Autonomous Movement

Technical Game Development II

Professor Charles Rich
Computer Science Department
rich@wpi.edu

[see Buckland, Ch. 3
Millington, Ch. 3
http://opensteer.sourceforge.net]

Introduction

- One of the most fundamental requirements of AI is to *move characters around* in the game sensibly
- For some games, e.g., FPS, realistic movement is pretty much all there is--there is no higher level decision making
- At other extreme, e.g., chess, there is no "movement" per se---pieces just placed
- We're going to treat everything in 2D today, since most game motion in gravity on surface (2 1/2 D)

Craig Reynolds

- The "giant" in this area---his influence cannot be overstated
  - **1987**: "Flocks, Herds and Schools: A Distributed Behavioral Model," *Computer Graphics*
  - **1998**: Winner of *Academy Award* in Scientific and Engineering category
  - **1999**: "Steering Behaviors for Autonomous Characters," *Proc. Game Developers Conference*
  - Currently at U.S. R&D group of Sony Computer Entertainment

The "Steering" Model

Action Selection

Choosing goals and plans, e.g.

- *"go here"*
- *"do A, B, and then C"*

Steering

- *Calculate trajectories to satisfy goals and plans*
- *Produce steering force that determines where and how fast character moves*

Locomotion/Vehicle

Mechanics ("how") of motion

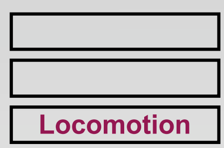
- *differs for characters, e.g., fish vs. horse*
- *independent of steering*

Individual Steering Behaviors

seek	flee
arrive	pursue
wander	evade
interpose	hide
avoid obstacles & walls	follow path

and combinations thereof.....

Updating the Vehicle Physics

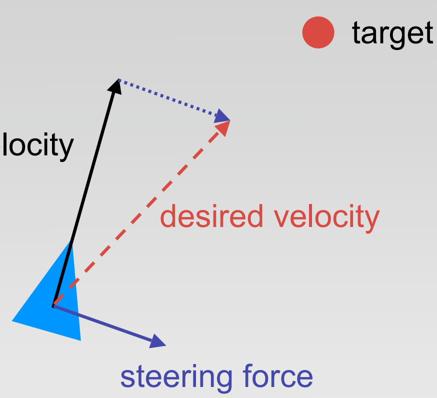



```
class Vehicle
   // point mass of rigid body
   mass         // scalar
   position     // vector
   velocity     // vector

   // orientation of body
   heading      // vector

   // properties of vehicle
   maxForce     // vector
   maxSpeed     // scalar
   maxRotation // scalar (not used)

   def update (dt) {
      force = ...; // combine forces from steering behaviors
      acceleration = force / mass;  // Newton's 2nd law
      velocity += truncate(acceleration * dt, maxSpeed);
      position += velocity * dt;
      // unless almost stopped
      if ( |velocity| > 0.00000001 )
         // update heading to face along velocity vector
         heading = ...velocity...;
   }
```

Steering Methods

```
class Vehicle
   def update (dt) {
      force = ...; // combine forces from steering behaviors
      ...}

   def seek (target) { ... return force; }

   def flee (target) { ... return force; }

   def arrive (target) { ... return force; }

   def pursue (vehicle) { ... return force; }

   def evade (vehicle) { ... return force; }

   def hide (vehicle) { ... return force; }

   def interpose (vehicle1, vehicle2) { ... return force: }

   def wander () { ... return force; }

   def avoidObstacles () { ... return force; }
   ...
```

Seek

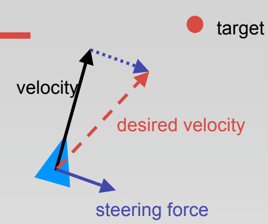



```
def seek (target) {
   // vector from here to target scaled by maxSpeed
   desired = truncate(target - position, maxSpeed);
   return desired - velocity;
}
```

Problem with Seek

- Overshoots target
- Amount of overshoot determined by ratio of maxSpeed to maxForce
- Intuitively, needs to decelerate as gets closer

