Game AI

Ming-Hwa Wang, Ph.D.
COEN 396 Interactive Multimedia and Game Programming
Department of Computer Engineering
Santa Clara University

Introduction to Game AI

- artificial intelligence
  - strong AI: create intelligent machine, gain better insight into the nature of human intelligence, conscience, emotions, solve problems, learn and adapt
  - weak AI: give machines specialized intelligent qualities
  - game AI: a weak AI which gives non-player characters human-level intellect, appearance of having different personalities, portraying emotions or various dispositions, etc., making game more immersive, challenging and fun
- game AI techniques:
  - deterministic: deterministic behavior or performance is specified and predictable, and there is no uncertainty; fast, easy to implement, understand, test, and debug, limits a game's play-life
  - nondeterministic: has a degree of uncertainty and is somewhat unpredictable; difficult to test and debug
  - hybrid: isolate the parts of deterministic and nondeterministic
- the most widely used AI trick in games is cheating, the computer team can have access to all information on it human opponents (privacy issue?) and helps give the computer an edge against intelligent human players; however, too much cheating causes player lost interest, cheating must be balanced to create just enough of a challenge for the player to keep the game interesting and fun
- related websites
  http://www.gameai.com
  http://www.aiguru.com
  http://www.gamasutra.com
  http://www.gamedev.net
  http://www.ai-depot.com
  http://www.generation5.org
  http://www.aaai.org
  http://www.oreilly.com/catalog/ai

Chasing and Evading

- the chasing and evading problem (evading is virtually the opposite of chasing) consists of 3 parts: the decision to initiate a chase or evade, the effecting the chase or evade, and obstacle avoidance
- the environment can be either tiled or continuous
  - tile-based: the game domain is divided into discrete tiles, and the player's position is represented by integers, the position is fixed to a discrete tile, movement goes tile by tile, the number of directions in which the player can make is limited, and characters appear to move faster when moving along a diagonal path
  - continuous environment: position is represented by floating-point coordinates, which can represent any location in the game domain, and the player also is free to head in any direction
- basic chasing: correcting the predator's coordinates based on the prey's coordinates so as to reduce the distance between their positions, the chasing or evading seems almost too mechanical

```
// basic chasing
if ( predatorX > preyX ) {
   predatorX--;
} else if ( predatorX < preyX ) {
   predatorX++;
}
if ( predatorY > preyY ) {
   predatorY--;
} else if ( predatorY < preyY ) {
   predatorY++;
}
```

- line-of-sight chasing appears more natural, direct, and intelligent than basic chasing
  - in a tiled-base game, there are only 8 possible direction of movements, to minimize the jaggy and sometimes jumpy appearance, it is important to move only adjacent tiles when changing positions
  - the Bresenham's algorithm will never draw 2 adjacent pixels along a line's shortest axis, and it will walk along the shortest possible path between the starting and ending points
  - once the target moves, the pre-calculated path becomes obsolete, and need to be re-calculated
  - the algorithm will determine which axis is the longest, then traverse the long axis, calculating each point of the line along the way

```
// path direction calculation
deltaRow = endRow - currRow;
deltaCol = endCol - currCol;
stepRow = ( deltaRow < 0 ) ? -1 : 1;
stepCol = ( deltaCol < 0 ) ? -1 : 1;
deltaRow = abs(deltaRow * 2);
deltaCol = abs(deltaCol * 2);
// path initialization
for ( currStep = 0; currStep < kMaxPathLength; currStep++ ) {
   pathRows[currStep] = pathCols[currStep] = -1;
}
currStep = 0;
pathRows[currStep] = currRow;
pathCols[currStep] = currCol;
currStep++;
// bresenham algorithm
if ( daltaCol > deltaRow ) {
   fraction = deltaRow * 2 - daltaCol;
   while ( currCol != endCol ) {
```

```
        if ( fraction >= 0 ) {
            currRow += stepRow;
            fraction -= deltaCol;
        }
        currCol += stepCol;
        fraction += deltaRow;
        pathRow[currentStep] = currRow;
        pathCol[currentStep++] = currCol;
    }
} else {
    fraction = deltaCol * 2 - daltaRow;
    while ( currRow != endRow ) {
        if ( fraction >= 0 ) {
            currCol += + stepCol;
            fraction -= deltaRow;
        }
        currRow += stepRow;
        fraction += deltaCol;
        pathRow[currentStep] = currRow;
        pathCol[currentStep++] = currCol;
    }
}
```

