Movement: Path Finding, Flocking, Formation

John Laird

9/27/10

Also based on talks by Lars Lidén and Damián Isla
Movement

• Movement layer figures out how the character should move in the world
• Avoid obstacles, follow others, …
• Does not figure out
  • where to move or
  • how to animate movement
Simple Movement

- Random motion
  - Just roll the dice to pick when and which direction to move
- Simple pattern
  - Follow invisible tracks: Galaxians
- Tracking
  - Pure Pursuit: Move toward agent’s current position
    - Heat seeking missile
  - Lead Pursuit: Move to position in front of agent
  - Collision: Move toward where agent will be
Lead Pursuit
Collision
Simple Movement - more

- Tracking
  - Weave: Every N seconds move X degree off opponent’s bearing
  - Spiral: Head 90-M degrees off of opponent’s bearing
- Evasive – opposite of any tracking
- Delayed or restricted sensing gives different effects
Moving in the World: Path Finding

- Just try moving toward goal *while avoiding obstacles.*
Problem
Create Avoidance Regions

Source

Goal
Path Planning Problem

• Given a graph with some cost function, find the shortest path between two vertices in the graph
• Arbitrary graph
• 2D world – node-based, or tile based, convex regions, squares, rectangles, hexes
• 2½D world – no arches – same thing
• 3D world – node-based, or treat mostly as 2D tiles with unusual connectivity
Path Finding

A: 10
C: 7
D: 6
E: 5
B: 6
F: 7
G: 5
H: 4
I: 5
J: 2
K: 5
L: 3
M: 0
Analysis

• Find the shortest path through a maze of rooms

• Approach is A*:
  • At each step, calculate the cost of each expanded path
  • Also calculate an estimate of remaining cost of path
  • Extend path with the lowest cost + estimate

• Cost can be more than just distance:
  • Climbing and swimming are harder (2x)
  • Monster filled rooms are really bad (5x)
  • Can add cost to turning – creates smoother paths
  • But must be a numeric calculation
  • Must guarantee that estimate is not an overestimate

• A* will always find shortest path
A* Advantages/Disadvantages

• Advantages:
  • Provably optimal
  • Usually limits search
    • Especially for simple graphs
    • Linear with straight lines
  • Can have complex cost function that includes many things
    • Visibility, effort, …

• Disadvantages:
  • Can spend a lot of resources getting optimal result
  • Can be compute and memory intensive
  • Exhaustive if no path
  • Requires admissible heuristic
A* Nuts and Bolts

- Best-first search (with dynamic programming)
  - Generating partial paths

- Evaluation function $f = g + h$
  - $g =$ actual cost from start node to current node along current path
  - Must be monotonic
  - $h =$ estimated cost from current node to goal node
  - $h$ must not overestimate the cost to the goal
  - Used to guide the search – closer to actual cost the smaller the search
Cost Functions

• Examples:
  • Distance
    • Geometric Distance
    • Manhattan distance (when moving through tiles)
  • Traversal speed based on terrain type
  • Danger (near enemy)
  • Visibility [Vision, Radar]
  • Amount of turning

• Things can’t do:
  • Negative cost (time travel)
Nodes

• Information associated with each node:
  • \( g \) = cost of shortest path from start to node
  • \( h \) = estimated cost to goal
  • \( f = g + h \)
  • \( Parent \) = parent on shortest path from start to node
    • Node is really a path!
  • \( Children \) = all children of this node
  • \( Other\ task\ dependent\ data - x,\ y,\ z\ location,\ ... \)

• Links have cost of traversal
Algorithm

1. Let P = starting point
2. Assign $f$, $g$, $h$ to P
3. Add P to Open list
4. Let $B =$ the best node from the Open list (lowest $f$)
   1. If $B$ is the goal node, then quit – a path is found
   2. If the Open list is empty, then quit – no path
5. Let $C =$ a valid node connected to $B$ (child)
   1. Assign $f$, $g$, $h$ to C
   2. Check if $C$ is part of path on Open or Closed List
      1. If so, check whether the new path is better (lower $f$)
         1. If so, recursively update all super-paths
         2. Else, add C to the Open list
   3. Repeat 5 for all valid children of $B$
   4. Add $B$ to Closed List
6. Repeat from step 4
Costly operations: Optimize execution

1. Let $B =$ the best node from the Open list (lowest $f$)
   1. Sorted open list: heap

2. Check if C is on Open or Closed List
   1. Have bit vector associated with map nodes – in open/closed
   2. Hash – heap sort

3. Else, add C to the Open list
   1. Heap

4. Add B to the Closed list
   1. Don’t sort Closed list – make a hashtable

5. If so, recursively update all super-path

Optimization: Cache top of open-list – don’t sort all of it – don’t expect to go through all of it
Lots of large spaces and obstacles

- Add all regions of high terrain cost
  - If the added region entirely covers the current cell, set the current cell’s cost to the new cost.
  - Otherwise, divide the current cell into four equal size quarters, and recursively add the new region to each child cell.

- Simplify Quad-trees
  - Recombine any cell that has all children of equal cost

- Extract the graph of neighboring cells
  - If impassable, skip the rest
  - If current cell is undivided, add it to list of cells, that border the n,s,e,w and skip
  - Else recur through all children
    - Create the list of internal cells that border our north border by merging the northborder lists of ne and nw children.
    - Create neighbor links.

