UNIVERSITÀ DEGLI STUDI
DI MILANO

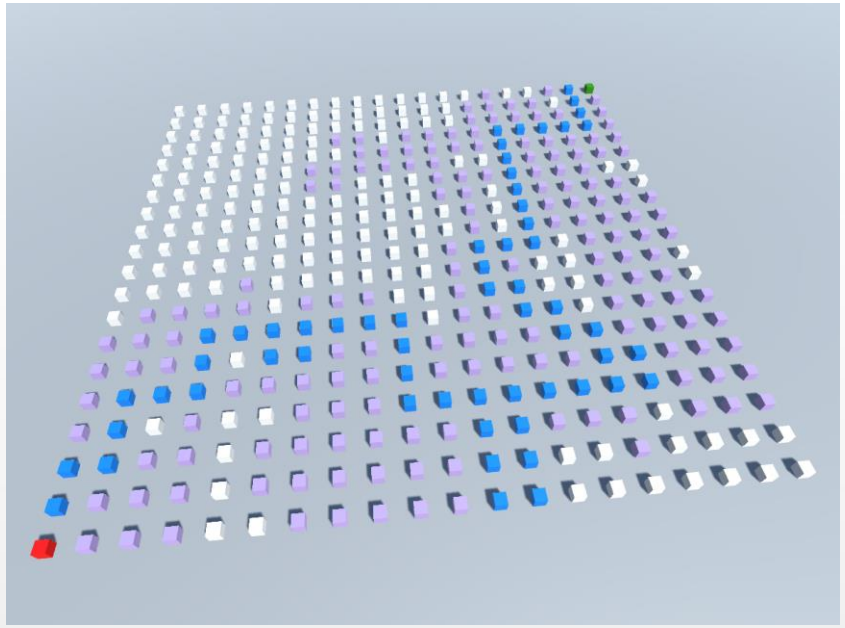
From Map to Graph

A.I. for Video Games

Dario Maggiorini (dario@di.unimi.it)

A Simple Problem with a Complex Solution

From this



… to this

Division Schemes

- A division scheme is a representation of a level where you split the area in linked regions

- Each division scheme is defined by:
  - Quantization and localization
  - Generation
  - Validity

Quantization and Localization

- We need to define a way to convert level locations into graph nodes and vice-versa

- Quantization
  - Linking a position to a graph node

- Localization
  - Linking a node to a position

Generation (i.e., How to Quantize)

- This is the policy (algorithm) we use to split a map and assign its pieces to graph nodes

- We have two options:
  1. Manually
     - The old school. We are not going to discuss this
  2. Algorithmically
     - Tile graphs
     - Dirichlet domains

> Synonymous with Dirichlet domain also Voronoĭ region, Wigner-Seitz cell, Brillouin zone (used in reciprocal space), domain of influence, or plesiohedron are used.

