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

- A fundamental requirement in many games is to *move characters (player avatar and NPC's) around realistically* and *pleasantly*
- For some games, e.g., FPS, realistic NPC movement is pretty much all there is (except shooting)--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*
  - Left U.S. R&D group of Sony Computer Entertainment in April 2012 after 13 years

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 <u>steering force</u> that determines where and how fast character moves*

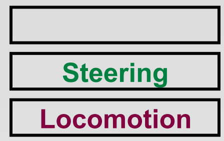
Locomotion

Mechanics (“how”) of motion

- *differs for characters, e.g., fish vs. horse (cf. animations)*
- *independent of steering*

Locomotion *Dynamics*

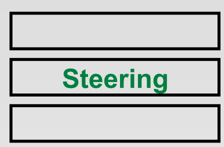
```
class Body
   // point mass of rigid body
   mass        // scalar
   position    // vector
   velocity    // vector

   // orientation of body
   heading     // vector

   // dynamic properties of body
   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...;
      ... // then render
   }
```



Individual Steering "Behaviors"

compute the forces:

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



and combinations thereof.....

So "Steering" in this context means

Making objects move by:

- Applying forces

  instead of

- Directly transforming their positions

Why?

...because it looks much more natural

e.g., "steering" does not mean just using the arrow/WASD keys to move an avatar, but doing the motion by applying forces

Or in Unity...

Add

Component > Physics > Rigidbody

with script, e.g.,

```
public class example : MonoBehaviour {
   ...
   void FixedUpdate() {
      rigidbody.AddForce(Vector3.up * 10);
   }
}
```

Steering Methods

```
class Body
   def update (dt) {
      force = truncate(..., // combine forces from steering behaviors
                       maxForce);
   ...}

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

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

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

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

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

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

   def interpose (body1, body2) { ... return force: }

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

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

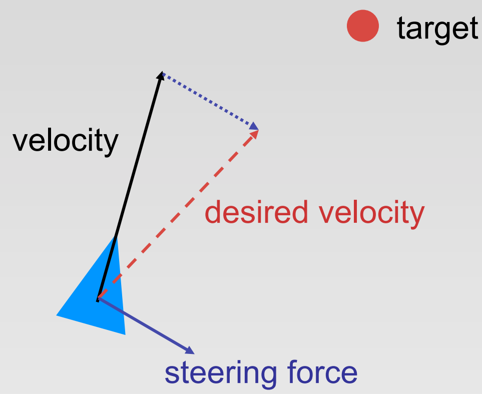

Reference Code in C++

Complete example code for this unit can be downloaded from:

http://samples.jbpub.com/9781556220784/Buckland_SourceCode.zip

See folder for Chapter 3

Seek: Steering Force




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

DEMO

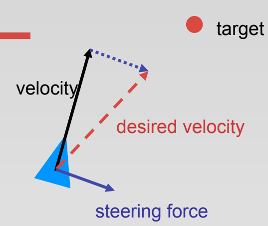
Problem with Seek

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

Arrive: Variant of Seek Behavior

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

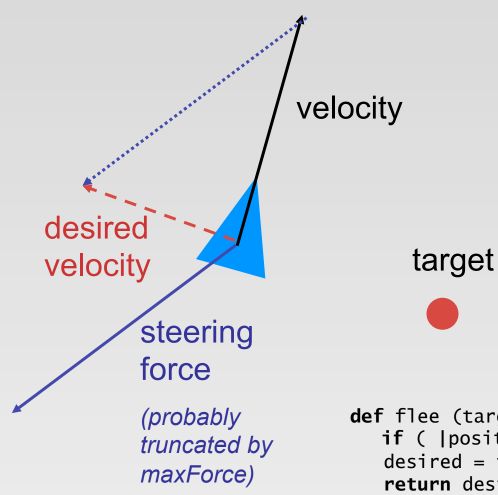
Arrive




