* maxAcceleration
    return movementOutput
```

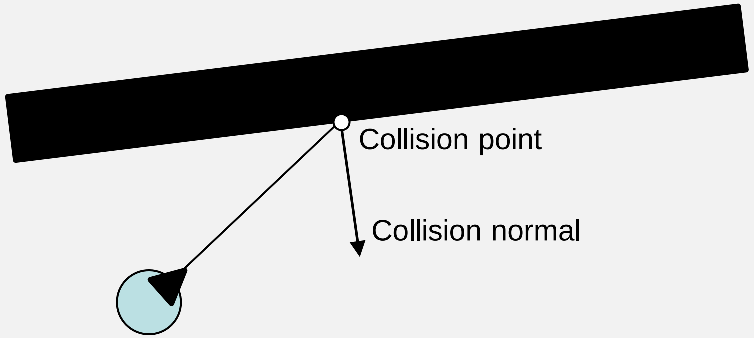
Character Avoidance with Multiple Targets

Dealing with Multiple Targets

```
Class CharacterAvoidance
    character, maxAcceleration, collisionRadius,targets,maxTimeLookAhead

def getMovement ()
    shortestTime = INF

    foreach (t in targets)
        deltaPos = t.position - character.position
        deltaVel = t.velocity - character.velocity
        deltaSqrSpeed = deltaVel.sqrMagnitude()
        if(deltaSqrSpeed==0) break continue
        timeToClosest = -(deltaPos.deltaVel)/deltaSqrSpeed
        if(timeToClosest > maxTimeLookAhead) break continue
        futureDeltaPos = deltaPos + deltaVel * timeToClosest
        futureDistance = futureDeltaPos.magnitude()
        if(futureDistance > 2*collisionRadius) break continue

        if(timeToClosest > 0 and timeToClosest < shortestTime)
            shortestTime = timeToClosest
            closestTarget = t
            closestFutureDistance = futureDistance
            closestFutureDeltaPos = futureDeltaPos
            closestDeltaPos = deltaPos
            closestDeltaVel = deltaVel

    if(shortestTime == INF) return empty MovementOutput
    if(closestFutureDistance <= 0 or closestDeltaPos.magnitude() < 2*collisionRadius)
        avoidanceDirection = character.position - closestTarget.position
    else
        avoidanceDirection = closestFutureDeltaPos*-1
    output = new MovementOutput()
    output.linear = avoidanceDirection.normalized()*maxAcceleration
    return output
```

Avoiding Collisions with walls and other obstacles

- Use rays
- Assume that the game engine will provide us a ray cast collision detection mechanism

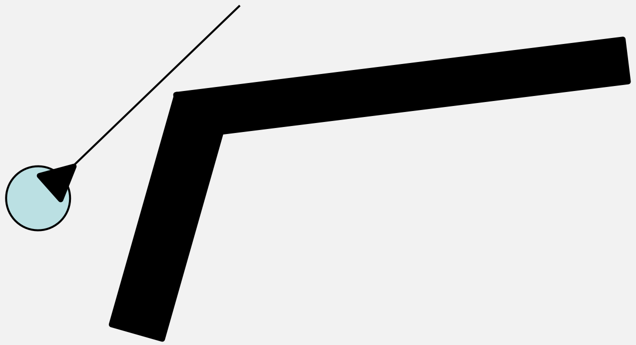


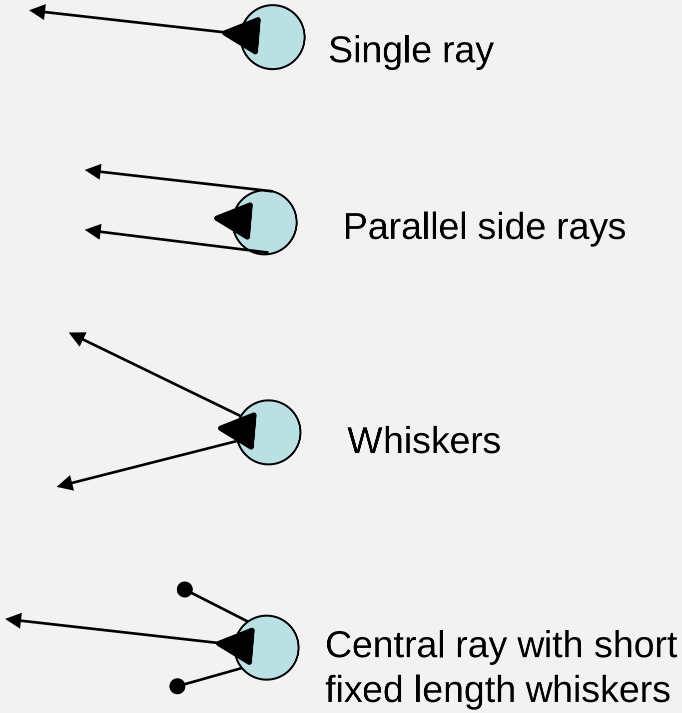
Obstacle Avoidance

