Autonomous Movement

IMGD 4000

With material from: Millington and Funge, *Artificial Intelligence for Games*, Morgan Kaufmann 2009 (Chapter 3), Buckland, *Programming Game AI by Example*, Wordware 2005 (Chapter 3), http://opensteer.sourceforge.net and http://gamedevelopment.tutsplus.com/series/understanding-steering-behaviors--gamedev-12732

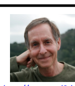
Introduction

- 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 core (along with shooting) → there is no higher level decision making!
- At other extreme (e.g., chess), no "movement" per se → pieces just placed

Note: as for pathfinding, we're going to treat everything in 2D, since most game motion in gravity on surface (i.e., 2 ½ D)

2

Craig Reynolds



Website: http://www.red3d.com/cwr/

- 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
    - Recognition of "his pioneering contributions to the development of three-dimensional computer animation for motion picture production"
  - **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
  - Now (2015) at *SparX* (eCommerce coding within *Staples*)

3

Outline

- Introduction (done)
- The "Steering" Model (next)
- Steering Methods
- Flocking
- Combining Steering Forces

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

Mechanics ("how") of motion
- Differs for characters, e.g., fish vs. horse (e.g., compare animations)
- Independent of steering

5

The "Steering" Model – Example

- Cowboys tend herd of cattle
- Cow wanders away
- Trail boss tells cowboy to fetch stray
- Cowboy says "giddy-up" and guides horse to cow, avoiding obstacles

Note, depending upon the game, player could control boss or cowboy (or both)!

- Trail boss decision represents action
  - Observes world – *cow is missing*
  - Setting goal – *retrieve cow*
- Steering done by cowboy
  - *Go faster, slower, turn right, left …*
- Horse implements locomotion
  - With signal, go in indicated direction
  - Account for mass when accelerating/turning
  - Provide animation

Action Selection

- Done through variety of means…
  - e.g., *decision tree* or *FSM*
  - (see earlier slide deck)
- Examples:
  - "Get health pack"
  - "Charge at enemy"
- Player input
  - "Return to base"
  - "Fetch cow"

Action Selection

Locomotion *Dynamics*

Steering

Locomotion

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

  // Orientation of body
  heading     // vector

  // Dynamic properties of body
  maxForce    // scalar
  maxSpeed    // scalar

  def update (dt) {
    force = ...;              // Combine forces from steering behaviors
    acceleration = force / mass;   // Update acceleration w/Newton's 2nd law
    velocity += truncate ( acceleration * dt, maxSpeed );  // Update speed
    position += velocity * dt;     // Update position
    if ( | velocity | > 0.000001 ) // If vehicle moving enough
      heading = normalize ( velocity ); // Update heading to velocity vector
    // render …
  }
```

```
// Scale vector to appropriate size (max)
vector truncate(vector v, int  max) {
  float f;
  f =  max / v.getLength();
  if (f < 1.0)
    f = 1.0
  v.scaleBy(f);
  return v;
}
```


Individual Steering "Behaviors"

Compute forces

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

Steering

Multiple behaviors combine forces (e.g., *flocking*)

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

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

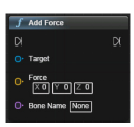
Adding Forces in UE4

Add force to a single rigid body

`virtual void AddForce (Fvector Force, Fname BoneName)`

- Force – force vector to apply. Magnitude is strength of force
- BoneName – name of body to apply it to ('None' to apply to root body)

```
void AMyCharacter::AddUpwardForce() {
  const float ForceAmount = 20000.0f;
  Fvector force(0.0f, 0.0f, ForceAmount);
  Fname bone; // defaults to "NAME_None"
  this->AddForce(force, bone);
}
```



Distance units are *centimeters*
i.e., earth gravity 981 cm/s^2

C++

Note: *max velocity* property of object

Blueprints

Steering Methods

```
class Body {
  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 (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; }
  ...
};
```

- Forces returned by each method are combined (shown later)
- Individual behaviors can be turned on/off (next slide)

Turning Steering Methods On & Off

- Action Selection controls which steering behaviors on/off

