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)

Outline

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

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)
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
- Mechanics (“how”) of motion
- Differs for characters, e.g., fish vs. horse (cf. 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 the cow, avoiding obstacles
  - Trail boss 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

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
// 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 with 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 along velocity vector
    // render ...
}
Individual Steering “Behaviors”

Compute forces

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

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

Adding Forces in UE4 (1 of 2)

1. Radial Force Actor can apply Constant Force, which acts every frame
   - Set force strength in editor
2. Radial Force Actor can also apply Impulse
   - Acts for only one tick
   - Done with blueprints or AddImpulse() function
   - Example for impulse for explosions:
     https://www.youtube.com/watch?v=IlkNCUtqTGU
3. Physics Thrusters, type of actor that simulates directional force on paired physics object
   - Example - thruster for rocket ship
4. AddForce() to apply any type of force other than impulse
   - (Next slide)

For steering, #1, #2 and #4 all options...

Adding Forces in UE4 (2 of 2)

Add force to a single rigid body

virtual void AddForce (FVector Force, FName BoneName)

- Force – force vector to apply. Magnitude is strength
- 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);
    bone; // defaults to “None”
    this->AddForce(force, bone);
}

C++

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 evade (body) { ... return force; }
    def interpose (body1, body2) { return force; }
    def wander () { ... return force; }
    def avoidObstacles () { ... return force; }
};

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();
        }
        return force; }

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.comseries/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
Seek: Steering Force

```python
def seek(target):
    # vector from here to target scaled by maxSpeed
    desired = truncate(target - position, maxSpeed);
    # return steering force
    return desired - velocity;  # vector difference
```

Problem with Seek

- Overshoots target
- 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., it closes at maximum speed

```
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;
```

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;
```
Flee: Opposite of Seek

- Produces curved (orange) path
- def flee (target) {
  desired = truncate ( position - target, m
  return desired - velocity;
}

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

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 10 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
  // note: this could be hardcoded (e.g., 100 ms) or use more sophisticated prediction
  dt = toBody / ( maxSpeed + abs ( 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 ) );
}

Evade: Opposite of Pursue (1 of 2)

- Almost same as Pursue, but this time evader flees predicted position

- def evade (target) {
  desired = truncate ( position - target, m
  return desired - velocity;
}

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)

- 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 10 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
  // note: this could be hardcoded (e.g., 100 ms) or use more sophisticated prediction
  dt = toBody / ( maxSpeed + abs ( 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 ) );
}
Evade: Opposite of Pursue (2 of 2)

```python
def evade(body):
    toBody = body.position - position;
    // no special case check for dead ahead
    dt = |toBody| / (maxSpeed + |body.velocity|);
    // flee predicted position
    return flee(body.position + (body.velocity * dt));
```

Pursue with Offset (1 of 2)

- Steering force to keep body at specified offset from target body
- Useful for:
  - Marking an opponent in sports
  - Staying docked with moving spaceship
  - Shadowing an aircraft
  - Implementing battle formations
- (This is not “flocking”, which we will see later)

Pursue with Offset (2 of 2)

```python
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

Interpose (3 of 3)

```python
def interpose(body1, body2):
    # lookahead time to current midpoint
    $dt = |\text{body1}.position + \text{body2}.position| / (2 \times \text{maxSpeed})$;
    # extrapolate body trajectories
    position1 = \text{body1}.position + \text{body1}.velocity * $dt$;
    position2 = \text{body2}.position + \text{body2}.velocity * $dt$;
    # steer to midpoint
    return $\text{arrive} (\text{position1} + \text{position2}) / 2$;
```

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

Path Following: Using Seek

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

```python
path = ...; // (circular) list of waypoints
current = \text{path}.first(); // current waypoint vector

def followPath() {
    if ( |current - position| < SEEK_DISTANCE )
        if ( \text{path}.isEmpty() )
            return $\text{arrive} (\text{current})$;
        else
            current = \text{path}.next();
    return $\text{seek} (\text{current})$;
}
```
Individual Steering “Behaviors”

Compute forces

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

Multiple behaviors combine forces

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

```python
// 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;
}
```
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)

```python
def computeAvoidForce (closestObstacle) {
  // convert to “local” space, so object is at origin
  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.get() ) * 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 - 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

  ![Diagram showing hide refines]

- Add “panic distance” so if super close, then flee

**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

![Diagram showing wall avoidance]

**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

- Each applied to all bodies based on neighbors (next)
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)

```python
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

```python
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

```python
def cohesion () {
    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 () {
    vector force = [0,0];
    vector force = separation() * separation;
    vector force = alignment() * a;
    vector force = cohesion() * co;
    vector force = wander() * wand;
    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)
  – Separation (to do)

• Animal simulation (e.g., sheep in RTS)
  – Wander
  – Obstacle avoidance (e.g., trees)
  – Flee (e.g., predator)

Combine Steering Forces

class Body {
    def update (dt) {
        force = calcForce();
        ...
    }
    def seek (target) {
        return truncate (force, ...
    }
    def flee (target) {
        return truncate (force, ...
    }
    def arrive (target) {
        return truncate (force, ...
    }
    def pursue (body) {
        return truncate (force, ...
    }
    def evade (body) {
        return truncate (force, ...
    }
    def hide (body) {
        return truncate (force, ...
    }
    def interpose (body1, body2) {
        return truncate (force, ...
    }
    def wander () {
        return truncate (force, ...
    }
    def avoidObstacles () {
        return truncate (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:
  \[ w_1 F_1 + w_2 F_2 + w_3 F_3 + ... \]
  - weights do not need to sum to 1
  - weights tuned by trial and error
- Final result will be limited (truncated) by `maxForce`

```cpp
vector Body::calcForce() {
    vector force;
    force += wander() * wander_weight;
    force += avoidObstacles() * avoid_weight;
    force += ...
    return truncate ( force, 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
  - 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* (saves CPU)

```cpp
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 )
        return truncate ( force, maxForce );
    return truncate ( force, maxForce );
}
```

Prioritized Steering – Variation

1. Add force. If less than `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) < 0.9) {
    force += avoidObstacles() * avoid_weight;
    if ( magnitude (force) >= maxForce )
      return truncate ( force, maxForce );
  }
  if ( random (0-1) < 0.2) {
    force += wander() * wander_weight;
    if ( magnitude (force) >= maxForce )
      ...
  }
  return force;
}
```

Another Problem - Judder

- Conflicting behaviors can alternate, causing “judder” (jitter/shudder – note, usually slight)
  - e.g., avoidObstacle and seek
    - avoidObstacle forces you away from obstacle until it is out of range
    - seek pushes you 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
t=1 t=2 t=3
```

DEMO – Big Shoal vs. Another Big Shoal with Smoothing