```
Class ObstacleAvoidance : Seek
   collisionDetector
   avoidDistance
   lookAhead

def getMovement ()

   rayVector = character.velocity.normalized()*lookAhead

   collision = collisionDetector.getCollision(character.position, rayVector)

   if(collision = null) return empty movement output

   base.target.position = collision.position + collision.normal * avoidDistance

   return base.getMovement()
```

Alternative Ray Configurations

- Detection problems with single ray

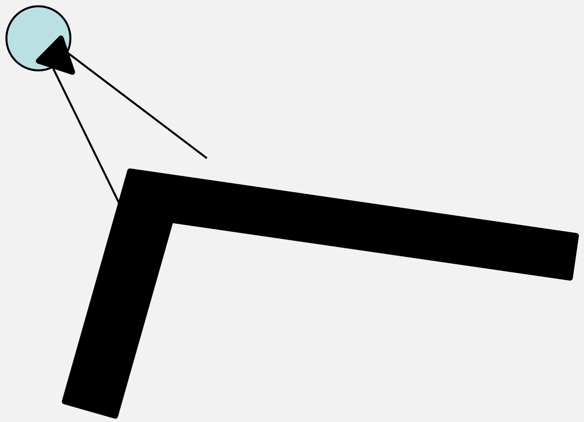




Alternative Ray Configurations

- Best ray configuration varies from scene to scene
  - Central ray with short whiskers usually gets good results
    - However, it might have problems with tight passages
  - Parallel rays
    - Has problems with the corner trap

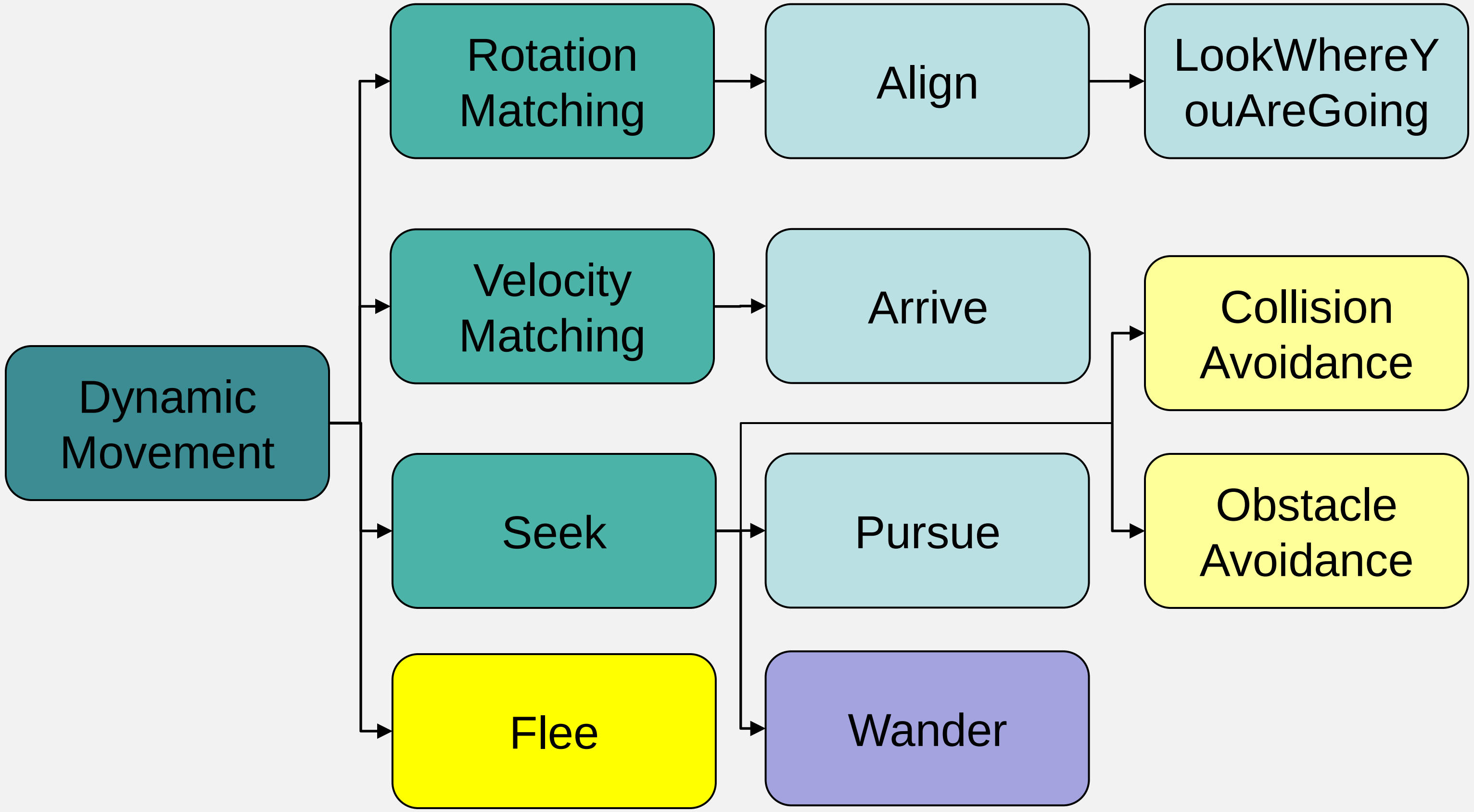
The corner trap



- Solutions
  - Collision detection with volumes
    - Computationally heavy
  - Adaptive fan angles
    - If there are no collisions
      - decrease fan angle
    - If a collision is detected
      - Increase fan angle

Combining Dynamic Movements

Combining Dynamic Movements



Combining Dynamic Movements

- A character needs to be able to combine several movements
  - Reach a desired target
  - Avoid obstacles
  - Turn in a particular direction

Blending

- Simplest form of combining movements
- Each movement returns its individual output as usual
- Individual outputs are then combined using weights.

Blended Movement

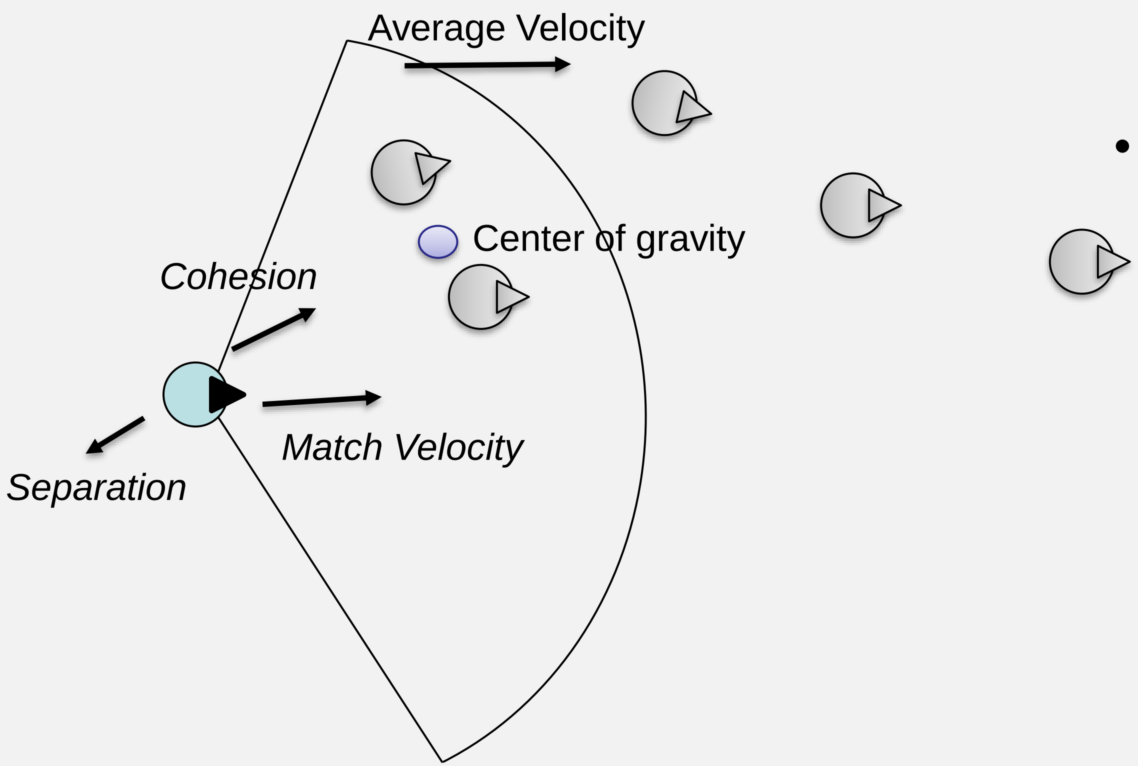
```
Class BlendedMovement : DynamicMovement
   maxRotation
   maxAcceleration
   movements : [DynamicMovementWithWeight] // new class!