     - Points of visibility
     - Navigation meshes

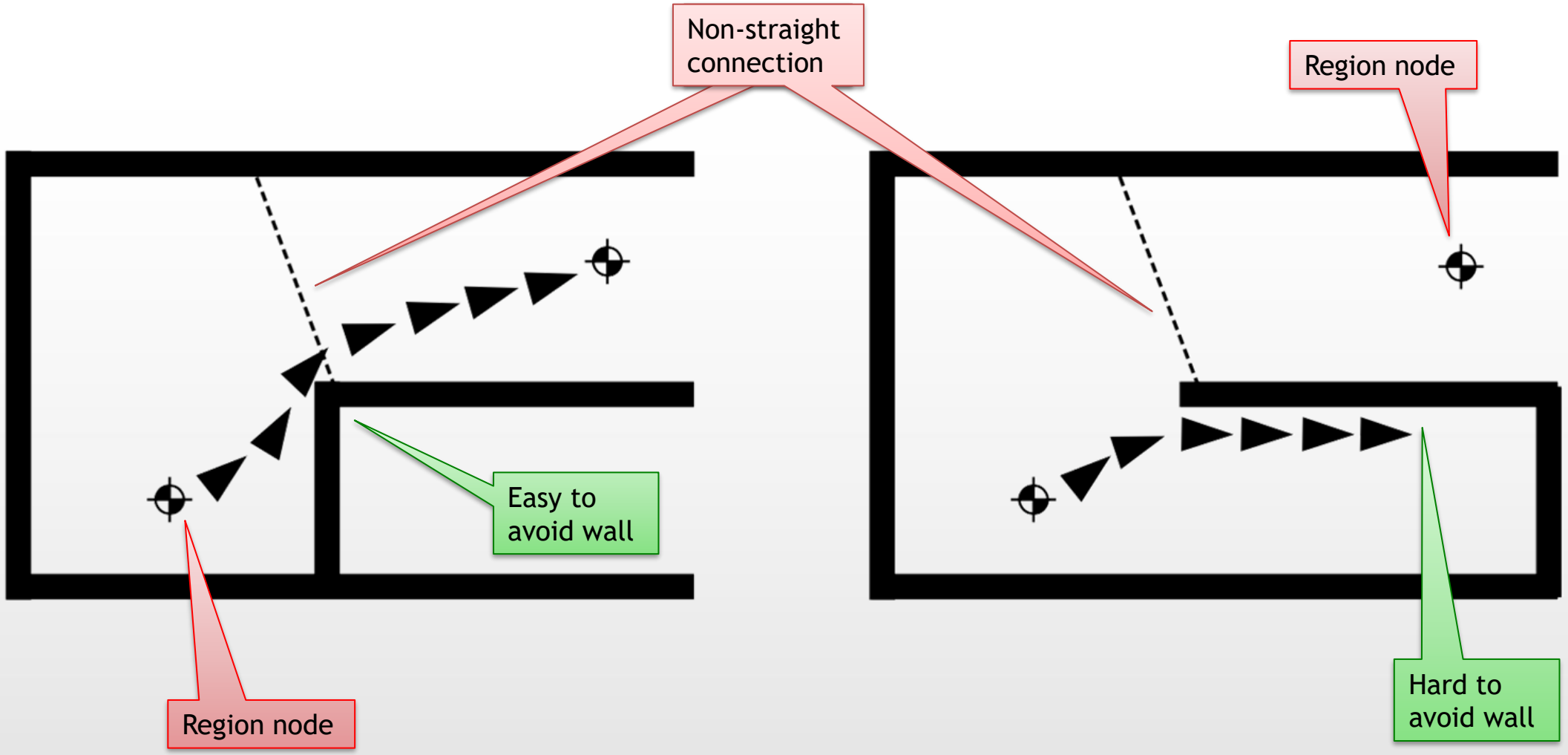
Validity

- Validity is about asking if quantization and generation are dependable for planning

- If a character needs to move from node X to node Y, will it (always) be able to carry out that movement?
  - If the quantized regions around X or Y are not allowing this, then the pathfinder is creating useless plans

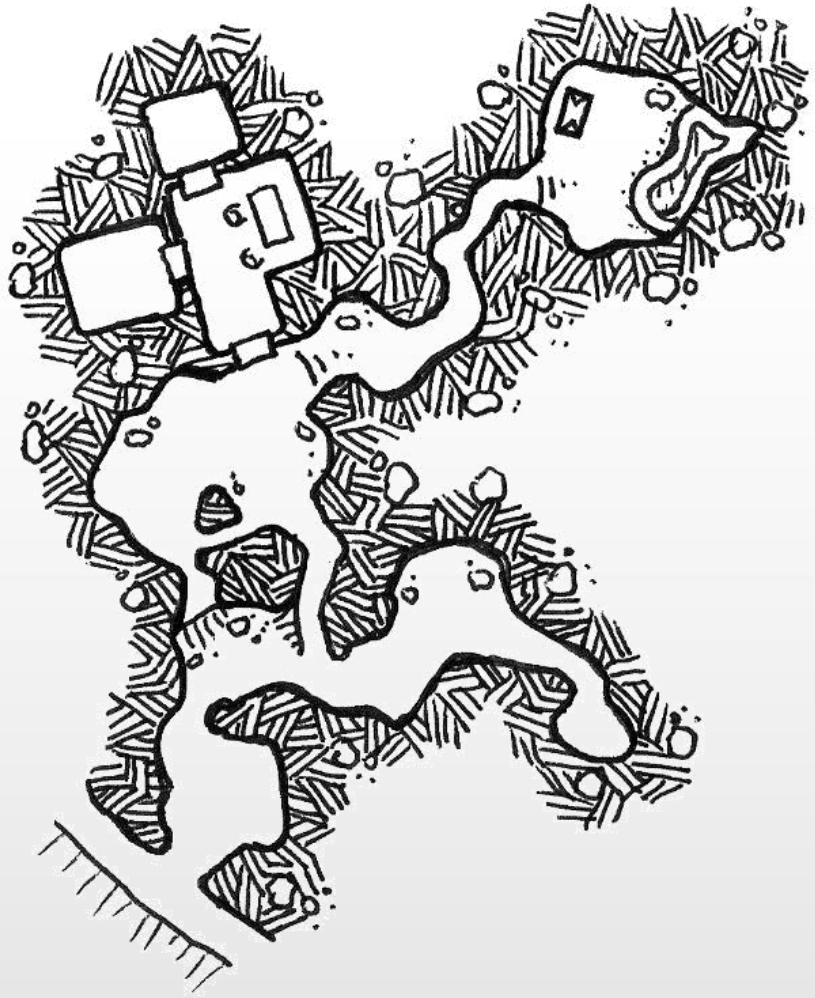
- Taking into consideration any couple of connected regions, a division scheme is valid if all point of one region can be reached from any point of the other region