- in a continuous environment, use physics engine, where the game entities are driven by applied forces and torques
- a simple 2-dimensional rigid-body physics engine, the line-of-sight algorithm controls the predator by applying thrust for forward motion and activate steering forces for turning so as to keep its heading toward the prey at all times
- the predator can't always turn on a dime, but turning radius is a function of their linear speed
- global and local coordinate systems
    - a global (earth-fixed) coordinate system is fixed and doesn't move, whereas a local (body-fixed) coordinate system rotates with the object to which it is attached and is locked onto objects that move around within the global coordinate system
    - x = X cosQ + Y sinQ; y = -X sinQ + Y cosQ;
    - where (x, y) are the local coordinates of the global point (X, Y)

```
// line-of-sight chasing
left = right = false;
v = vRotate2D(
    -predator.fOrientation, prey.vPosition - predator.vPosition
);
v.normalize( );
if ( v.x < -TOL ) {
    left = true;
} else if ( v.x > TOL ) {
    right = true;
}
predator.setThrusters(left, right);
```

- the unit vector v points from the predator directly toward the prey
- if the x-position of the prey, in terms of the predator's local coordinate system, is negative, the prey is somewhat to the starboard side of the predator (turn right); if the prey's x-coordinate is positive, it is somewhat on the port side of the predator (turn left)
- the predator always head directly toward the prey and most likely end up right behind it, unless it is moving so fast that it overshoots the prey, in which case it will loop around and head toward the prey again
- to prevent overshooting, use speed control logic to allow the predator to slow down as it gets closer to the prey
- heading directly toward the prey is not always the shortest path in terms of range to target, or, perhaps, time to target

Pattern Movement

- pattern movement in tiled environment: use Bresenham's line algorithm to calculate various patterns of movement by only providing vertices, paths will be made up of line segments, each new segment will begin where the previous one ended, the last segment ends where the first one begins to make the troll moves in a repeating pattern

```
// normalizePattern
origRow = pathRows[0];
origCol = pathCols[0];
for ( i = 0; i < pathSize; i++ ) {
pathRows[i] -= origRow;
pathCols[i] -= origCol;
}
```

    - the patterns are represented in terms of relative coordinates instead of absolute coordinates, so the pattern is not tied to any specific starting position in the game world, once the pattern is built and normalized, we can execute it from anywhere
    - at certain points along the track there is more than one valid direction for the troll, randomly select a new coordinate from the valid points found, to avoid abrupt back-end-forth movement, track the troll's previous location and exclude that position when building the array of valid moves
- pattern movement in physically simulated environment: do not specify an object's position explicitly during the simulation, only need to apply appropriate control forces to the object to drive it to where you want it to go
    - control forces have to act over time to effect the desired motion, the pattern array would be huge because the time steps typically taken in a physics-based simulation are very small, to get around this, the pattern array also contains information of how long the forces should be applied, the physics engine processes those instructions each time step through the simulation until the conditions specified in the given set of instructions are meet, then the next set of instructions from the pattern array are selected and applied

- to make smooth turn and avoid overshooting, start the turn with full force but then gradually reduce the steering force as get closer to the heading change target
- if in the midst of executing the patrol pattern, other logic in the game detects an enemy in the patrol area, the troller will stop patrolling and begin chasing the enemy

Flocking

move in cohesive groups rather than independently, grazing in a flock rather than walking aimlessly