def getMovement ()

   totalWeight = 0
   output = new movementOutput

   for (movement in movements)
      tempOutput = movement.getMovement()
      if(tempOutput.magnitude() > 0) //checks both linear and angular!
         totalWeight += movement.weight
         output += movement.weight * tempOutput

   if (totalWeight > 0)
      normalizationFactor = 1/totalWeight
      output *= normalizationFactor

   return output
```

Boids (Craig Reynolds)

- Example of application of blending
  - Used to simulate the movement patterns of flocks of birds

- Blends 3 simpler types of dynamic movement
  - Separation
    - Move away from boids that are too close
  - Match velocity/align
    - Move in the same direction and at the same velocity with the flock
  - Cohesion
    - Move towards the center of mass of the flock

Boids



- Consider just the boids inside a neighborhood circle
- Or a cone

Separation

```
Class Separation : DynamicMovement
    character,flock,separationFactor,radius,maxAccel

def getMovement ()
    output = new movemenOutput

    foreach(boid in flock.members)
        if(boid != character)
            direction = character.position - boid.position
            distance = direction.magnitude()
            if(distance < radius)
                separationStrength =
                           min(separationFactor/(distance^2),maxAccel)
                direction.normalize()
                ouput.linear += direction*separationStrength

    if(output.linear > maxAcceleration)
         output.linear = output.linear.normalize()* maxAcceleration

    return output
```

Cohesion

```
Class Cohesion: DynamicArrive
    flock,radius,fanAngle

def getMovement ()
    massCenter = new vector()
    closeBoids = 0
    foreach(boid in flock.members)

        if(character != boid)
            direction = boid.position – character.position

            if(direction.magnitude() <= radius)
                angle = VectorToOrientation(direction)
                angleDifference = ShortestAngleDifference(character.orientation, angle)

                if(abs(angleDifference) <= fanAngle)
                    massCenter += boid.position
                    closeBoids ++

