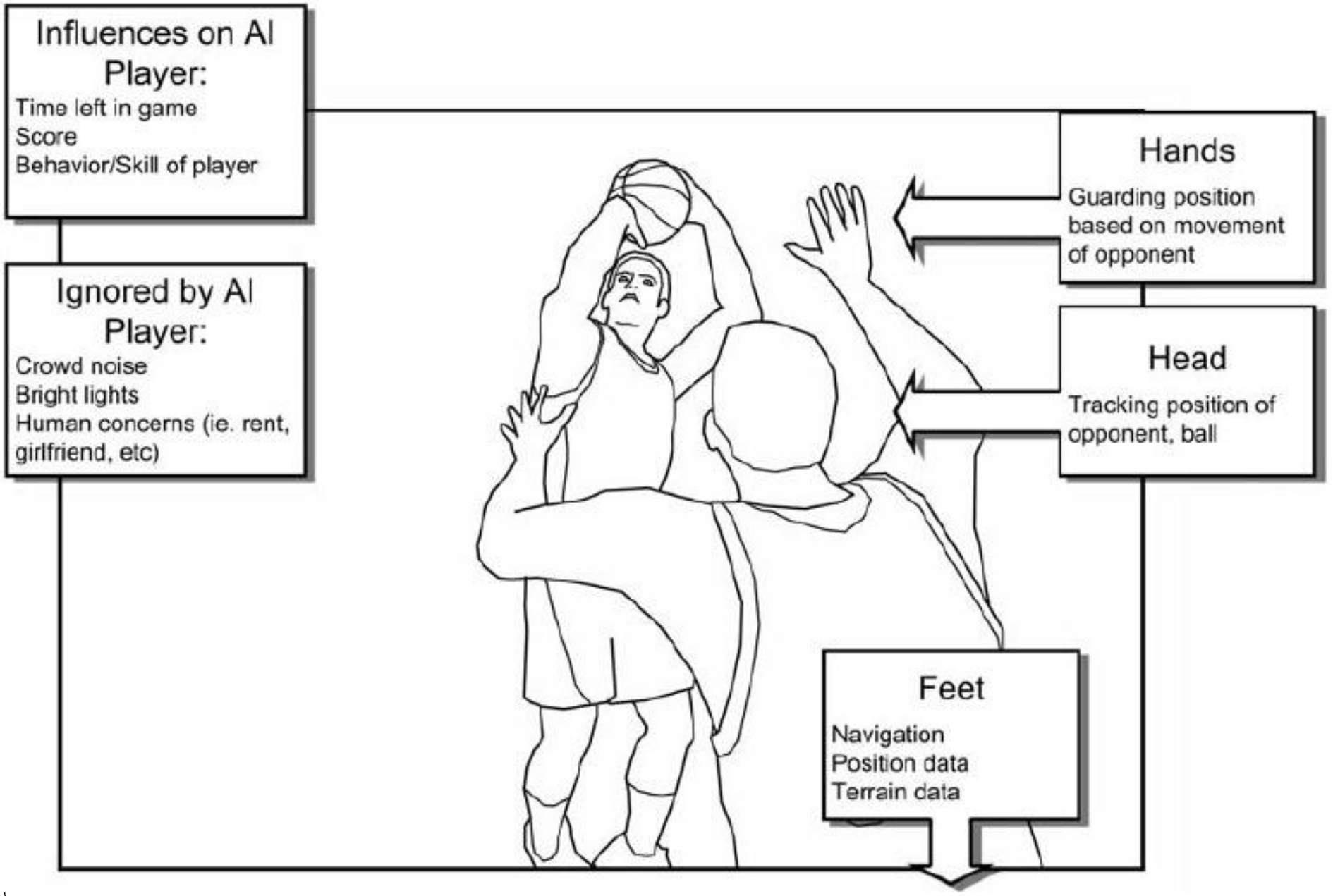
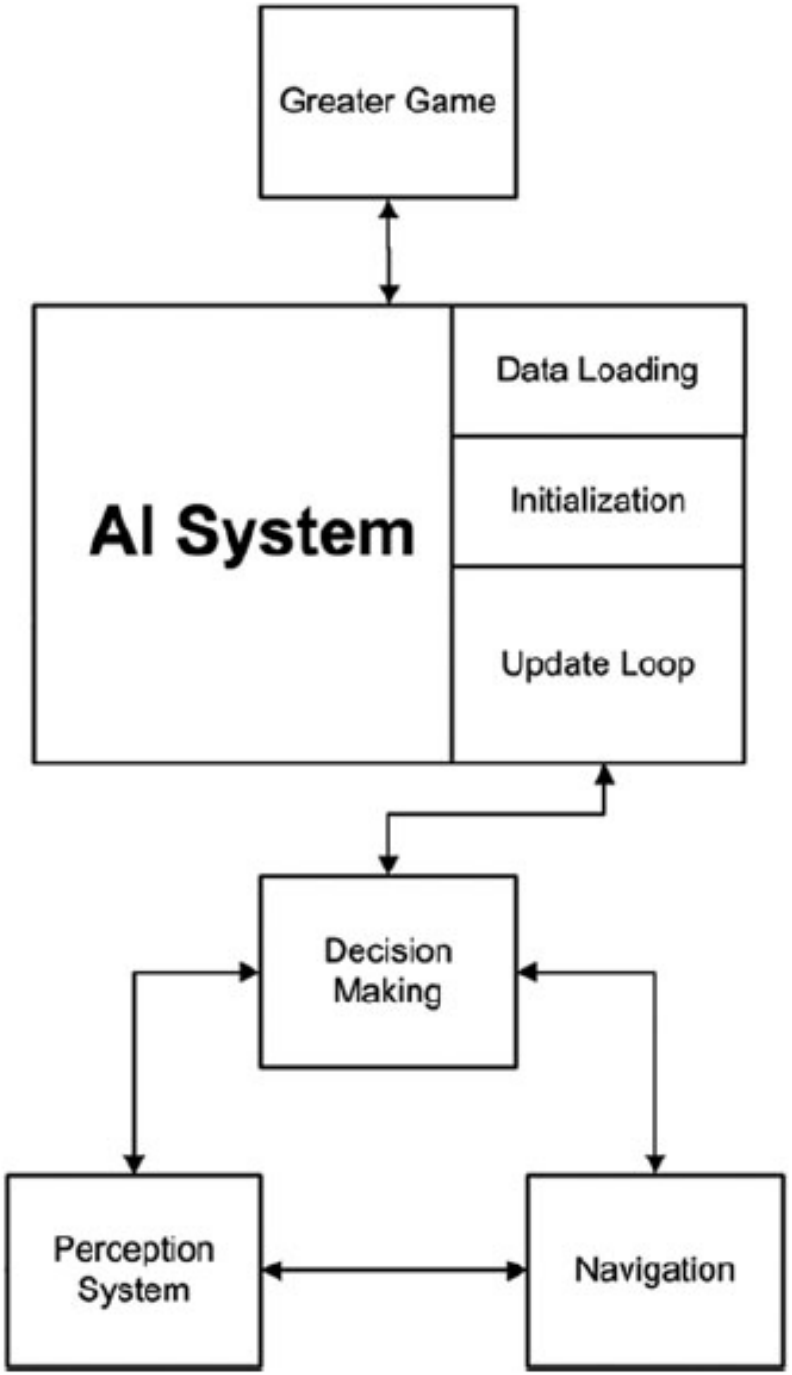
Artificial Intelligence: Navigation in the world!

Behrouz Minaei

Iran University of Science and Technology


Influences on AI Player:
Time left in game
Score
Behavior/Skill of player
Ignored by AI Player:
Crowd noise
Bright lights
Human concerns (ie. rent, girlfriend, etc)
Hands
Guarding position based on movement of opponent
Head
Tracking position of opponent, ball
Feet
Navigation
Position data
Terrain data

Basic AI engine Layout



Navigation

- AI navigation is the art of getting from point A to point B.

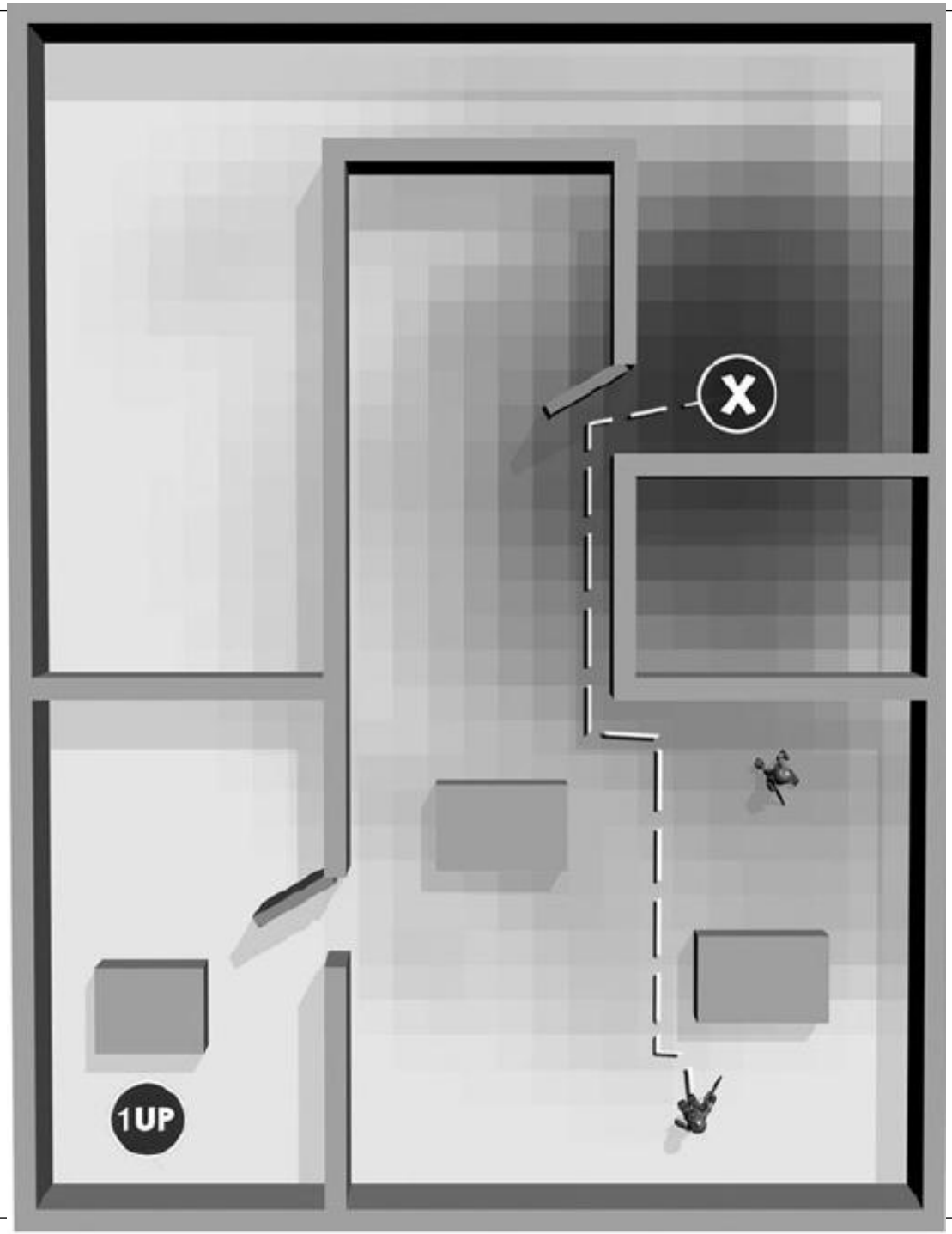
- In our search for more realistic /thrilling /dramatic games, the worlds of modern games commonly involve large, complex environments with a variety of terrains, obstacles, movable objects, and the like.

