Pathfinding Lists

- Here are some possible implementation choices for open/closed lists:
  - Unsorted arrays or linked lists
  - Sorted arrays
  - Sorted linked lists
  - Sorted skip lists (hierarchy of linked lists)
  - Indexed arrays
  - Hash tables
  - Binary heaps
  - Splay trees
  - HOT (Heap On Top) queues

Pathfinding Lists

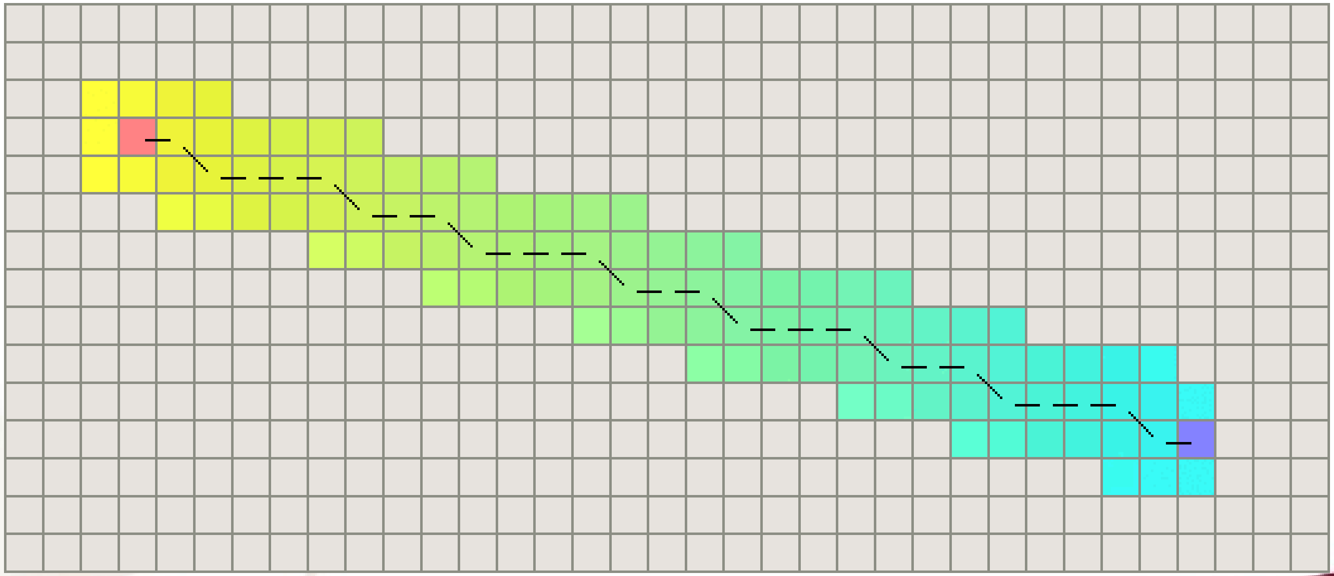
- To get the best performance, you will want a hybrid data structure.
- For example1, you can use an indexed array for O(1) membership test and a binary (min-)heap for O(log l) insertion and O(log l) to find and remove the node with the smallest estimated-total-cost.
- For cost-so-far updates, use an indexed array for an O(1) test whether an update is needed (by storing the cost-so-far value in the indexed array), and then use an O(l) update (rarely done) on the binary heap. Better: you can also use the indexed array to store the location in the heap of each node; this would give you O(log l) for update.

Tampering with heuristic function

- A* can theoretically produce non-optimal results
- Choice of heuristic function can determine the outcome of results
  - Use a good heuristic function (with additional termination rules) to ensure better optimal results
  - Deliberately allow sub-optimal results (by using a mediocre heuristic) to give *faster execution*

Choosing a Heuristic

- The more accurate the heuristic, the less fill A* will have (i.e., *a smaller l*), the faster it will run
- Perfect heuristic → ***always gives correct answer***
- In only a few cases will a practical heuristic be accurate.



Choosing a Heuristic

- **Underestimating heuristic**
  - **Gives *cost less than or equal to actual cost***
  - ***Biased towards cost-so-far***
  - ***Increases running time***
  - **A* will *prefer to examine nodes closer to the start node*, rather than those closer to the goal**
  - **It generates the *exact same path that the Dijkstra* algorithm would generate.**
  - **Best if accuracy *more important than performance***

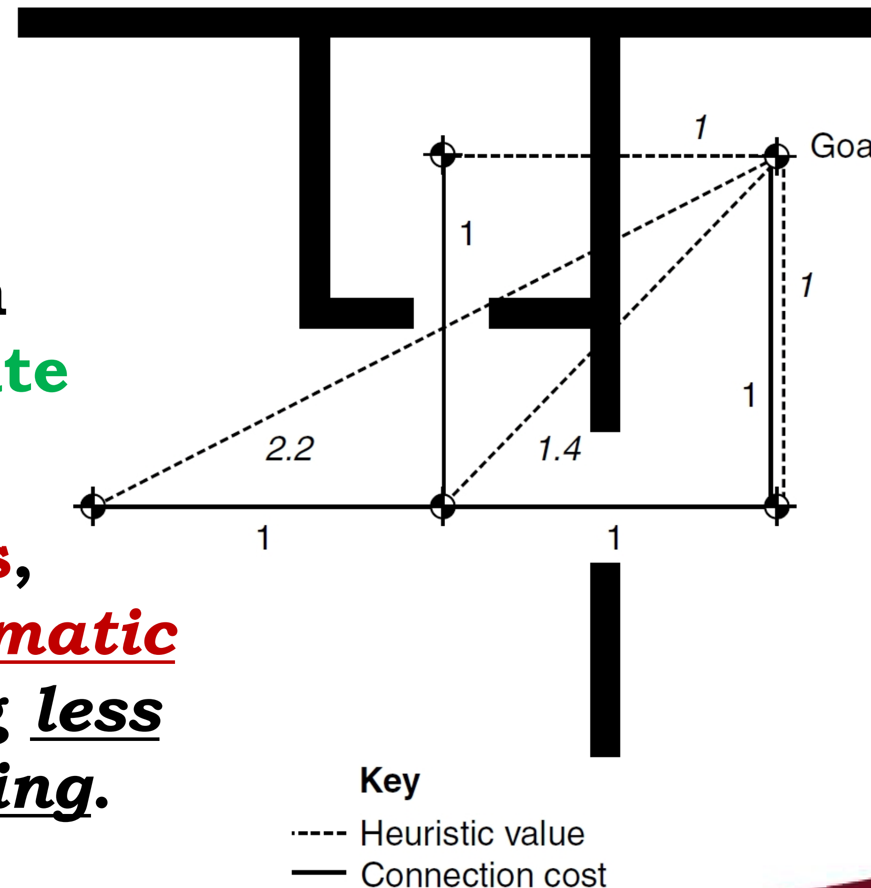
Choosing a Heuristic

- **Overestimating heuristic**
  - **Gives *cost greater than actual cost***
  - ***Biased towards heuristic value***
  - ***Faster search towards goal***
  - **The A* algorithm will pay *proportionally less attention to the cost-so-far* and will tend to favour nodes that have less distance to go (i.e., closer to the goal if heuristic reflects this).**
  - **Definitely sub-optimal**
  - **Best if *performance* and speed is *more important than accuracy***

Concordia University

Heuristic – Euclidean Distance

- Or "as the crow flies" or "straight-line distance"
- Guaranteed to be *underestimating*
- In outdoor settings: with few constraints on movement: very accurate with fast pathfinding.
- In indoor environments, as shown: can be a *dramatic underestimate*, causing *less than optimal pathfinding*.

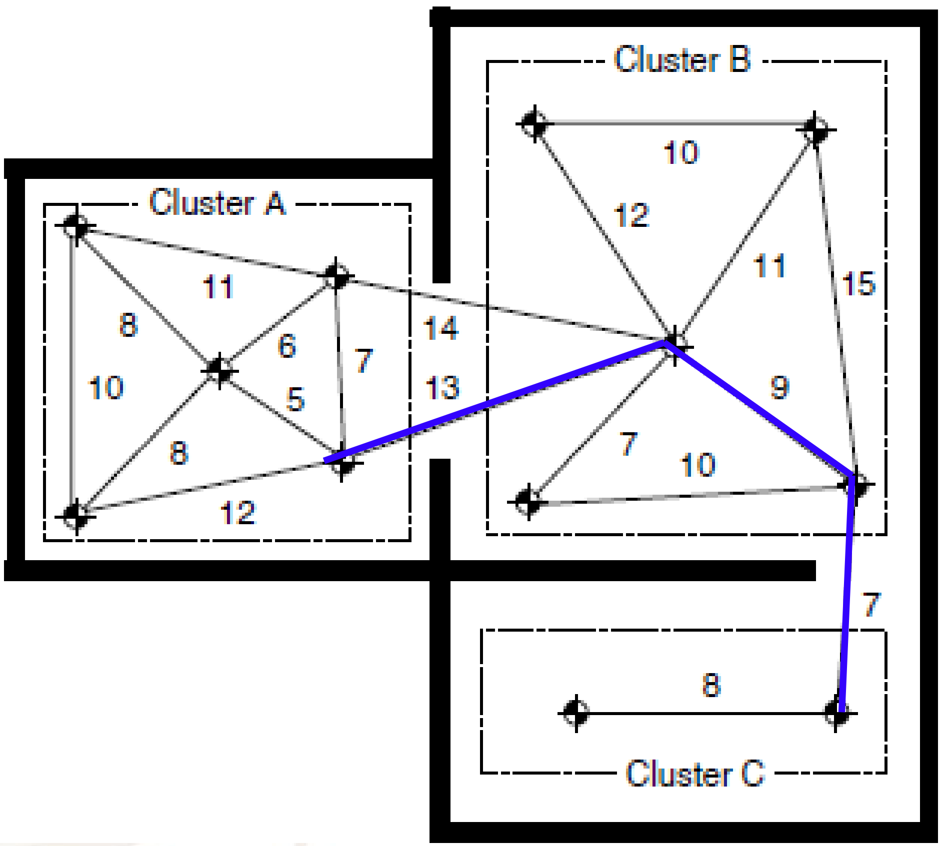


Heuristic – Cluster Heuristic

- **Works by grouping nodes together in clusters**
- **Nodes in a cluster represent some region of level (room, corridor, building, *etc.*) that is *highly interconnected***
  - **Clustering can be done by some graph clustering algorithms (beyond the scope of this course),**
  - **but often clustering is *manual* or a *by-product of the level design***
- **Create a lookup table with the smallest path length between each pair of clusters. An offline step based on running lots of pathfinding trials between nodes in pairs of clusters.**