    if(closeBoids == 0) return empty movement output

    massCenter /= closeBoids
    base.target.position = massCenter

    return base.getMovement()
```

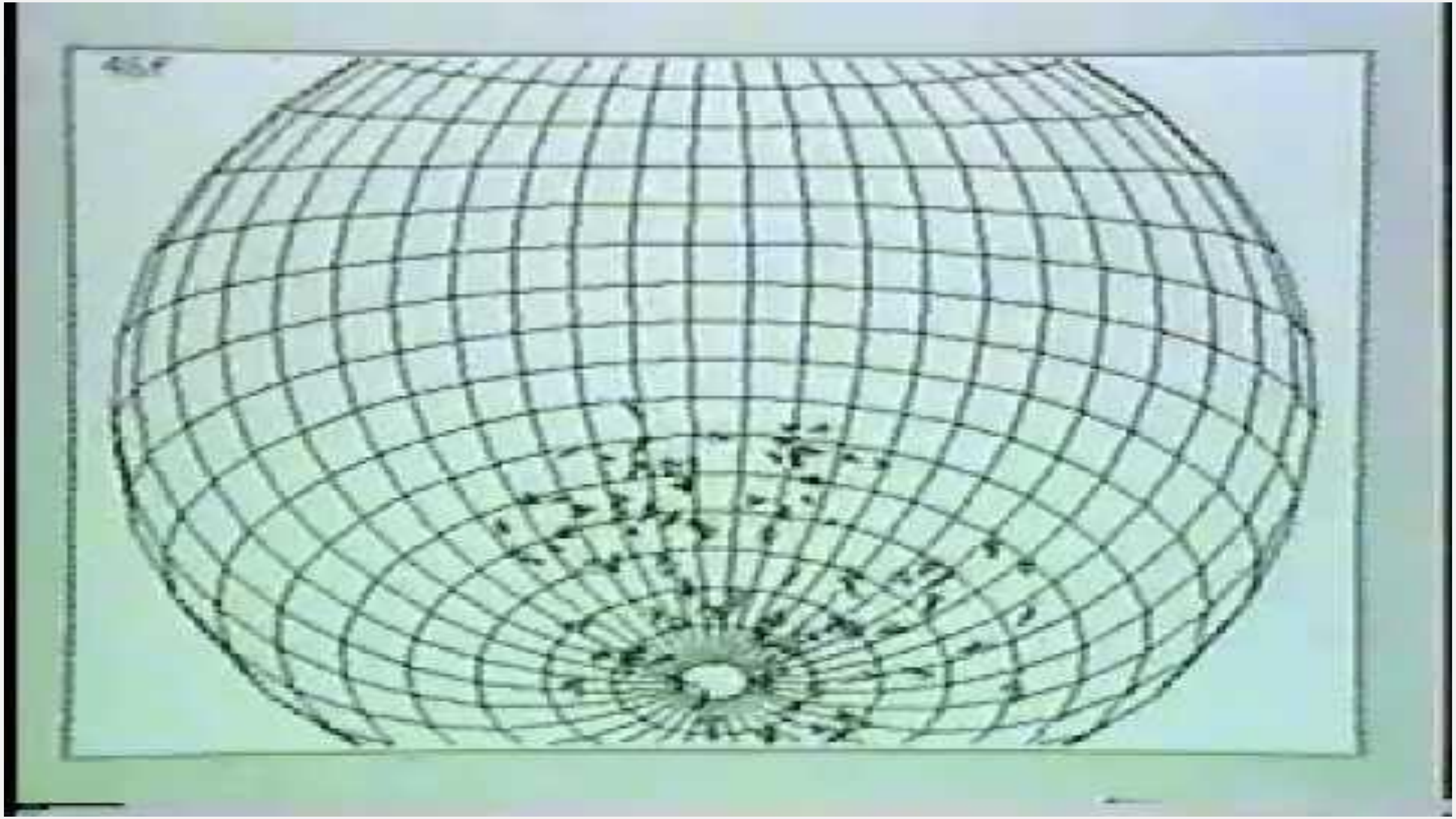
```
def ShortestAngleDifference (source, target)

    delta = target – source
    if(delta > PI) delta-=360
    else if(delta < -PI) delta+=360

    return delta
```

Velocity Matching

```
Class FlockVelocityMatching: DynamicVelocityMatch
    flock,radius,fanAngle

def getMovement ()
    averageVelocity = new vector()
    closeBoids = 0

    foreach(boid in flock.members)
        if(character != boid)
            direction = boid.position - character.position

            if(direction.magnitude() <= radius)
                angle = VectorToOrientation(direction)
                angleDifference = ShortestAngleDifference(character.orientation,angle)

                if(abs(angleDifference) <= fanAngle)
                    averageVelocity += boid.velocity
                    closeBoids ++

    if(closeBoids == 0) return Empty Movement Output

    averageVelocity /= closeBoids
    base.target.velocity = averageVelocity

    return base.getMovement()
```

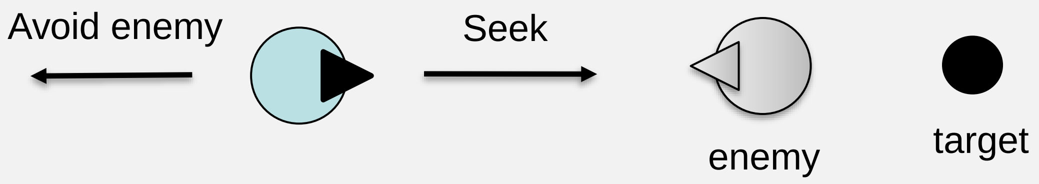
Boids

- Veloc



Problems with blending

- Opposing movements can cause undesired equilibriums
- Environments with lots of obstacles



Priority Movement

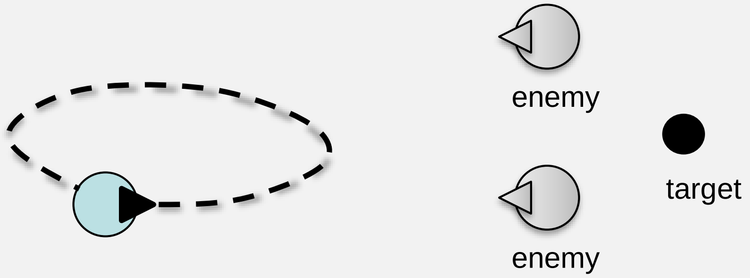
- Uses priorities instead of weights
- Movements divided into groups
  - Which can use blending to combine their output
  - A priority is assigned for each group
- Groups ordered by Priority
  - If a group returns an output > 0, then use that output to move the character
  - Otherwise, check the next group
- A wander movement is normally used as fallback in case all other movements fail

Priority Movement