- In the previous lecture you saw A* as the most common path finding algorithm is video games.
  - A* is from a class of navigation problems called Grid based Navigation

Navigation Grid

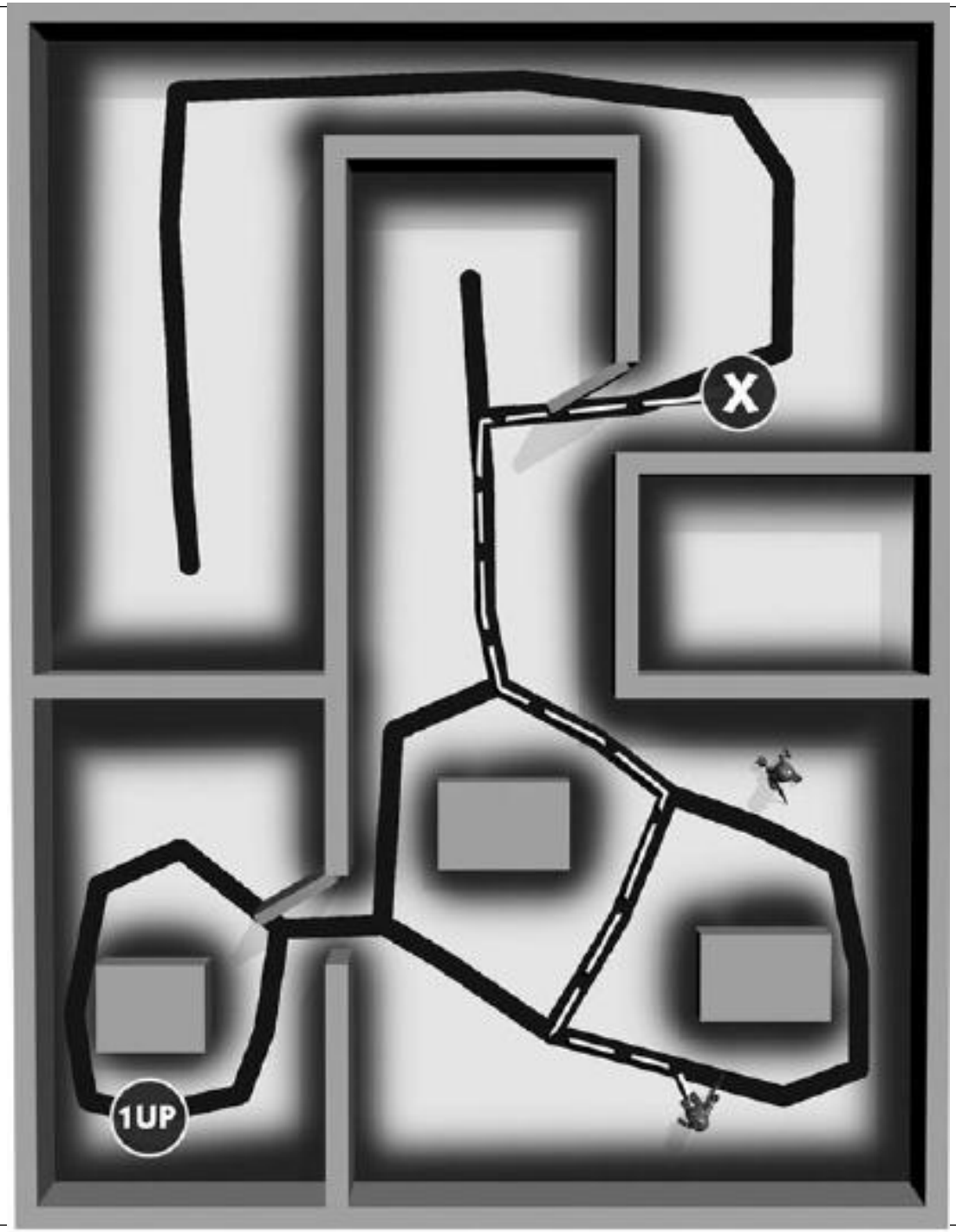
- In a grid-based system, the world is divided up into an even grid, usually either square or hexagonal, and the search algorithm A* or some close relative is used to find the shortest path using the grid.

- In the next slide you'll see an example of a navigation grid in general


X
1UP

Simple avoidance and potential field

- With simple avoidance and potential fields, you again separate the map into a grid.
- You then associate a vector with each grid area that exerts a push or pull on the AI character from areas of high potential to areas of low potential value.
- In an open with convex obstacles, this technique can be preprocessed, leading to an almost optimal Voronoi diagram of the space
  - (that is, a mathematically sound optimal "partition" of the space) providing good quality, fast pathfinding.
- The paths are extracted from the map by simply following the line of decreasing potential .


X
1UP

Grid Based Navigation

- With concave obstacles, however, you cannot preprocess because the vector would depend on a particular character's approach angle and direction of travel.

- In this case, the pressure is now on the runtime potential field generator.

Map Node networks (waypoint path finding)

- With this method, the level designers, during world construction, actually lay down a series of connected waypoints that represent interconnectedness among the rooms and halls that make up a particular game space.

- Then, just like the grid-based method, a search algorithm (most likely A*) will be used to find the shortest connected path between the points.

- In effect, you are using the same technique as described earlier, but are reducing the state space in which the algorithm will operate tremendously.

- The memory cost is much less for this system, but there is a cost. The node network becomes another data asset that has to be created correctly to model intelligent paths, and maintained if the level is changed.

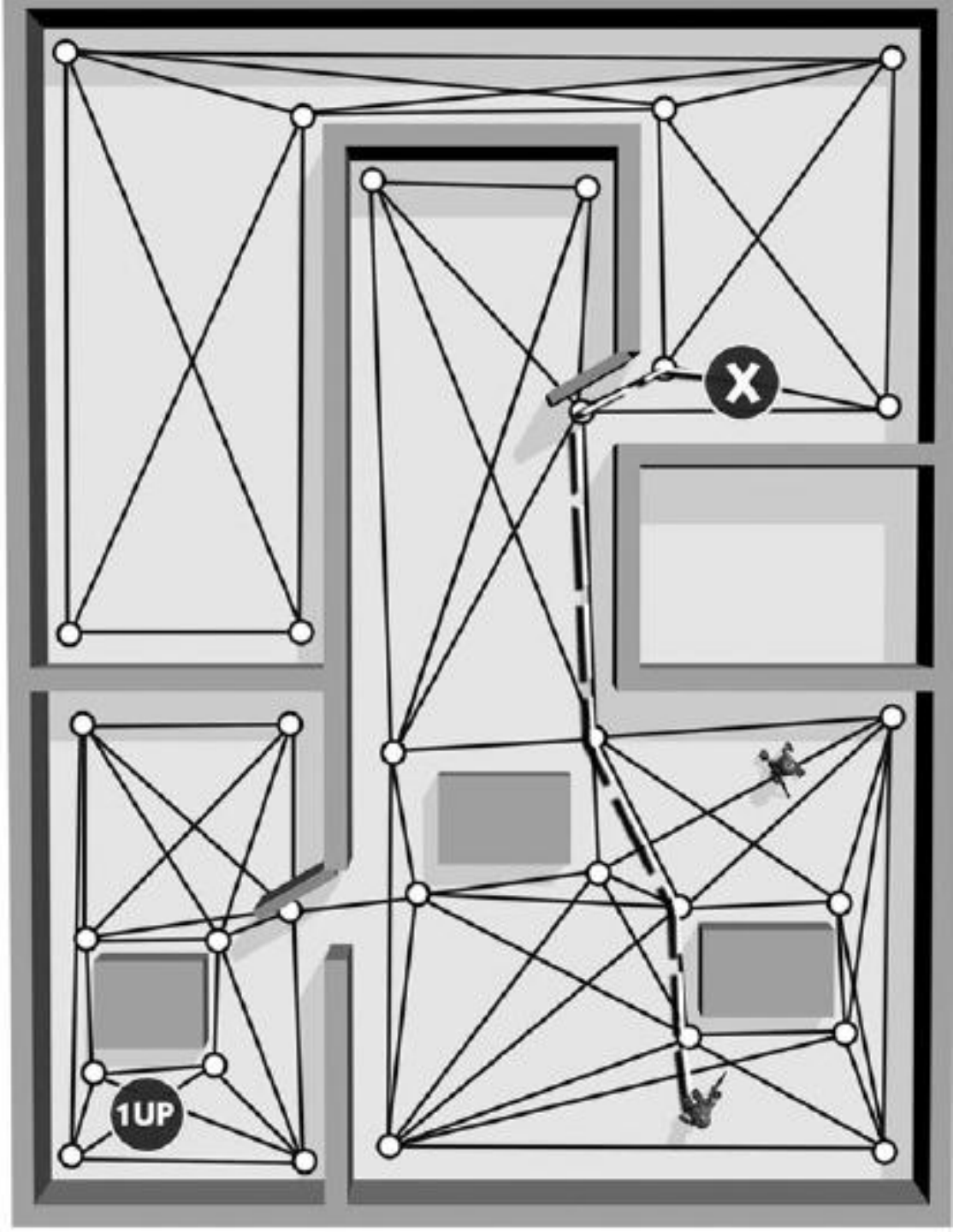
Map Node networks (waypoint path finding)

- Also, this method doesn't lend itself well to dynamic obstacles, unless you don't mind inserting/removing the dynamic object locations into and out of the node network.

- A better way is to use some form of obstacle avoidance system to take care of moving objects, and use the node network to traverse the static environment.