```
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 body 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;
}
```

DEMO

Flee: Opposite of Seek

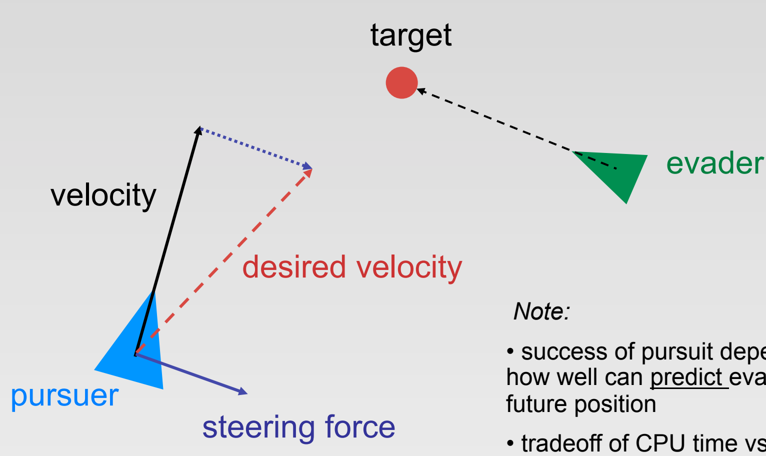



```
def flee (target) {
   if ( |position - target| > PANIC ) return [0,0];
   desired = truncate(position - target, maxSpeed);
   return desired - velocity;
}
```

DEMO

IMGD 4000 (B 12) 15

Pursue: Seek a Predicted Position

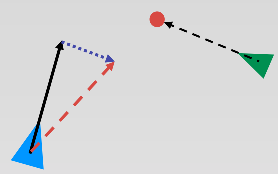


Note:

- 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

IMGD 4000 (B 12) 16

Pursue



```
def pursue (body) {

   toBody = body.position - position;

   // if within 20 degrees ahead, simply seek
   bearing = heading * toBody.heading;
   if ( bearing > 0 && bearing < -0.95 )
      return seek(body.position);

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

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

DEMO

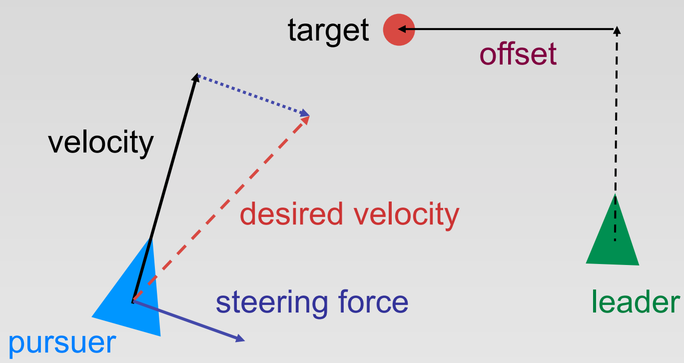
Evade: Opposite of Pursue

```
def evade (body) {

     toBody = body.position - position;

     // no special case check for dead ahead

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

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

Pursue with Offset

- Steering force to keep body at specified offset from target body
- 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 (body, offset) {
   // calculate lookahead time based on distance and speeds
   dt = |position - (body.position + offset)|
        / (maxSpeed + |body.velocity|);
   // arrive at predicted offset position (vs. seek)
   return arrive(body.position + offset + (body.velocity * dt));
}
```

DEMO

Interpose

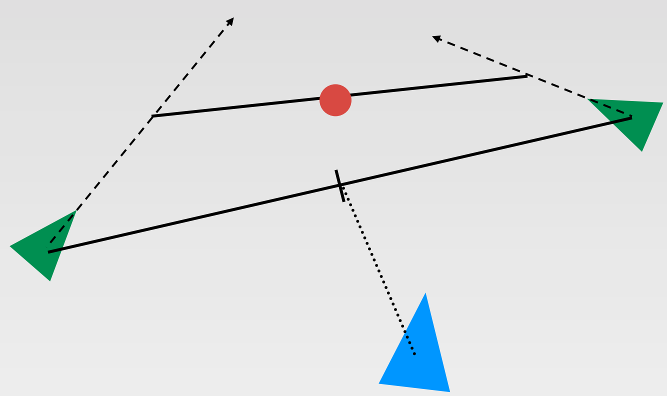
- Similar to pursue
- Return steering force to move body to midpoint of imaginary line connecting two bodies
- Useful for:
  - bodyguard taking a bullet
  - soccer player intercepting a pass
- Like pursue, main trick is to estimate lookahead time (dt) to predict target point

Interpose

(1) Bisect line between bodies

(2) Calculate dt to bisection point

(3) Target arrive at midpoint of predicted positions



Interpose

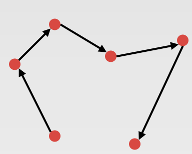
```
def interpose (body1, body2) {

   // lookahead time to current midpoint
   dt = |body1.position + body2.position| / (2 * maxSpeed);

   // extrapolate body trajectories
   position1 = body1.position + body1.velocity * dt;
   position2 = body2.position + body2.velocity * dt;

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

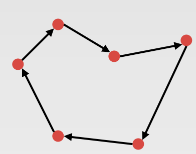
DEMO

Path Following

- Create steering force that moves body 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



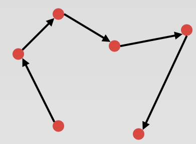


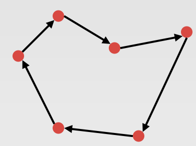
looped path

Path Following: Using Seek

- 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 locomotion model)
  - tighter path following for interior corridors
  - looser for open outdoors

DEMO

Individual Steering "Behaviors"

compute the forces:

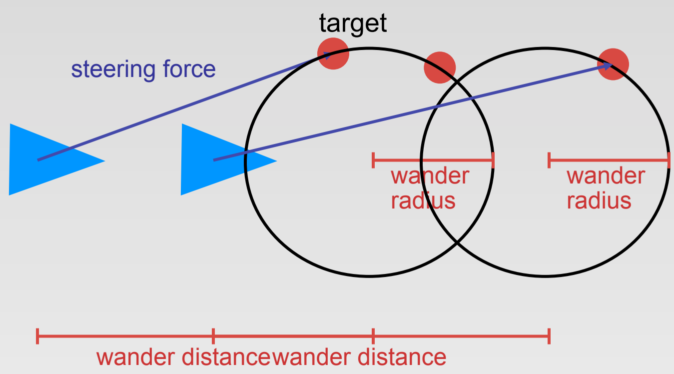
seek	flee
arrive	pursue
interpose	evade
follow path	wander
hide	avoid obstacles & walls

and combinations thereof.....

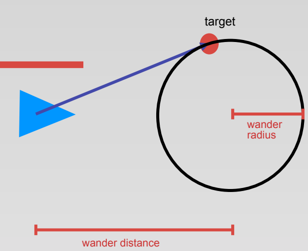
Wander

- 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 body
  - steer towards a *randomly moving target* constrained 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 += bodyToWorldCoord([DISTANCE, 0]);

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

DEMO

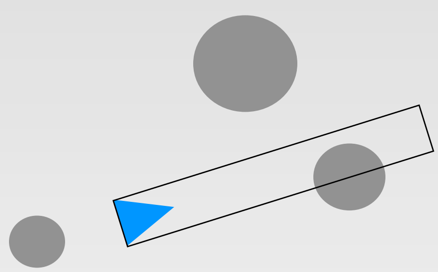
Interacting with Environment

- Obstacle Avoidance
- Hide
- Wall Avoidance

Obstacle Avoidance

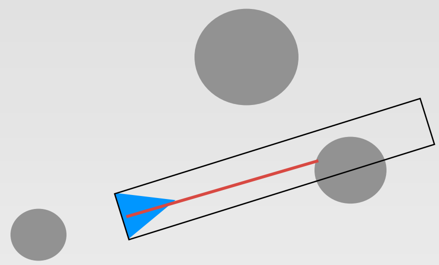
- Treat obstacles as circular bounding volumes
- *Basic idea:* extrude "detection box" in front of body in direction of motion

  (cf. intersection collision detection algorithm)



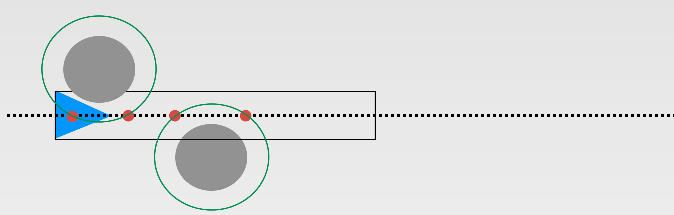
Obstacle Avoidance Algorithm

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



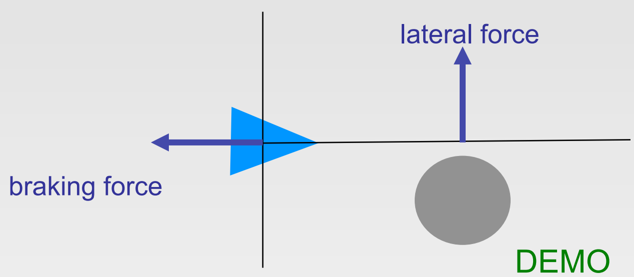
Obstacle Avoidance Algorithm

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 body



Obstacle Avoidance Algorithm

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



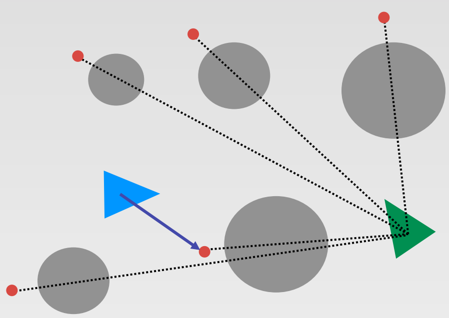
DEMO

Hide

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

Hide

for each obstacle, determine hiding spot
  if no hiding spots **then** invoke 'evade'
  else invoke 'arrive' to closest hiding spot

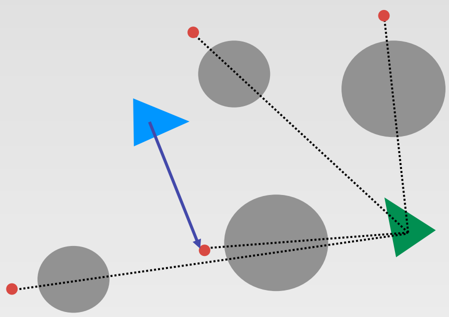


Hide - Possible Refinements

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

Hide - Possible Refinements

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



Hide - Possible Refinements

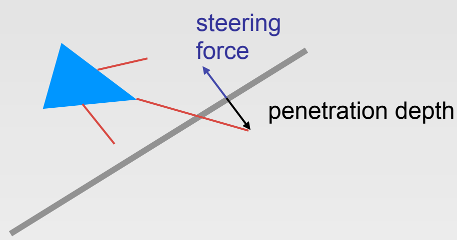
- Add “panic distance” (like flee behavior)

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

DEMO

Wall Avoidance

1. test for intersection of three “feelers” with wall
2. calculate *penetration depth* of closest intersection
3. return steering force perpendicular to wall with magnitude equal to penetration depth



Combining Steering Behaviors: Examples

- battle tanks
  - path following
  - wall avoidance
  - separation (to do)
- animal simulation (e.g., sheep)
  - wander
  - obstacle avoidance (e.g., trees)
  - flee (e.g., predator)

Combining Steering Forces