Heuristic – Cluster Heuristic

- **Intra-cluster Euclidean distances & Cluster distance lookup table**



	A	B	C
A	x	13	29
B	13	x	7
C	29	7	x

Lookup table

Heuristic – Cluster Heuristic

- In a game,
  - If start and goal nodes are *in same cluster*, Euclidean distance (or some other simpler fallback such as a *null heuristic* — i.e., use Dijkstra within the cluster) is used to get result
  - Otherwise, *look up the table for the distance between clusters* (this distance is the minimum distance between any node in the current cluster and any node in the destination cluster; e.g., the distance 29 on the previous slide).

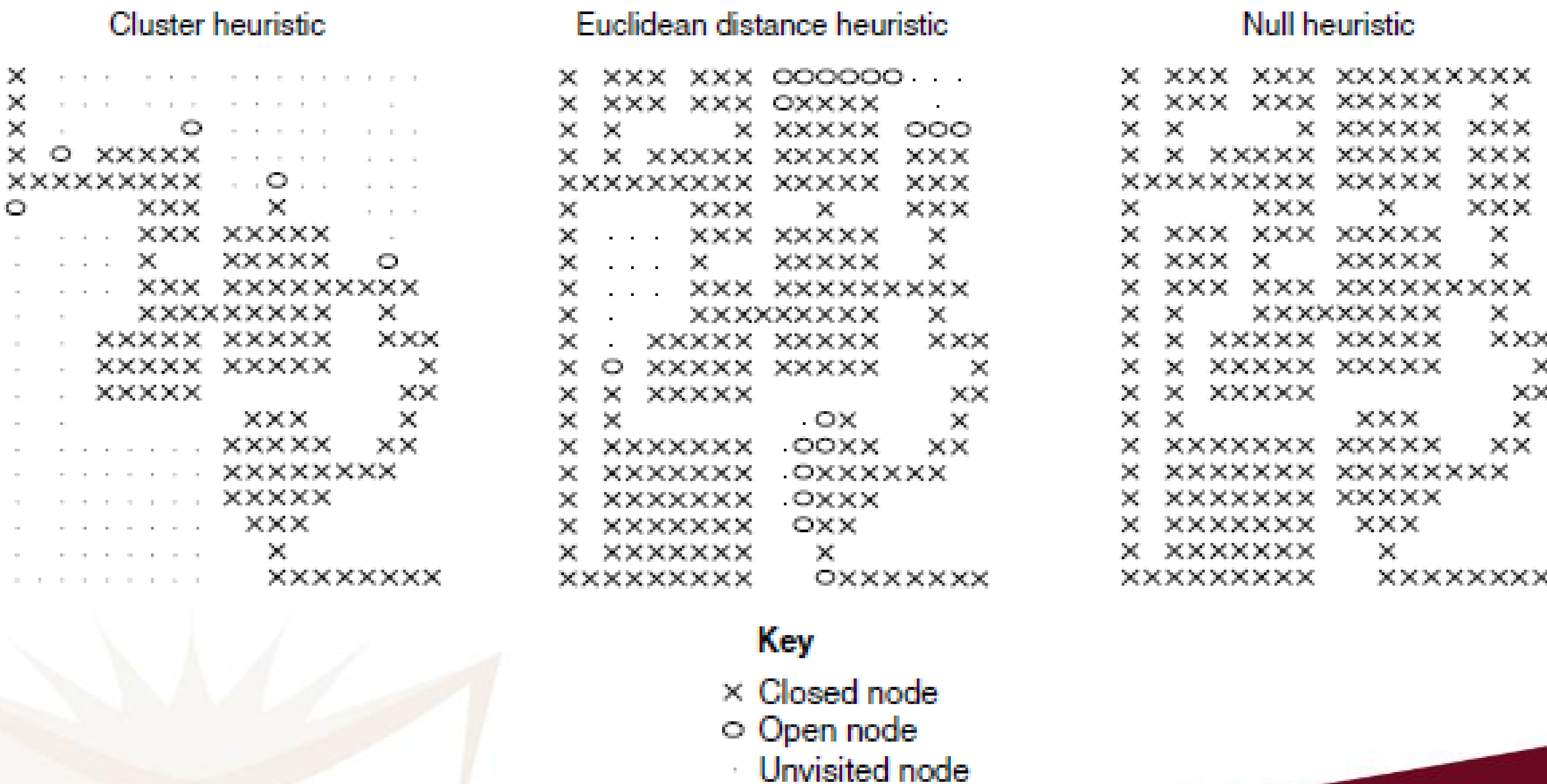
Heuristic – Cluster Heuristic

- Do you foresee any problem since all nodes in a cluster are given the <u>same cluster heuristic value</u>?
- What are the implications if the cluster sizes are
  - Small (i.e., using lots of clusters)?
  - Large (i.e., using only a few clusters)?
- There are ways to improve the cluster heuristic, but ***<u>no accepted techniques for reliable improvement</u>***; essentially a matter of experimenting within the context of your game's level design.

Fill Patterns in A*

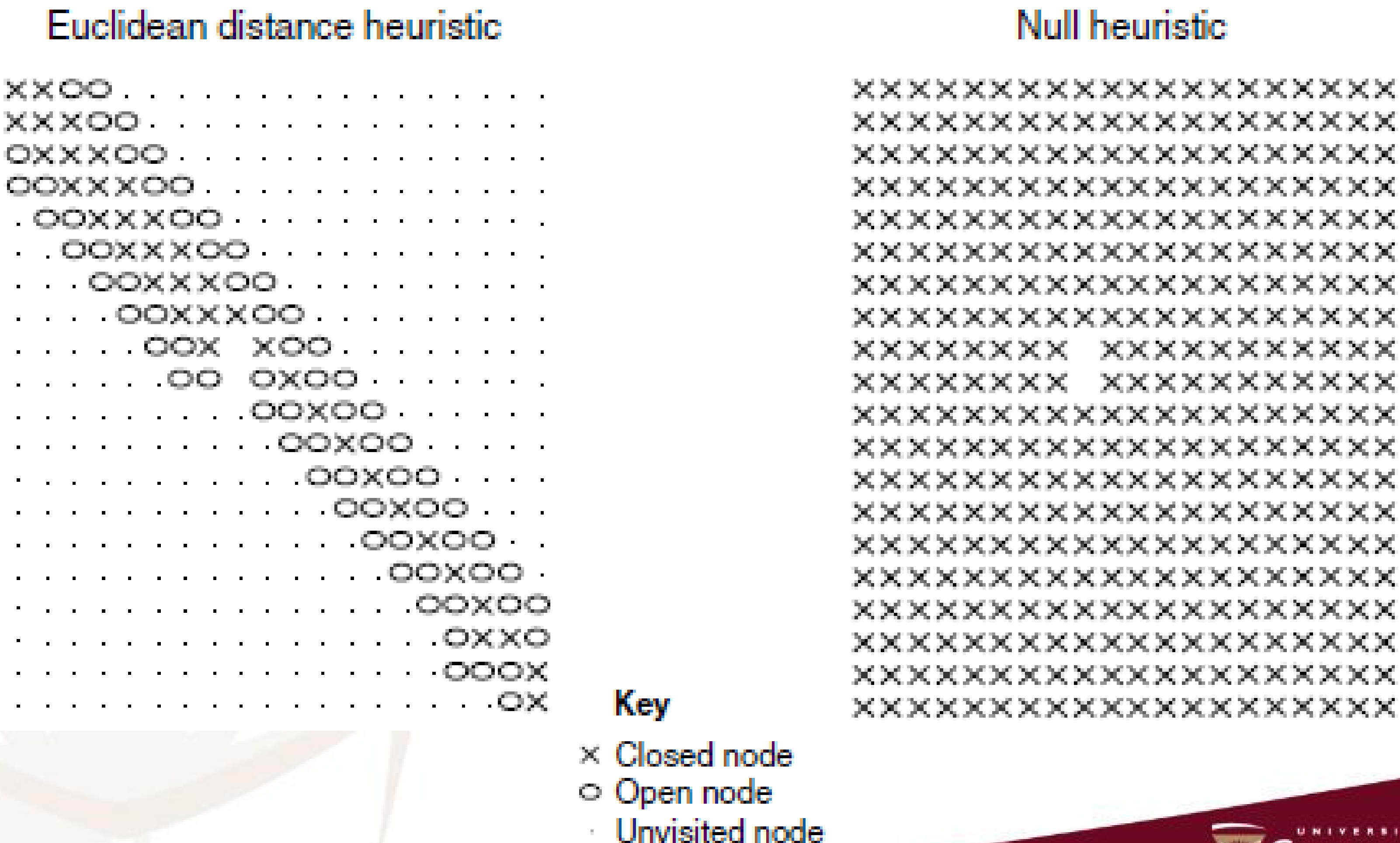
- **Indoor fill patterns of a tile-based level**