Map Node networks

- Waypoint pathfinding



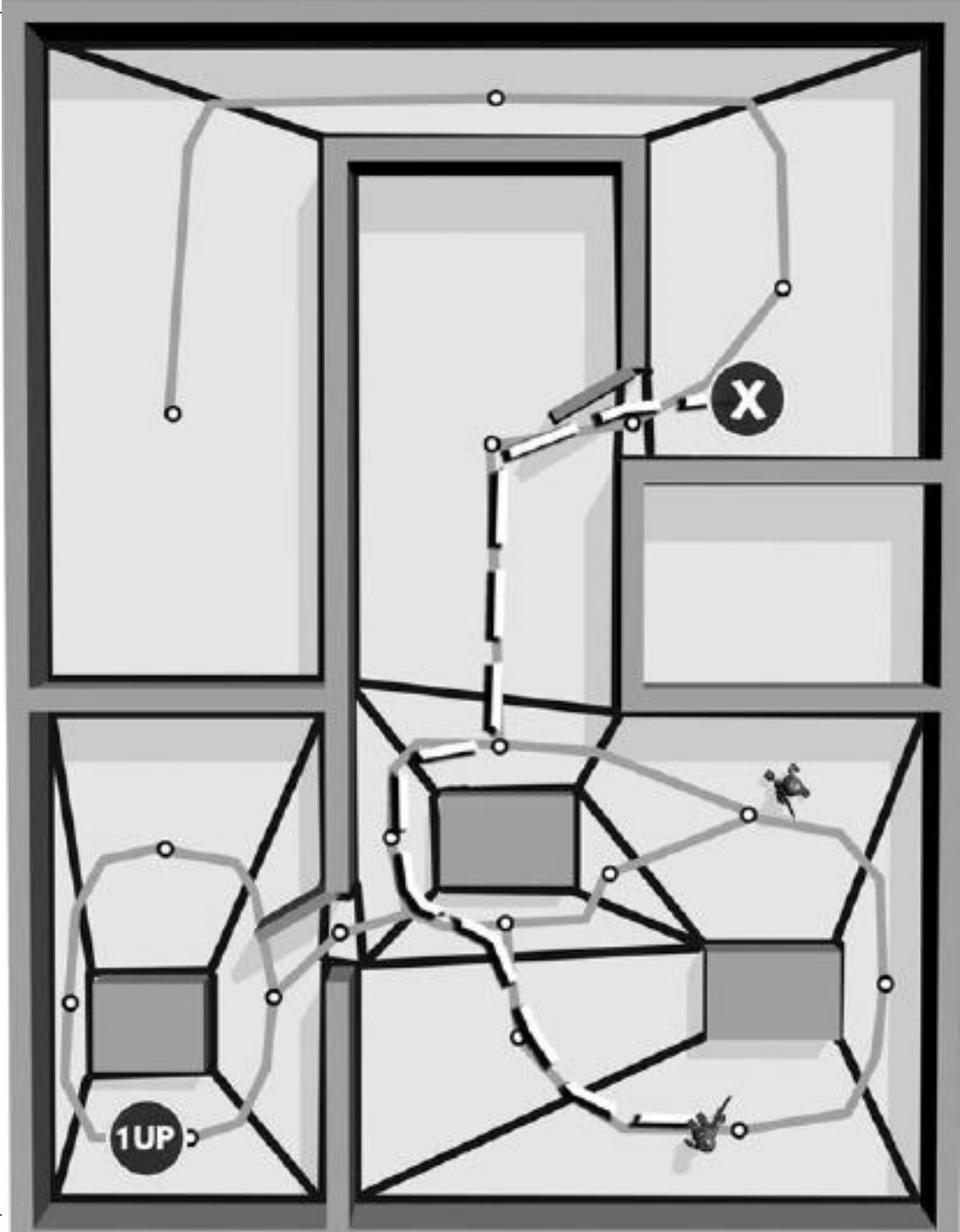
Navigation Mesh

- A navigation mesh system tries to get all the advantages of the map node system, without having to create or maintain the node network.

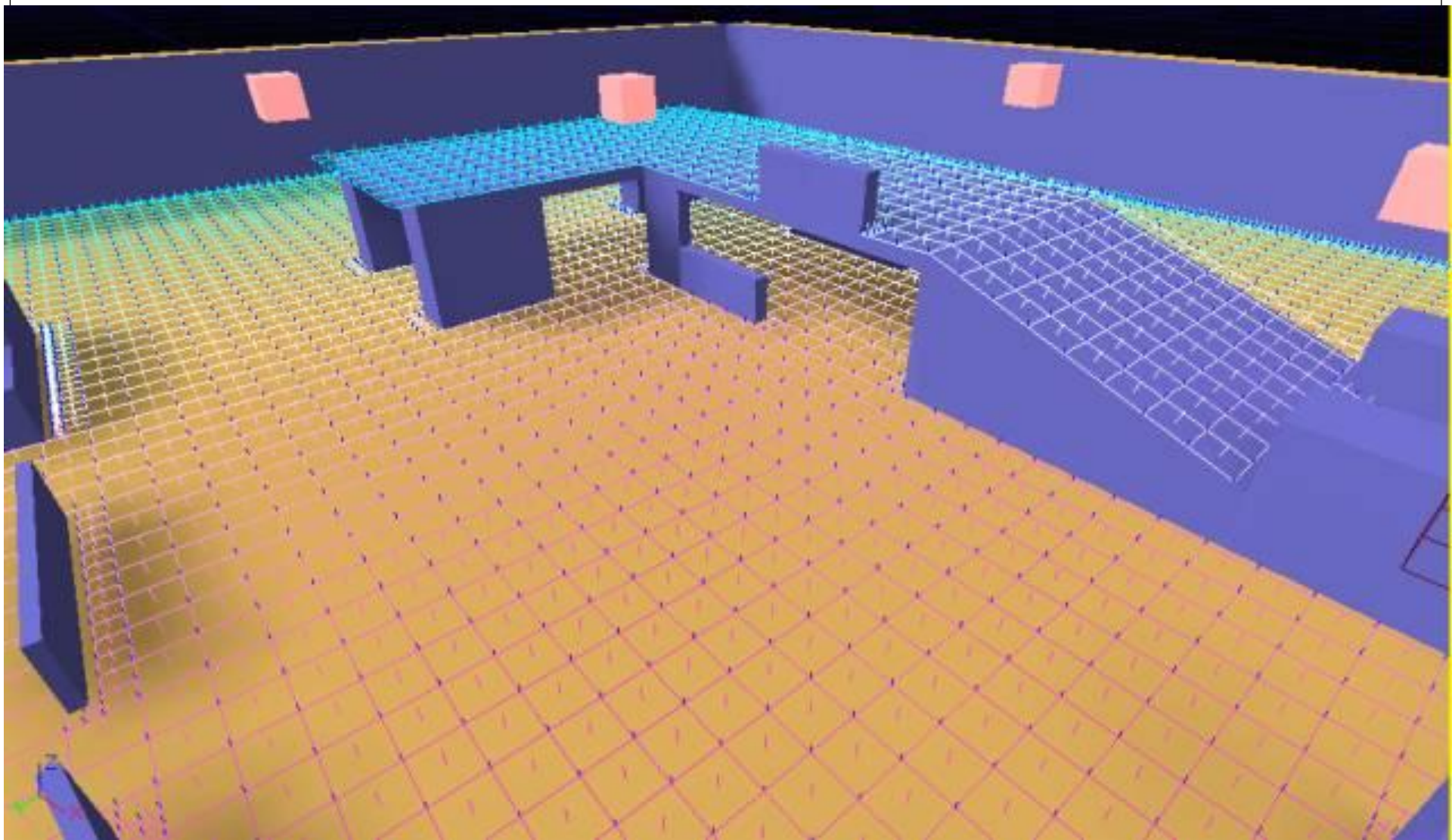
- By using the actual polygons used to build the map, this system algorithmically builds a path node network that the AI can use.

- This type of system is best used for simple navigation, because game play- specific path features (such as stairs or elevators) can be difficult to extract with a general algorithm.

Navigation Mesh



Navigation Mesh



Navigation Graphs

- An alternative approach to navigation meshes.

- A reminder of navigation mesh
  - Agents in game can use a navigation mesh to find the path from location A to location B.

  - First, given the initial agent's location A, the algorithm finds what triangle Ta of the mesh contains A and what triangle Tb contains B (the destination).

  - Then, a shortest path across the mesh can be built. This path is a list L of connected triangles from triangle Ta to triangle Tb. Usually this is done using A* (a-star) algorithm.

Navigation Graphs

- A reminder of navigation mesh
  - Navigation meshes can often be naturally derived from collision geometry in an automated or completely automatic manner.
  - It can be updated in run time as well, but this may be challenging due to the performance constraints unless we deal with only small portions of the mesh at a time.

Navigation Graphs

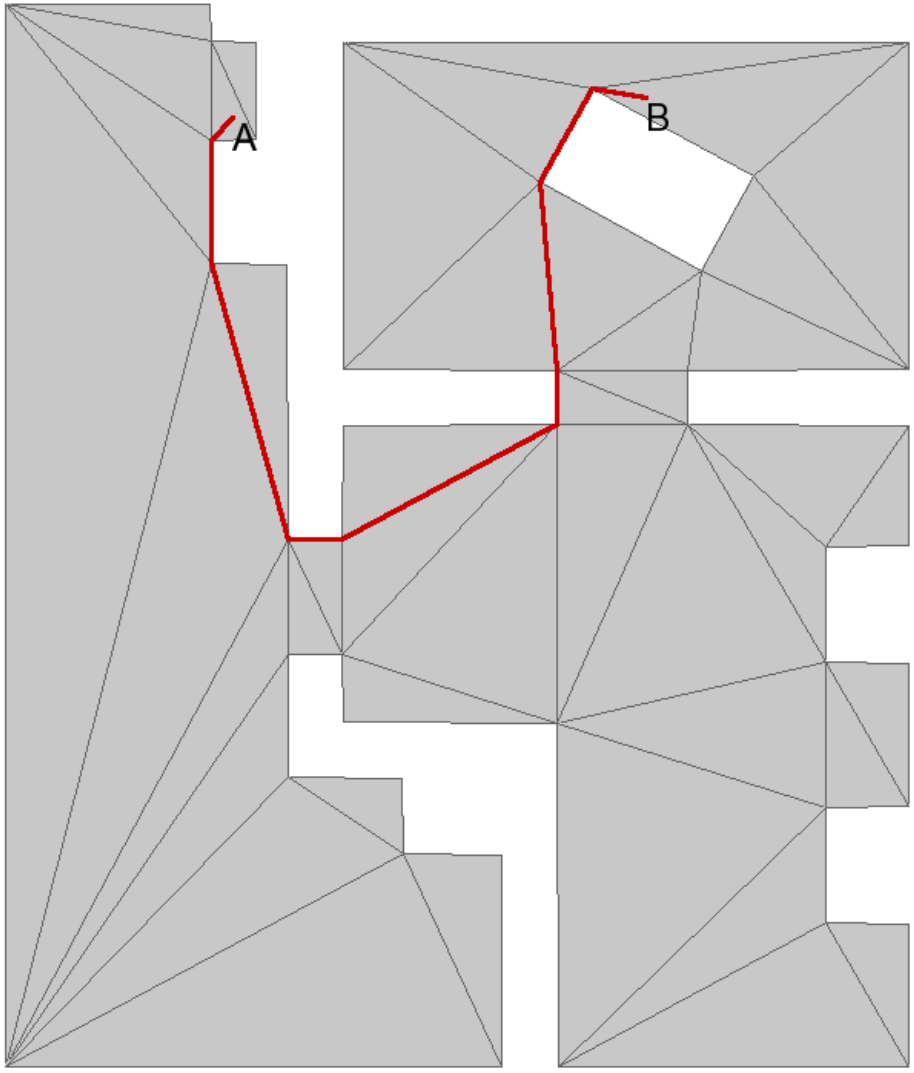
- Namely, the graph is based on nodes, which are not intended to cover all navigable geometry but rather provide useful markers and serve as waypoints.

- set of 2D primitives (circles, quads and the like) can be used to represent nodes in a navigation graph.

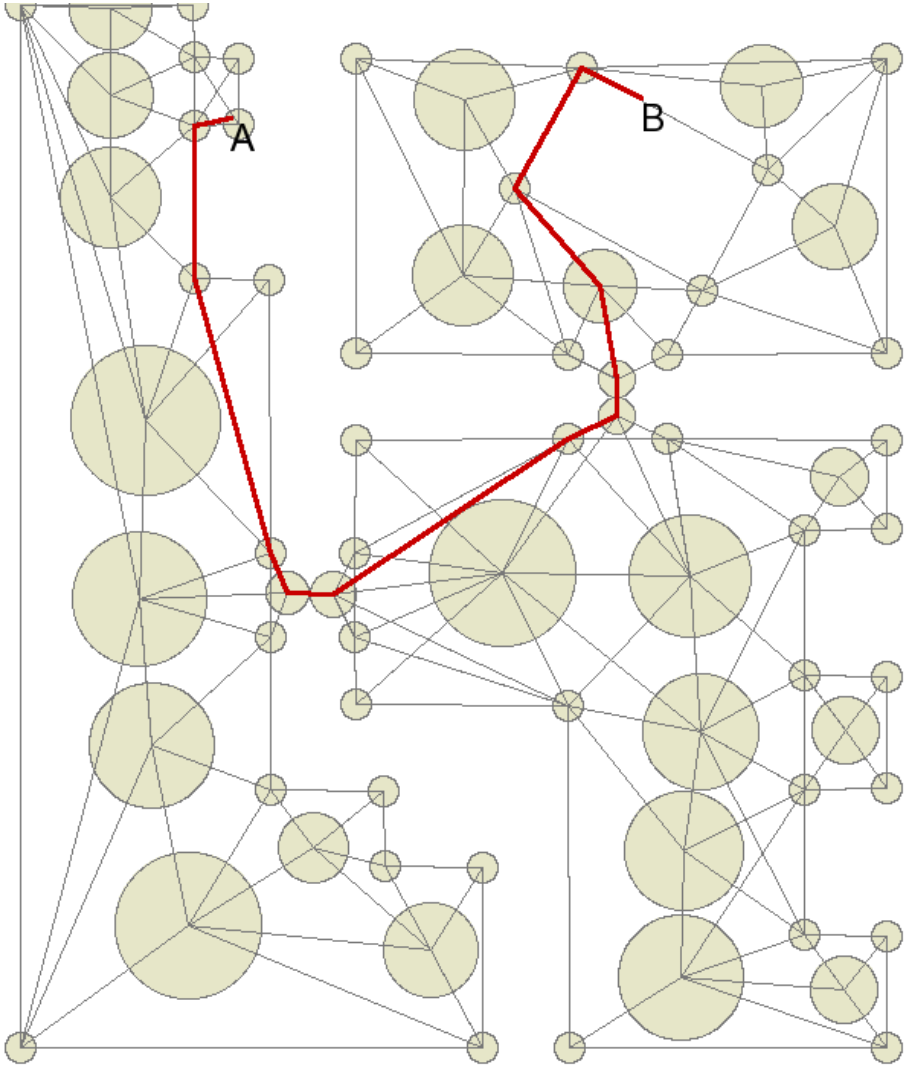
- Any path inside a node is considered valid. Also, a path (usually - a straight line) from any point in the environment to any point inside any node is considered to be valid provided it satisfies some additional constraints.
  - The most common constraint is that the path does not intersect collision geometry.

Navigation Graph

Navigation Mesh



Navigation Graph

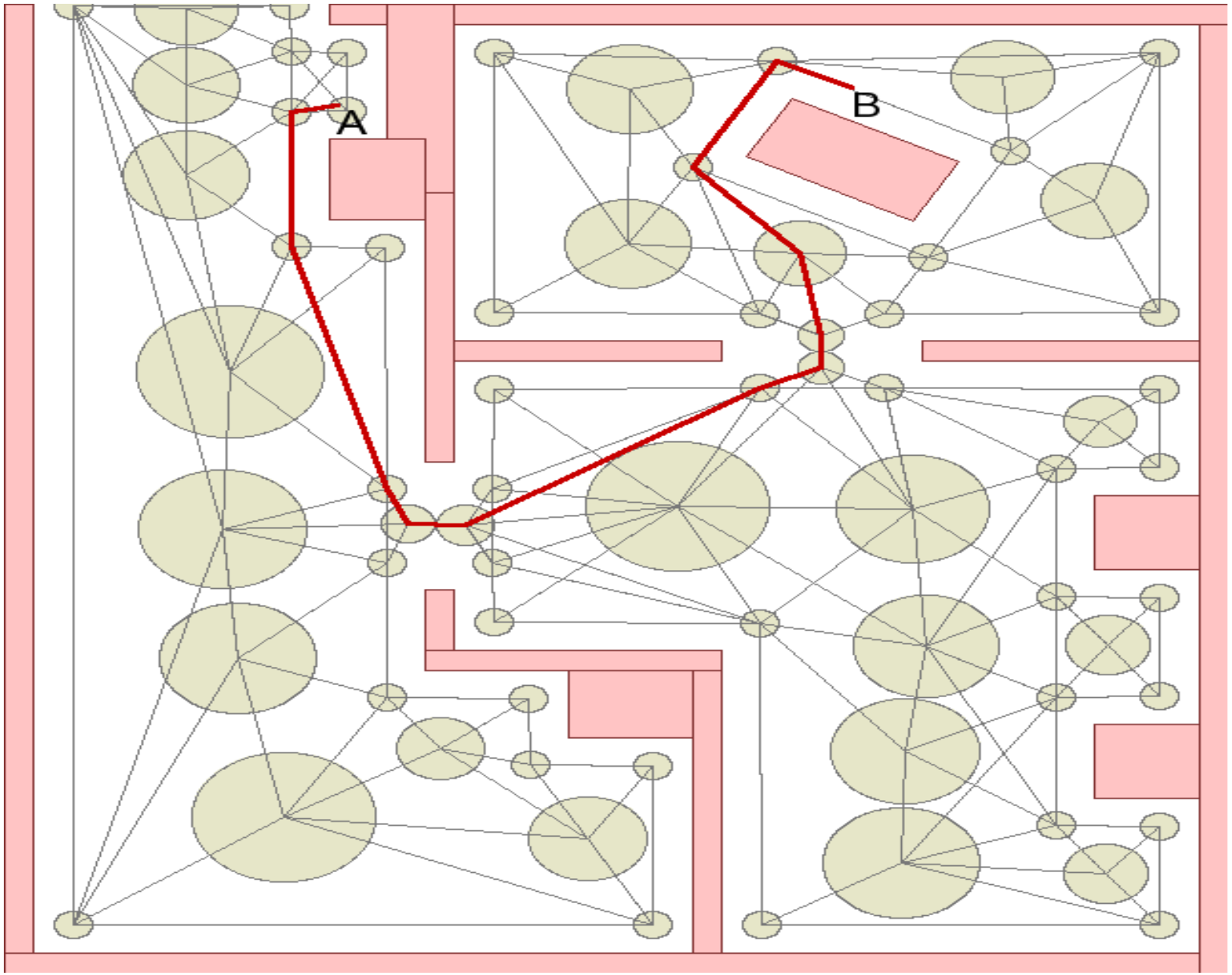


Navigation Graph

- More often than not, checking against collision and other terrain properties along the path could be expensive.
- Such checks can be replaced by checking against much simpler representations of un-navigable geometry via blocking primitives.
- The blocking primitives are by far less-detailed and are often limited to boxes (oriented or axis-aligned) and cylinders.
- This allows for very efficient testing of ray intersection against blocking primitives.
- Note that the navigation mesh usually doesn't require addition of blocking primitives as all information about the navigable area is already included in the navigation mesh

Navigation Graph

- Given a navigation graph and a set of blocking primitives, we can construct a path using the following algorithm:
  - From starting point A we look for the closest navigation node Na such that the line of view from A to Na is not blocked by any primitive.
  - In a similar manner we find node Nb for destination point B.
  - Then using a-star we find a list of nodes connecting Na to Nb.


A
B

Navigation Graph Advantageous

- Navigation graphs may offer certain advantages over navigation meshes.
- To start with, the graph can easily incorporate navigation slots of objects placed into the environment.
  - Say, we need an agent to approach a door and orient itself in a certain way to successfully open it. With navigation graphs the slots indicating desired position near the door can become just another node in the graph.
  - Second, since the navigation graph is not tied to the level geometry in the same direct way as navigation mesh, it can offer extra flexibility during level design. The nodes are usually represented as objects in the level design tools. As such they can be easier manipulated than triangles in a navigation mesh where designers should care about preserving topological properties of the mesh.

Navigation Graph Disadvantageous

- Needless to say, manual generation of a navigation graph and its tweaking can be very time consuming.

- Another downside is that such handcrafted graphs are difficult to update if the geometry changes during the design process and they require extra care in case of run-time modifications in game.

Navmesh v.s Navgraph

- Summarizing, we may conclude that :
  - navigation mesh is more appropriate when navigation data is generated automatically
  - and a navigation graph may be more preferred when level designers' input is needed or necessary.

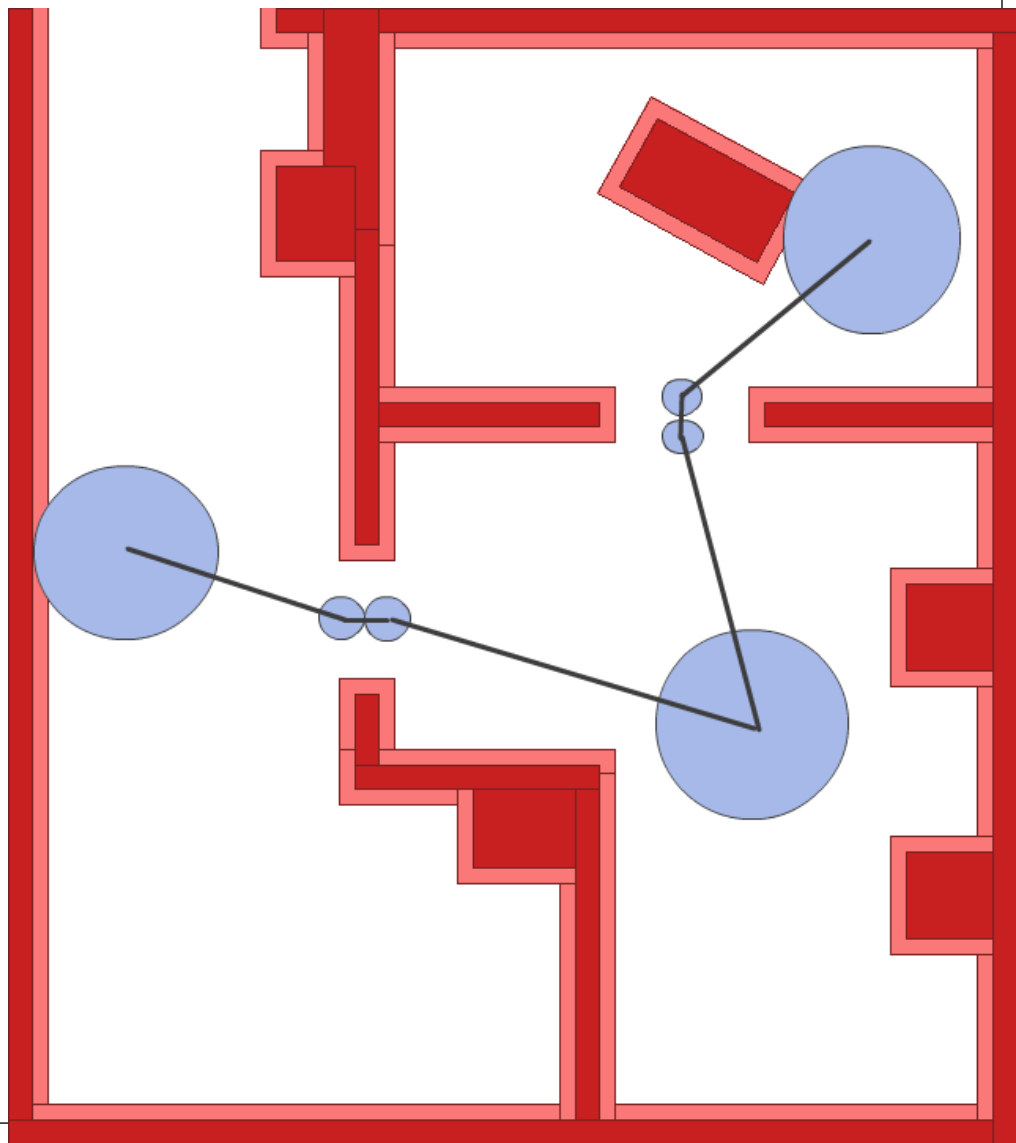
Hierarchical Navigation Graphs

- Navigation graphs can be used hierarchically

- This is the most useful when we have sparse outdoor environments that themselves are indoors locations.
  - In this case we can use a higher level navigation graph for determining some "Portals" in our environment, which themselves are detailed navigation graphs.

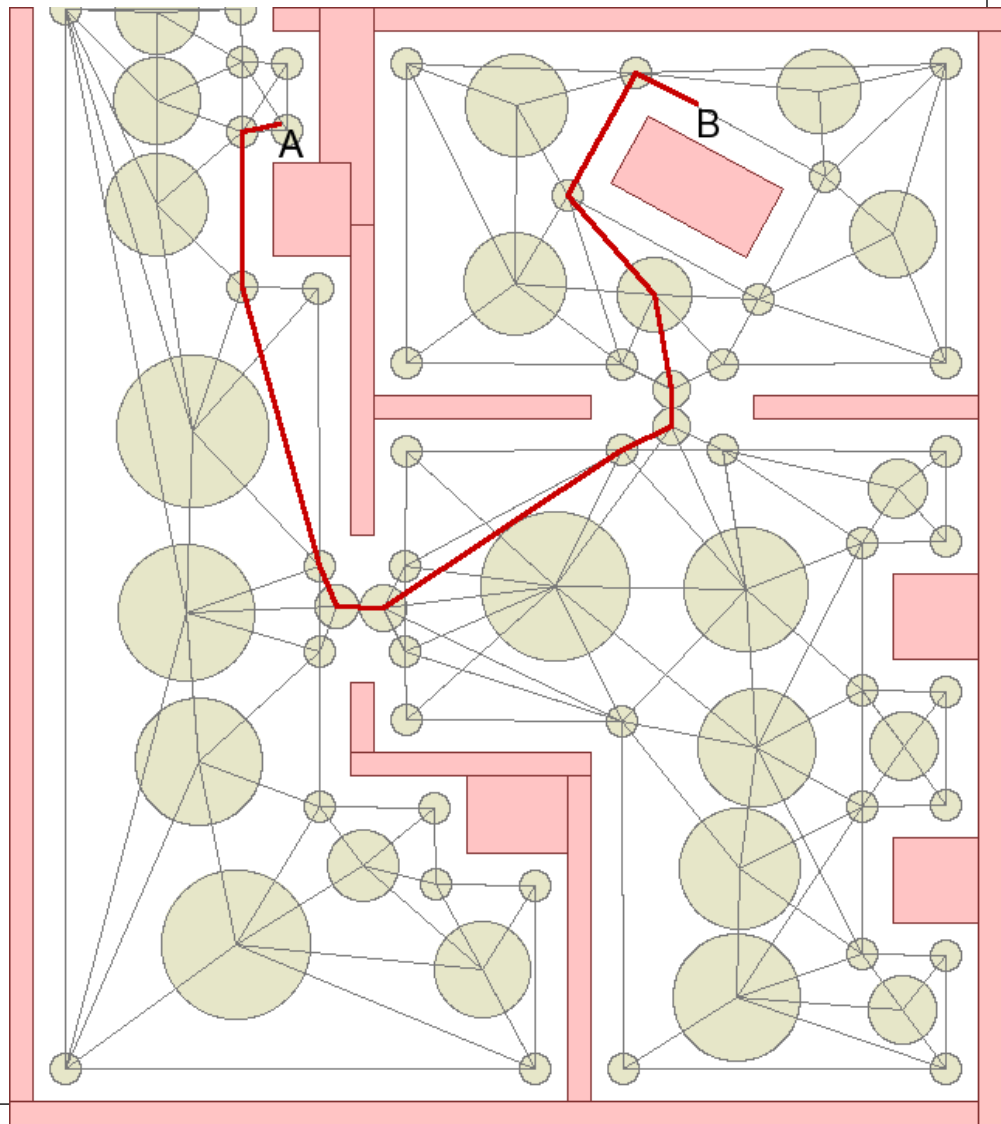
Higher level navigation graph

- As you see super nodes determine the entry points of portals in our game



Detailed Navigation Graph

- Each super node is decomposed to finer grained navigation graphs.



Navigation Graph Advantageous

- Partitioning of the level into regions corresponding to the nodes of super graph has additional benefits.

- One of them is that we can worry less about situations like when the nearest node is picked across blocked terrain.

- If the search for nearest node is limited to the current region then such situations will happen much less often and the search itself will speed up.

  Apparently two-level hierarchy can be generalized to more levels but this is rarely justified in practice.

Influence Mapping

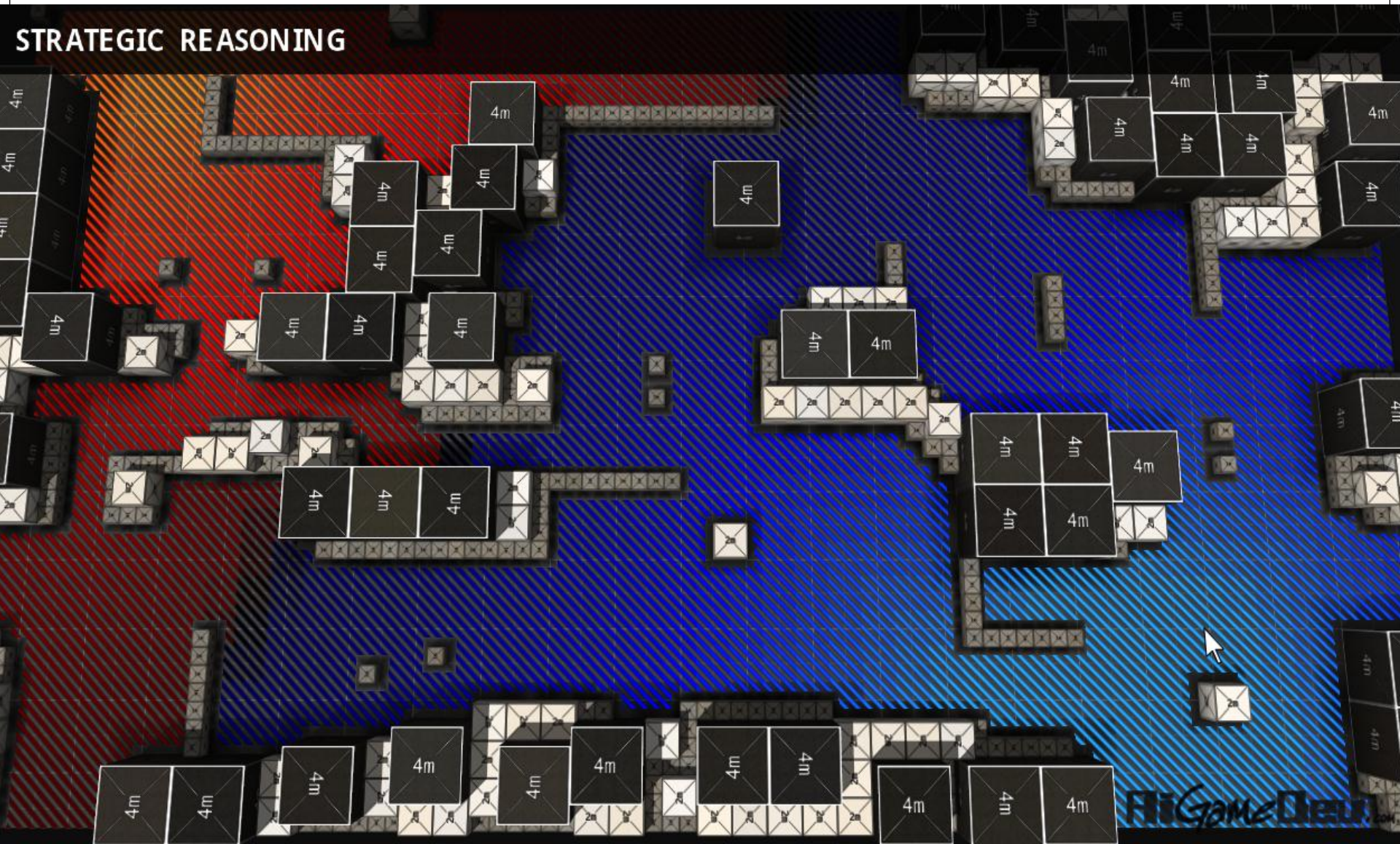
- The main method of navigation in Killzone 2 and 3!!

Influence Mapping: Motivation

- Influence maps ultimately help your AI make better decisions by providing useful information about the world.

- In particular, influence maps provide three different types of information that are particularly useful for decision making:
  - **Situation Summary** — Influence maps do a great job of summarizing all the little details in the world and making them easy to understand at a glance. Who's in control of what area? Where are the borders between the territories? How much enemy presence is there in each area?
  - **Historical Statistics** — Beyond just storing information about the current situation, influence maps can also remember what happened for a certain period of time. Was this area being assaulted? How well did my previous attack go?
  - **Future Predictions** — An often ignored aspect of influence maps, they can also help predict the future. Using the map of the terrain, you can figure out where an enemy would go and how his influence would extend in the future.

Influence Mapping



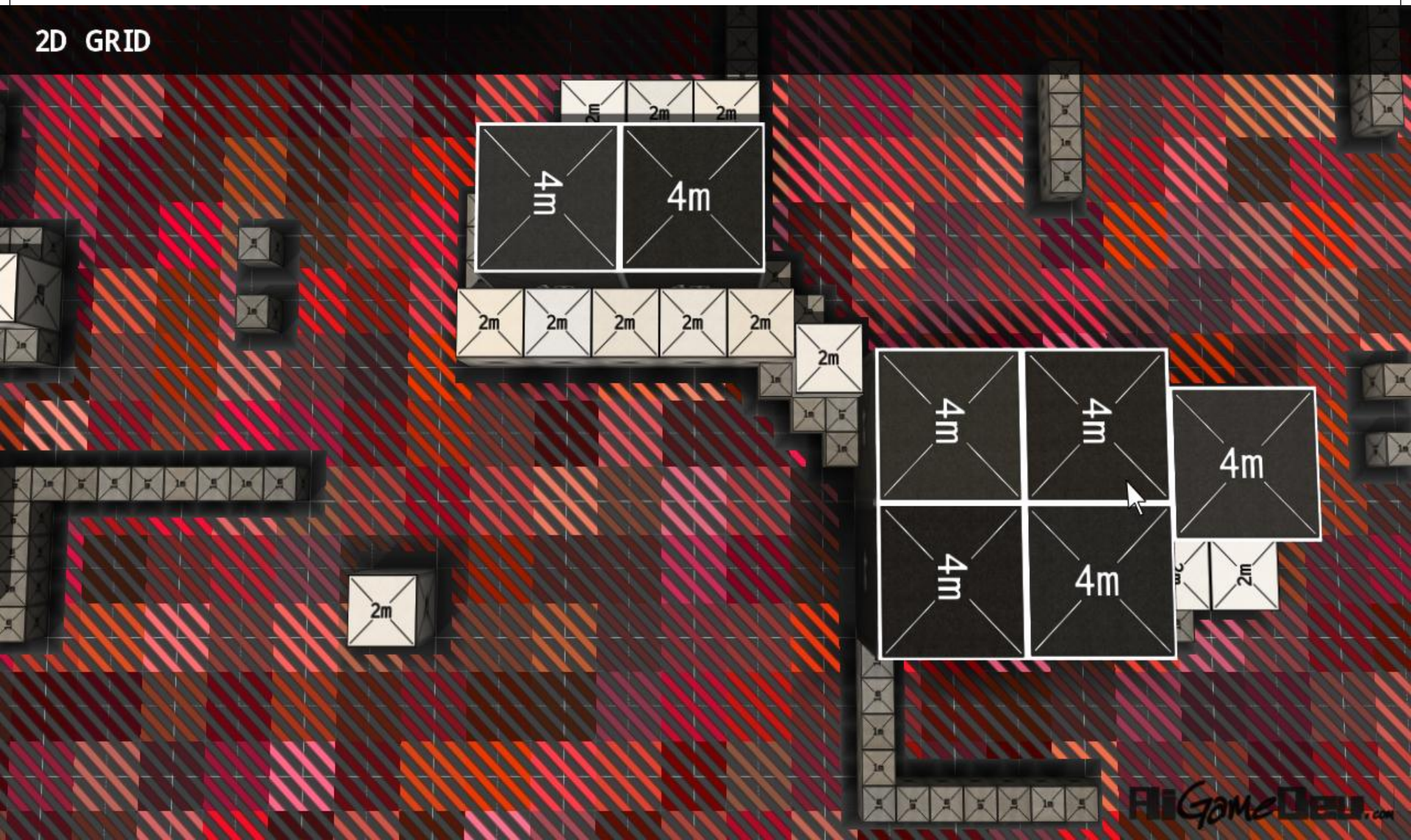
Influence Mapping: representation

- Generally there are two things you need from your influence map representation:

  - **Spatial Partition** — A way to easily and efficiently partition space and store information (i.e. influence) for each partition. This allows the influence map to store information about the past, gathering statistics about what happened.

  - **Connectivity** (optional) — An indication of connectivity between these spatial partitions in space. The connections allow the influence mapping algorithm to predict how influence could spread through the level, predicting what could happen in the future.

Influence Mapping: 2D grid representation



Influence Mapping: 2D grid representation

- 2D grid is a great default representation if you can map your world to a mostly 2D environment.
- It's a very fast representation to process and very simple to implement.
- The downsides, however, are that you may waste memory if you have sparse environments.

Influence Mapping: Area Graphs representation

- If you have a navigation hierarchy in place already, then you can use that as the basic representation for the influence map.
- The advantage of this approach is that it's not a very intrusive change and can easily be incorporated into most game engines.
- The disadvantages are the lack of precision in cases where details are required for the decision making.

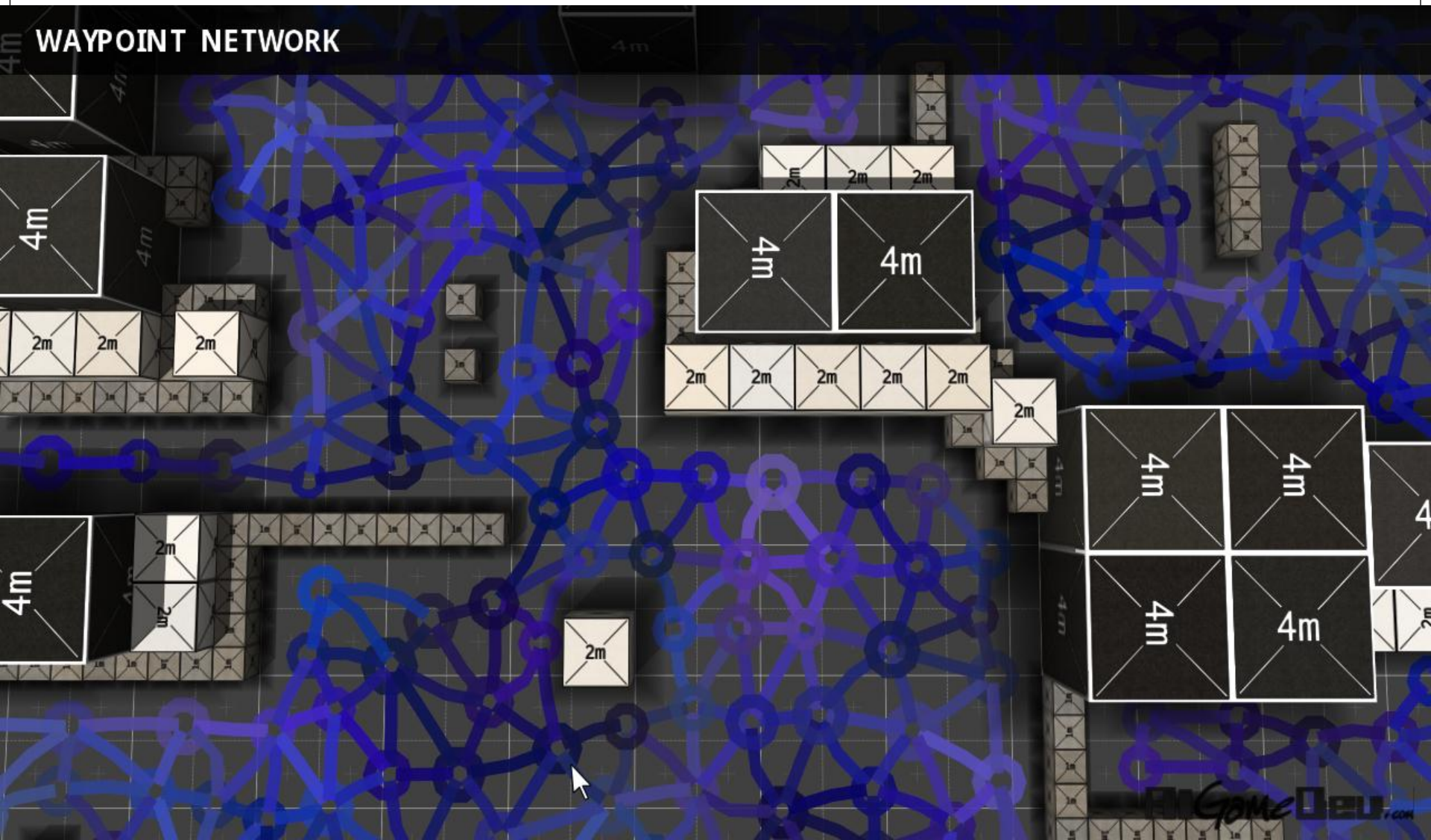
Influence Mapping: Area Graphs representation

Influence Mapping: waypoint network representation

- Using a full waypoint network in 3D resolves some of the problems with 2D grids, as you can easily wrap a waypoint network over multiple levels of a building or upstairs.

- However, the downside is that it's much less efficient to process than both a grid or an area graph.

