Advanced Pathfinding
IMGD 4000

Finding a Path

• Often seems obvious and natural in real life
  – e.g., Get from point A to B
    → go around lake
• For computer controlled player, may be difficult
  – e.g., Going from A to B goes through enemy base!
• Want to pick “best” path
• Need to do it in real-time

http://www.codeofhonor.com/blog/the-starcraft-path-finding-hack

Finding a Path

• Path – a list of cells, points or nodes that agent must traverse to get to from start to goal
  – Some paths are better than others
    → measure of quality
• A* is commonly used heuristic search
  – Complete algorithm in that if there is path, will find
  – Using “distance” as heuristic measure, then guaranteed optimal


A* Pathfinding Search

• Covered in detail in IMGD 3000
• Basic A* is a minimal requirement for solo project
  – You may use any reference code as a guide, but not copy and paste (cf. academic honesty policies)
• An advanced pathfinding feature will be optional, but required for an A
  – This slide deck

Practical Path Planning

• Sometimes, basic A* is not enough
• Also, often need:
  – Navigation graphs
    • Points of visibility (poiv) – lines connecting visible nodes
    • Navigation mesh (navmesh) – models traversable areas of virtual map
  – Path smoothing
  – Compute-time optimizations
  – Hierarchical pathfinding
  – Special case methods
• Some of these count as optional requirement

Tile-Based Navigation Graphs

• Common, especially if environment already designed in squares or hexagons
• Node center of cell; edges to adjacent cells
• Each cell already labeled with material (mud, river, etc.)
• Downsides:
  – Can burden CPU and memory
    • E.g., Modest 100x100 cell map has 12,000 nodes and 78,000 edges
  – Especially if multiple AI’s calling at same time

Most of slide deck is survey about how to do better...
Outline

• Introduction (done)
• Navigation Graphs (next)
• Navigation Mesh
• Pathfinding Tuning
• Pathfinding in UE4

Point of Visibility (POV) Navigation Graph

• Place graph nodes (usually by hand) at **important** points in environment
• Such that each node has **line of sight** to at least one other node

POV Navigation

• Find closest visible node (a) to current location
• Find closest visible node (b) to target location
• Search for least cost path from (a) to (b), e.g. A*
• Move to (a)
• Follow path to (b)
• Move to target location

DEMO (COARSE) Note, some “backtracking”

Blind Spots in POV

• No POV point is visible from red spots!
• Easy to fix manually in small graphs
• A problem in larger graphs

DEMO (COARSE)

POV Navigation

• **Advantage**
  — Obvious how to build and expand
• **Disadvantages**
  — Can have “blind spots”
  — Can have “jerky” (backtracking) paths
  — Can take a lot of developer time, especially if design is rapidly evolving
  — Problematic for random or user generated maps
• **Solutions**
  1. Automatically generate POV graphs
  2. Make finer grained graphs
  3. Path smoothing

Automatic POV by Expanded Geometry

(A) Expand geometry
— By amount proportional to bounding radius of moving agents
(B) Connect all vertices
(C) Prune non-line of sight points
→ Avoids objects hitting edges when pathing

Note: works best if bounding radius similar for all units
Finely Grained Graphs

- **Upside**: Improves blind spots and path smoothness
- **Downside**: Back to similar performance issues as tiled graphs
- **Upside**: Can often generate automatically using "flood fill" (next slide)

Flood Fill to Produce Finely Grained Graph

- Place "seed" in graph
- Expand outward
  - e.g., 8 directions
  - Making sure nodes and edges passable by bounding radius
- Continue until covered
  - Produces a finely grained graph
- Note, same algorithm used by "paint" programs to flood fill color

Path Finding in Finely Grained Graph

- Use A* or Dijkstra depending on whether looking for specific or multiple general targets.
  - e.g., Find exit? A* typically faster than Dijkstra’s since latter is exhaustive
  - e.g., Find one of many rocket launchers? A* would need to be re-run for each, then chose minimum.

Problem: Kinky Paths

Solution: Path smoothing,
- Simple fix to “penalize” change in direction
- Others work better (next)

Simple Smoothing Algorithm (1 of 2)

- Check for “passability” between adjacent edges
- Also known as "ray-cast" since if can cast a ray between A and C then waypoint B is not needed
Navigation Mesh (NavMesh)

- Partition open space into network of convex polygons
  - Why convex? → guaranteed path from any point to any point inside
  - Instead of network of points, have network of polygons
- Can be automatically generated from arbitrary polygons
- Becoming very popular (e.g., UE4)

NavMesh Example (1 of 3)

Waypoint

NavMesh

NavMesh has more information (i.e., can walk anywhere in polygon)

NavMesh Example (2 of 3)

Waypoint

NavMesh

Waypoint needs lots of points
NavMesh needs fewer polygons to cover same area

NavMesh Example (3 of 3)

Waypoint

NavMesh

Plus smoothing, else zigzag
Note, smoothing for navmesh works, too
NavMesh Performance

- But isn’t it slower to do pathfinding on NavMesh?
- No. NavMesh is also a graph, just like waypoints.
- Difference? Navmesh has polygon at each graph node
- A* runs on any graph
  - Square grid
  - Waypoint
  - Navmesh