```
Class PriorityMovement : DynamicMovement
   epsilon
   groups // groups are ordered by priority

def getMovement ()

   for (group in groups)
      movementOutput = group.getMovement()

      if(movementOutput.magnitude() > epsilon)
         return movementOutput

   return movementOutput
```

Problems with Priority

- Solves problems with unstable equilibriums
- But still has problems with more complex stable equilibriums



Problems with Priority

- Problems with complex collision detection scenarios

- Illustrate with example from 1st project

Problems with Priority

Problems with Priority

Improved Collision Detection

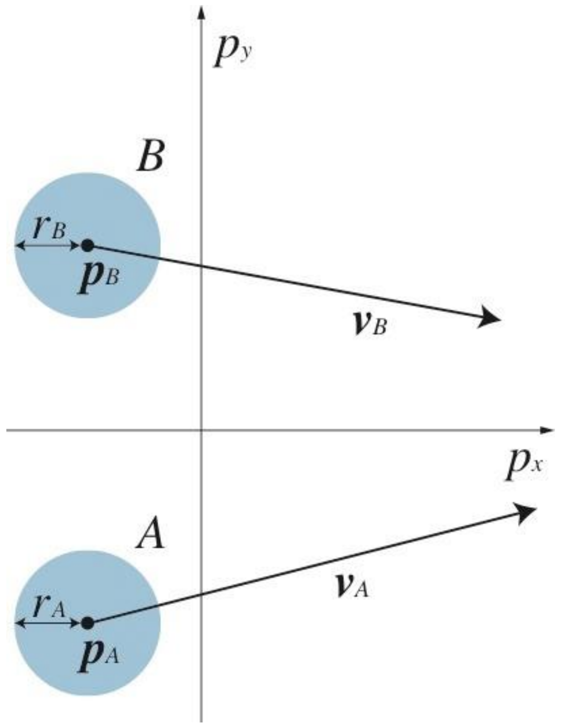
- In scenarios with large numbers of characters

- We need a more complex collision detection algorithm