Examples of Invalid Quantization



Tile Graphs

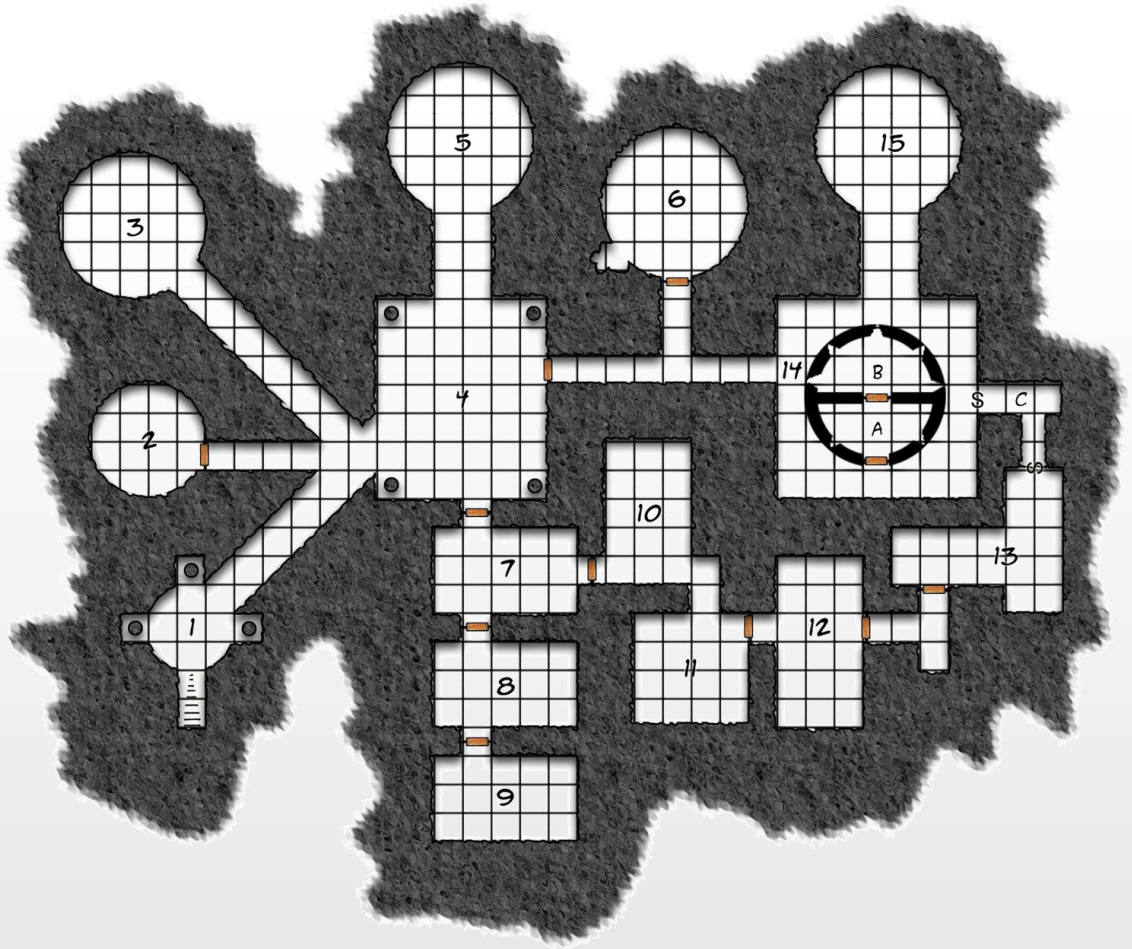
From this



Tile Graphs

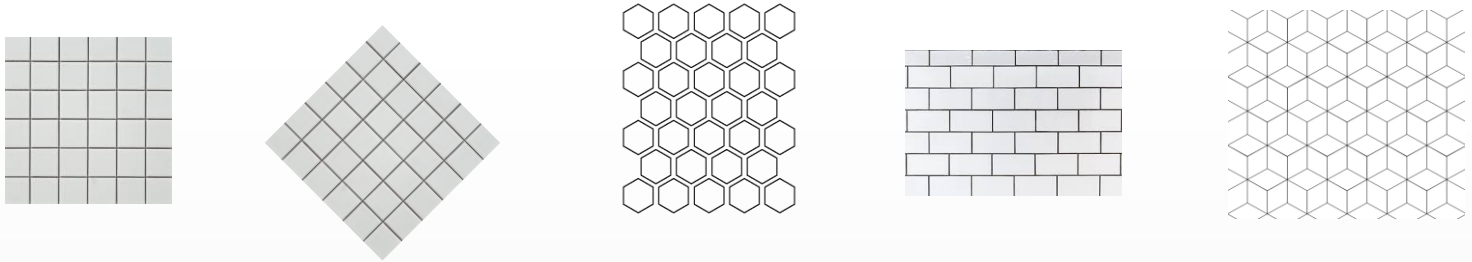
… to this

… and That is Why in Boardgames …

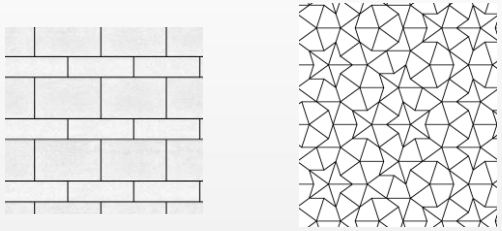


Tiles

- Tiles for tile graphs can be ANY REGULAR SIZE



- And, technically, they can be also mixed



- In the following, we are going to use squares, just because they are both convenient to manage and quite common in games
  - In your games, never underestimate the choice of your tiles

(Square) Tile Graphs

- Quantization
  - Extract a tile (node) number/coordinate from the position

```
float xPosition, yPosition;
float tileSize;

int xTile = (int)Math.Floor (xPosition / tileSize);
int yTile = (int)Math.Floor (yPosition / tileSize);
```

A tile (a node) is associated to a “tileSize x tileSize” area. The tiles are set as in a matrix covering all the space and (xTile, yTile) is the coordinate of the target tile

- Localization
  - The reverse calculation returning a location in the tile (the center?)