```
class Body
   def update (dt) {
      force = truncate(..., // combine forces from steering behaviors
                       maxForce);
   ...
   }
   def seek (target) { ... return force; }

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

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

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

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

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

   def interpose (body1, body2) { ... 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:

  $$w_1F_1 + w_2F_2 + w_3F_3 + ...$$

  - 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

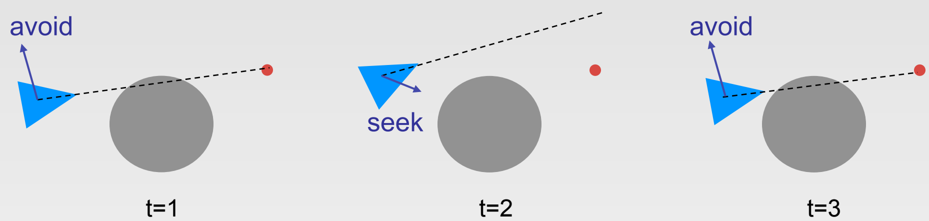
- *Intuition:* Many of steering behaviors only return a force in appropriate conditions
- *Algorithm:*
  - Sort steering methods into priority order
  - Call methods one at a time until first one returns non-zero force
  - Apply that force and *stop evaluation* (saves CPU)
- *Variation:*
  - Define groups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