(Dijkstra)



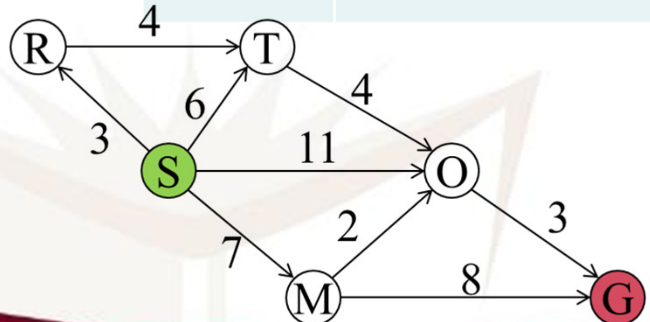
Fill Patterns in A*

- **Outdoor** fill patterns



A* Example

Current Node	Open List, Format: (Node, Cost-so-far, Connection, Estimated-total-cost)	Closed List, Format: (Node, Cost-so-far, Connection)
S	(T,6,ST,6+6=12), (O,11,SO,11+3=14), (M,7,SM,7+8=15), (R,3,SR,3+15=18)	(S,0,-)
T	**(O,10,ST,13)**, (M,7,SM,15), (R,3,SR,18)	(S,0,-), (T,6,ST)
O	(M,7,SM,15), (R,3,SR,18), (G,13,OG,13)	(S,0,-), (T,6,ST), (O,10,TO)
G	(M,7,SM,15), (R,3,SR,18)	(S,0,-), (T,6,ST), (O,10,TO), (G, 13, OG)
M	(R,3,SR,18), **(O,9,MO,12)**	(S,0,-), (T,6,ST), (G, 13, OG), (M,7,SM)
O	(R,3,SR,18), **(G,12,OG,12)**	(S,0,-), (T,6,ST), (M,7,SM), (O,9,MO)
G	(R,3,SR,18)	(S,0,-), (T,6,ST), (M,7,SM), (O,9,MO), (G,12,OG)



Node	Heuristic
S	10
T	6
O	3
M	8
G	0
R	15

Quality of Heuristics

- Producing a heuristic is far <u>more of an art</u> than a science.
- Most AI developers just drop in a <u>***simple Euclidean distance***</u> heuristic without thought and hope for the best.
- The only sure-fire way to get a decent heuristic is to <u>visualize the fill</u> of your algorithm.
- Current research is exploring ***automatically generating heuristics*** based on examining the structure of the graph and its connections. But so far, the results have yet to prove compelling.

World Representations

- To squeeze your game level into the pathfinder you need to do some translation—*from geometry of the map and the movement capabilities* of your characters *to the nodes and connections of the graph and the cost function* that values them.
- For each pathfinding world representation, we will *divide the game level into linked regions* that correspond to nodes and connections.
- The different ways this can be achieved are called **division schemes**. Each division scheme has three important properties we'll consider in turn:
- quantization/localization, generation, and validity

World Representations

Quantization:

- The process of converting *locations/positions* in the game into *nodes in the graph*

Localization:

- The conversion of *nodes in the pathfinder path* back into *game world locations*

World Representations

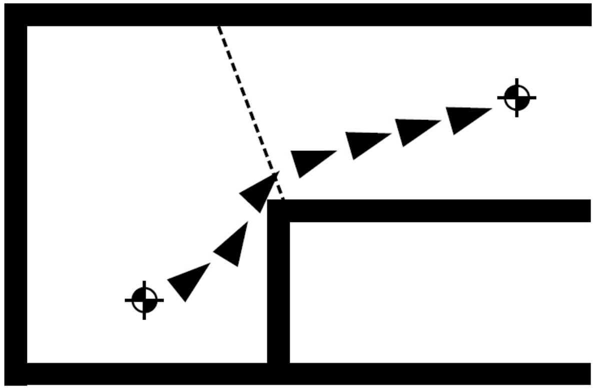
Generation:

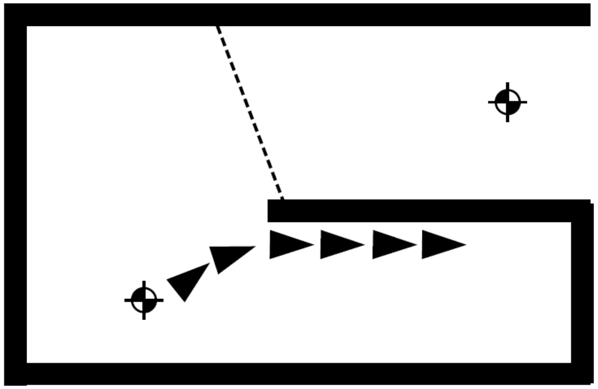
- There are ***many ways of dividing up a continuous space into regions and connections*** for pathfinding.
- Standard methods used regularly:
  - **Manual Techniques:**
    - ✓ Dirichlet domain
  - **Algorithmic Methods:**
    - ✓ Tile graphs,
    - ✓ Points of visibility, and
    - ✓ Navigation meshes (NavMeshes)

Université Concordia University

Validity

- If a plan tells a character to move along a connection from node A to node B, then the *character should be able to carry out that movement*.
- A division scheme is valid *if all points in two connected regions can be reached from each other*.
- In practice, most division schemes **don't enforce validity**.
- There can be different levels of validity:





Tile Graphs

- Tile-based levels ***rarely appear in mainstream games***
- **A large number of games, though, use a regular grid in which they place 3D characters.**
- **This grid can be simply turned into a tile-based graph**

- **Many RTS games still use tile-based graphs**
- **Tile-based levels split the whole world into regular (usually square) regions.**

Tile Graphs

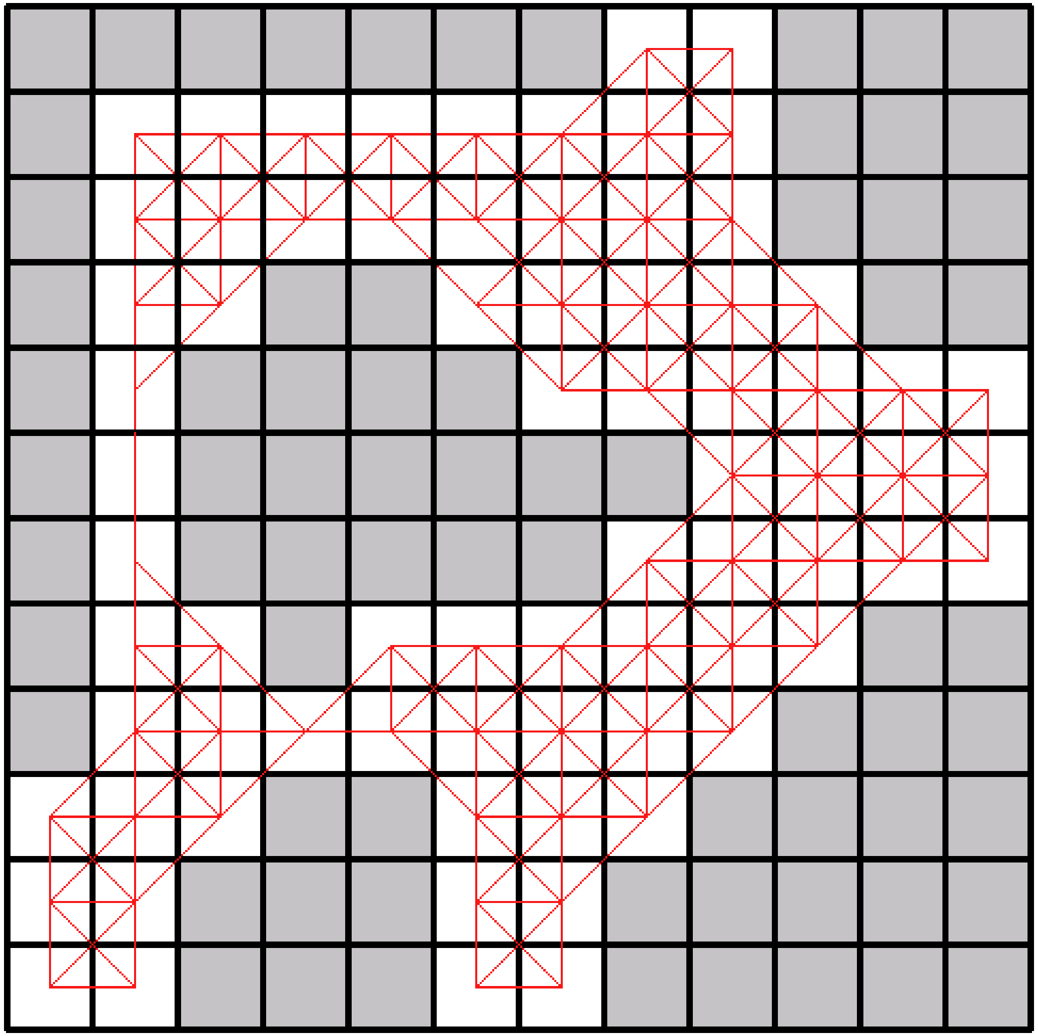


Image from http://www.codeproject.com/Articles/14840/Artificial-Intelligence-in-Games

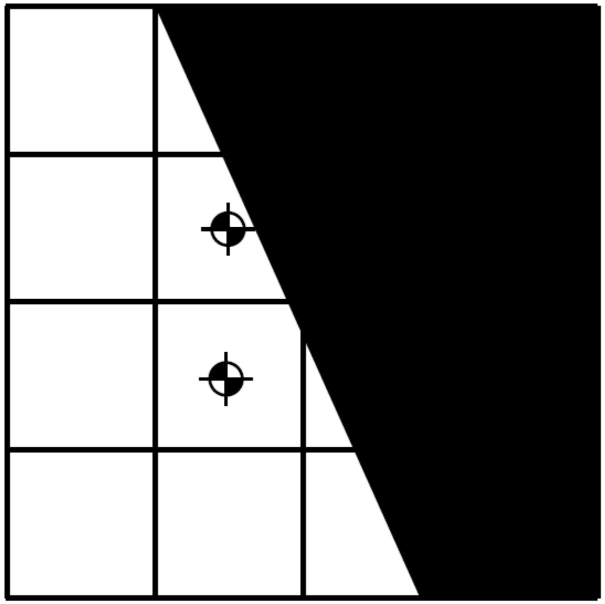
- **Division Scheme**: **Nodes** in the pathfinder's graph represent **tiles in the game world**. The connections between nodes ***connect*** to their **immediate** (obvious set of) **neighbours**.