Arrive



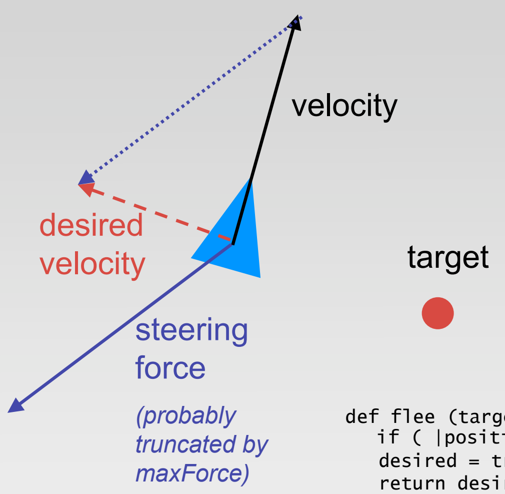

```
def arrive (target) {

   distance = |target - position|;  // to target
   if ( distance == 0 ) return [0,0];

   // current speed required to arrive at rest at target
   // deceleration time is a "tweak" variable
   speed = distance / DECELERATION;

   // current speed cannot exceed vehicle maxSpeed
   speed = min(speed, maxSpeed);

   // vector from here to target scaled by speed
   desired = (target - position) * speed / distance;

   // return steering force as in seek
   return desired - velocity;
}
```

Arrive Behavior

- When vehicle is far away from target, it behaves just like **seek**, i.e., it closes at maximum speed
- Deceleration only comes into effect when the vehicle get close to the target, i.e. when ‘speed’ becomes less than ‘maxSpeed’ in:

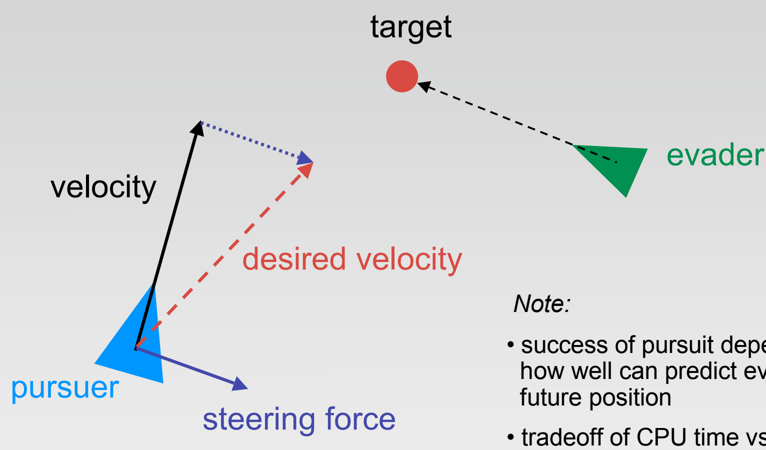
```
speed = min(speed, maxSpeed);
```


Flee (Opposite of Seek)




```
def flee (target) {
   if ( |position - target| > PANIC ) return [0,0];
   desired = truncate(position - target, maxSpeed);
   return desired - velocity;
}
```

Pursue (Seek Predicted Position)



Note:

- success of pursuit depends on how well can predict evader's future position
- tradeoff of CPU time vs. accuracy
- special case: if evader almost dead ahead, just seek

Pursue

```
def pursue (vehicle) {

   toVehicle = vehicle.position - position;

   // if within 20 degrees ahead, simply seek
   if ( toVehicle * heading > 0
        && heading * vehicle.heading < -0.95 )
      return seek(vehicle.position);

   // calculate lookahead time based on distance and speeds
   dt = |toVehicle| / (maxSpeed + |vehicle.velocity|);

   // seek predicted position
   return seek(vehicle.position + (vehicle.velocity * dt));
}
```