```
int xTile, yTile;
float tileSize;

float xPosition = (xTile + .5f) * tileSize;
float yPosition = (yTile + .5f) * tileSize;
```

Reverse calculation but the geometric position is associated to the center of the tile (hence, the + 0.5 to the position)

PRO TIP: In Unity, using a Vector2 can make your life a bit easier for lerping and general math

But Maybe …

It’s more convenient to build it around Node in the Graph class

A Dictionary associates each tiles to its coordinates

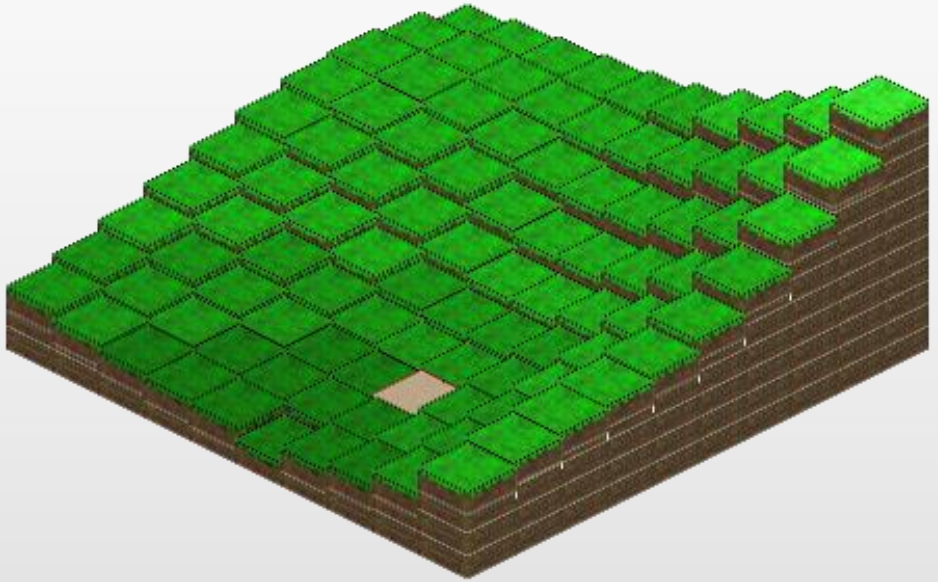
See accessors in C# for details about how these works

```
protected float tileSize;
protected Node[,] matrix;
protected Dictionary<Node, float[]> map;

// quantization
public Node this [float x, float y] {
    get {
        return matrix[
                    (int) Math.Floor (x / tileSize),
                    (int) Math.Floor (y / tileSize)
                    ];
    }
}

// localization
public float[] this [Node n] {
    get {
        if (!map.ContainsKey (n)) return null;
        return map [n];
    }
}
```