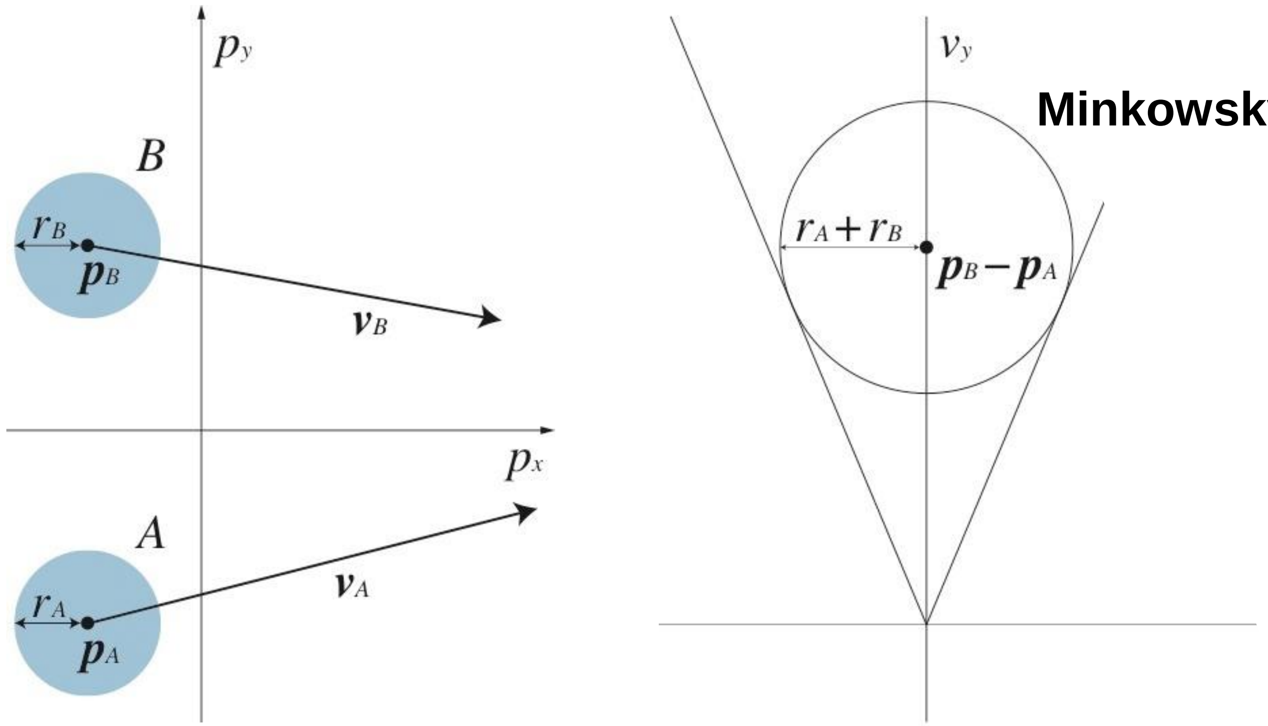
Velocity Obstacle Algorithms

- **Not available in the book**
- Inspired in research with robots
- Based on the concept of Velocity Obstacles (VO)
  - **Reciprocal Velocity Obstacles (RVO) [2008]**
  - Truncated cone Velocity Obstacles (FVO)
    - ClearPath
  - Hybrid Reciprocal Velocity Obstacles (HRVO)
    - HRVO Library (C++)
  - Optimal Reciprocal Collision Avoidance (ORCA)
    - RVO2 Library (C++/C#)

VO Definition

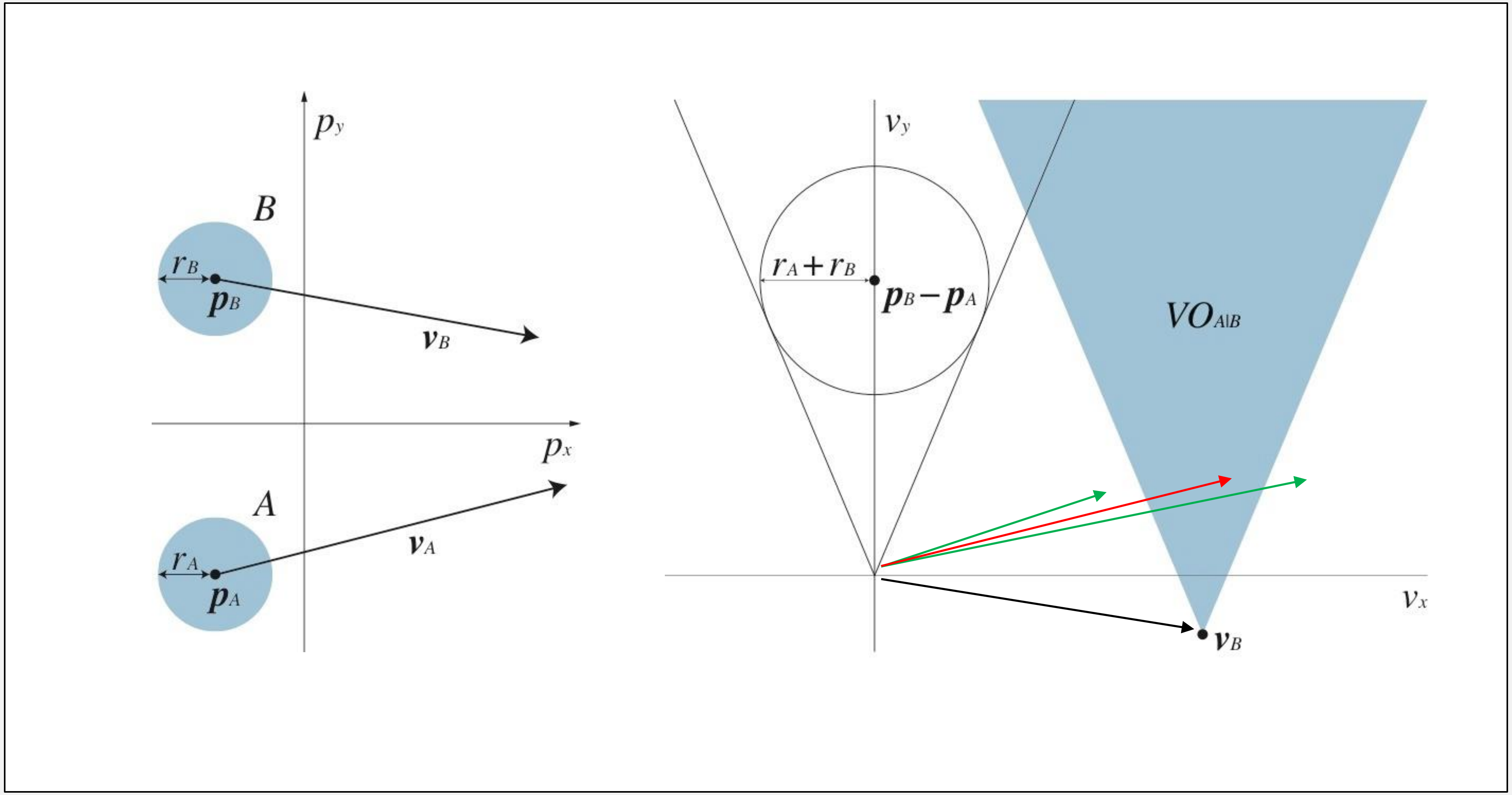


VO Definition



Minkowsky Sum $\mathrm{B} \oplus -A$

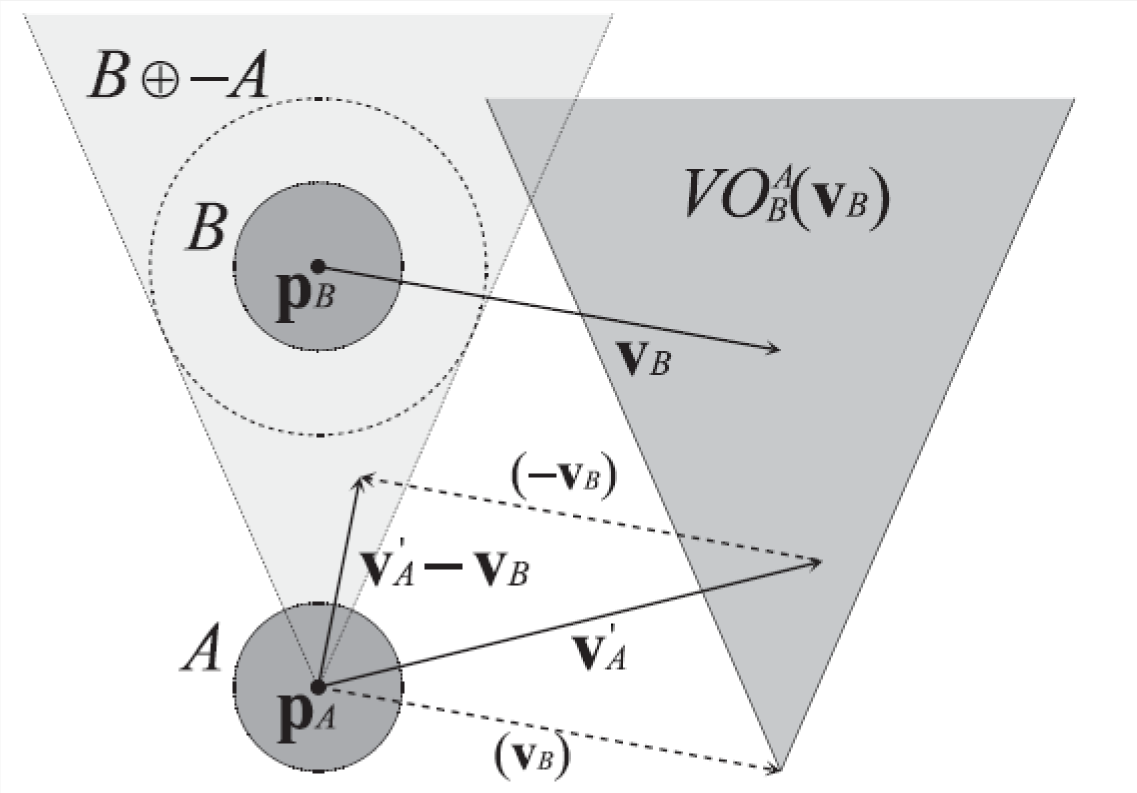
VO Definition



Velocity Obstacle

- Set of all velocities for the agent that will result in a collision with a moving obstacle