Tile Graphs

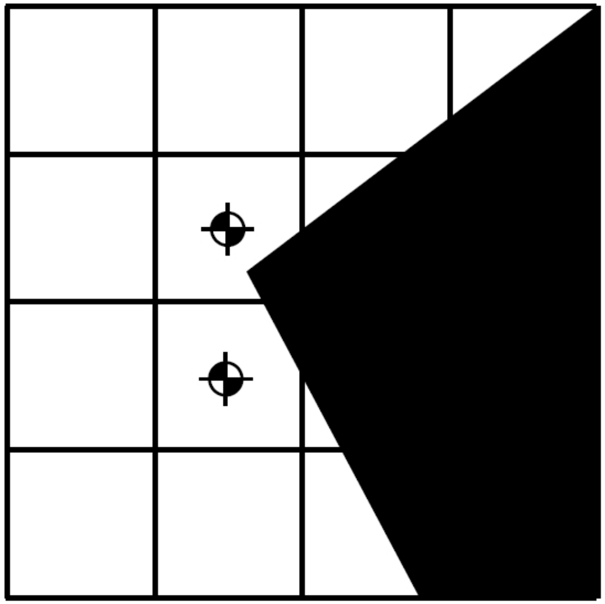
- **Quantization and Localization**: Simple with a regular grid.
- **Generation**: Tile-based graphs are ***generated automatically*** (even at runtime if needed). Connections can be generated as requested by the pathfinding (some ***connections to blocked tiles will not be available***).
- For tile-based grids representing outdoor height fields (a rectangular grid of height values), the ***costs often depend on gradient*** (and distance).

Tile Graphs

Validity: **if only *empty tiles* are connected, then the graph will be *guaranteed to be valid*. When a graph node is only partially blocked, then the graph may not be valid, depending on the shape of the blockage.**



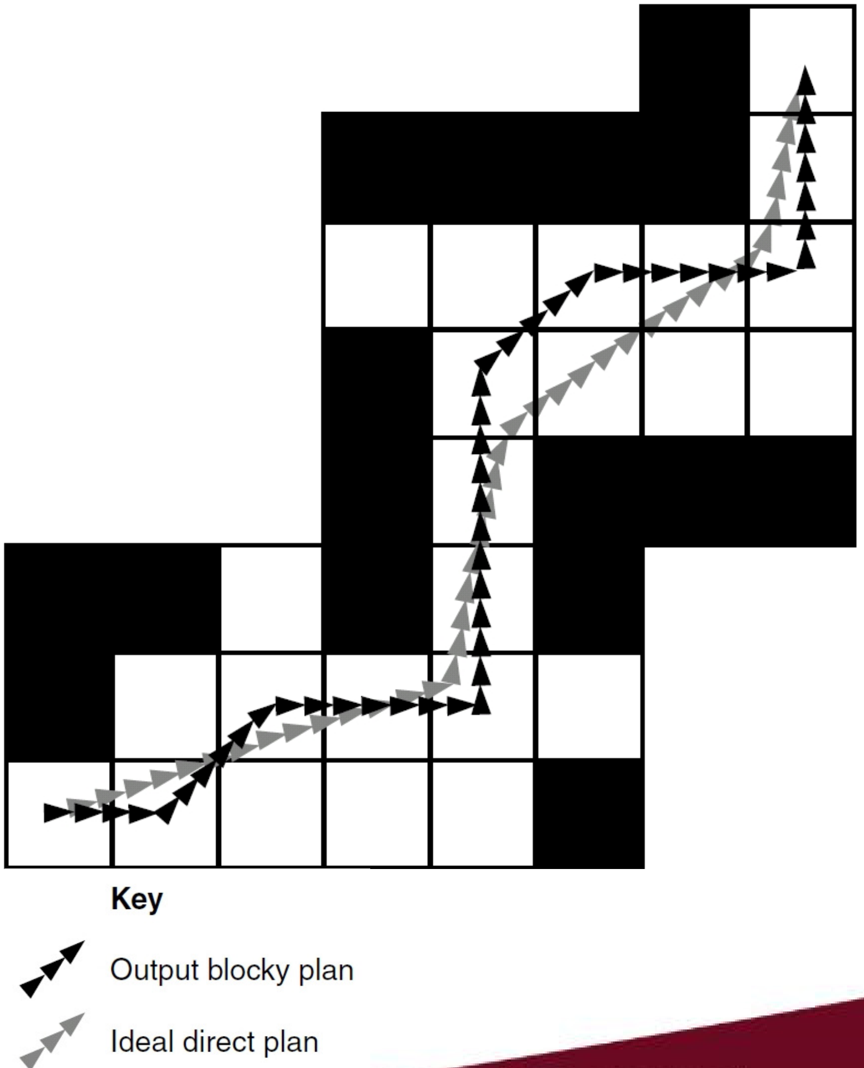
Valid partial blockage



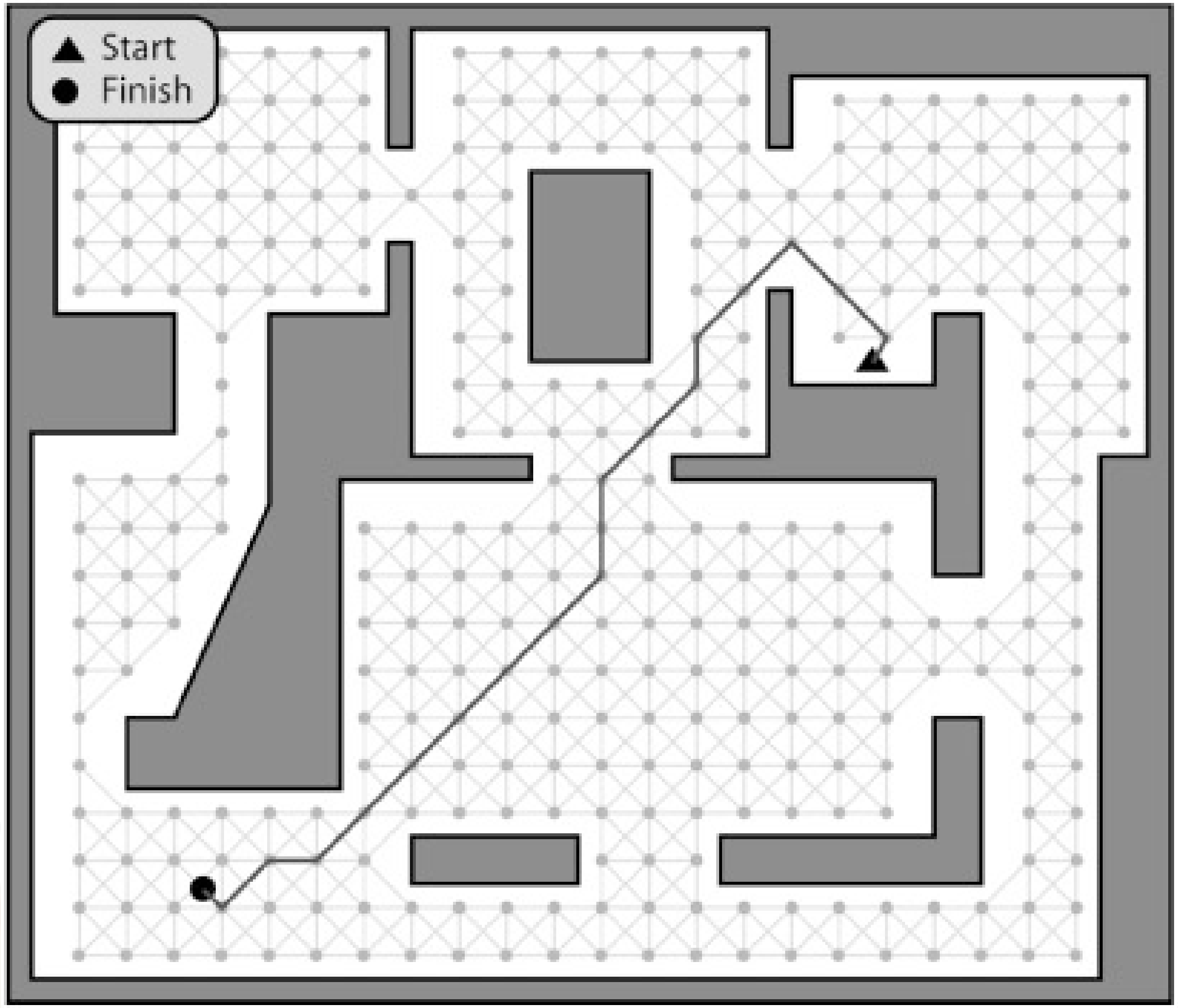
Invalid partial blockage

Tile Graph Pathfinding

- When the *paths* returned by the *pathfinder* are drawn on the graph (using localization foreach node in the plan), they **can appear blocky and irregular**.
- Modest 100 x 100 cell map: 10,000 nodes and 78,000 edges
- *can burden CPU and memory*, especially if multiple AI's calling in