Tile Graphs Generation

- Tiles can be generated at runtime, when needed
  - Connections between tiles also, if you do it completely random

- If we you set a *y* (height in Unity) dimension, the steepness should contribute to the distance (weight) calculation



Generation of a Square Tile Graph

Source: TileGraph
Folder: Pathfinding/Graph

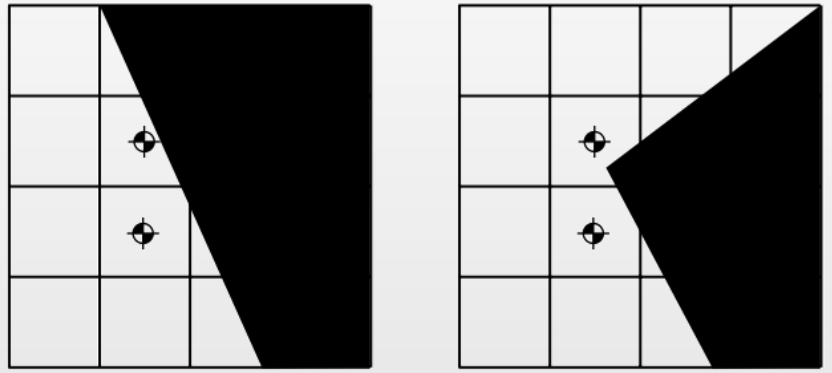
```
public class TileGraph {

    protected float tileSize;
    protected Node[,] matrix;
    protected Dictionary<Node, float[]> map;

    public TileGraph(float x, float y, float ts) {

        tileSize = ts;

        matrix = new Node[
          (int) Math.Floor (x / tileSize) + 1,
          (int) Math.Floor (y / tileSize) + 1
          ];

        map = new Dictionary<Node, float[]> ();
        for (int i = 0; i < matrix.GetLength(0); i += 1) {
            for (int j = 0; j < matrix.GetLength(1); j += 1) {
                Node n = new Node ("" + i + "," + j);
                matrix [i, j] = n;
                map [n] = new float[] { (x + .5f) * tileSize, (y + .5f) * tileSize };
            }
        }
    }
```

First, we create a tile matrix, then we populate the dictionary.

Graph interrogation is performed through the accessors presented two slides ago

Tile Graphs Validity

- Depends on your level design
- If a tile is either completely empty or full, then the graph is guaranteed to be valid
- If a tile may be partially blocked, it will depend on the shape of the blockage and your movement system