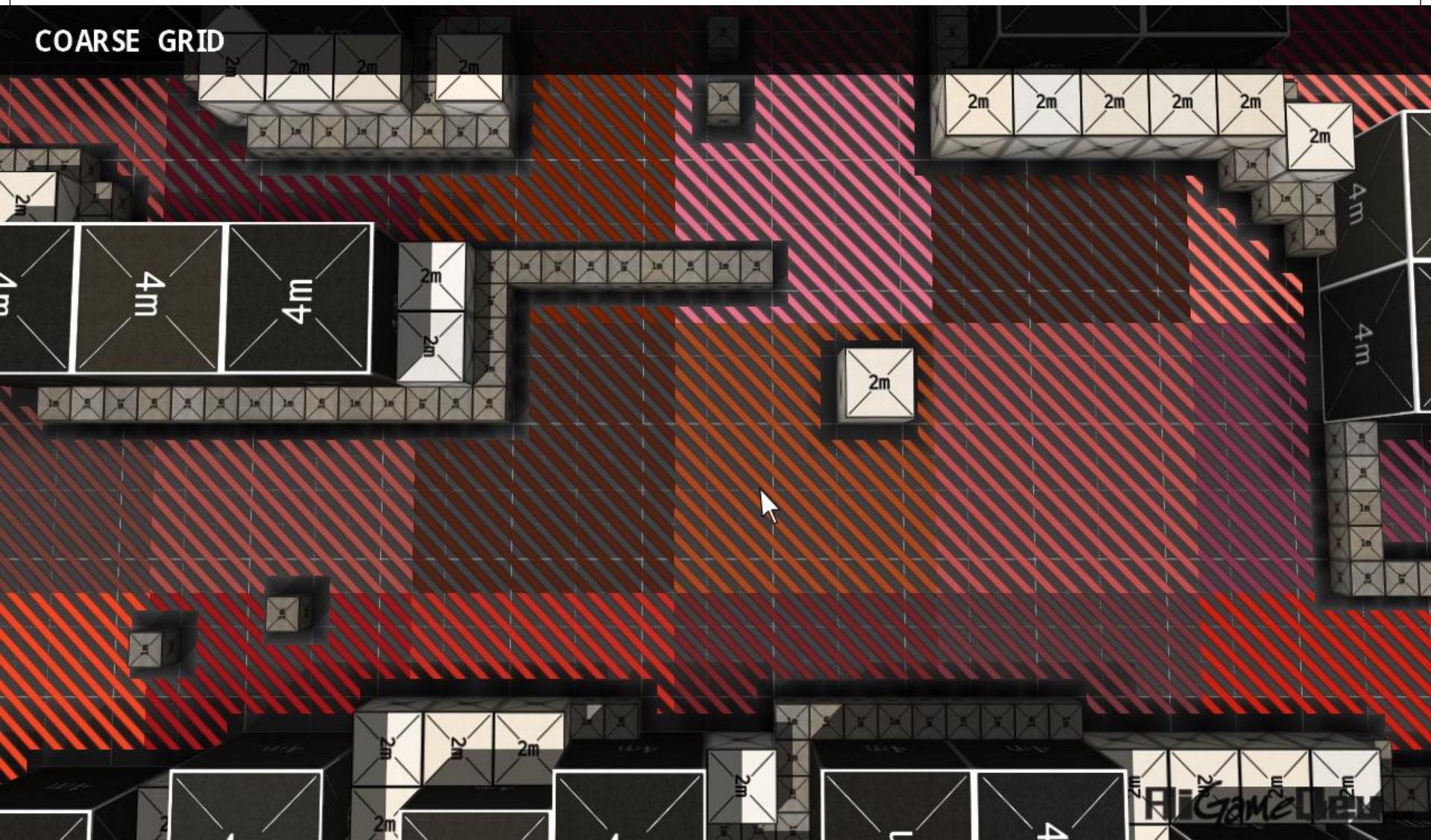
Influence Mapping: waypoint network representation



Influence Mapping: coarse grid representation

- One last option is to decrease the resolution of the grid.
- Unfortunately, this causes certain grid cells to span over obstacles, which can cause problems when updating the influence.
- The alternative is to remove those split cells from the representation, but certain connectivity may be lost because of it.
- Using these cells as disconnected buckets in space to store influence is also an option, though it rules out predicting the spread of the influence in the future.

Influence Mapping: coarse grid representation



Influence Mapping Algorithm

- Fundamentally, the algorithm is similar to a blurring process that you can find in photo editing software like Photoshop.

- You start by setting the influence values in your map, and then repeatedly blur the map to spread the influence from the source towards neighboring nodes.

  The influence mapping algorithm is actually pretty flexible. It's made up of two different steps, but you can run those steps in any order you see fit and customize them quite extensively.

Influence Mapping Algorithm

- Step1 : Setting the influence!
- Typically, the first step is to set the influence sources inside the representation you chose.

- It's as easy as storing or updating a floating point number in an array of influence values.

- The only challenge is figuring out the influence sources for your game. Often these sources can be:
  - Entities, both friendly and enemy soldiers, semi-permanent turrets, etc.

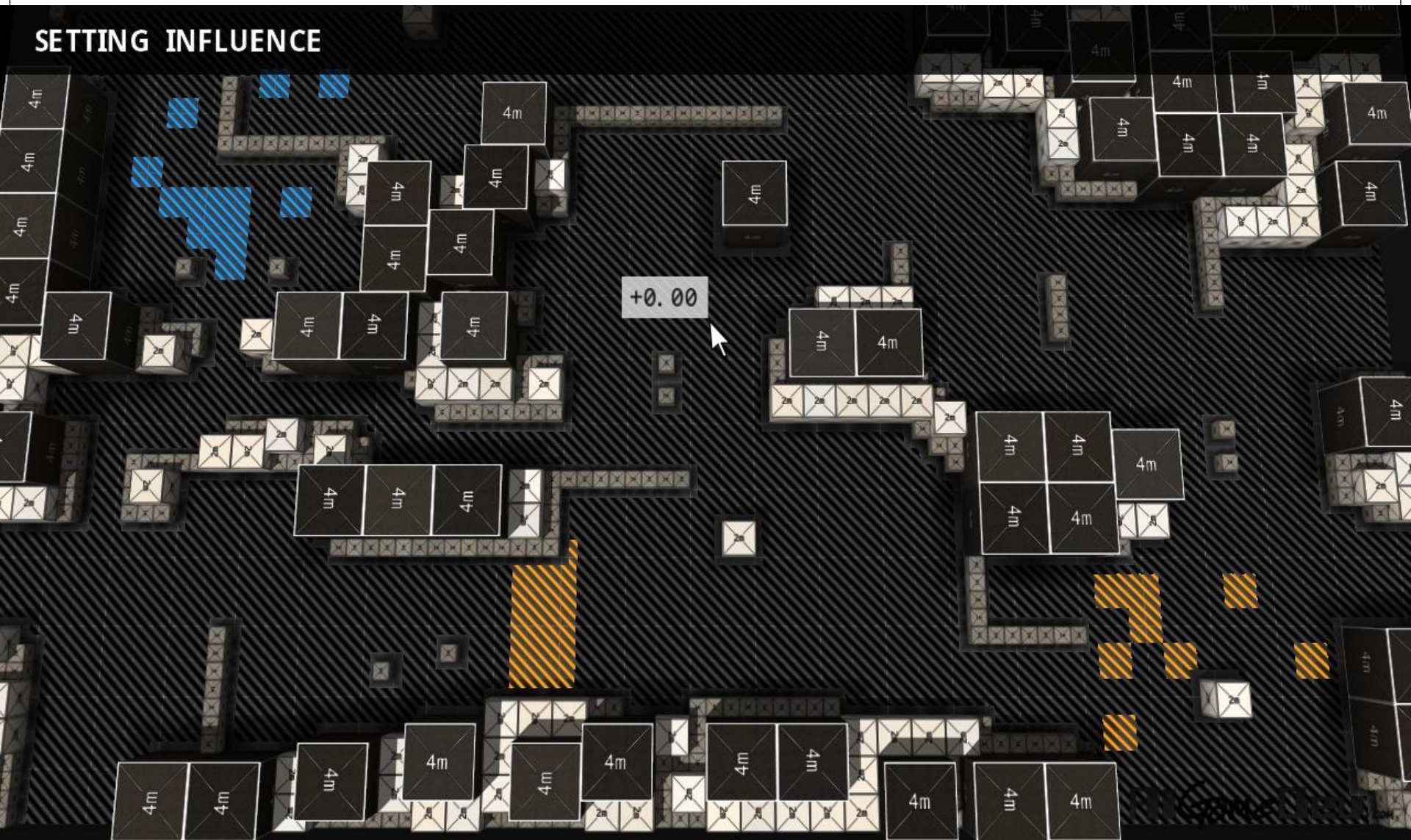
  - Events like grenade explosions, bullet fire, taking damage.

Influence Mapping Algorithm

- You'll also need to figure out which of these influence sources are additive (layered on top of the existing influence) and which are reference (set as the base influence value).

- That depends on your game, but temporary events like bullet fire are well suited to additive influence.