NavMesh with other Paths

- NavMesh can be used with waypoints
- Use waypoints for higher-order locations
  - E.g.:
    - Soldiers need patrol path
    - Old man needs fishing path
    - Cover points for hiding
- NavMesh to get there

Outline

- Introduction
  - (done)
- Navigation Graphs
  - (done)
- Navigation Mesh
  - Generating a NavMesh
    - (next)
- Pathfinding Tuning
- Pathfinding in UE4

Generating NavMesh

- Can be generated by hand
  - e.g., lay out polygons (e.g., squares) to cover terrain for map
  - Takes a few hours for typical FPS map
- Can be generated automatically
  - Various algorithm choices
  - One example [Leo14]

Generating NavMesh – Walkable Area

- Use collision grid to compute walkable area
  - Prepare 2d array, one for each pixel
  - Sample each pixel → if collide, then black else white

Generating NavMesh – Contour

- Run marching squares to get contour
  - “marching squares” is graphics algorithm that generates contours for 2d field
  - Parallelizes really well
- Contour points used as vertices for triangles for NavMesh

Outline

- Introduction
  - (done)
- Navigation Graphs
  - (done)
- Navigation Mesh
  - Generating a NavMesh
    - (next)
- Pathfinding Tuning
- Pathfinding in UE4

Generating NavMesh

- Can be generated by hand
  - e.g., lay out polygons (e.g., squares) to cover terrain for map
  - Takes a few hours for typical FPS map
- Can be generated automatically
  - Various algorithm choices
  - One example [Leo14]
Generating NavMesh – Simplified Contour

- Simplify contour by removing points along same horizontal/vertical line
- Don’t remove all redundant points to avoid super-long edges (can produce odd navigation) in triangles
  - Define max distance between points

Simplifying contour points, max distance 128

Generating NavMesh – Triangles

- Fit squares → Loop
  - Find point not in mesh
  - Create square at point
  - Expand until hits edge or other square
  - Done when no more points
- Split squares into triangles
- Connect triangle to all other triangles in neighbor squares
- Now have graph for pathfinding (e.g., A*)

NavMesh generated using rectangle expansion. Red lines show neighbors.

Generating NavMesh – Path

- Using mid-points, path will zig-zag (see right)

Solution? → Path smoothing
  A. Simple ray-cast
  B. Funnel

Path generated using midpoints of triangles

Generating NavMesh – Path Smoothing by Ray-cast

- Ray-cast as for “simple” smoothing shown earlier (see right)

Path generated using ray-cast to remove unnecessary points

Generating NavMesh – Path Smoothing by Funnel

- Smooth path from start to goal
- Move edges along triangle
- If can ray-cast, then not path “corner” so continue
- If cannot, then found “corner”

Outline

- Introduction (done)
- Navigation Graphs (done)
- Navigation Mesh (done)
- Pathfinding Tuning (next)
- Pathfinding in UE4
Possible Pathfinding Load Spikes

- Could be many AI agents, all moving to different destinations
- Each queries and computes independent paths
- Can lead to spikes in processing time
  → Game loop can’t keep up!

**Solution?** Reduce CPU load by:
1) Pre-compute paths
2) Hierarchical pathfinding
3) Grouping
4) Time slice  
(Talk about each briefly, next)
Generalized Search Class

```cpp
enum SearchType { Astar, Dijkstra; 
enum SearchResult { found, not_found, incomplete; 
class GraphSearch {
private:
    SearchType search_type;
    Position target;
public:
    GraphSearch (SearchType type, Position target);
    virtual SearchResult cycleOnce() = 0;
    virtual double getCost() const = 0;
    virtual std::list<PathEdge> getPath() const = 0;
};
```
Getting Out of Stuck Situations

- Calculate distance to Object’s next waypoint each update step
- If this distance remains about same or consistently increases
  - Probably stuck
  - Replan
- Alternatively – estimate arrival time at next waypoint
  - If takes longer, probably stuck
  - Replan

Advanced Pathfinding Summary
- Not necessary to use all techniques in one game
- Only use whatever game demands and no more
- An advanced pathfinding feature is an optional project requirement
- For reference C++ code see [http://samples.jbpub.com/9781556220784/Buckland_SourceCode.zip](http://samples.jbpub.com/9781556220784/Buckland_SourceCode.zip) (Chapter 8 folder)

Outline
- Introduction (done)
- Navigation Graphs (done)
- Navigation Mesh (done)
- Pathfinding Tuning (done)
- Pathfinding in UE4 (next)

Navigation in UE4
- Has NavMesh
  - Auto generated initially
  - Tweaked by hand
- NavLinkProxy to allow “jumping”
- Auto re-generates when static objects move

UE4 NavMesh
- Modes Panel → Create → Volumes
- Translate/scale to encapsulate walkable area
- Press “P” to view
  - Green shows mesh

(More in upcoming slides)
Automatic Update as Design Level

Automatic Update at Runtime

- Edit → Project Settings → Navigation Settings → Rebuild at Runtime

NavLinkProxy

- Tell Pawns where can temporarily leave NavMesh (e.g., to jump off edges)

Use NavMesh with Character

- Setup AI Character and AI Controller Blueprint
- Place character in scene
- “Move to location” – E.g., waypoints
- “Move to actor” – E.g., follow or attach character