  - Assuming the obstacle will maintain a constant velocity
  - If the agente velocity selected
    - is outside of the velocity obstacle, there is no collision
    - is inside the velocity obstacle, then the agent will potentially collide

VO Definition



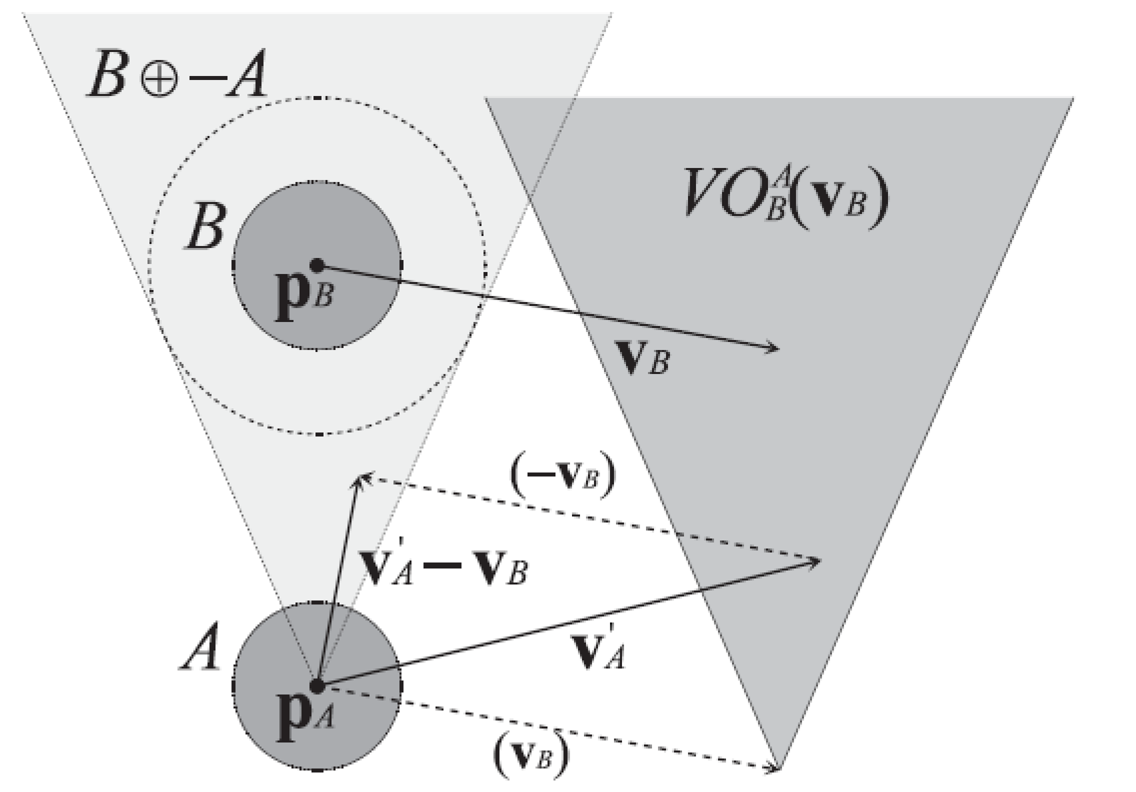
VO Definition

- $\lambda(p, v)$ - Ray starting at p, with direction v
- $\lambda(p, v) = \{\mathrm{p} + t\mathrm{v} | t \geq 0\}$

- $VO_B^A(v_B) = \{v'_A | (\lambda(p_A, v'_A - v_B) \cap B \oplus -A) \neq \emptyset\}$
  - The set of all velocities v'_A such that a ray originating at p_A, with direction $v'_A - v_B$ intersects with $B \oplus -A$

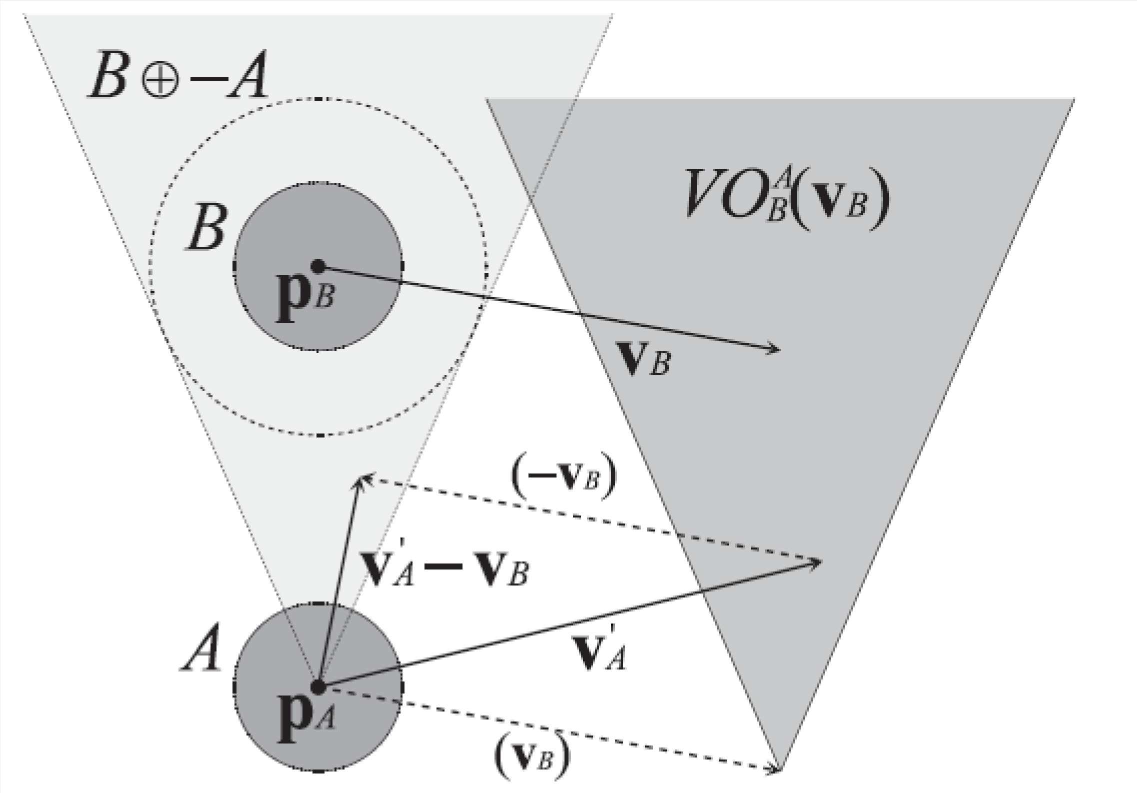
- p_A, p_B - current positions of A and B
- v_B - current velocity of B