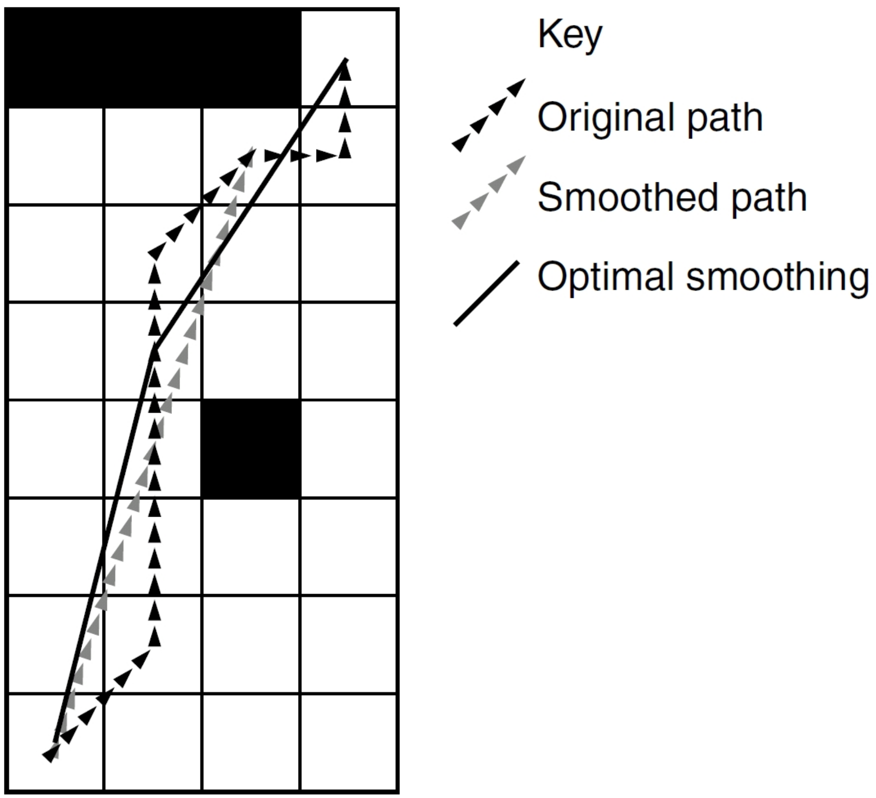
Tile Graph Pathfinding



- Can lead to "kinky paths"
- Solution: ***path smoothing***

Path Smoothing

- A path that travels from node to node through a graph ***can appear erratic***. Sensible node placing can give rise to very ***odd looking paths***.
- The final appearance also depends on ***how characters act on the path***. E.g., a path following steering behaviour will gently smooth out the path.
- Aka ***string pulling*** or ***funnel algorithm***.

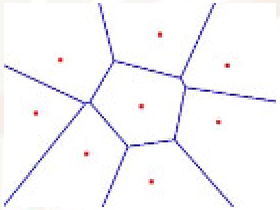


Path Smoothing

- Assume there is a clear route between adjacent nodes in the input path (i.e., subdivision is valid)
- Add start node (i = 1) to output path
- Until the end node is reached do:
  - Starting with the third node (i = 3) in the input path, cast a ray *from the last node of the output path to consecutive nodes* in the input list until the ray fails with the *i^{th}* node. Put (i-1)th node on output path
- Put end node on output path
- The result is a smooth path, but perhaps *not the (optimally) "smoothest"*.

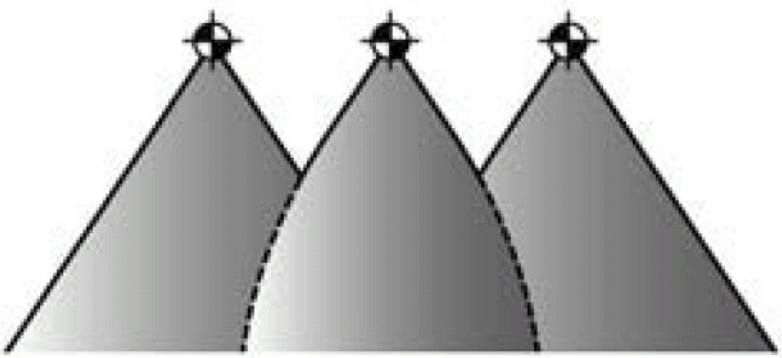
Dirichlet Domains

- A Dirichlet domain, also referred to as a ***Voronoi polygon*** in two dimensions, is a region around one of a finite set of source points whose interior consists of everywhere that is closer to that source point than any other.
- **Division Scheme**: Pathfinding nodes have an associated point in space called the ***characteristic point***, and the quantization takes place by mapping all locations in the point's Dirichlet domain to the node.
- The set of characteristic points is usually specified by a level designer as part of the level data.

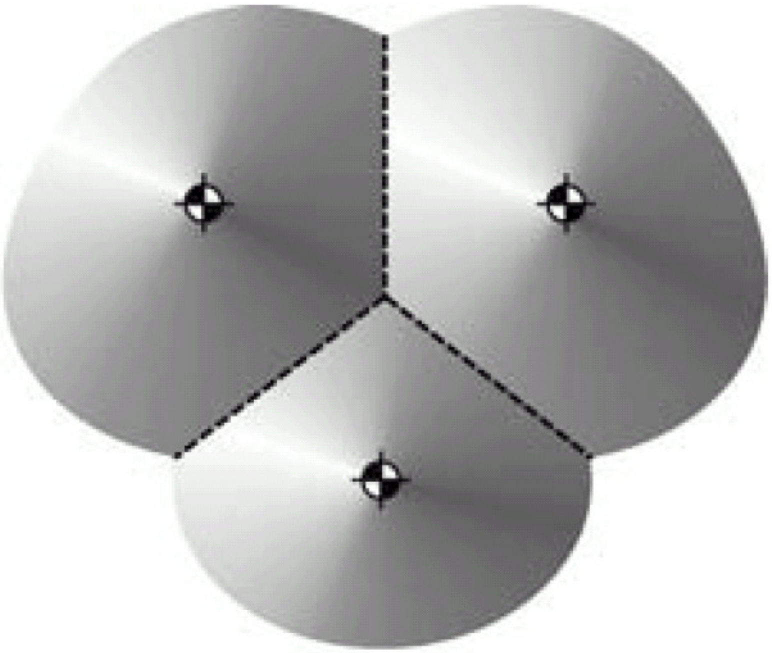


Dirichlet Domains

- **Division Scheme**: **Can be viewed as being *cones originating from the characteristic points*.**