```
class Body {

private:
   bool seek_on;

public:
   void setSeek(bool on=true);
   bool doSeek();
   …
}
```

```
vector Body::calcForce() {
      if ( doSeek() ) {
         force += seek();
      …
}
```

13

Reference Code in C++

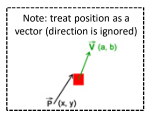
- Complete example code for this unit from Buckland's book can be downloaded from:
  http://samples.jbpub.com/9781556220784/Buckland_SourceCode.zip
  - Folder for Chapter 3
- See also learning guide's "Understanding Steering Behaviors":
  http://gamedevelopment.tutsplus.com/series/understanding-steering-behaviors--gamedev-12732
  - Similar concepts, slightly different code implementation

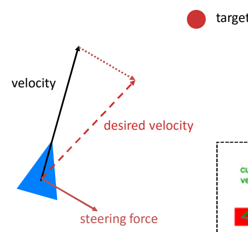
14

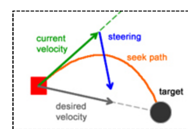
Outline

- Introduction (done)
- The "Steering" Model (done)
- Steering Methods (next)
- Flocking
- Combining Steering Forces

Seek: Steering Force







Will result in smooth path!

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

 // return steering force
 return desired - velocity;  // vector difference
}
```

DEMO
(Force: INS/DEL,
Speed: HOME/END)

16

Problem with Seek?

- What happens when reaches target?
- How bad is it?

17

Problem with Seek

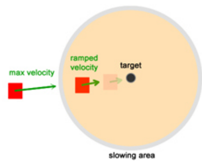
- What happens when reaches target?
  - Overshoots target
- How bad is it?
  - Amount of overshoot determined by ratio of maxSpeed to maximum force applied
- Intuitively, should decelerate as gets closer to target

  → Arrive

18

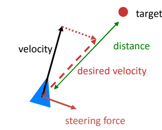
Arrive: Variant of Seek Behavior

- When body is far away from target, it behaves just like seek, i.e., closes at maximum speed



- Deceleration only when close to target, e.g., 'speed' reduced below 'maxSpeed' when within range

Arrive




```
def arrive (target) {

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

  // slow down linearly with distance.
  // DECELERATION allows tweaking (larger is slower)
  speed = distance / DECELERATION;

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

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

  // return steering force as in seek (note, if heading
  // directly at target already, this just decelerates)
  return desired - velocity;
}
```

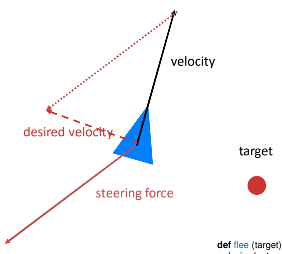
Note, when at target, desired velocity is zero.
→ Steering force becomes -velocity
Added to force, stops moving!

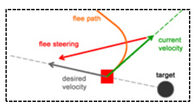
DEMO

Decelerates linearly.
Example: Max speed 4, initial distance 10.

	DECELERATION 2	DECELERATION 1
Dist	Speed	Speed
10	4	4
9	4	4
8	4	4
7	3.5	4
6	3	4
5	2.5	4
4	2	4
3	1.5	3
2	1	2
1	0.5	1
0	0	0

Flee: Opposite of Seek





Produces curved (orange) path

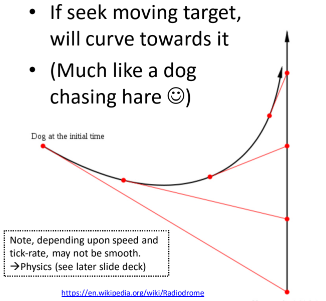
```
def flee (target) {
  desired = truncate ( position - target, maxSpeed );
  return desired - velocity;
}
```

Note: Buckland adds "range" to only flee if near, but that is really an Action Selection decision.

DEMO

Seek and Ye Shall Find?

