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

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 a path, will find
  – Using “distance” as heuristic measure, then guaranteed optimal

A* Pathfinding Search

• Covered in detail in IMGD 3000
• Basic A* is a technical requirement for final project
  – You may use any reference code as a guide, but not copy and paste (cf. academic honesty policies)
• An advanced pathfinding feature is one of the optional tech requirements
  – This slide deck
Practical Path Planning

- Sometimes, basic A* is not enough
- Also often need:
  - Navigation graphs
    - Points of visibility (pov) – lines connecting visible nodes
    - Navigation mesh (navmesh) – models traversable areas of virtual map
  - Path smoothing
  - Compute-time optimizations
  - Hierarchical pathfinding
  - Special case methods
- Any one of these counts 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.)
- Downside:
  - Can burden CPU and memory
    - e.g., Modest 100x100 cell map has 10,000 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

Note, some "backtracking"

DEMO (COARSE)

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 take a lot of developer time, especially if design is rapidly evolving
  - Problematic for random or user generated maps
  - Can have “blind spots”
  - Can have “jerky” (backtracking) paths

- 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 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)

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.

Flood Fill

• Place “seed” in graph
• Expand outward
  – Making sure nodes and edges passable by bounding radius
• Continue until covered
• Note, same algorithm used by “paint” programs to flood fill color

Problem: Kinky Paths

Path chosen not “natural”

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

Simple Smoothing Algorithm (2 of 2)

1. Grab source edge \( E_1 \)
2. Grab destination \( E_2 \)
3. If agent can move between,
   a) Assign destination \( E_1 \) to destination \( E_2 \)
   b) Remove \( E_2 \)
   c) Advance \( E_2 \)
4. If agent cannot move
   a) Assign \( E_2 \) to \( E_1 \)
   b) Advance \( E_2 \)
5. Repeat until destination \( E_1 \) or destination \( E_2 \) is endpoint

Outline

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

- Partition open space into a 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)

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

NavMesh Example (2 of 3)

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

NavMesh Example (3 of 3)

- 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
- Run A* runs on graph
  - Square grid
  - Waypoint
  - Navmesh

NavMesh with other Paths

- NavMesh can be used with waypoints
- E.g.,
  - Soldiers need patrol path
  - Old man needs fishing path
  - Cover points for hiding

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

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 to make triangles for NavMesh

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 in triangles
  – Define max distance between points

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*)
Generating NavMesh – Path

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

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

Generating NavMesh – Path
Smoothing by Ray-cast

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

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

Reduce CPU Overhead – Precompute

- If static paths, pre-generate paths and costs (e.g., using Dijkstra’s)
- Time/space tradeoff

- e.g., path A to D?

- Shortest path table

- Path cost table

Reduce CPU Overhead – Hierarchical

- Typically two levels, but can be more
- First plan in high-level, then refine in low-level
- E.g., Navigate Atlanta to Richmond
  - States Georgia and Virginia
  - State navigation: Georgia → South Carolina → North Carolina → Virginia
  - Fine grained pathfinding within state

Reduce CPU Overhead – Grouping

- In many cases, individuals do not need to independently plan path
  - E.g., military has leader
- So, only have leader plan path
  - Normal A*
- Other units then follow leader
  - Using steering behaviors (later slide deck)

(Sketch of how next)
Reduce CPU Overhead – Time Slice (1 of 3)

- Evenly divide fixed CPU pathfinding budget between all current callers
  - Must be able to divide up searches over multiple steps
- Considerable work required!
  - But can be worth it since pathfinding load constant

Reduce CPU Overhead – Time Slice (2 of 3)

- Create PathPlanner for Objects to use for pathing
- Pathfinding generalized
  - Grab next node from priority queue
  - Add node to shortest paths tree
  - Test to see if target
  - If not target, examine adjacent nodes, placing in tree as needed
- Call the above a “cycle”

(See example next slide)

Reduce CPU Overhead – Time Slice (3 of 3)

- Create PathManager to allocate cycles
- Object registers with PathManager
  - Gets instance of path
- Each tick, PathManager distributes cycles among all
- When path complete, send message (event) to Object

Time Slice Example

- Object requests path to location
- PathPlanner
  - Provides node
  - Creates search (A*, also could be Dijkstra)
  - Registers with PathManager
- PathManager allocates cycles
- When done, returns path (or fail)

- Note time slicing, implies that caller may have to wait for answer
  - Delay proportional to size of graph (number of cycles needed) and number of other Objects pathing
- What should Object do while waiting for path?
  - Stand still, but often looks bad (e.g., player expects unit to move)
  - So, start moving, preferably in “general direction” of target
    - “Seek” as a behavior (see later slide deck)
    - “Wander” as a behavior (ditto)
  - When path returns, “smooth” to get to target (example next slide)
Time Slicing needs Smoothing

- Object registers pathfinding to target
- Starts seeking towards target
- When path returns, Object will backtrack. Bad!
- Solution? → Simple smoothing described earlier to remove

Without smoothing

Smoothed

Time Slicing with Seek Fail

- When waiting for path, head “wrong” direction
  - May even hit walls!
  - Looks stoopid
- Alternative can be to return path to node closer to Object
  - Start moving along that path
- When complete path returned, adjust

Seek heads in obvious (to player) wrong direction

Getting Out of Stuck Situations

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

- Not necessarily use *all* techniques in *one* game
- Only use whatever game demands and no more
- An advanced pathfinding feature is one of the [optional tech requirements](http://samples.jbpub.com/9781556220784/Buckland_SourceCode.zip (Chapter 8 folder))
- For reference C++ code see

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

(More in upcoming slides)

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

Move static mesh (bridge)

NavMesh automatic update

Automatic Update at Runtime

• Edit → project settings → navigation settings → rebuild at runtime

Unreal Engine 4 Tutorial - NavMesh Rebuild Realtime
https://www.youtube.com/watch?v=UpbaCHTcNPA

NavLinkProxy

• Tell Pawns where can temporarily leave NavMesh (e.g., to jump)

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

Unreal Engine 4 Tutorial - Basic AI Navigation
https://www.youtube.com/watch?v=4k2zrBu6Y