Setting the influences



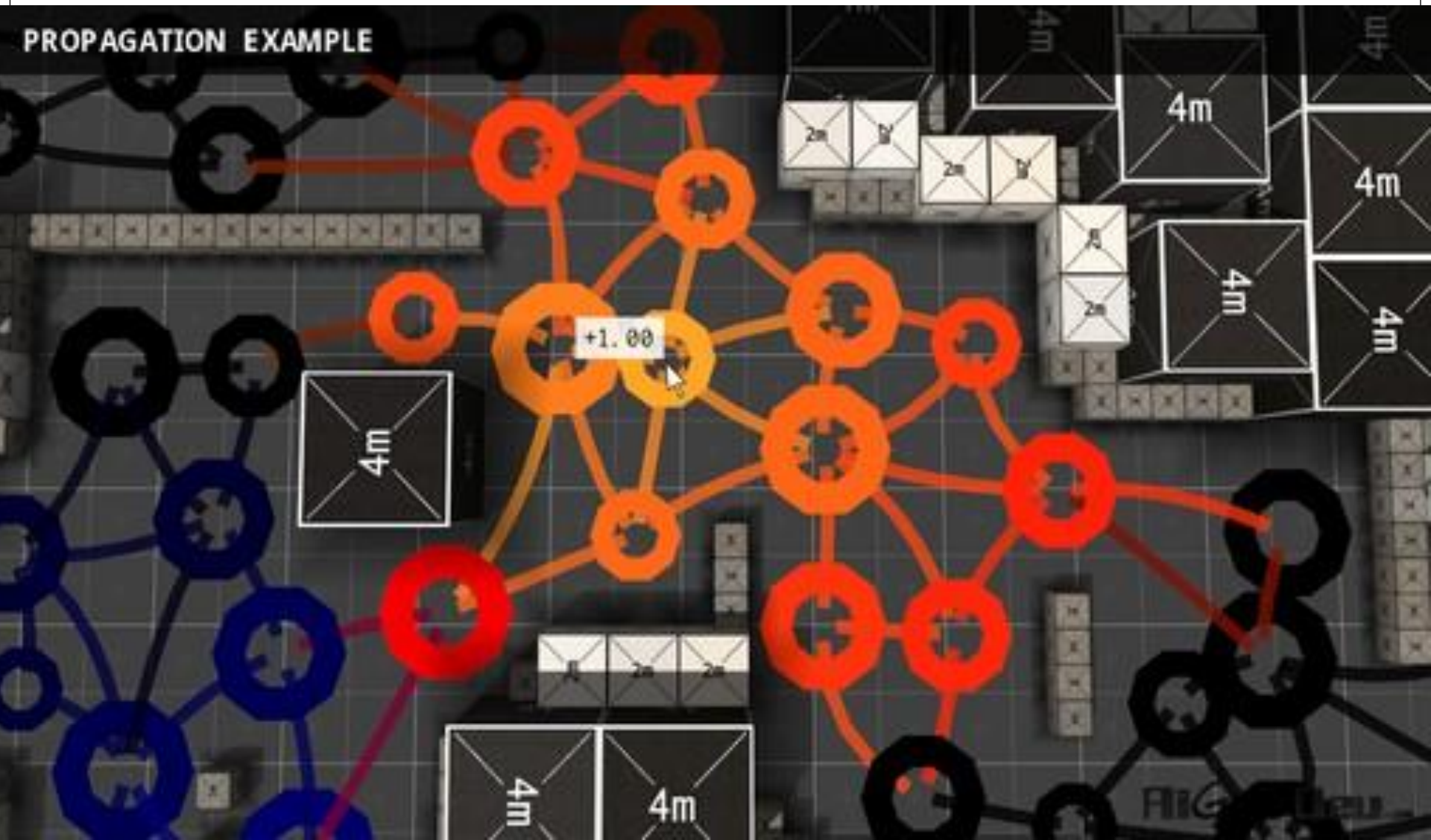
Influence Mapping Algorithms

- Step2: Propagation

```
void InfluenceMap::propagateInfluence()
{
  for (size_t i = 0; i < m_pAreaGraph->getSize(); ++i)
  {
        float maxInf = 0.0f;
        Connections& connections = m_pAreaGraph->getEdgeIndices(i);
        for (Connections::const_iterator it = connections.begin();
                it != connections.end(); ++it)
        {
          const AreaConnection& c = m_pAreaGraph->getEdge(*it);
          float inf = m_Influences[c.neighbor] * expf(-c.dist * m_fDecay);
          maxInf = std::max(inf, maxInf);
        }

        m_Influences[i] = lerp(m_Influences[i], maxInf, m_fMomentum);
  }
}
```

Propagation of influence



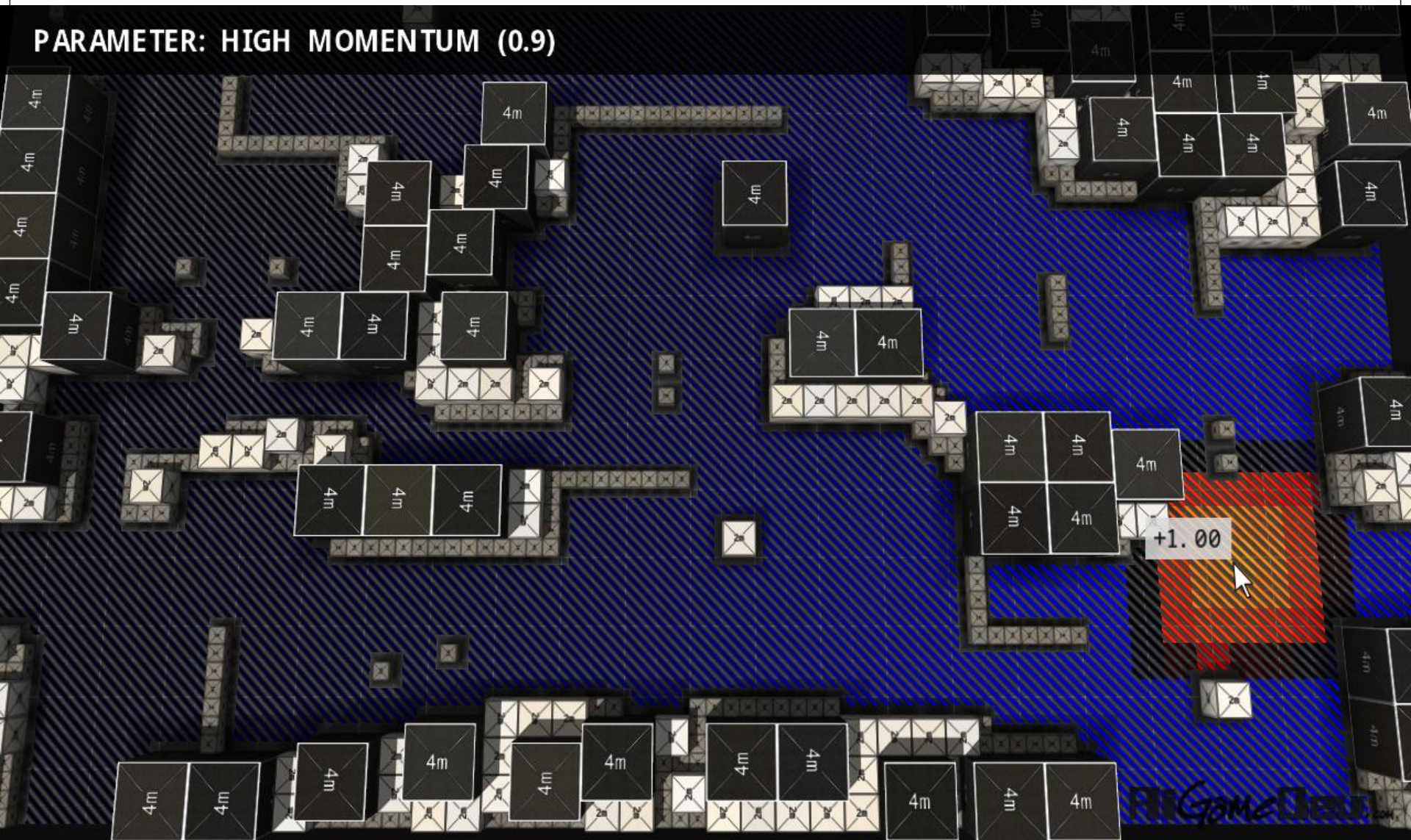
Important Parameters

- Some Parameters in the previous algorithm is important
  - Momentum
  - Decay Rate
  - Update Rate

Algorithm Parameter

- Momentum
  - When you update the influence value, how much do you bias the update towards the existing value compared to the new value?
  - The code above uses linear interpolation to blend from the current value to the new value, and then relies on the momentum parameter to control the result.
  - If you set the momentum to high (closer to 1.0) then the algorithm will bias towards the historical values of the influence, which is particularly well suited to storing statistics about previous attacks. Use this for things like high-level strategic maps.
  - Conversely, if you set the momentum parameter to low (closer to 0.0) then the algorithm biases towards the currently calculated influence, so the propagation happens quicker and the prediction is more accurate. Use this for low-level influence maps for individual positioning for example.

Momentum



Algorithm Parameter: Decay

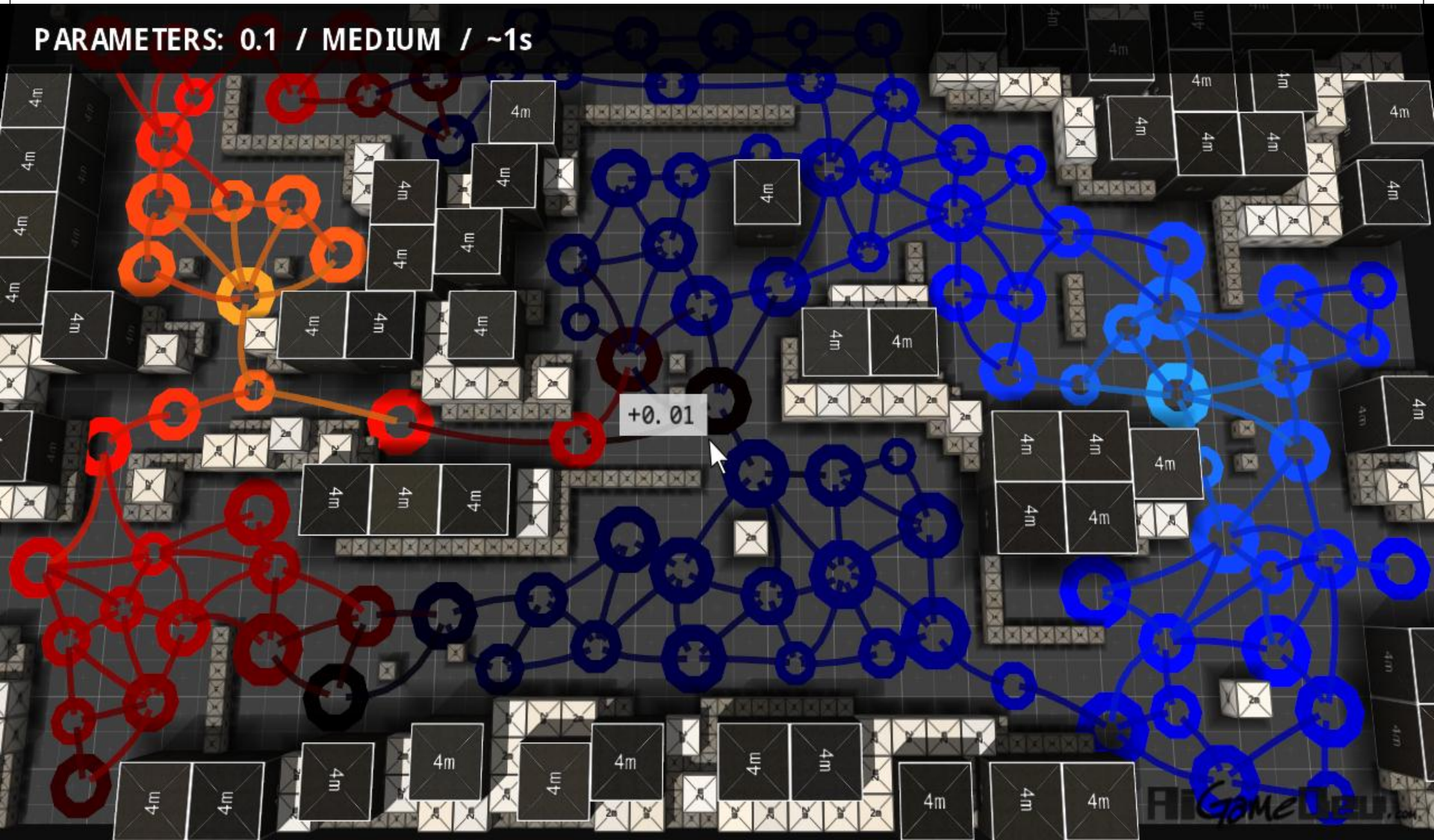
- How quickly should the influence decay with distance? In the code snippet above, this is controlled by a multiplier to the distance before it's passed to the exponential function, so you can control how quickly the influence fades.

- Typically, you'll use different decay values based on the size of your influence map. Lower decay for larger strategic maps, and higher decay when the map is localized and used for tactical purposes.

Algorithm Parameter: Update Frequency

- The parameter you'll have the least control over, is the update frequency.
- This will depend on how many resources you have at your disposal for updating the AI.
- Luckily, influence maps can scale down relatively well, but there's a base amount of computation that needs to be done to get good quality information.

Final Result!



Conclusion

- This is pretty much everything you need to know about game world navigation.

- There are some more advance concepts left. Because of our time limit, we will skip those concepts.