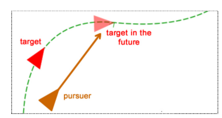
- If seek moving target, will curve towards it
- (Much like a dog chasing hare ☺)



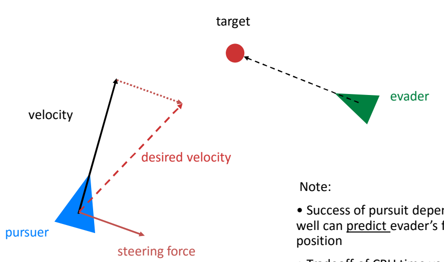
Note, depending upon speed and tick-rate, may not be smooth.
→Physics (see later slide deck)

https://en.wikipedia.org/wiki/Radiodrome

- Instead, seek to target location *in the future*



Pursue: Seek Predicted Position (1 of 2)



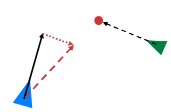
Note:

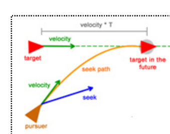
- Success of pursuit depends on how well can predict evader's future position
- Tradeoff of CPU time vs. accuracy
- *Special case:* if evader almost dead ahead, just seek

Pursue: Seek Predicted Position (2 of 2)

Longer distance, then higher time (dt)
→ Pursuer seek point far ahead
And vice-versa

```
def pursue (body) {

  toBody = body.position - position;

  // if within ~20 degrees ahead, simply seek
  facing = computeFacing(heading, body);
  if ( facing  > -10 && facing < -10 )
    return seek ( body.position );

  // calculate lookahead time based on distance and speeds
  // note: this could be hardcoded (e.g., 100 ms) or use more
  // sophisticated prediction
  dt = | toBody | / ( maxSpeed + | body.velocity | );

  // seek predicted position, assuming body moves in straight line
  // note: again, this could use more sophisticated prediction
  return seek ( body.position + ( body.velocity * dt ) );
}
```

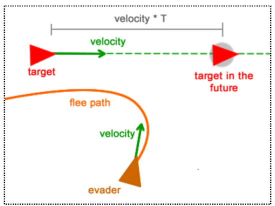




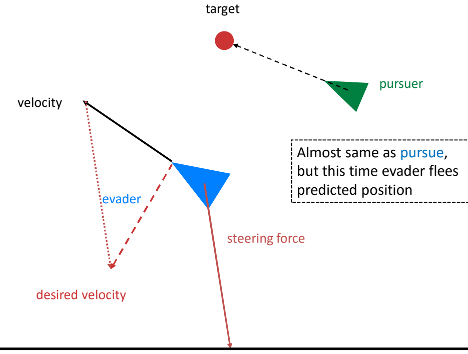
DEMO

Don't Just Flee, Evade!

- Predict where target will be
- Move in opposite direction

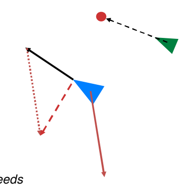


Evade: Opposite of Pursue (1 of 2)



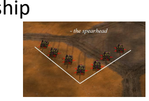
Evade: Opposite of Pursue (2 of 2)

```
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 (1 of 2)

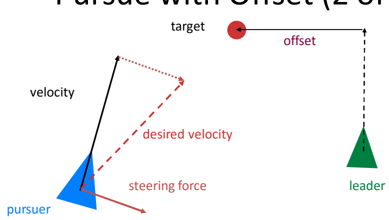
- What if don't want to intercept, but be near?
  - Marking an opponent in sports
  - Staying docked with moving spaceship
  - Shadowing an aircraft
  - Implementing battle formations



CC Generals

- Solution → Pursue with Offset
  - Steering force to keep body at specified offset from target body
- (This is not "flocking", which we will see later)

Pursue with Offset (2 of 2)




```
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 (1 of 3)

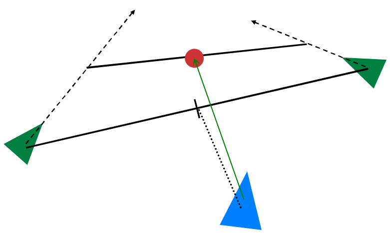
- Similar to pursue
- Return steering force to move body to midpoint of imaginary line connecting two bodies
- Useful for:
  - Bodyguard taking a bullet
  - Soccer player intercepting pass
- Like pursue, main trick is to estimate lookahead time (dt) to predict target point

Interpose (2 of 3)

(1) Bisect line between bodies

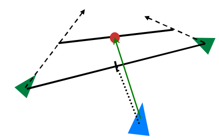
(2) Calculate dt to bisection point

(3) Target arrive at midpoint of predicted positions



Interpose (3 of 3)

```
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 );
}
```

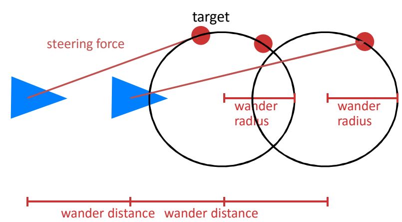


DEMO

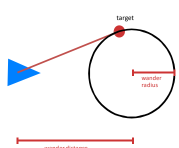
Wander

- Goal is to produce steering force which gives impression of random walk though agent's environment
- Naive approach:
  - Calculate *random steering force* each update step
  - Produces unpleasant "jittery" behavior
- Reynold's approach:
  - Project circle in front of body
  - Steer towards *randomly moving target* constrained along perimeter of the circle

Wander



Wander



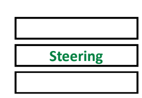

```
// initial random point on circle
wanderTarget = ...;

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

Individual Steering "Behaviors"

Compute forces

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

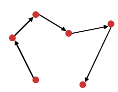


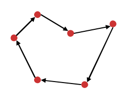
Multiple behaviors combine forces

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

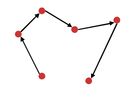
A path can be described by an array of vectors.

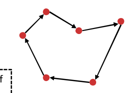
37

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);
}
```





Sensitive to SEEK_DISTANCE and ratio of maxForce to maxSpeed (in underlying locomotion model)
- tighter path for interior corridors
- looser for open outdoors

DEMO

38

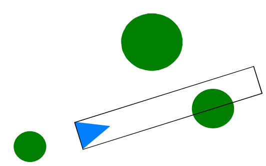
Mini-Outline

- Interacting with the Environment
  - Obstacle Avoidance
  - Hide
  - Wall Avoidance

39

Obstacle Avoidance

- Treat obstacles as circular bounding volumes
- *Basic idea:* extrude "detection box" (width of body, length proportional to speed) in front of body in direction of motion (like intersection testing)

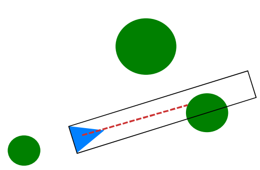


40

Obstacle Avoidance Algorithm Overview

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

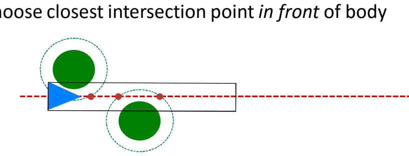
(expand each next)



41

Obstacle Avoidance Algorithm (1 of 3)

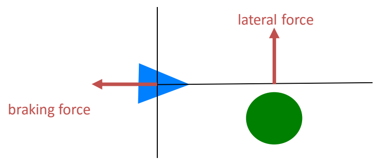
1. Find closest intersection point
   (a) discard all obstacles which do not overlap with detection box
   (b) expand obstacles by half width of detection box
   (c) find intersection points of trajectory line and expanded obstacle circles
   (d) choose closest intersection point *in front* of body



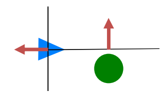
42

Obstacle Avoidance Algorithm (2 of 3)

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



Obstacle Avoidance Algorithm (3 of 3)



```
def computeAvoidForce ( closestObstacle ) {

  // convert to "local" space, so object is at origin

  // the closer it is, the stronger the force away
  multiplier = 1 + ( box.getLength() – closestObstacle.getX() ) / box.getLength()

  // calculate lateral force
  force.y = ( closestObstacle().getRadius() – closestObstacle().getY() ) * multiplier

  // apply braking force proportional to obstacles distance
  brakingWeight = 2.0
  force.x = ( closestObstacle().getRadius() – closestObstacle.getX() ) * brakingWeight

  // convert vector back to world space
  return vectorToWorld ( force )
}
```