Prioritized Dithering (Reynolds)

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

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*

Turning Steering Methods On & Off

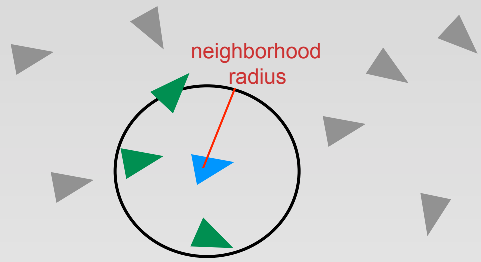
```
class Body
   seekTarget = null;
   fleeTarget = null;
   ...
   wanderOn = false;
   ...

   def think () { ... }

   def update (dt) {
      think(); // the AI part!
      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; }
   ...
```


"Flocking" = *Group* Steering Behaviors

- Combination of three steering behaviors:
  - cohesion
  - separation
  - alignment
- Each applied to all bodies based on neighbors

DEMO

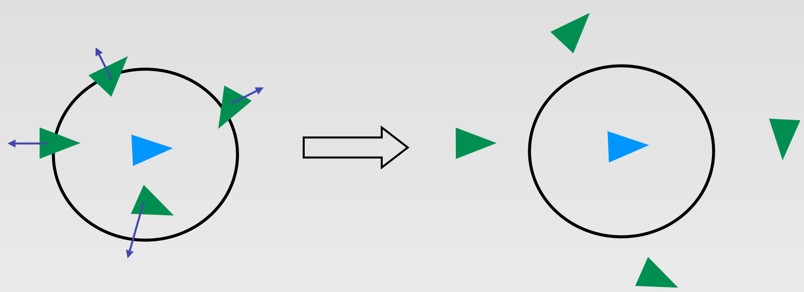
Neighbors



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

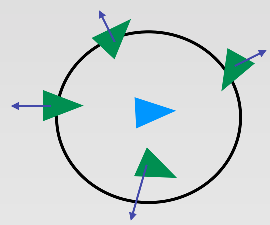
Separation

- Add force that steers body away from others in neighborhood



Separation

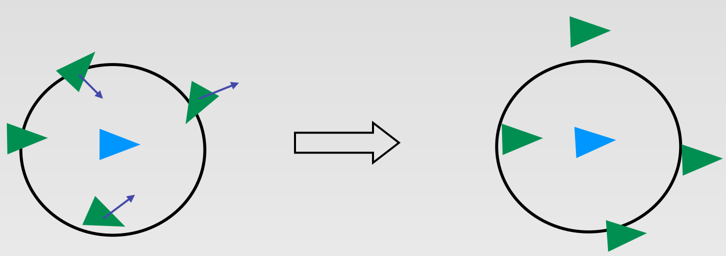
- Vector to each neighbor is normalized and divided by the distance (i.e., stronger force for closer neighbors)



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


Alignment

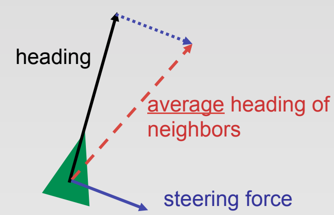
- Attempt to keep body's heading aligned with its neighbors headings



Alignment

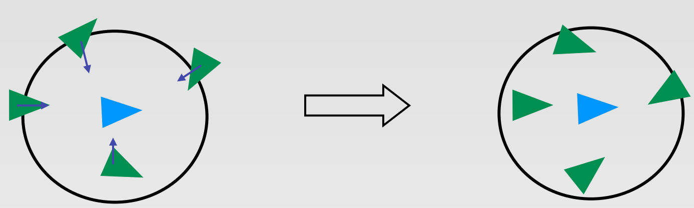
- Return steering force to correct towards *average* heading vector of neighbors

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

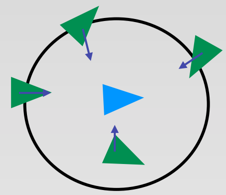


Cohesion

- Produce steering force that moves body 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

DEMO