- v'_A - new velocity of A to be selected

VO Definition



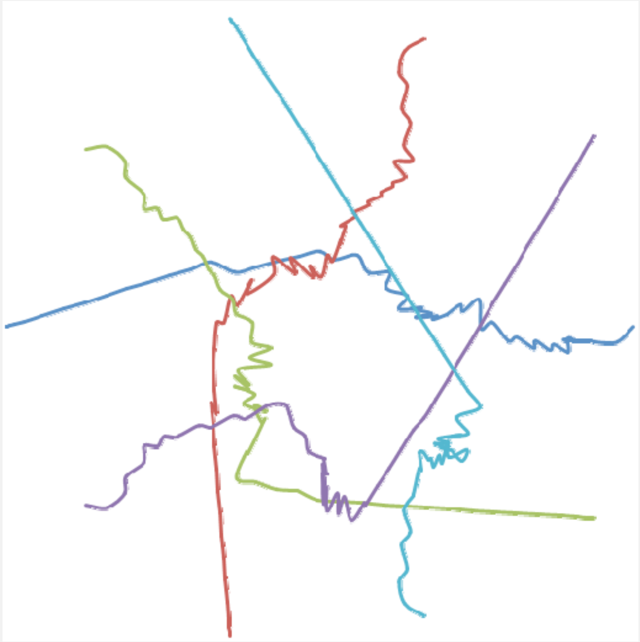
VO Time to Collision

- $\lambda(p_A, v'_A - v_B) = Circle(p_B, r_A + r_B)$



VO Trajectories

- Problem with VO
  - Assumes that target's velocity does not change
  - When modeling groups of characters
    - All of them will proactively try to avoid the collision
    - Using VO will cause many oscillations



Reciprocal Velocity Obstacle (RVO)

- The character will take half the responsibility for avoiding collision
  - **Assume the other character** takes care of the other half

- In order to avoid the collision
  - Select a new velocity that is the average of the current velocity and a velocity outside the VO
    - The one you would have to take without reciprocity