Side view

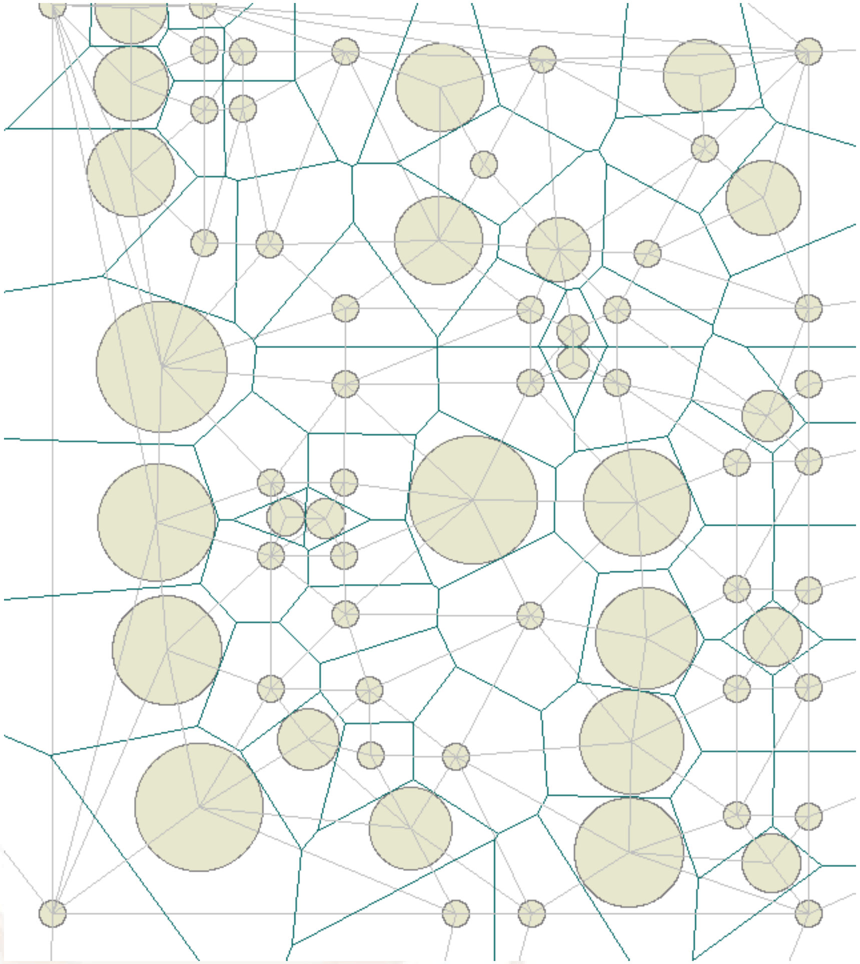


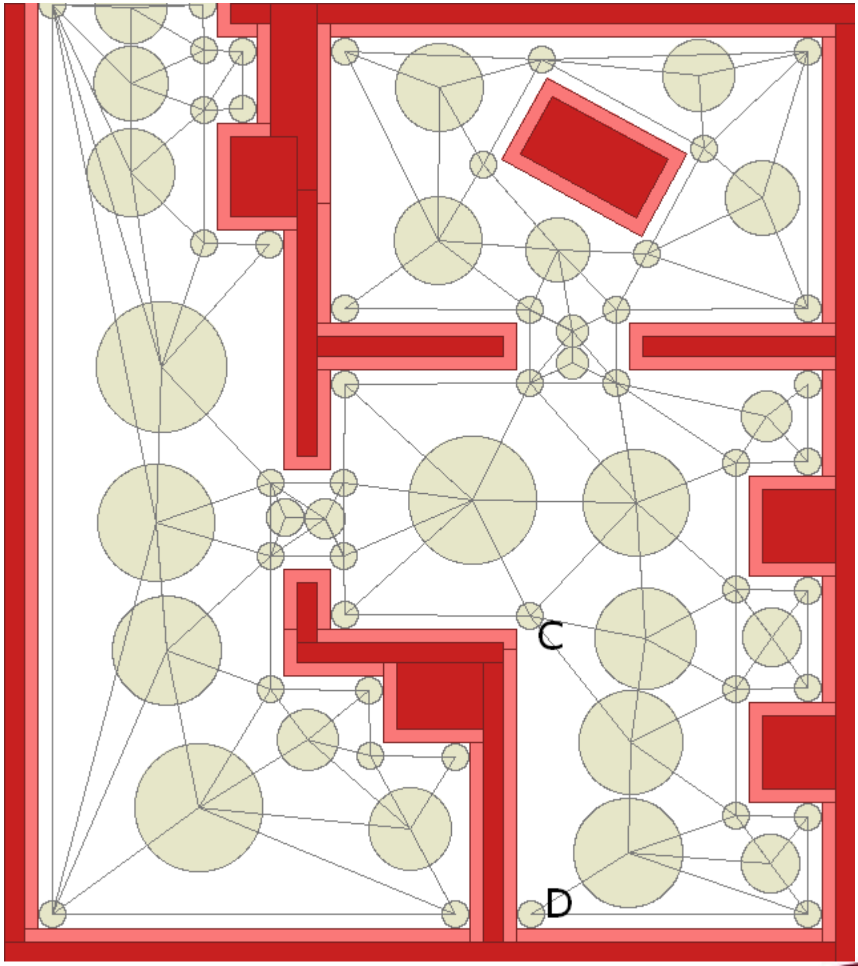
Top view

Dirichlet Domains

- **Division Scheme**: ***Connections are placed between bordering domains*** **(this creates a *Delaunay triangulation*). You will need to check that domains that touch can actually be moved between (e.g., *there might be a wall in the way*).**
- **Frequently, either**
  - ***make the artist specify connections*** **as part of their level design, or**
  - ***ray cast between points to check for connections*** **(see the points of visibility method below).**

Removing Blocked Edges





Images from http://www.gamedev.net/page/resources/_/technical/artificial-intelligence/navigation-graph-generation-r2805

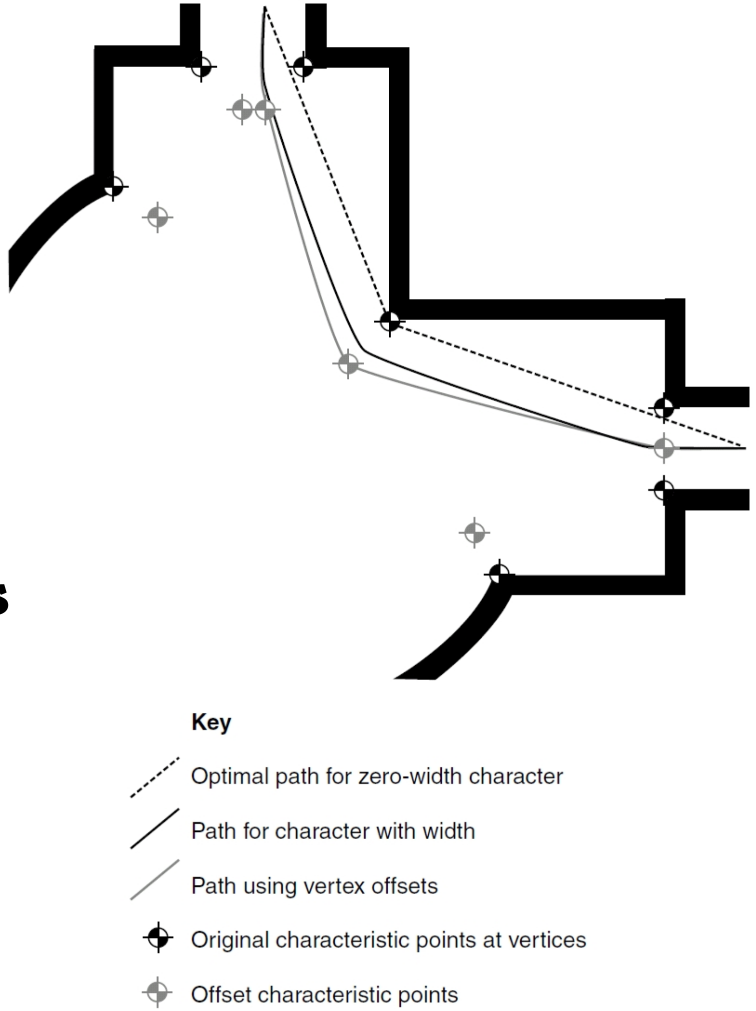
Dirichlet Domains

- **Quantization and Localization**: Positions are quantized by finding the *characteristic point that is closest* (aided by a spatial partition data structure). The localization of a node is given by the *position of the characteristic point that forms the domain* (i.e., the one that is closest).

- **Validity:** To ensure the graph is valid, the *placement of nodes* is *often based on the structure of obstacles*. Obstacles are not normally given their own domains, and so the *invalidity of the graph is rarely exposed*.

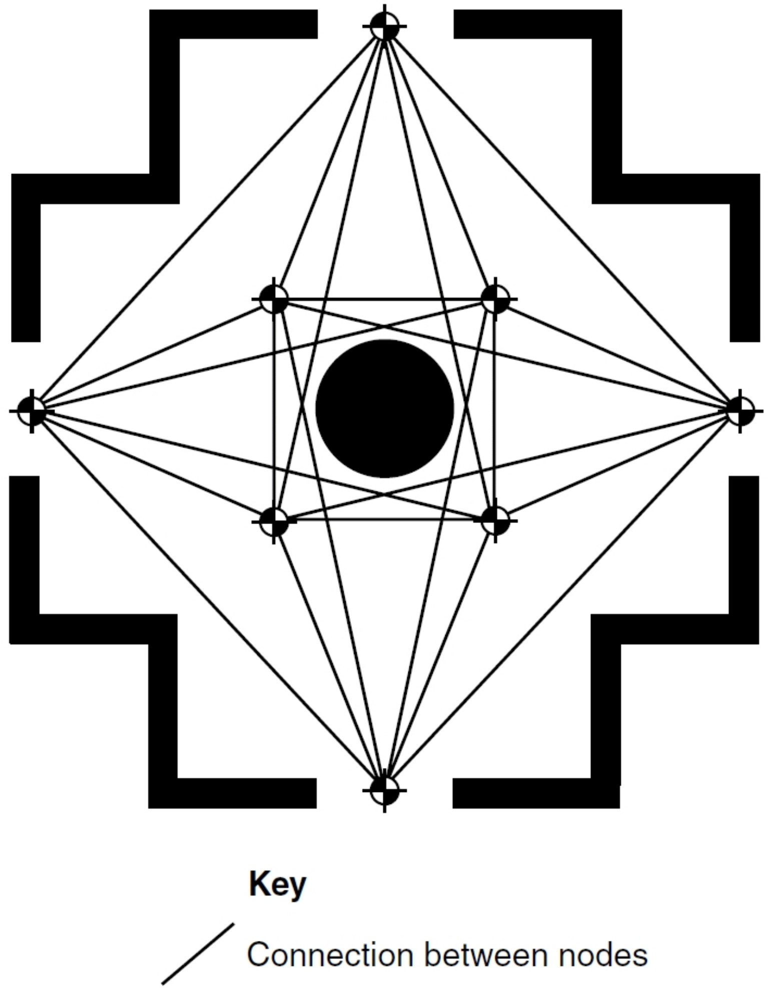
Points of Visibility (POV)

- **The optimal path through any 2D environment *will always have inflection points* (i.e., points on the path where *the direction changes*) at convex vertices in the environment. If the character that is moving has some radius, these *inflection points are replaced by* arcs of a circle at a distance away from the vertex.**