Valid partial blockage

Invalid partial blockage

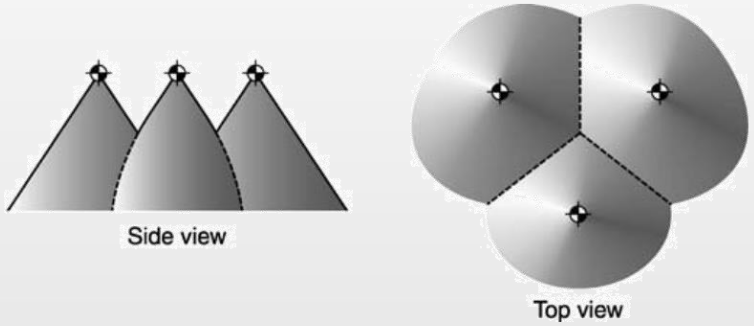
(Optional) Exercise

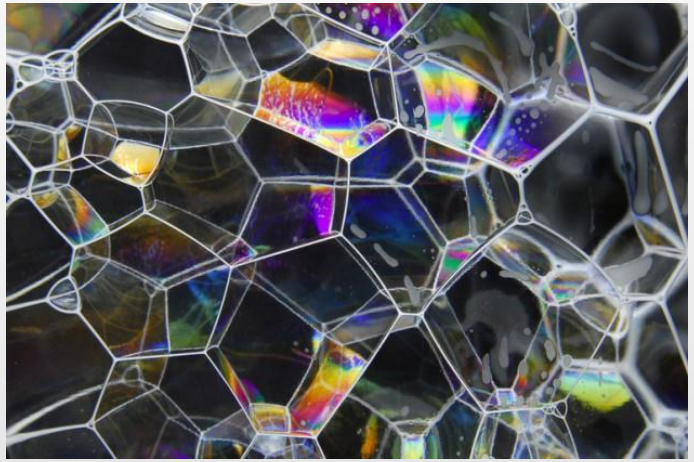
- Change the code to make it use hexagons
  - Think about the *Catan* board game

Dirichlet Domains

Once we can identify a set of characteristic points on the map, we surround them with an area. This area will be made of all locations that are closer to that point than to any other

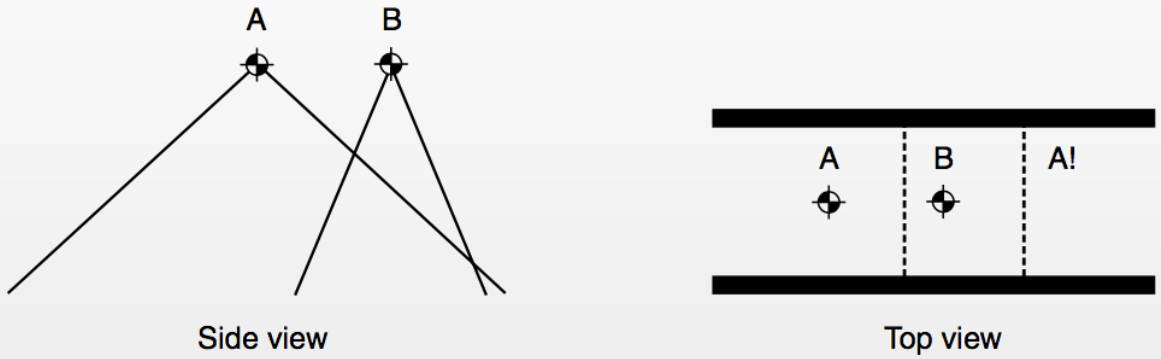
A Dirichlet domain, also referred to as a Voronoi polygon in two dimensions, is a region around one of a finite set of source points whose interior consists of everywhere that is closer to that source point than any other.





Connection Between Dirichlet Domains

- Can be solved algorithmically using *Delaunay triangulation* to find bordering domains
  - Quite complicated
  - Each domain can be weighted to tune cone's falloff
    - Cone falloff is causing problem with overlapping cones (see below)



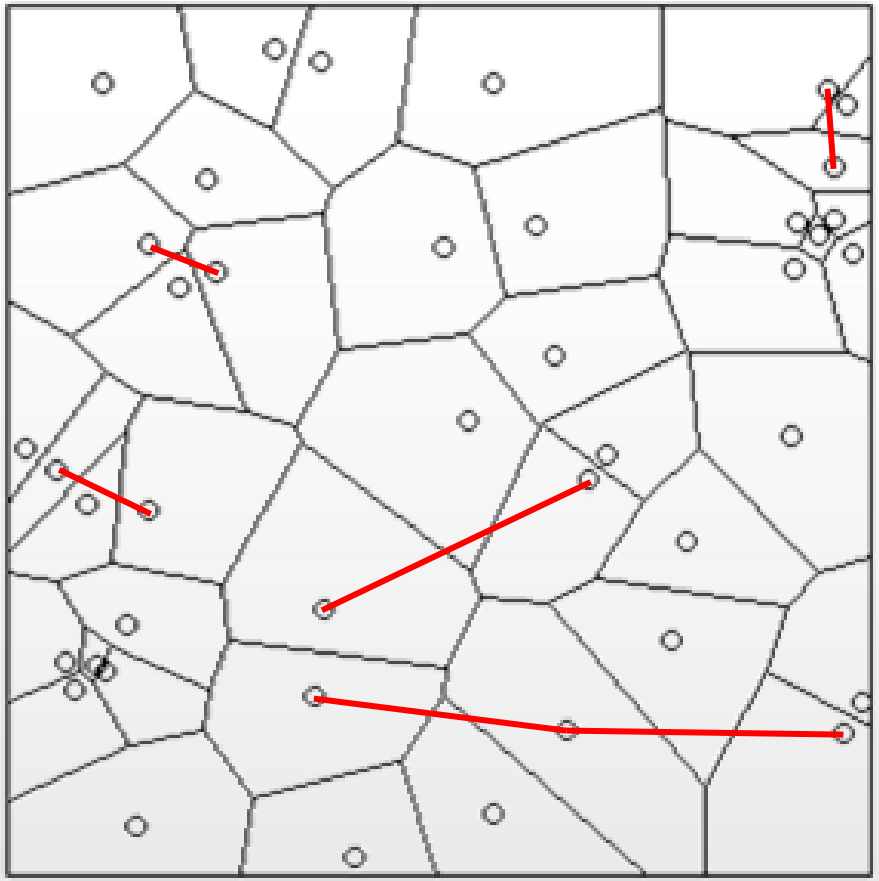
  - This approach cannot take obstacles into account
  - Usually, we prefer to delegate this to the lever designer