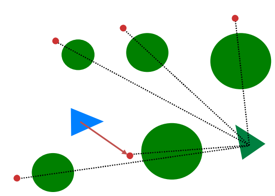
DEMO

Hide

- Attempt to position body so 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)

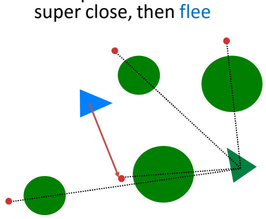
Hide

for each obstacle, determine hiding spot (projected point opposite each obstacle)
if no hiding spots **then** invoke 'evade'
else invoke 'arrive' to closest hiding spot



Hide - Possible Refinements

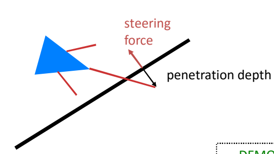
- Action selection decisions to …
- Only hide if can "see" other body
  - tends to look dumb (i.e., agent has no memory)
  - can improve by adding time constant, e.g., hide if saw other body in last <n> seconds
- Only hide if can "see" other body *and* other body can "see" you
- Add "panic distance" so if super close, then flee



DEMO

Wall Avoidance

1. Test for intersection of three "feelers" with wall (like cat whiskers)
2. Calculate *penetration depth* of closest intersection
3. Return steering force perpendicular to wall with magnitude equal to penetration depth



DEMO

Outline

- Introduction (done)
- The "Steering" Model (done)
- Steering Methods (done)
- Flocking (next)
- Combining Steering Forces

"Flocking" = *Group* Steering Behaviors

- Combination of three steering behaviors:
  - cohesion
  - separation
  - alignment

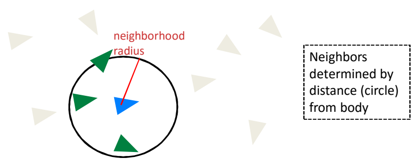
DEMO

- Each applied to all bodies based on neighbors

(next)

50

Neighbors



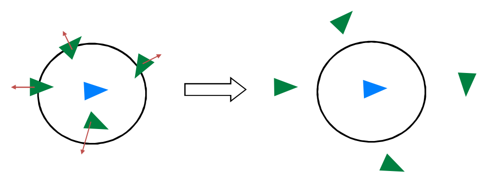
- Variation:
  - Restrict neighborhood to field of view (e.g., 180 deg.) in *front*

  (May be more realistic in some applications)

51

Separation (1 of 2)

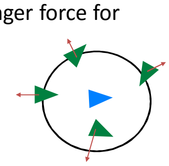
- Add force that steers body away from others in neighborhood



52

Separation (2 of 2)

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



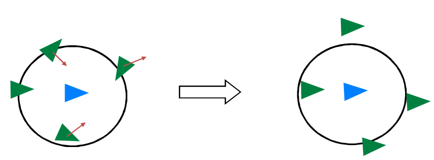
```
def separation () {
  force = [0,0];
  for each neighbor
    direction = position - neighbor.position;
    force += normalize(direction) / | direction |;
  return force;
}
```

Divide by bigger number when farther, smaller number when closer

53

Alignment (1 of 2)

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

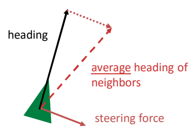


54

Alignment (2 of 2)

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

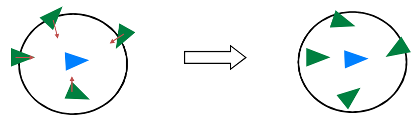
```
def alignment () {
  average = [0,0];
  for each neighbor
    average += neighbor.heading;
  average /= |neighbors|;
  return average - heading;
}
```



55

Cohesion

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



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

56

Flocking Force Combination

- Combine flocking forces with weights
  - Different weights give different behaviors
  - (Related to next topic)
- Note, if isolated neighbor out of range, will do nothing
  - Add "wander" behavior

```
def flock () {
  vector force = [0,0];
  vector force =
        separation() * separation_weight
      + alignment() * alignment_weight
      + cohesion() * cohesion_weight
      + wander() * wander_weight;
  return force;
}
```

