Autonomous Movement

IMGD 4000


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., 2½ D)

Craig Reynolds

- The “giant” in this area — his influence cannot be overstated
- 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”
- Left U.S. R&D group of Sony Computer Entertainment in April 2012 after 13 years
- Now (2015) at SparX (eCommerce coding within Staples)

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”
    - “go to A, B, and then C”
  - Calculate trajectories to satisfy goals and plan
  - Produce steering force that determines where and how fast character moves
- Steering
  - Mechanics (“how”) of motion
  - Differs for characters, e.g., fish vs. horse (e.g., compare animations)
  - Independent of steering
- Locomotion

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
- 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

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

Locomotion Dynamics

- Class `Body`
  - Position of rigid body
  - Velocity // vector
  - Orientation of body
  - Heading // vector

- Dynamic properties of body
  - Mass // scalar
  - Max Speed // scalar

- Method `update` (dt)
  - Force = ...
  - // Combine forces from steering behaviors
  - Acceleration = Force / Mass; // Update acceleration
  - Velocity += Acceleration * dt; // Update speed
  - Position += Velocity * dt; // Update position

- Methods:
  - Seek // Vector to target
  - Flee // Vector from target
  - Arrive // Vector from target
  - Pursue // Vector from (target - seeker)
  - Evade // Vector from (target - seeker)
  - Interpose // Vector from (target - seeker)
  - Wander // Vector from [0 0 0]
  - Avoid Obstacles // Vector from (target - seeker)

- 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)

- Individual Steering “Behaviors”

  Compute forces
  - Seek, Flee, Pursue, Evade, Wander, Interpose, Avoid Obstacles, Follow Path

  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

- Steering Methods

  - Forces returned by each method are combined.
  - Individual behaviors can be turned on/off (next slide)
Turning Steering Methods On & Off

- **Action Selection** controls which steering behaviors on/off

```cpp
class Body {
  private:
    bool seek_on;
  public:
    void setSeek(bool on=true);
    bool doSeek();
    ...
}
```

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](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](http://gamedevelopment.tutsplus.com/series/understanding-steering-behaviors--gamedev-12732) – Similar concepts, slightly different code implementation

Outline

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

Problem with Seek?

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

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**
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

```
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;
}
```

---

Flee: Opposite of Seek

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

Seek and Ye Shall Find?

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

```
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));
}
```

---

Pursue: Seek Predicted Position (2 of 2)

Note:
- Calculation of look ahead time based on distance and speeds
- Note: CPU time is also taken into consideration (e.g., 100 ms) in actual implementation
- Special case: if pursuer already in front of evader, just seek

DEMO
Don’t Just Flee, Evade!

- Predict where target will be
- Move in opposite direction

Evade: Opposite of Pursue (1 of 2)

Almost same as pursue, but this time evader flees predicted position

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
- 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));
}

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 \( \text{arrive} \) at midpoint of predicted positions

\[
\text{def } \text{interpose}(\text{body1}, \text{body2}) \{
\begin{align*}
&\text{// lookahead time to current midpoint} \\
&\quad dt = \frac{\text{body1.position} + \text{body2.position}}{2 \times \text{maxSpeed}}; \\
&\text{// extrapolate body trajectories} \\
&\quad \text{position1} = \text{body1.position} + \text{body1.velocity} \times dt; \\
&\quad \text{position2} = \text{body2.position} + \text{body2.velocity} \times dt; \\
&\text{// steer to midpoint} \\
&\quad \text{return } \text{arrive}(\frac{\text{position1} + \text{position2}}{2});
\end{align*}
\]

Interpose (3 of 3)

\[
\text{def } \text{interpose}(\text{body1}, \text{body2}) \{
\begin{align*}
&\text{// lookahead time to current midpoint} \\
&\quad dt = \frac{\text{body1.position} + \text{body2.position}}{2 \times \text{maxSpeed}}; \\
&\text{// extrapolate body trajectories} \\
&\quad \text{position1} = \text{body1.position} + \text{body1.velocity} \times dt; \\
&\quad \text{position2} = \text{body2.position} + \text{body2.velocity} \times dt; \\
&\text{// steer to midpoint} \\
&\quad \text{return } \text{arrive}(\frac{\text{position1} + \text{position2}}{2});
\end{align*}
\]

Wander

- Goal is to produce steering force which gives impression of random walk through agent’s environment
- Naïve approach:
  - Calculate random steering force each update step
  - Produces unpleasant “jittery” behavior
- Reynolds’s approach:
  - Project circle in front of body
  - Steer towards randomly moving target constrained along perimeter of the circle

\[
\text{def } \text{wander}() \{
\begin{align*}
&\text{// initial random point on circle} \\
&\quad \text{wanderTarget} = \ldots; \\
&\text{// displace target random amount} \\
&\quad \text{wanderTarget} += \text{[random(0, JITTER), random(0, JITTER)]}; \\
&\text{// project target back onto circle} \\
&\quad \text{wanderTarget} = \text{wanderTarget} \cdot \text{RADIUS}; \\
&\text{// move circle wander distance in front of agent} \\
&\quad \text{wanderTarget} += \text{bodyToWorldCoord(\text{DISTANCE, 0})}; \\
&\text{// steer towards target} \\
&\quad \text{return } \text{wanderTarget} - \text{position};
\end{align*}
\]

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

A path can be described by an array of vectors.

Path Following: Using Seek
• Invoke ‘seek’ on each waypoint until ‘arrive’ at finish (if any)

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

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

Mini-Outline
• Interacting with the Environment
  – Obstacle Avoidance
  – Hide
  – Wall Avoidance

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)

Obstacle Avoidance Algorithm Overview
1. Find closest intersection point
2. Calculate steering force to avoid obstacle (expand each next)

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
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)

```java
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 )
}
```

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)
  – to avoid being shot by player
  – can improve by adding time constant

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

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
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

Neighbors

• Variation:
  — Restrict neighborhood to field of view (e.g., 180 deg.) in front
  (May be more realistic in some applications)

Separation (1 of 2)

• Add force that steers body away from others in neighborhood

Separation (2 of 2)

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

Alignment (1 of 2)

• Attempt to keep body’s heading aligned with its neighbors headings

```python
def separation(p: np.ndarray):  # assuming p is a list of neighbors
    force = np.zeros(2)
    for neighbor in p:
        direction = p - neighbor
        force += np.linalg.norm(direction) / np.linalg.norm(direction)
    return force
```
Alignment (2 of 2)

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

```python
def alignment(def 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

```python
def cohesion(def center = [0.0];
for each neighbor
  center += neighbor.position;
center /= |neighbors|;
return seek(center);
```}

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

```python
def flock(def vector force = [0.0];
vector force =
  separation() * separation_weight
+ alignment() * alignment_weight
+ cohesion() * cohesion_weight
+ wander() * wander_weight;
return force;
```}

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

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)
Combine Steering Forces

```c
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?

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:
  \[ w_1 F_1 + w_2 F_2 + w_3 F_3 + \ldots \]
  - weights do not need to sum to 1
  - weights tuned by trial and error
- Final result will be limited (truncated) by \( \text{maxForce} \)

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

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

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

```
class Body {
  calcForce () {
    vector force = avoidObstacles() * avoid_weight;
    return truncate (force, maxForce);
  }
```

Prioritized Steering – Variation

1. Add force. If less than \( \text{maxForce} \), continue. Otherwise, stop evaluation and apply force.
   - Additional variation can apply weights to forces
2. 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

```
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;
  }
}
```

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
    - ...

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

```
avoid seek avoid
```

DEMO – Big Shoal vs. Another Big Shoal with Smoothing