Dirichlet Domains

- Quantization
  - We need to find the closest characteristic point on the map
  - Complexity will be O(n)
  - We usually want to partition the map so that we only perform calculation for nearby points
- Localization
  - Given a domain, we return its characteristic point
- No panic, we are not going to calculate any Dirichlet or Voronoy diagrams today

Dirichlet Domains Validity

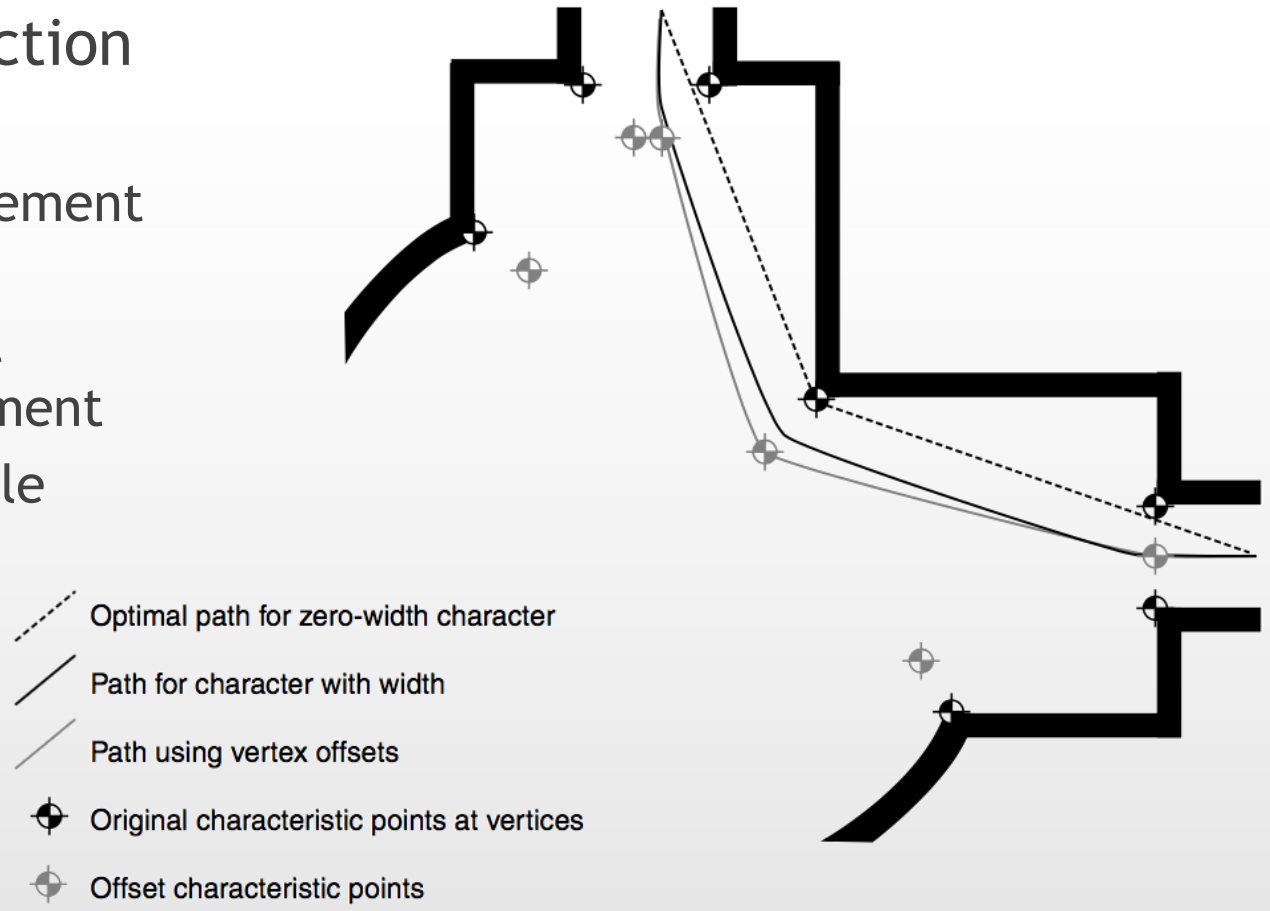
- There is no formal way to demonstrate validity
  - Domains will form an intricate tessellation
  - There is no formal way to demonstrate that the line connecting two characteristic points belonging to two connected domains is not passing through a third domain
- Delegating points selection to the level designer might solve the issue
  - Obstacles can have their own domain
  - The steering behaviour of your NPC can help you