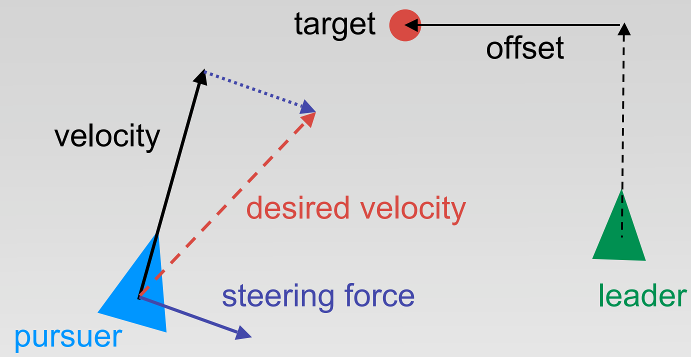
Evade (Opposite of Pursue)

```
def evade (vehicle) {

     // no special case check for dead ahead

     // calculate lookahead time based on distance and speeds
     dt = |position - vehicle.position| / (maxSpeed + |vehicle.velocity|);

     // flee predicted position
     return flee(vehicle.position + (vehicle.velocity * dt));
  }
```


Pursue with Offset

- Steering force to keep vehicle at specified offset from target vehicle
- Useful for:
  - marking an opponent in a sports simulation
  - docking with a spaceship
  - shadowing an aircraft
  - implementing battle formations
- NB: This is not “flocking”, which we will see later

Pursue with Offset




```
def pursue (vehicle, offset) {
   // calculate lookahead time based on distance and speeds
   dt = |position - (vehicle.position + offset)|
        / (maxSpeed + |vehicle.velocity|);
   // arrive at predicted offset position (vs. seek)
   return arrive(vehicle.position + offset + (vehicle.velocity * dt));
}
```


Interpose

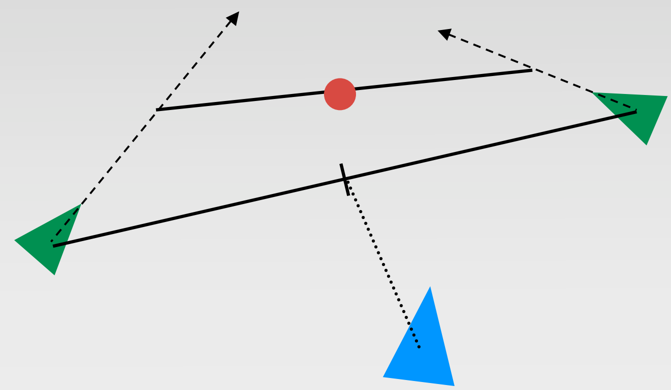
- Similar to pursue
- Return steering force to move vehicle to midpoint of imaginary line connecting two vehicles
- Useful for:
  - bodyguard taking a bullet
  - soccer player intercepting a pass
- Like pursue, main trick is to estimate lookahead time (dt) to predict target point

Interpose

(1) Bisect line between vehicles

(2) Calculate dt to bisection point

(3) Target arrive at midpoint of predicted positions



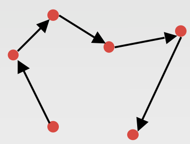
Interpose

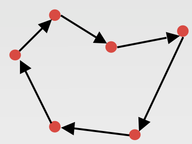
```
def interpose (vehicle1, vehicle2) {

   // lookahead time to current midpoint
   dt = |vehicle1.position + vehicle2.position| / (2 * maxSpeed);

   // extrapolate vehicle trajectories
   position1 = vehicle1.position + vehicle1.velocity * dt;
   position2 = vehicle2.position + vehicle2.velocity * dt;

   // steer to midpoint
   return arrive(position1 + position2 / 2);
}
```

Path Following

- Create steering force that moves vehicle along a series of *waypoints* (open or looped)
- Useful for:
  - patrolling (guard duty) agents
  - predefined paths through difficult terrain
  - racing cars around a track

open
path



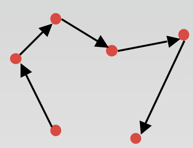


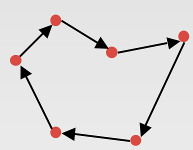
looped
path

Path Following

- Invoke ‘seek’ on each waypoint until ‘arrive’ at finish (if any)

```
path = ...;                 // (circular) list of waypoints
current = path.first() ; // current waypoint vector

def followPath () {

   if ( |current - position| < SEEK_DISTANCE )
      if ( path.isEmpty() )
         return arrive(current);
      else
         current = path.next();

    return seek(current);
}
```





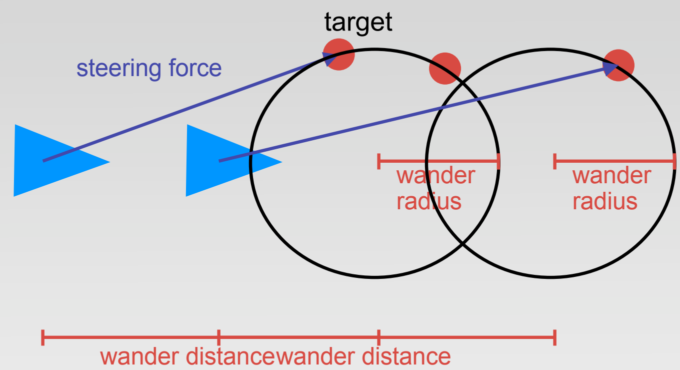
Path Following

- Very sensitive to `SEEK_DISTANCE` and ratio of maxForce to maxSpeed (in underlying vehicle locomotion model)
  - tighter path following for interior corridors
  - looser for open outdoors

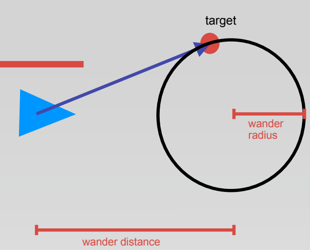
Wander

- Goal is to produce a steering force which gives impression of a random walk though the agent's environment
- Naive approach:
  - calculate *random* steering force each update step
  - produces unpleasant "jittery" behavior
- Reynold's approach:
  - project a circle in front of vehicle
  - steer towards target constrained to move along perimeter of the circle

Wander



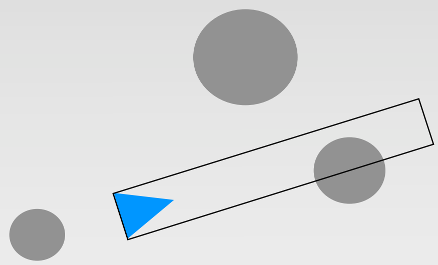
Wander




```
// initial random point on circle
wanderTarget = ...;

def wander () {

   // displace target random amount
   wanderTarget += [ random(0, JITTER), random(0, JITTER) ];

   // project target back onto circle
   wanderTarget.normalize();
   wanderTarget *= RADIUS;

   // move circle wander distance in front of agent
   wanderTarget += vehicleToWorldCoord([DISTANCE, 0]);

   // steer towards target
   return wanderTarget - position;
}
```

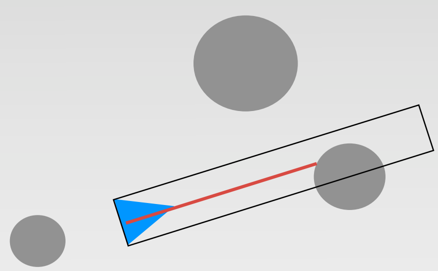
Obstacle Avoidance

- Treat obstacles as circular bounding volumes
- Basic idea: extrude "detection box" in front of vehicle in direction of motion



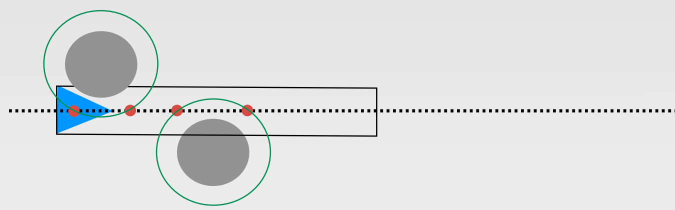
Obstacle Avoidance Algorithm

1. Find closest intersection point
2. Calculate steering force to avoid obstacle



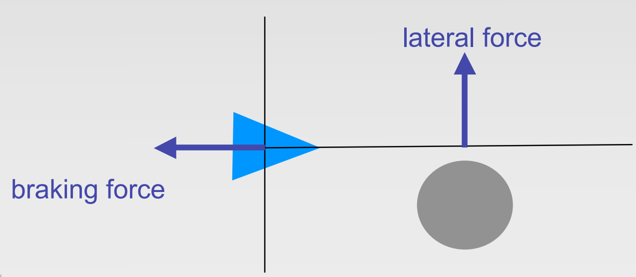
Obstacle Avoidance Algorithm

1. Find closest intersection point
   (a) discard all obstacles which do not overlap with detection box
   (b) expand obstacles by half width of detection box
   (c) find intersection points of trajectory line and expanded obstacle circles
   (d) choose closest intersection point *in front* of vehicle



Obstacle Avoidance Algorithm

2. Calculate steering force
   (a) combination of lateral and braking force
   (b) each proportional to vehicle's distance from obstacle (needs to react quicker if closer)

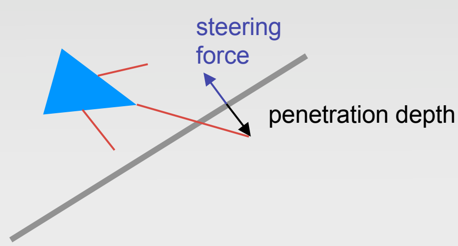


Wall Avoidance

(a) test for intersection of three "feelers" with wall

(b) calculate *penetration depth* of closest intersection

(c) return steering force perpendicular to wall with magnitude equal to penetration depth

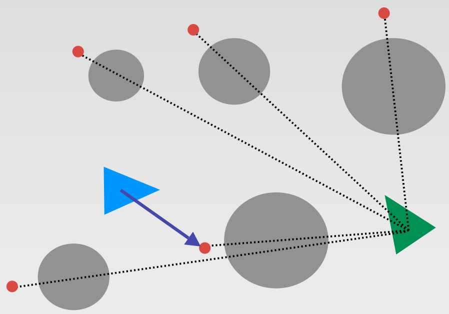


Hide (Behind Obstacles)

- Attempt to position vehicle so that an obstacle is always between itself and other vehicle
- Useful for:
  - NPC hiding from player
    - to avoid being shot by player
    - to sneak up on player (combine hide and seek)

Hide

(a) for each obstacle, determine hiding spot
(b) if no hiding spots, invoke ‘evade’
(c) otherwise, invoke ‘arrive’ to closest hiding spot

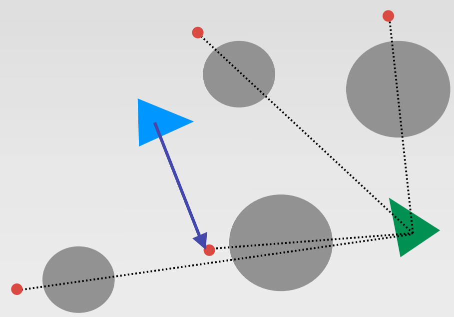


Hide - Possible Refinements

- Only hide is you can “see” other vehicle
  - tends to look dumb (i.e., agent has no memory)
  - can improve by adding time constant, i.e., hide if you saw other vehicle in last <n> seconds
- Only hide if you can “see” other vehicle *and* other vehicle can see you

Hide - Possible Refinements

- Instead of always choosing *closest* hiding spot, favor spots that are *behind* or to *side* of other vehicle



Hide - Possible Refinements

- Add “panic distance” (like flee behavior)

```
def hide (vehicle) {
   if ( |position - target| > PANIC ) return [0,0];
   ...
}
```

Combining Steering Forces

```
class Vehicle
   def update (dt) {
      force = ...; // combine forces from steering behaviors
      ...}

   def seek (target) { ... return force; }

   def flee (target) { ... return force; }

   def arrive (target) { ... return force; }

   def pursue (vehicle) { ... return force; }

   def evade (vehicle) { ... return force; }

   def hide (vehicle) { ... return force; }

   def interpose (vehicle1, vehicle2) { ... return force: }

   def wander () { ... return force; }

   def avoidObstacles () { ... return force; }
   ...
```


