DM842

Computer Game Programming: AI

Lecture 2

Movement Behaviors

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Outline

Outline

Steering – Intro

- movement algorithms that include accelerations (linear and angular)
- present in driving games but always more in all games.
- range of different behaviors obtained by combination of fundamental behaviors: eg. seek and flee, arrive, and align.
- each behavior does a single thing, more complex behaviors obtained by higher level code
- often organized in pairs, behavior and its opposite (eg, seek and flee)

Input: kinematic of the moving character + target information
(moving char in chasing, representation of the geometry of the world in obstacle avoidance, path in path following behavior; group of targets in flocking – move toward the average position of the flock.)

Output: steering, ie, accelerations

Variable Matching

- Match one or more of the elements of the character's kinematic to a single target kinematic (additional properties that control how the matching is performed)
- To avoid incongruencies: individual matching algorithms for each element and then right combination later. (algorithms for combinations resolve conflicts)

Seek and Flee

Seek tries to match the position of the character with the position of the target. Accelerate as much as possible in the direction of the target.

```
struct Kinematic:
    position
    orientation
    velocity
    rotation

    def update(steering, maxSpeed, time):
        position += velocity * time
        orientation += rotation * time
        velocity += steering.linear * time
        orientation += steering.angular * time
        if velocity.length() > maxSpeed:
            velocity.normalize()
            velocity *= maxSpeed # trim back
```

```
struct SteeringOutput
    linear # accleration
    angular # acceleration
```

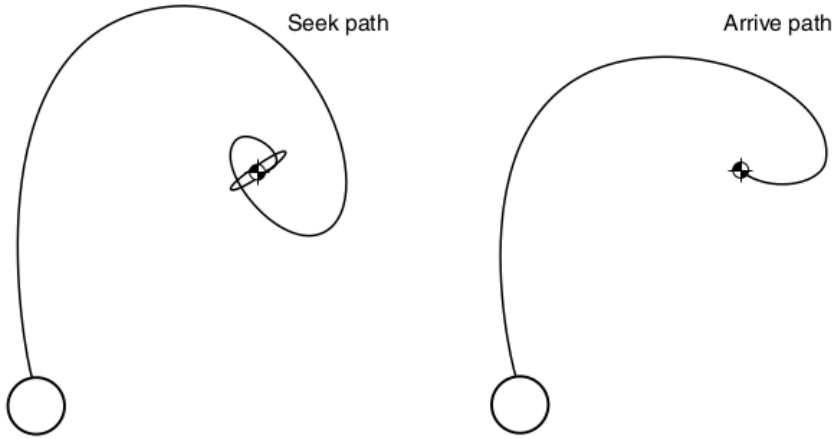
```
class Seek:
    character # kinematic data
    target # kinematic data
    maxAcceleration

    def getSteering():
        steering = new SteeringOutput()
        steering.linear = target.position -
                          character.position #
                                change here for
                                 flee
        steering.linear.normalize()
        steering.linear *= maxAcceleration
        steering.angular = 0
        return steering
```

Demo

Note, orientation removed: like before or by matching or proportional

Arrive

Seek always moves to target with max acceleration. If target is standing it will orbit around it. Hence we need to slow down and arrive with zero speed.



Two radii:

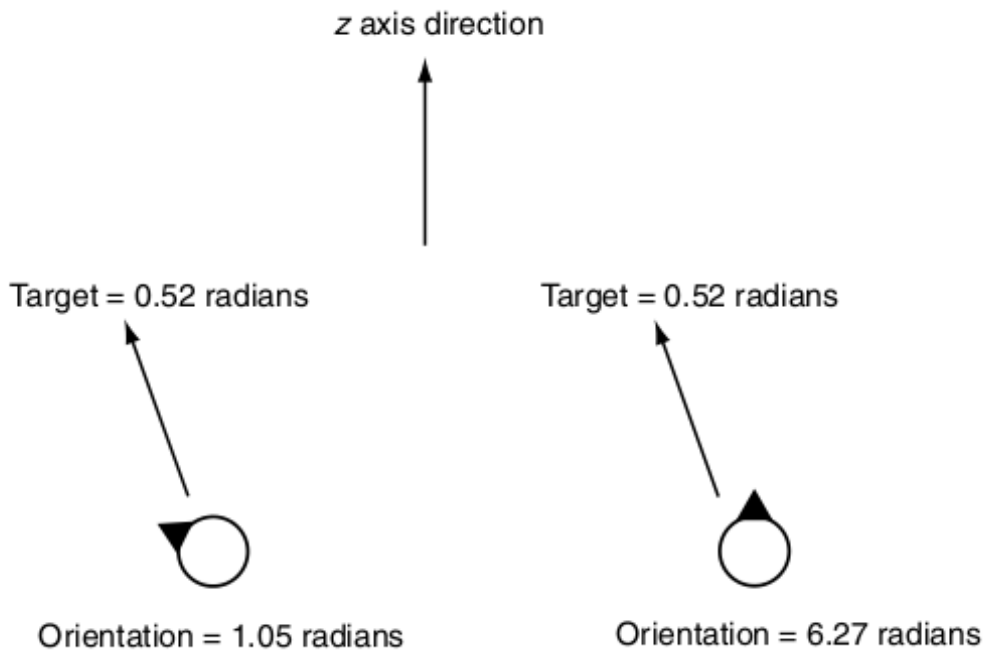
- arrival radius, as before, lets the character get near enough to the target without letting small errors keep it in motion.
- slowing-down radius, much larger. max speed at radius and then interpolated by distance to target

Direction as before
Acceleration dependent on the desired velocity to reach in a fixed time (0.1 s)

```
class Arrive:
    character # kinematic data
    target
    maxAcceleration
    maxSpeed
    targetRadius
    slowRadius
    timeToTarget = 0.1 # time to arrive at target
    def getSteering(target):
        steering = new SteeringOutput()
        direction = target.position - character.position
        distance = direction.length()
        if distance < targetRadius
            return None
        if distance > slowRadius:
            targetSpeed = maxSpeed
        else:
            targetSpeed = maxSpeed * distance / slowRadius
        targetVelocity = direction
        targetVelocity.normalize()
        targetVelocity *= targetSpeed
        steering.linear = targetVelocity - character.velocity
        steering.linear /= timeToTarget
        if steering.linear.length() > maxAcceleration:
            steering.linear.normalize()
            steering.linear *= maxAcceleration
        steering.angular = 0
        return steering
```

Align

Match the orientation of the character with that of the target (just turn, no linear acceleration). Angular version of Arrive.
Issue:
avoid rotating in the wrong direction because of the angular wrap



convert the result into the range $(-\pi, \pi)$ radians by adding or subtracting $m \cdot 2\pi$

```
class Align:
    character
    target
    maxAngularAcceleration
    maxRotation
    targetRadius
    slowRadius
    timeToTarget = 0.1
    def getSteering(target):
        steering = new SteeringOutput()
        rotation = target.orientation - character.orientation
        rotation = mapToRange(rotation)
        rotationSize = abs(rotationDirection)
        if rotationSize < targetRadius
            return None
        if rotationSize > slowRadius:
            targetRotation = maxRotation
        else:
            targetRotation = maxRotation * rotationSize / slowRadius
        targetRotation *= rotation / rotationSize
        steering.angular = targetRotation - character.rotation
        steering.angular /= timeToTarget
        angularAcceleration = abs(steering.angular)
        if angularAcceleration > maxAngularAcceleration:
            steering.angular /= angularAcceleration
            steering.angular *= maxAngularAcceleration
        steering.linear = 0
        return steering
```

Velocity Matching

- So far we matched positions
- Matching velocity becomes relevant when combined with other behaviors, eg. flocking steering behavior
- Simplified version of arrive

```
class VelocityMatch:
    character
    target
    maxAcceleration
    timeToTarget = 0.1
    def getSteering(target):
        steering = new SteeringOutput()
        steering.linear = ( target.velocity – character.velocity ) / timeToTarget
        if steering.linear.length() > maxAcceleration:
            steering.linear.normalize()
            steering.linear *= maxAcceleration
        steering.angular = 0
        return steering
```

Summary

- AI Introduction
- Movement behaviours
  - Representation: static, kinematic
  - Kinematic Movement
    - Seeking
    - Wandering
  - Steering Behaviors
    - Seek and Flee
    - Arrive
    - Align
    - Velocity Matching

Delegated Behaviors

- we saw the building blocks: seek and flee, arrive, align and velocity matching
- next we will see delegated behaviors: calculate a target, either position or orientation, and delegate the steering
- example: arrive can be obtained by creation of a velocity target and application of veleocity matching
- author advocates polymorphic style of programming (inheritance, subclasses) to avoid duplicating code
- Pursue and evade, Face, Looking where you are going, Wander, Path following

Resume

Kinematic Movement

- Seek
- Wandering

Steering Movement

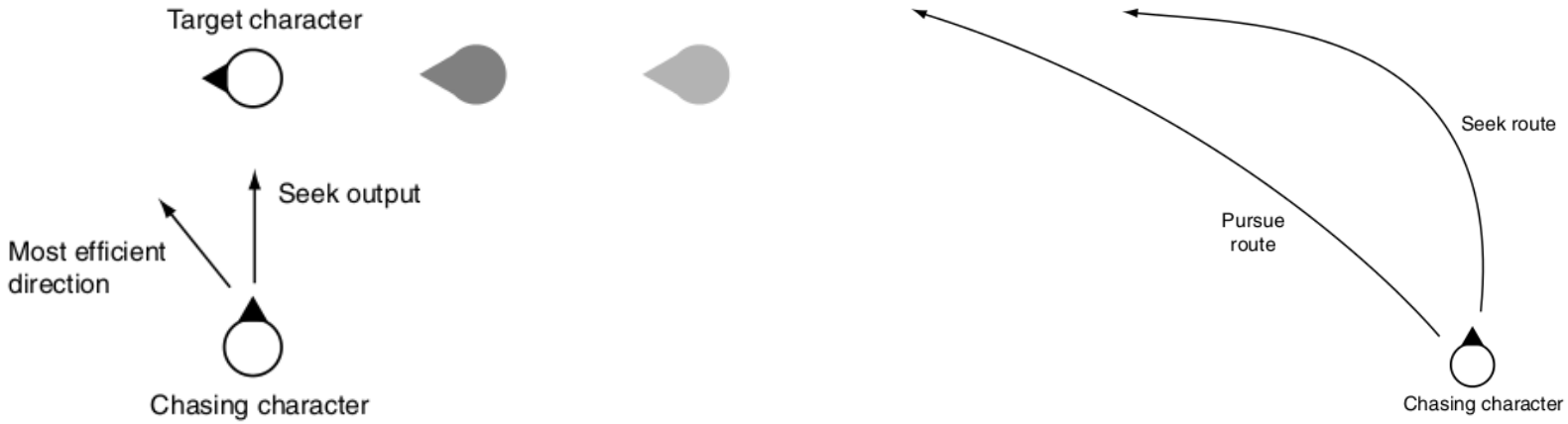
- Variable Matching
- Seek and Flee
- Arrive
- Align
- Velocity Matching

Outline

Outline

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): # derived from Seek
    maxPrediction # max lookahed time
    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

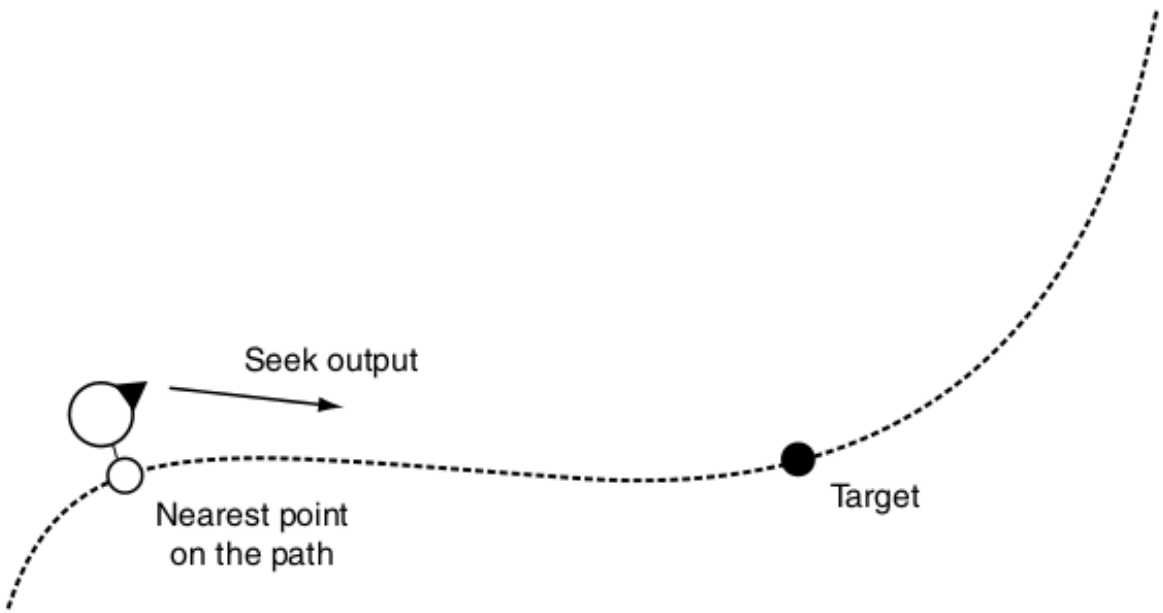

```
class Wander (Face):
    wanderOffset # forward offset of the wander
    wanderRadius
    wanderRate # max rate of change of the orientation
    wanderOrientation # current orientation
    maxAcceleration
    # ... Other data is derived from the superclass ...
    def getSteering():
        wanderOrientation += ( random(0,1)-random(0,1) ) * 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

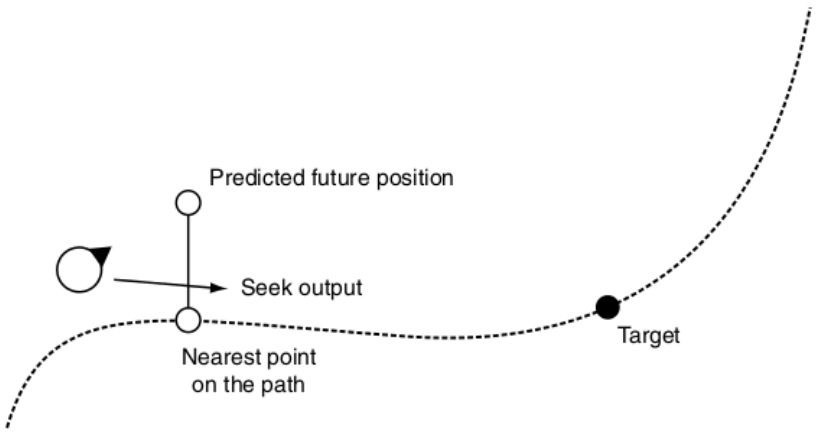
- Takes a whole path (line segment or curve splines) as target (eg, a patrol rute). Resulting behavior: move along the path in one direction

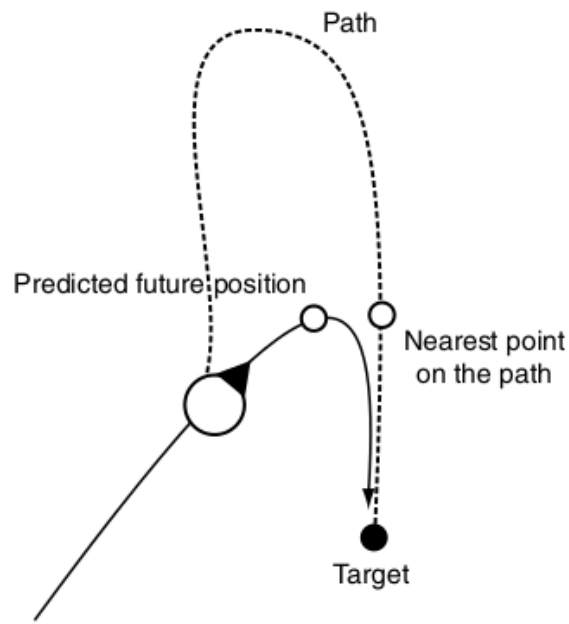
- Delegated:

  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
```

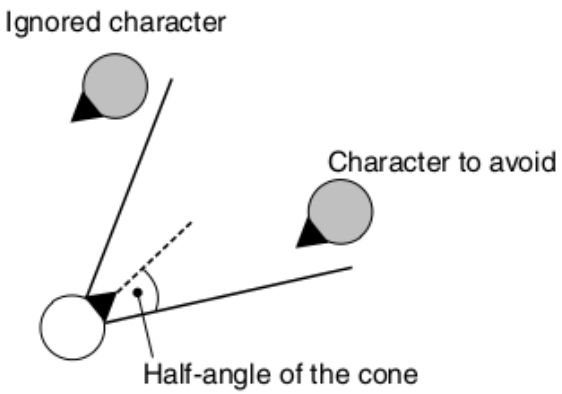
Separation


```
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 to find neighbors: Multi-resolution maps, quad- or octrees, and binary space partition (BSP) trees

Collision Avoidance

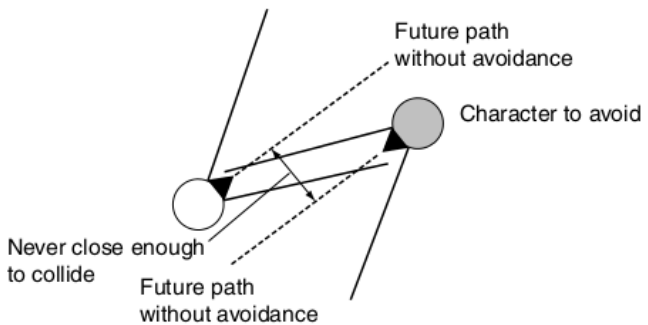
- with large numbers of characters moving around: only engage if the target is within a cone in front of the character.
- average position and speed of all characters in the cone and evade that target. Alternatively, closest character in the cone.

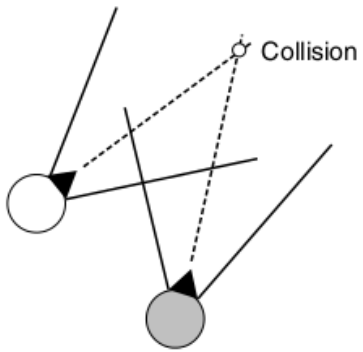


cone checked by dot product

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

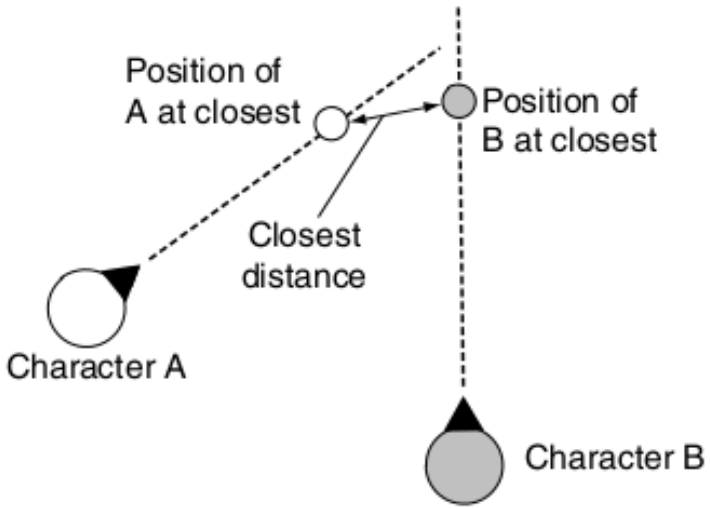
Two problematic situations:





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.



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

$$\boldsymbol{v} = \frac{\boldsymbol{r}}{t} \Longrightarrow t = \frac{\boldsymbol{r} \cdot \boldsymbol{v}}{|\boldsymbol{v}|^2}$$

position at time of closest approach:

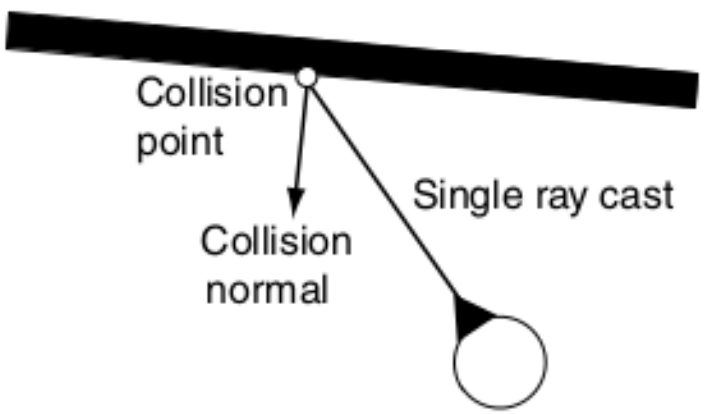
$$\boldsymbol{r}'_c = \boldsymbol{r}_c + \boldsymbol{v}_c t$$
$$\boldsymbol{r}'_t = \boldsymbol{r}_t + \boldsymbol{v}_t t$$

With group of chars: search for the character whose closest approach will occur first and 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

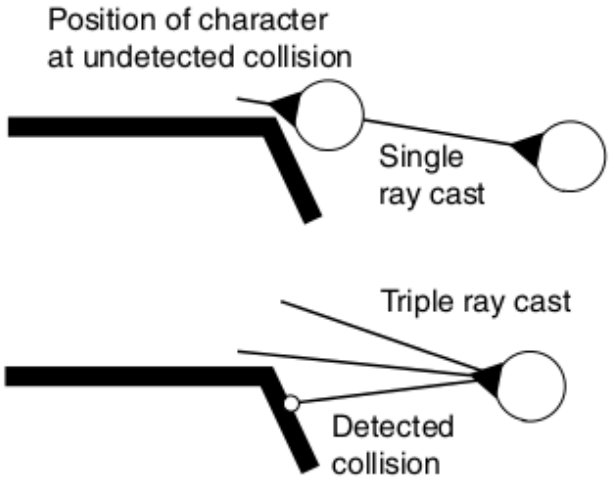
- So far targets are spherical and center of mass
- More complex obstacles, eg, walls, cannot be easily represented in this way.
- cast one or more rays out in the direction of the motion.
- If these rays collide with an obstacle,
  then create a target to avoid the collision, and do seek on this target.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

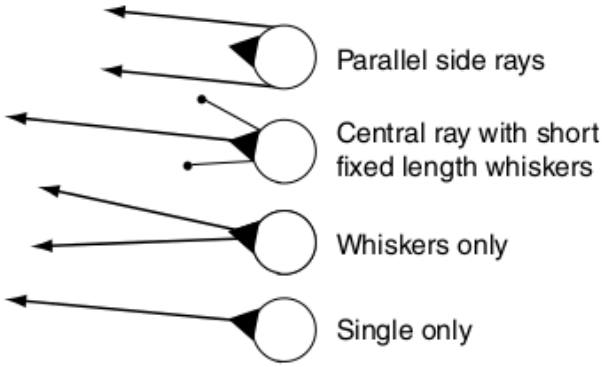


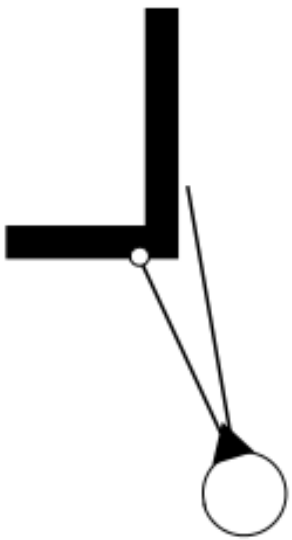
```
class ObstacleAvoidance (Seek):
    collisionDetector
    avoidDistance
    lookahead
    # ... Other data from superclass ...
    def getSteering():
        rayVector = character.velocity
        rayVector.normalize()
        rayVector *= lookahead
        collision = collisionDetector.getCollision(character.position,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

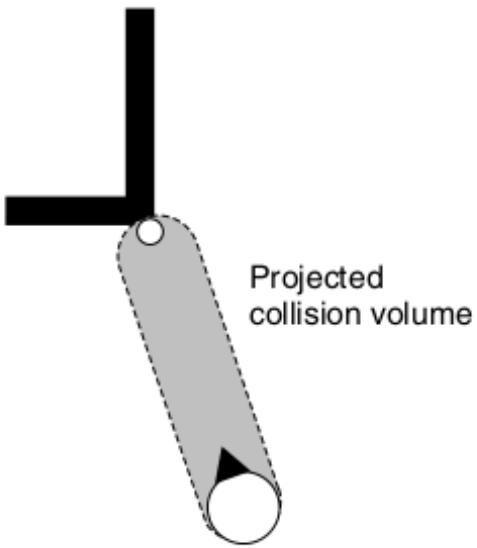
getCollision implemented by casting a ray from position to position + moveAmount and checking for intersections with walls or other obstacles.

Problems and Work Around

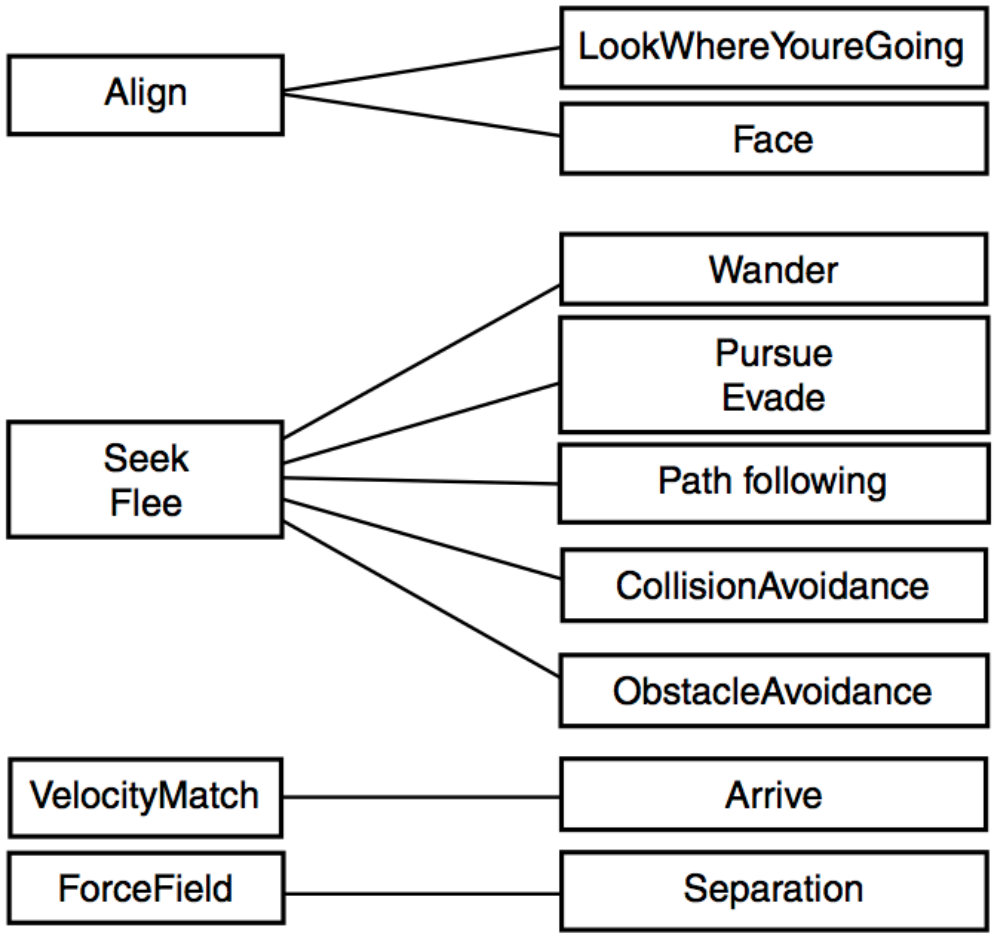








Summary


Align
LookWhereYoureGoing
Face
Seek
Flee
Wander
Pursue
Evade
Path following
CollisionAvoidance
ObstacleAvoidance
VelocityMatch
Arrive
ForceField
Separation