- classic flocking 3 elegantly simple rules:
  - cohesion: have each unit steer toward the average position of its neighbors
  - alignment: have each unit steer so as to align itself to the average heading of its neighbors
  - separation: have each unit steer to avoid hitting its neighbors
- treating the units as rigid bodies enables you to take care of orientation rather than use particles to represent the units which you don't have to worry about rotation
- the visibility arc is defined by 2 parameters: the arc radius (r) and the angle (θ)
  - large radius will allow the unit to see more of the group and results in a more cohesive flock, a smaller radius tends to increase the likelihood of temporarily lose sight of their neighbors and results in a larger flock with possible detached units which need to be re-join, navigating around obstacles also can cause a flock to break up
  - and angle θ measures the field of view of each unit, a view of 360 degree is easier to implement but the resulting flocking behavior might be somewhat unrealistic, a more common field of view is having a distinct blind spot behind each unit, the very narrow field of view will degenerate a flock into units that one follows another and form a curved line
- steer model: each rule makes a small contribution to the total steering force and the net result is applied to the unit, require some tuning to make sure no single rule dominates, tuning is done by trial and error
- we want the avoidance rule steering force contribution to be small when the units are far away from each other, but we want the avoidance rule steering force contribution to be relatively large when the units are dangerously close to each other, the avoidance steering force is inversely proportional to the separation distance
- for alignment we will consider the angle between a given unit's current heading relative to the average hading of its neighbors, if the angle is small, we want to make only a small adjustment to its heading, whereas if the angle is large, a large adjustment is required
- the arrangement of the units in a flock will change constantly, each unit must update its view of the world each time through the game loop, we must cycle through all the units in acquire each unit's unique perspective, this neighbor search can become computational expensive as the number of units grow large

- field-of-view check:

```
if ( wideView ) {
    inView = ((w.y > 0) || (w.y < 0) && (fabs(w.x) > fabs(w.y) *
backViewAngle));
}
if ( limitedView ) {
    inView = (w.y > 0);
}
if ( narrowView ) {
    inView = ((w.y > 0) && (fabs(w.x) < fabs(w.y) * backViewAngle));
}
```

- cohesion rule:

```
if ( doFlock && (n > 0) ) {
    pAve = pAve / n;
    v = unit[i].vVelocity;
    v.normalize( );
    u = pAve - units[i].vPosition;
    u.normalize( );
    w = vRotate2D(-units[i].fOrientation, u);
    if ( w.x < 0 ) m = -1;
    if ( w.x > 0 ) m = 1;
    if ( fabs(v * u) < 1 ) {
        fs.x += m * steeringForce * acos(v * u) / pi;
    }
}
```

- alignment rule:

```
if ( doFlock && (n > 0) ) {
    vAve = vAve / n;
    u = vAve;
    u.normalize( );
    v = units[i].vVelocity;
    v.normalize( );
    w = vRotate2D(-units[i].fOrientation, u);
    if ( w.x < 0 ) m = -1;
    if ( w.x > 0 ) m = 1;
    if ( fabs(v * u) < 1 ) {
        fs.x += m * steeringForce * acos(v * u) / pi;
    }
}
```

- separation rule:

```
for ( j = 1; j < maxNumUnits; j++ ) {
    if ( inView ) {
if ( d.magnitude( ) <= units[i].fLength * radiusFactor ) {
            pAve += units[j].vPosition;
        vAve += units[j].vVelocity;
        } elsif ( d.magnitude() <= units[i].fLength*separationFactor ) {
        if ( w.x < 0 ) m = -1;
        if ( w.x > 0 ) m = 1;
        fs.x += m * steeringForce * (units[i].fLength *
```

```
                separationFactor) / d.magnitude( );
            }
            }
    }
```

- obstacle avoidance:
  - simple idealization of an obstacle: cycle
  - outfit our units with virtual feelers, they will stick out in front of the units, and if they hit something, this will be an indication to the units to turn
  - the vector v, represent the feeler, the vector a is the difference between the unit's and the obstacle's positions, project a onto v by taking their do product and produce vector p, subtract vector p from a yields vector b, to test whether v intersects the circle somewhere we need to test 2 conditions: the magnitude of p must be less than the magnitude of v, and the magnitude of b must be less than the radius r of the obstacle
  - the steering force is a function of the prescribed maximum steering force times the ratio of the magnitude of v to the magnitude of a, this will make the steering correction greater the closer the unit is to the obstacle
- follow the leader
  - we won't explicitly designate any particular unit as a leader, but instead we'll let some simple rules sort out who should be or could be a leader; if no other units are directly in front of a given unit, it becomes a leader; if at least one unit is in front of and within view of the current unit, the current unit can't become a leader and it must follow the flocking rule