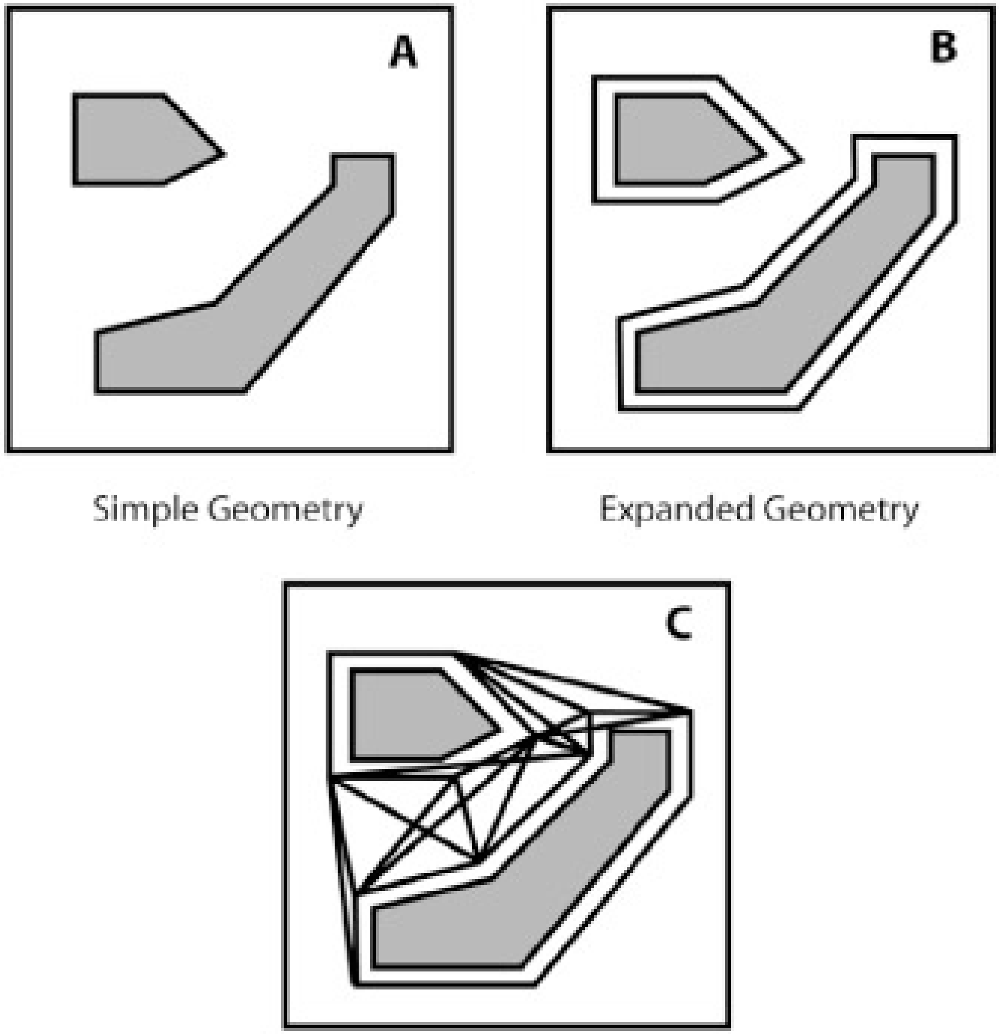


Points of Visibility (POV)

- Since these *inflection points naturally occur in the shortest path*, we can use them as nodes in the pathfinding graph.
- To work out how these points are connected, *rays are cast between them*, and a connection is made *if the ray doesn't collide* with any other geometry.
- The *resulting graph can be large*.



Points of Visibility (POV)



Simple Geometry — Expanded Geometry — The finished POV graph

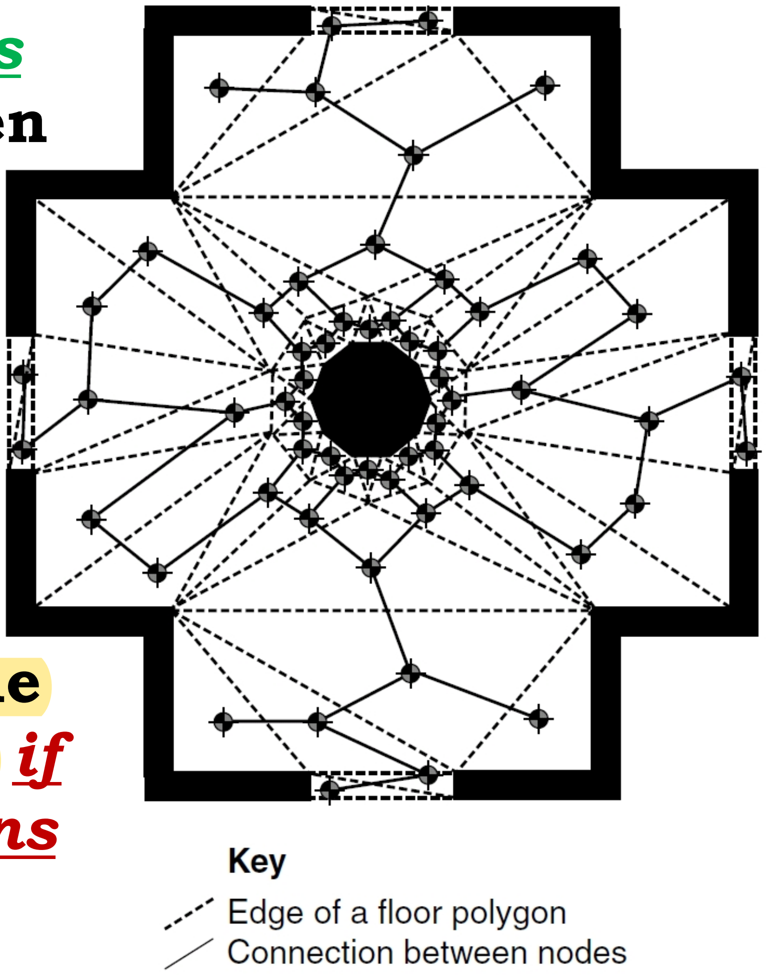
1. **Expand geometry** by an amount ***proportional to bounding radius of the NPCs*** (this expansion is called the **Minkowski sum** of the two geometries)
2. **Add vertices to graph**
3. ***Prune non-line-of-sight points***

Points of Visibility

- **Quantization and Localization**: ***Points of visibility*** **are usually taken** ***to represent the centers of Dirichlet domains*** **for the purpose of** ***quantization*****.**
- **Validity: In addition, if Dirichlet domains are used for quantization, points quantized to two connected nodes** ***may not be able to reach each other*****. As with Dirichlet domains above, this means that the graph is** ***strictly invalid*****.**
- **Although a relatively popular method for automatic graph generation, [AI for G, M]** ***don't think the results are worth the effort*** **and suggest** ***Navigation Meshes instead*****…**

Navigation Meshes

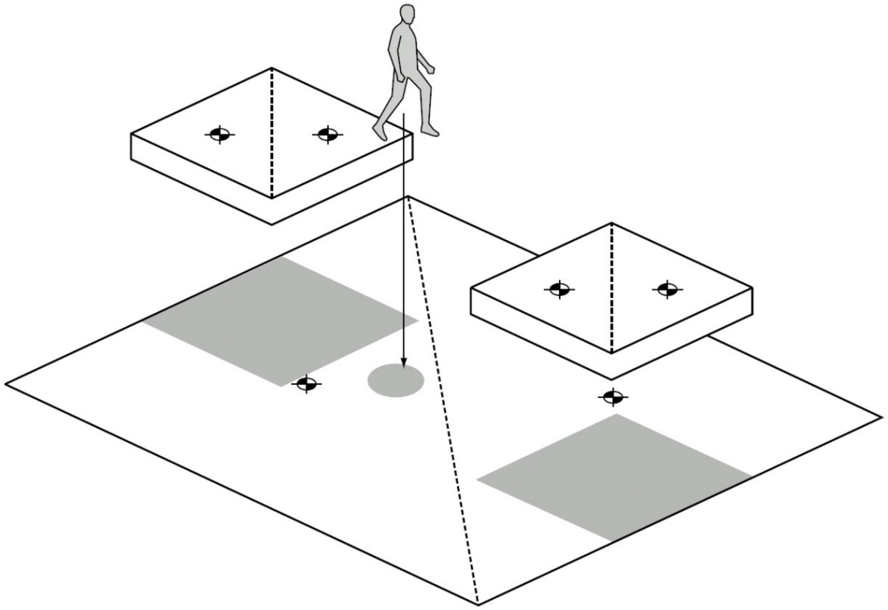
- A ***majority of modern games use navigation meshes*** (often abbreviated to "navmesh") for pathfinding. Since levels are usually made up of polygons, ***these are used as the basis of the graph***.
- **Division Scheme**: Each polygon acts as a node in the graph. Nodes are connected ***if their corresponding polygons share an edge***.



Navigation Meshes

- **Quantization and Localization**: A *position is localized into the floor polygon that contains it*. The only wrinkle occurs *when a character is not touching the floor*. We can simply *find the first polygon below it and quantize it to that*.