DEMO

Flocking – Summary

- An "emergent behavior"
  - Looks complex and/or purposeful to observer
  - But actually driven by fairly simple rules
  - Component entities don't have "big picture"
- Tunable to different kinds of flocks
- Often used in films
  - Bats and penguins in *Batman Returns*
  - Orc armies in *Lord of the Rings*

58

Outline

- Introduction (done)
- The "Steering" Model (done)
- Steering Methods (done)
- Flocking (done)
- Combining Steering Forces (next)

Combining Steering Behaviors: Examples

- FPS bots
  - Path following (point A to point B)
  - Obstacle avoidance (crates, barrels)
  - Pursue with offset (formation)
  - Separation
- Animal simulation (e.g., sheep in RTS)
  - Wander
  - Obstacle avoidance (e.g., trees)
  - Flee (e.g., predator)

60

Combine Steering Forces

```
vector Body::calcForce() {
    vector force;
    force += wander();
    force += avoidObstacles();
    force += …
    return truncate ( force, maxForce );
}
```

```
class Body {
  def update (dt) {
    force = calcForce();
    …
  }

  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; }
  ...
};
```

Other choices for combination?

61

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*

62

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

```
vector Body::calcForce() {
    vector force;
    force += wander() * wander_weight;
    force += avoidObstacles() * avoid_weight;
    force += …
    return truncate ( force, maxForce );
}
```

63

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
  - e.g., vehicle by wall and neighbors – separation force may be great so hits wall. If tweak avoid wall weight higher, when alone near wall may act odd
- Note: can work well in limited cases (e.g., flocking) where there are few conflicts

64

Prioritized Steering

- Intuition: Many of steering behaviors only return force in appropriate conditions
  - e.g., vehicle with separation, alignment, cohesion, wall avoidance, obstacle avoidance. Should give priority to wall avoidance and obstacle avoidance.
- 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*
    - Helps with consistent behavior
    - Plus saves CPU

DEMO – Big Shoal

65

Prioritized Steering – Variation

1. Add force. If less than maxForce, continue. Otherwise, stop evaluation and apply force.
   - Additional variation can apply weights to forces

```
vector Body::calcForce() {
    vector force;
    force += avoidObstacles() * avoid_weight;
    if ( magnitude (force) >= maxForce )
      return truncate ( force, maxForce );
    force += wander() * wander_weight;
    if ( magnitude (force) >= maxForce )
    …
}
```

2. Define <u>groups</u> of behaviors with <u>blending inside</u> each group and <u>priorities between</u> groups

Prioritized Dithering (Reynolds)

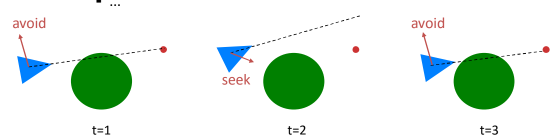
- In addition to priority order, associate a probability with each steering method
- Use random number and probability to sometimes skip some methods in priority order (on some ticks)
- Gives lower priority methods some influence without problems of blending

```
vector Body::calcForce() {
  vector force;
  prob_avoid = 0.9;
  prob_wander = 0.2;

  if ( random ( 0-1 ) < prob_avoid) {
    force += avoidObstacles() * avoid_weight;
    if ( magnitude (force) >= maxForce )
      return truncate ( force, maxForce );
  }
  if ( random ( 0-1 ) < prob_wander) {
    force += wander() * wander_weight;
    if ( magnitude (force) >= maxForce )
    …
  }
}
```

67

Another Problem – Judder

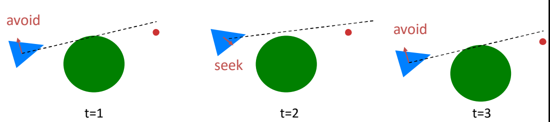
- Conflicting behaviors can alternate, causing "judder" (jitter/shudder – note, usually *slight*)
  - e.g., avoidObstacle and seek
    - avoidObstacle forces away from obstacle until it is out of range
    - seek pushes back into range
    - …



68

Judder Solution – Smoothing

- 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)
  - → Smaller oscillations
  - Not perfect solution, but produces adequate results at low cost



DEMO – Big Shoal vs. Another Big Shoal with Smoothing