- Need to smooth paths
  - Find best corners of cells
  - Use last-visible point –
    - Draw a straight line from the current point to
    - The earliest point on path and not exit cells in our path
**Time-Slicing**

- Can’t always compute complete path in one frame
  - Increase resources the longer it takes
- Do a very quick search to start
  - Search N nodes or search for M msec
- Do full search while moving through first N nodes
  - If don’t complete, take best so far
- Splice from current [N] to final
Details

• Cache paths – reuse and splice onto them
  • PRECOMPUTATION!

• If tried same failed path multiple times – kill unit, …

• Design maps so there aren’t big obstacles in the middle

• If maps have choke points, can precompute
Is optimal really necessary?

• What can we do to get a pretty good solution?
• If find a path to the goal, just take it
• Bias search by inflating $h$ [dangerous, but possible]
Dynamic Terrain

• Re-plan when get to changed part of path
• Periodically re-plan
• Set trigger for terrain that is used and then re-plan
  • Expensive if terrain will change and change back
Movement: Pathfinding Tools

- **Waypoint**
  - Position in a map that is used for navigation
  - Usually placed in word manually by a level designer
  - A* is the preferred pathfinding algorithm for quickly finding a short path between two waypoints

- **Link**
  - Connection between two waypoints
  - Often annotated with the required navigation type (Jump, Swim, Climb)
  - For a given NPC, two waypoints are linked when:
    - The NPC has room enough to move from one node to another without colliding with the world geometry
    - The NPC has the required navigation ability

- **Node Graph**
  - Data structure holding all waypoints and links
  - Either generated manually by a level designer or automatically by the computer and annotated by a level designer
Movement: Node Graph
Spatial Feature Extraction

A lot of features we’re interested in can be extracted automatically …

- Surface categorization / characterization
- Surface connectivity
- Overhang detection
- Interior/exterior surfaces
- Ledges
- Wall-bases
- “Leanable” walls
- Corners
- “Step” sectors
- Thresholds
- Local environment classification
  - Captures the “openness” of the environment at firing positions
Spatial Feature Extraction

... and a lot can’t. So we make the designers do it.

Designer “hints”:

- Jumping
- Climbing
- Hoisting
- “Wells”
- Manual fix-up for when the automatic processes fail:
  - Cookie-cutters
  - Connectivity hints
Configuration space approach
Embedded Info, Animation, Sound

Embedded info can be used in path finding:

- Node graph is an example embedded environment information
- embed in path how to jump over crevices or how to open door *(Soldiers of Fortune 2)*
- illusion of coordination by “reserving” a path
Embedding Animation

Diagram showing Points A, B, C, and D with labels such as 'rapelling edge', 'jump edge', 'vault edge', and 'crevice geometry'.
Quake III Arena

- Released in 1999 by id Software
- Designed to be a multiplayer only game
- The player battles computer-controlled opponents, or bots
- Bots developed by Jan Paul van Waveren
Quake III Bot AI

- FSM based – Uses a stack for short-term goals
- Use Fuzzy Logic for some decision making
  - Collecting weapons and armor
  - Choosing a weapon for combat
- Fuzzy Relations were selected using Genetic Algorithms
- Each bot has a data file containing weapon preferences and behavior-controlling variables
Bot Network

Figure 15.1: AI network.
Quake III Bot Navigation

- AAS (Area Awareness System)
  - Level is subdivided into convex hulls that contain no obstacles
  - Connections between areas are formed

Figure 18.2: Jump reachability
Figure 18.3: Jump pad reachability
Simple Animate Objects

- Bugs
  - Never fly in a straight line
  - Don’t always flap their wings
  - Attracted toward certain objects
  - Avoid moving objects
  - Affected by wind/breezes

- Solitary birds: Soaring Bird of Prey
  - Flap their wings rarely: glide/soar
  - Move to area to catch thermals
  - Move in circle
  - Dive to get food (either bird or ground animal)
  - Stay away from planes, etc.

- Ground animals
  - Stay in limited area [bounding box]
  - Feed most of the time
    - look for food, Eat food, Bring food to nest, Fight over food
  - Startled/alerted by motion of bigger animal

- Flocks
- Schools of Fish
- Primary and secondary animals
Flocking Things:
What to measure/model

- Nearby entities
- Current path
- Interaction with weather
- Simple model of hunger, thirst...
- *Don’t really need physics – can “embed” physics in behavior*
Flocking

Separation: steer to avoid crowding local flockmates
Alignment: steer towards the average heading of local flockmates
Cohesion: steer to move toward the average position of local flockmates
Avoidance: Steer to avoid running into local obstacles or enemies
Issues

- Stateless
- Leader can have more influence than others
- Roll, pitch, yaw
- Perceptual range
- Acceleration – change velocity by up to some fixed percent
Schools of fish

- Sudden change of motion every 2-10 seconds
- Random targets to head toward
- Targets change as they are achieved
Swarms

- Lots of agent
- Flies toward $N$ nearest neighbors – don’t avoid collisions
- No alignment, computationally more efficient
Formations

- Organized movements
  - Usually same orientation – but fixed position
Issues

• Movement
• Spacing Distance
• Field of fire
• Protection of range attack
• Different turning radius
• Mixed types

• User control – automatic control
Web sites

• http://www.red3d.com/cwr/boids/ - reynolds boids
• http://www.riversoftavg.com/flocking.htm - flocking
• http://www.riversoftavg.com/formation_flocking.htm - formation