Combining Steering Forces

- Two basic approaches:
  - blending
  - priorities
- Advanced combined approaches:
  - weighted truncated running sum with prioritization [Buckland]
  - prioritized dithering [Buckland]
  - pipelining [Millington]
- All involve significant *tweaking* of parameters

Blending Steering

- ***All*** steering methods are called, each returning a force (could be [0,0])
- Forces combined as linear weighted sum:

  $$w_1F_1 + w_2F_2 + w_3F_3 + \ldots$$

  - weights do not need to sum to 1
  - weights tuned by trial and error
- Final result will be limited (truncated) by maxForce

Blended Steering - Problems

- Expensive, since all methods called every tick
- Conflicting forces not handled well
  - tries to "compromise", rather than giving priority
  - e.g., avoid obstacle and seek, can end up partly penetrating obstacle
- Very hard to tweak weights to work well in all situations

Prioritized Steering

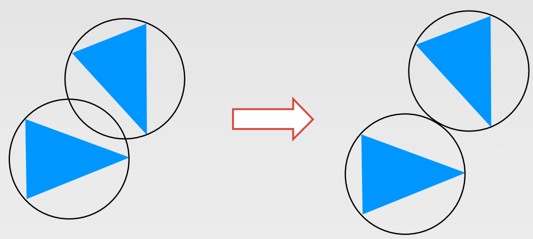
- *Intuition:* Many of steering behaviors only return a force in appropriate conditions
- *Algorithm:*
  - Sort steering methods into priority order
  - Call methods one at a time until first one returns non-zero force
  - Apply that force and *stop evaluation* (saves CPU)
- *Variation:*
  - Define <u>groups</u> of behaviors with <u>blending inside</u> each group and <u>priorities between</u> groups

Prioritized Dithering (Reynolds)

- In addition to priority order, associate a <u>probability</u> with each steering method
- Use random number and probability to sometimes <u>skip</u> some methods in priority order (on some ticks)
- Gives lower priority methods some influence without problems of blending

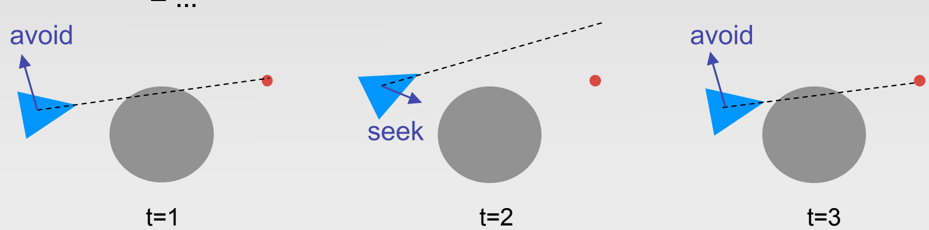
Ensuring Zero Overlap

- Often, when combining behaviors in the presence of multiple vehicles, the vehicles will occasionally overlap one another (they're not obstacles!)
- If bounding spheres overlap, just "teleport" to touching distance (ignore dynamics)



Smoothing - The Problem

- Conflicting behaviors can alternate, causing "judder" (jitter/shudder)
  - e.g., avoidObstacle and seek
    - avoidObstacle forces you away from obstacle until it is out of range
    - seek pushes you back into range
    - ...



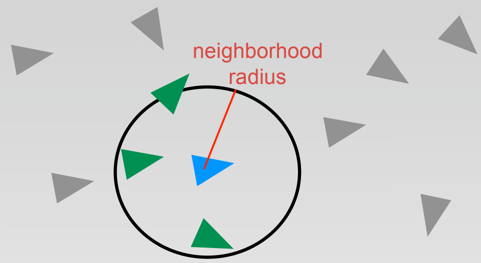
Smoothing - The Solution

- Ideally to avoid problem, forsee conflict ahead of time--but can be complicated and expensive to compute
- Simple hack (per Robin Green, Sony):
  - *decouple* heading from velocity vector
  - average heading over “several” ticks
  - tune number of ticks for smoothing (keep small to minimize memory and CPU)
  - not perfect solution, but produces adequate results at low cost

Group Steering Behaviors - “Flocking”

- Combination of three behaviors:
  - cohesion
  - separation
  - alignment
- Each applied to neighbors

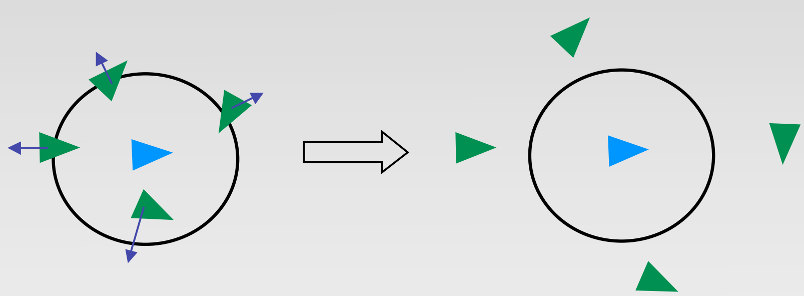
Neighbors



- Variation:
  - restrict neighborhood to field of view (e.g., 270 deg.) in *front*
  - may be more realistic in some applications

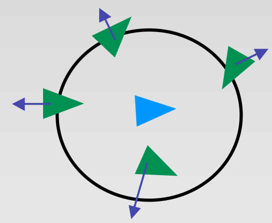
Separation

- Create force that steers vehicle away from others in neighborhood



Separation

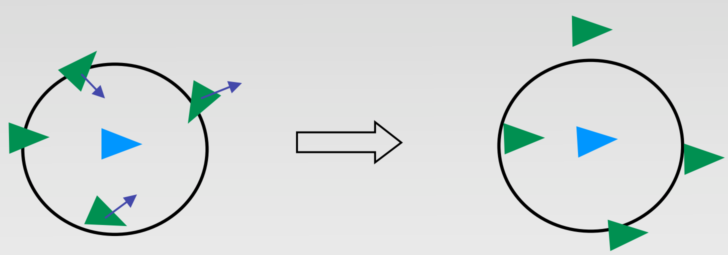
- Vector to each neighbor is normalized and divided by the distance (i.e., stronger force for closer neighbors)



```
def separation () {
   force = [0,0];
   for each neighbor
      direction = position - neighbor.position;
      force += normalize(direction) / |direction|;
   return force;
}
```


Alignment

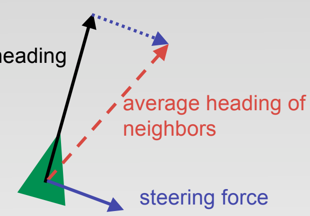
- Attempt to keep vehicle's heading aligned with its neighbors headings



Alignment

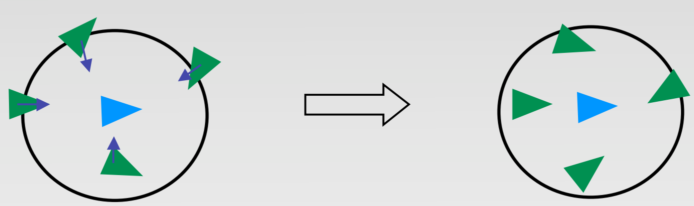
- Return steering force to correct towards average heading vector of neighbors

```
def alignment () {
   average = [0,0];
   for each neighbor
      average =+ neighbor.heading;
   average /= |neighbors|;
   return average - heading;
}
```

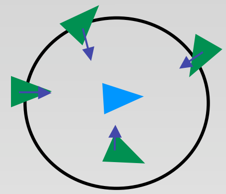


Cohesion

- Produce steering force that moves vehicle towards center of mass of neighbors



Cohesion



```
def cohesion () {
   center = [0,0];
   for each neighbor
      center += neighbor.position;
   center /= |neighbors|;
   seek(center);
}
```


Flocking

- An “emergent behavior”
  - looks complex and/or purposeful to observer
  - but actually driven by fairly simple rules
  - component entities don’t have the big picture
- Often used in films
  - bat and penguins in Batman Returns
  - orc armies in Lord of the Rings

Connecting Steering to Action Selection

Action Selection

Steering

Locomotion/Vehicle

Choosing goals and plans, e.g.

- *"go here"*
- *"do A, B, and then C"*

Steering Methods

```
class Vehicle
   def update (dt) {
      force = ...; // combine forces from steering behaviors
      ...}

   def seek (target) { ... return force; }

   def flee (target) { ... return force; }

   def arrive (target) { ... return force; }

   def pursue (vehicle) { ... return force; }

   def evade (vehicle) { ... return force; }

   def hide (vehicle) { ... return force; }

   def interpose (vehicle1, vehicle2) { ... return force: }

   def wander () { ... return force; }

   def avoidObstacles () { ... return force; }
   ...
```

Turning Steering Methods On & Off

```
class Vehicle
   seekTarget = null;
   fleeTarget = null;
   ...
   wanderOn = false;
   ...

   def think () { ... }

   def update (dt) {
      think();
      force = [0,0];
      if ( seekTarget != null ) force = combine(force, seek(seekTarget));
      if ( fleeTarget != null ) force = combine(force, flee(fleeTarget));
      ...
      if ( wanderOn ) force = combine(force, wander());
      ...
   }

   def seek (target) { ... return force; }
   def flee (target) { ... return force; }
   ...
   def wander () { ... return force; }
   ...
```


Thinking with Hierarchical Goals

[see Buckland, Ch. 9]

- A brief sketch to “bridge” to next week’s lecture on goal-based AI in Halo 3.
- Basic concepts:
  - goal decomposition (hierarchy)
  - competing goals (arbitration)
  - success and failure of goals

Everyday Example

Goal decomposition hierarchy

- visit the cinema
  - leave house
    - walk to closet
    - open closet door
    - remove coat from hook
    - put coat on
    - walk to front door
    - open front door

    ...
  - travel to cinema
  - enter cinema

Everyday Example

Competing goals

- go to cinema
- eat dinner
- do your homework
- call your mother

...

FPS Example

Competing goals

- explore environment
- get health
- get weapon (for various weapon types)
- attack target

...

FPS Example

Goal decomposition hierarchy

- attack target
  - hunt target
    - move to target position offset
      - follow path
        - traverse edge
        - traverse edge
        - ...
  - strafe target

Primitive vs. Composite Goals

- Primitive goals are at leaves of decomposition tree, e.g,. traverse edge
- Other goals are called “composite”, e.g., hunt target

- attack target
  - hunt target
    - move to target position offset
      - >follow path
        - »traverse edge
        - »traverse edge
        - ...
  - strafe target

Implementation

```
class Goal

  Goal (vehicle)

  enum status = { inactive, active, completed, failed }

  // queue up subgoals and change state to active
  def activate ()

  // invoke and sequence subgoals
  def process ()

  // cleanup and determine if completed or failed
  def terminate ()
```

Wander (Primitive Goal)

```
class Wander : Goal

   def activate () {
      status = active;
      vehicle.wanderOn = true;
   }

   def process () {}

   def terminate () {
      status = completed;
      vehicle.wanderOn = false;
   }
```


Attack Target (Composite Goal)

```
class Attack : Goal

   Attack (target)

   def activate () {
      status = active;
      // make sure target not already dead
      if ( target.isDead() )
         status = completed;
         return;
      if ( target.isInRange() )
         addSubgoal(new Strafe(target));
      else
         addSubgoal(new HuntTarget(target));
   }

   ...
}
```

Goal Execution Engine ("Think")

- For each toplevel goal (explore, attack, etc.), evaluate its desirability condition
- Choose the most desirable (goal arbitration)
- Execute that goal (call activate, process, terminate)
  - engine methods called in process method will keep track of active goals, re-queue failed goals, etc.
- Repeat as needed

Desirability Conditions

- For get health goal:

$$k_{health} \times \frac{1 - health}{distToHealth}$$

- For get weapon goal:

$$k_{weapon} \times \frac{health \times (1 - weaponStrength)}{distToWeapon}$$

- For attack goal:

$$k_{attack} \times totalWeaponStrength \times health$$

Need to tune the k's ! ("bucket of floats")

Putting It All Together

[see Buckland, Raven game]