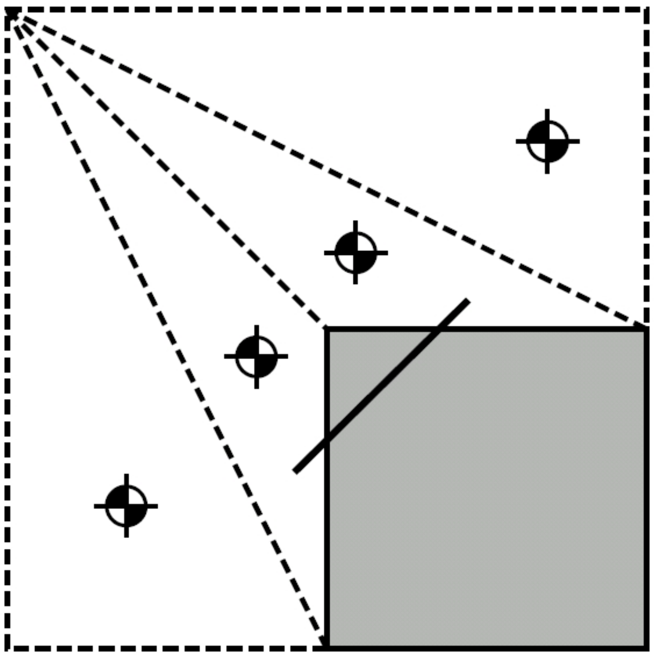
Unfortunately, it is possible for the character to be placed in a completely ***inappropriate ode as it falls or jumps***



Université Concordia University

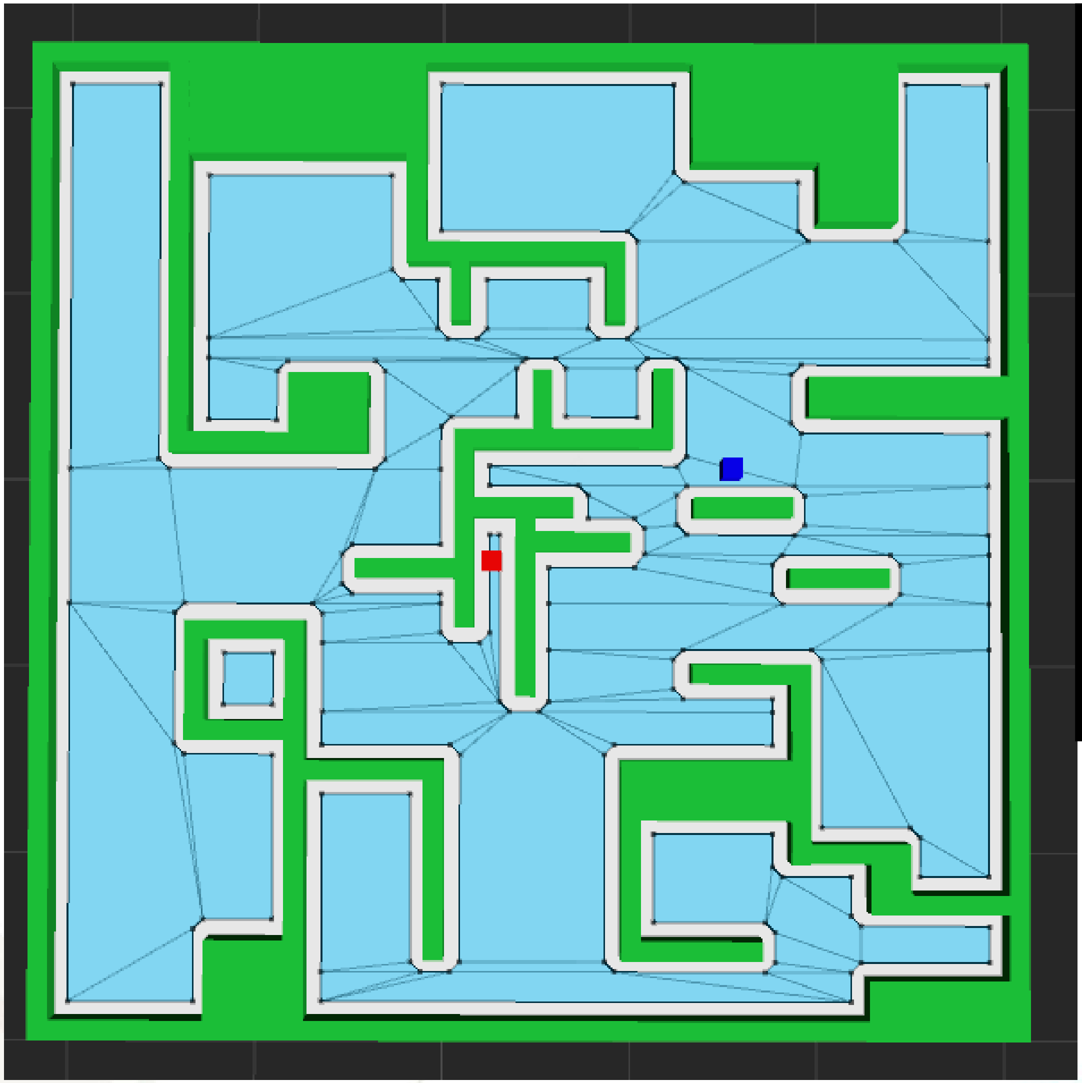
Navigation Meshes

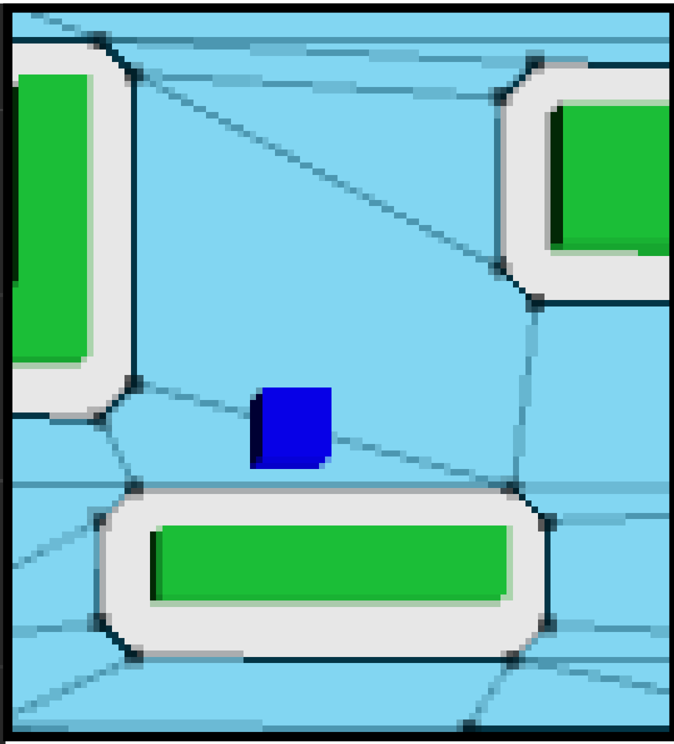
- **Quantization and Localization**: Localization can choose any point in the polygon, but normally uses the geometric center (the average position of its vertices).



- **Validity**: The regions generated by navigation meshes *can be problematic*. We have assumed that any point in one region can move directly to any point in a connected region, but *this may not be the case*. A level designer can *create geometry that avoid this*

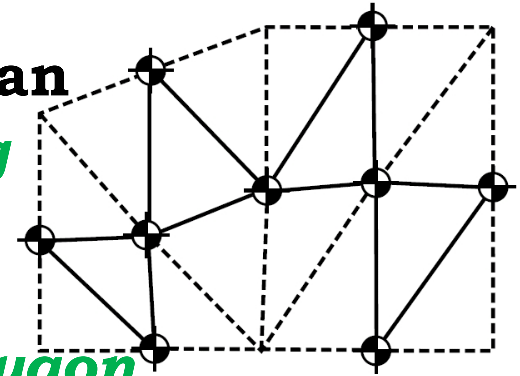
Unity's Navmesh



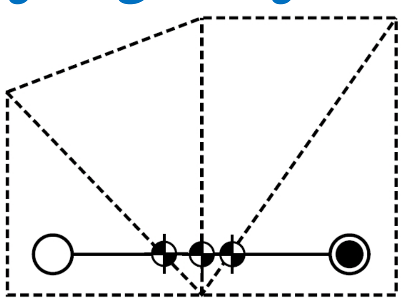


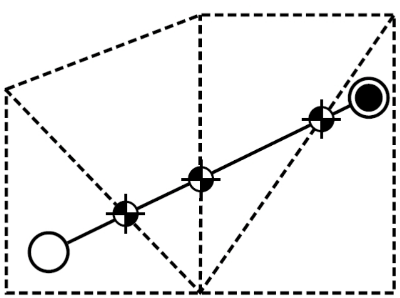
http://www.binpress.com/tutorial/unity3d-ai-navmesh-navigation/119

Navigation Meshes



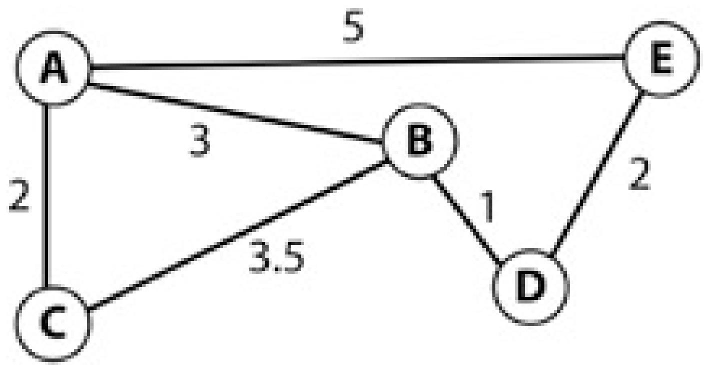
- **Edges as Nodes**: Floor polygons can also be *converted into a pathfinding graph* by *assigning nodes to edges between polygons* and *using connections across face of each polygon*
- This approach is also commonly used in association with portal-based rendering, where nodes are assigned to portals and where *connections link all portals within the line of sight of one another*.
- The nodes on the edges of floor polygons *can be placed dynamically in the best position* for the path-finder.





Reducing CPU Overhead

- Time/space trade-off



	A	B	C	D	E
A	A	B	C	B	E
B	A	B	C	D	D
C	A	B	C	B	B
D	B	B	B	D	E
E	A	D	D	D	E

shortest path table

	A	B	C	D	E
A	0	3	2	4	5
B	3	0	3.5	1	3
C	2	3.5	0	4.5	6.5
D	4	1	4.5	0	2
E	5	3	6.5	2	0

path cost table