Red lines outline invalid connected domains

Points of Visibility

- Any optimal path (in a 2D environment) will have inflection points
  - In these inflection points, movement direction will change
  - Inflection points are located at convex vertices in the environment
  - Avatar shape becomes a variable here



Point of visibility

- Since inflection points are part of the shortest path, we can use them to build our navigation graph

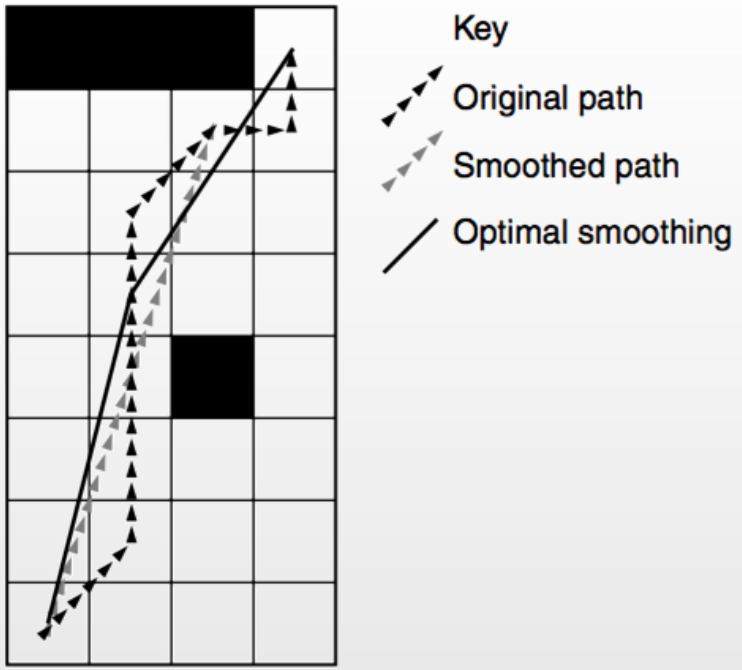
- We can use all convex point to build a level graph
  - They are way too many! We usually need to simplify the structure
    - Take them from colliders only
      - But then you must test your level for wall-walking
    - Pick them by hand
      - Once again, the level designer must do it for you

- To understand connections, just use a ray cast between points
  - If the ray can hit the target point without bouncing in another object (the two points are in line of sight), you have a connection
  - Hence, the name

- Points of visibility can be used as characteristic points to build Dirichlet Domains

Walking the Path (a.k.a. Smoothing)

- Walking straight from a tile center to another tile center may result in a very unrealistic movement
- Identifying intermediate points and reducing changes for movement direction will help increase the realism
- A steering behaviour may also be your friend here



A Simple Smoothing Algorithm

1. Find a path using Dijkstra or A*
2. Start from the first node
3. Select the next node and perform a raycast from current position to its position
4. If the destination is NOT visible
   - Move to the last visible position
5. If the destination is visible
   - If it is the last position, move to destination and terminates
6. Go to step 3

- This can be done inline or calculated before movement to be better smoothed through a steering behaviour

- **NOTE:** this algorithms takes division scheme validity for granted

References

- On the textbook
  - § 4.4
  - § 4.4.1
  - § 4.4.2
  - § 4.4.3
  - § 4.4.7