Potential Function Based Movement

- use potential function to control the behavior of computer-controlled game units, a single function handles both chasing and evading, can also handle obstacle avoidance
- very simple to implement, but CPU-intensive for large numbers of interacting game units $O(n^2)$
- Lenard-Jones potential function $U = - A / r^n + B / r^m$, where U is the inter-atomic potential energy and is proportional to the separation distance (r) between molecules, A and B are strength parameters, as are the attenuation exponents m and n
- if we take the derivative of this potential function, we get a function representing a force $F = - dV/dr = - nA / r^{n+1} + mB / r^{m+1}$, the term involving A and n represents the attraction force component of the total force, while the term involving B and m represents the repulsive force component
- the repulsive component acts over a relatively short distance, r, from the object, but it has a relatively large magnitude when r gets small; the attractive component has relative smaller magnitude, but it acts over a much greater range of separation, r
- hard-coded the parameters to some constant values, and then tuning by adjusting them by trial and error until the desired results are achieved
- reduce the strength of the attraction component to yield behavior resembling the interception algorithm; increase the strength of attraction to yield behavior resembling the basic line-of-sight algorithm; reduce the attraction force and increase the repulsive force to yield behavior resembling flocking; give different parameter settings to different units to lend some variety to their behaviors
- set the attraction force to 0 effective enable us to simulate spherical, rigid objects
- swarming: the attractive components of those forces will make the units come together (cohesion), while the repulsive components will keep them from running over each other (avoidance); use to model crowd behavior; also can combine leaders with this algorithm
- optimization suggestions:
  - not perform the force calculation for objects that are too far away from the unit
  - divide your game domain into a grid containing cells of some prescribed size and assign each cell an array to store indices to each obstacle that falls within that cell
  - while the units moves around, only compute the forces between the unit and those obstacles contained within that cell and the immediately adjacent cells
  - once you compute the force between the pair of i and j, you don't need to recalculate it for the pair j and i

Basic Pathfinding and Waypoints

- the basic path-finding is simply a process of moving the position of a game character from it initial location to a desired destination

```
if ( positionX > destinationX ) {
    positionX--;
} else if ( positionX < destinationX ) {
    positionX++;
}
if ( positionY > destinationY ) {
    positionY--;
} else if ( positionY < destinationY ) {
    positionY++;
}
```

  - the game character moves diagonally toward the goal until it reaches the point where it is on the same X- or Y-axis as the destination position, then moves in a straight horizontal or vertical path until it reaches its destination; this produces an unnatural-looking path to the destination
  - a better approach would be to move in a more natural line-of-sight path using Bresenham line algorithm
- random movement obstacle avoidance: random movement can be a simple and effective method of obstacle avoidance

```
if ( player in line-of-sight ) {
    follow straight path to player
} else {
    move in a random direction
```

```
}
```

- if there are a few obstacles in the scene, it is likely the player will be in the line of sight the next time through the game loop

- tracing around obstacles: the computer-controlled character follows a simple path-finding algorithm in an attempt to reach its goal; it continues along its path until it reaches an obstacle; at that point it switches to a tracing state; follow the edge of the obstacle in an attempt to work its way around it
  - tracing the outskirts of the obstacle until the line connecting the starting point and the desired destination is crossed
  - another method is to incorporate a line-of-sight algorithm with the previous tracing method; at each step along the way, we utilize a line-of-sight algorithm to determine if a straight line-of-sight path can be followed to reach the destination
- breadcrumb path-finding: each time the player takes a step, he unknowingly leaves an invisible marker or breadcrumb on the game world; when a game character comes in contact with a breadcrumb, the game character will follow in the footsteps of the player until the player is reached
  - the breadcrumb method also is an effective and efficient way to move groups of computer-controlled characters; instead of having each member of the a group use an expensive and time-consuming path-finding algorithm, you can simply have each member follow the leader's breadcrumb
  - loop shifts all the position in the array, it deletes the oldest position and makes the first array element available for the current player position
  - starting the search from the most recent player position; the troll will always look for the adjacent tile containing the most recent breadcrumb and skip over breadcrumb whenever possible
- path following: