CA-MIRI
Computer Animation
2nd Project

Exercise 3 – Pathfinding

Alejandro Beacco Porres

alejandro.beacco@upc.edu

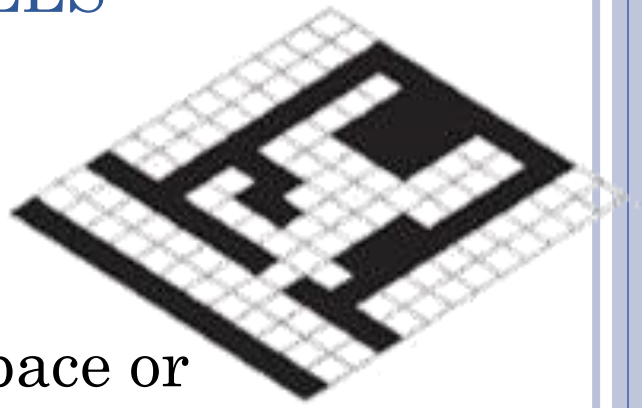
Exercise Statement

- For this exercise, a base code with abstract and template classes for pathfinding will be provided. You are free to use it to have a more general implementation of pathfinding algorithms.
- 1-Generate a custom random regular grid scene with obstacles (you can instantiate any *GameObject* to represent the obstacles).
- 2-Implement A* to search paths over that grid (you will need a graph representation of the grid conforming your navigation mesh).
- 3-Implement at least another pathfinding method improving A* from those explained in class (bidirectional search, jump ahead, ARA* …).
- 4-Integrate path finding with your crowd: assign way points over the navigation mesh.
- 5-Modify your heuristic to take into account the density of agents inside a cell.
- 6-Extra: try to replace your grid with a triangulated navigation mesh (*NavMeshTriangulation*).

Outline

- 1-Random Grid with Obstacles
- 2-A*
- 3-A* Variation
- 4-Pathfinding Integration
- 5-Agents Density
- 6-Triangulated Navigation Mesh

1-Random Grid with Obstacles



- Navigation Mesh → Regular Grid
- 2D grid, all cells have the same size
  - minX, maxX, minY, maxY, cellSize
- Value per cell representing empty space or obstacle → bool

- Loop through all cells
- Random obstacles or not
  - Random.Range(0.0f,1.0f) <= obstacleProbability
    - 0 <= obstacleProbability <= 1

- If obstacle → Instantiate(ObstaclePrefab)
  - ObstaclePrefab → size of cell

1-Random Grid with Obstacles

- Navmesh → Graph Representation of the Grid
- Template code → Add it to your project Assets folder

- **using namespace PathFinding;**

- GridCell → Derive from Node: **public class GridCell : Node { … }**

```
public abstract class Node{
// Abstract class that represents a node with an id

        public Node(int i){ id = i; }
        public Node(Node n){ id = n.id; }

        public int getId(){ return id; }

        public int id;
};
```

1-Random Grid with Obstacles

EXAMPLE

```
public class GridCell : Node {
// Your class that represents a grid cell node derives from Node

    // You add any data needed to represent a grid cell node
   protected float xMin;
   protected float xMax;
   protected float zMin;
   protected float zMax;

   protected bool occupied;

   protected Vector3 center;

   … // You also add any constructors and methods to implement
   your grid cell node class

   }
```

1-Random Grid with Obstacles

- CellConnection → Connection : **public class CellConnection : Connection<GridCell> {** … **}**

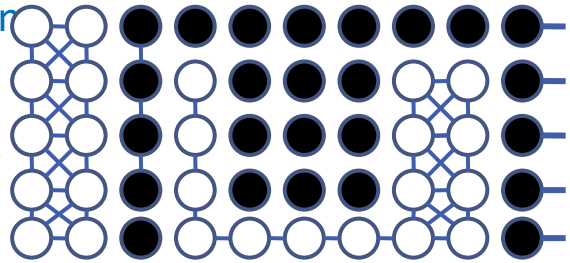
```
public abstract class Connection <TNode>
where TNode : Node
{
// Abstract class that represent the connection between 2 nodes

        public TNode fromNode; // reference to the origin node
        public TNode toNode; // reference to the destination node
        public int fromNodeId; // id of the origin node
        public int toNodeId; // id of the destination node

        public float cost; // the cost of using that connection in a path

        public Connection(TNode from, TNode to){
            fromNode = from;
            toNode = to;
            fromNodeId = fromNode.id;
            toNodeId = toNode.id;
        }
```

1-Random Grid with Obstacles

```
        …

                public TNode getFromNode(){
                        return fromNode;
                }

                public TNode getToNode() {
                        return toNode;
                }

                public float getCost() {
                        return cost;
                }

                public void setCost(float c){
                        cost = c;
                }
        };

}
```

1-Random Grid with Obstacles

```
public class NodeConnections<TNode,TConnection
        where TNode : Node
        where TConnection : Connection<TNode>
        {
           // Class to handle a list of connections

                public List<TConnection> connections;
                …
        }

public class GridConnections : NodeConnections
        <GridCell, CellConnection> { … }
```



1-Random Grid with Obstacles

```
public abstract class Graph<TNode, TConnection, TNodeConnections>
          where TNode : Node
          where TConnection : Connection<TNode>
          where TNodeConnections : NodeConnections<TNode,TConnection>
          {
// Abstract class that represents a graph (infinite or not), that is at least
// a function that returns a list of connections for any node

                    public   Graph(){ }
                    public abstract TNodeConnections getConnections (TNode fromNode);

          };
```

1-Random Grid with Obstacles

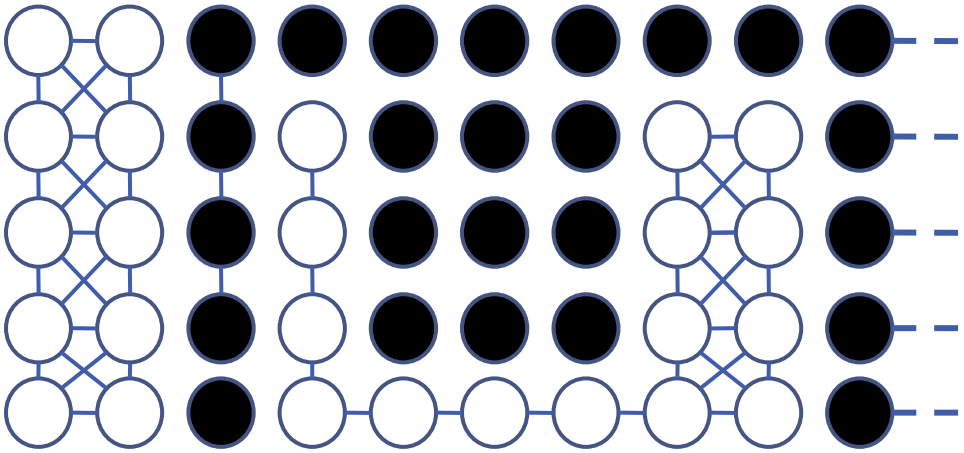
```
public abstract class FiniteGraph<TNode, TConnection, TNodeConnections> :
                            Graph<TNode, TConnection, TNodeConnections>
        where TNode : Node
        where TConnection : Connection<TNode>
        where TNodeConnections : NodeConnections<TNode,TConnection>
        {
// Abstract class that represents a finite graph, where there is a known set
// of nodes and a known set of connections between those nodes.

                public List<TNode> nodes;
                public List<TNodeConnections> connections;

                public   FiniteGraph():base(){
                         nodes = new List<TNode> ();
                         connections = new List<TNodeConnections> ();
                }

                …
```

1-Random Grid with Obstacles

```
        …

        public TNodeConnections getConnections(int fromNodeIndex) {
                return connections[fromNodeIndex];
        }

        public override TNodeConnections getConnections(TNode fromNode)
        {
                return getConnections (fromNode.getId ());
        }

        public int getNumNodes() { return nodes.Count; }
        public TNode getNode(int i)  {
                if (i < getNumNodes())
                        return nodes[i];
                else
                        return null;
        }
};
```

1-Random Grid with Obstacles

```
public abstract class FiniteGraph<TNode, TConnection, TNodeConnections> :
                      Graph<TNode, TConnection, TNodeConnections>
        where TNode : Node
        where TConnection : Connection<TNode>
        where TNodeConnections : NodeConnections<TNode,TConnection>

public class Grid : FiniteGraph<GridCell, CellConnection, GridConnections>
```



2-A*

```
public abstract class Heuristic <TNode> where TNode : Node{
  // Abstract class that represents a Heuristic function to estimate the cost of going from
  // one node to another

          protected TNode goalNode; //the goal node that this heuristic is estimating for

          // constructor takes a goal node for estimating
          public Heuristic(TNode goal){
                    goalNode = goal;
          }

          // generates an estimated cost to reach the stored goal from the given node
          public abstract float estimateCost(TNode fromNode);

          // determines if the goal node has been reached by node
          public abstract bool goalReached (TNode node);
};
```

public class GridHeuristic : Heuristic <GridCell> → Euclidian distance?

2-A*

EXAMPLE

```
public class GridHeuristic : Heuristic <GridCell>

        // constructor takes a goal node for estimating
        public GridHeuristic(GridCell goal):base(goal){
                goalNode = goal;
        }

         // generates an estimated cost to reach the stored goal from the given node
        public override float estimateCost(GridCell fromNode){
                return …
        }

        // determines if the goal node has been reached by node
        public override bool goalReached(GridCell node){
                return …
        }

};
```

2-A*

```
public abstract class PathFinder <TNode, TConnection, TNodeConnections, TGraph, THeuristic>
          where TNode : Node
          where TConnection : Connection<TNode>
          where TNodeConnections : NodeConnections<TNode,TConnection>
          where TGraph : Graph<TNode,TConnection,TNodeConnections>
          where THeuristic : Heuristic<TNode>
{
// Abstract class that represents a path finding algorithm.
// It must have a function to find a path between two nodes using a heuristic function

          public    PathFinder() { }

          // returns, if found, a path from start to end in graph using heuristic
          // found < 0 --> path not found
          // found == 0 --> path not found or incomplete
          // found > 0 --> path found
          public abstract List<TNode> findpath(TGraph graph, TNode start, TNode end,
                    THeuristic heuristic, ref int found);
};
```

2-A*

```
public class
A_Star<TNode,TConnection,TNodeConnection,TGraph,THeuristic> :
        PathFinder<TNode,TConnection,TNodeConnection,TGraph,THeuristic>
        where TNode : Node
        where TConnection : Connection<TNode>
        where TNodeConnection : NodeConnections<TNode,TConnection>
        where TGraph : Graph<TNode,TConnection,TNodeConnection>
        where THeuristic : Heuristic<TNode>
{
  // Class that implements the A* pathfinding algorithm
  // You have to implement the findpath function.
  // You can add whatever you need.

        …
```

2-A*

```
…

protected List<TNode> visitedNodes;
// list of visited nodes
protected NodeRecord currentBest; // current best node found
protected enum NodeRecordCategory{ OPEN, CLOSED, UNVISITED };

protected class NodeRecord{
// You can use (or not) this structure to keep track of the information that we need
// for each node
            public NodeRecord() {}
                        public TNode node;
            public NodeRecord connection; // connection traversed to reach this node
            public float costSoFar; // cost accumulated to reach this node
            // estimated total cost to reach the goal from this node
            public float estimatedTotalCost;
            // category of the node: open, closed or unvisited
            public NodeRecordCategory category;
            public int depth; // depth in the search graph
};

…
```

2-A*

```
…

public A_Star(int maxNodes, float maxTime, int maxDepth) : base()
{
        visitedNodes = new List<TNode> ();
}

public virtual List<TNode> getVisitedNodes()
{
        return visitedNodes;
}

public override List<TNode> findpath(TGraph graph, TNode start,
        TNode end, THeuristic heuristic, ref int found)
{
        List<TNode> path = new List<TNode>();

        // TO IMPLEMENT

        return path;
}
};
```

2-A*

```
public class Grid_A_Star :
        A_Star <GridCell, CellConnection,
                        GridConnections, Grid, GridHeuristic> { };
```

- In some component script:
  - `Grid grid = new Grid(); // Create and fill grid with obstacles`
    …
  - `Grid_A_Star gridAStar = new Grid_A_Star(maxNodes, maxTime, maxDepth);`
  - `int found = -1; // init as not found`
  - `GridCell Start;`
  - `GridCell Goal;`
  - `GridHeuristic heuristic = new GridHeuristic(Goal);`
  - `List<GridCell> path = gridAStar.findPath(grid, Start, Goal, heuristic, ref found);`

2-A*

- There are 2 sets: OPEN and CLOSED
  - OPEN contains nodes that are candidates for expanding
  - CLOSED contains nodes that have been expanded
  - Initially OPEN={S} and CLOSED={}
  - Each node keeps a pointer to its parent node
  - Iteratively:
    - Pulls out the best node n in OPEN
      - The one with lowest $f(n) = g(n) + h(n)$
    - And expands it

2-A*

OPEN = priority queue containing START
CLOSED = empty set

while lowest rank in OPEN is not the GOAL :

current = remove lowest rank item from OPEN
add current to CLOSED

foreach neighbor in neighbors of current:

cost = g(current) + movementcost(current, neighbor)

if neighbor in OPEN and cost < g(neighbor):
remove neighbor from OPEN //because new path is better

if neighbor in CLOSED and cost < g(neighbor):
remove neighbor from CLOSED

if neighbor not in OPEN and neighbor not in CLOSED:
g(neighbor) = cost
add neighbor to OPEN
set priority queue rank to g(neighbor) + h(neighbor)
set neighbor's parent to current

reconstruct reverse path from goal to start by following parent pointers

http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

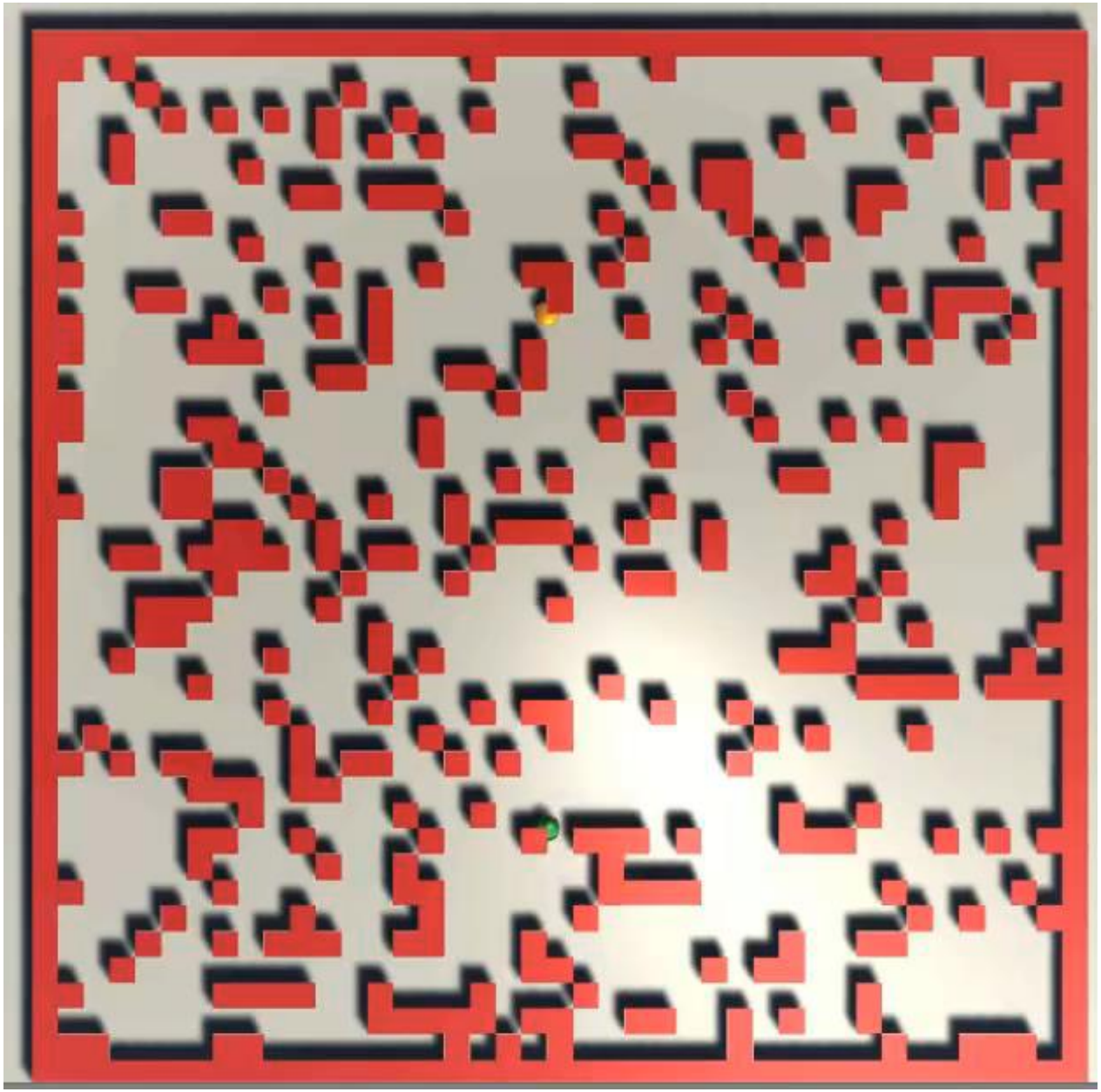
2-A*

- using System.Linq;

- SortedList
- Queue

- Remember debug visuals
  - void OnDrawGizmos( )
    {
    // Draw any debug visuals here
    // Debug OPEN and CLOSED nodes
    }
  - Gizmos.Color = Color.red;
  - Gizmos.DrawSphere(position, radius);

3.4-A*



3-A* Variation

- Bidirectional Search
- Jump Point Search
- ARA*

3-A* Variation Bidirectional Search

- Two searches in parallel
  1. from S to G
  2. from G to S
- When they meet, they should have a good path
- Advantage: better 2 small trees than one big
- Front-to-Front:
  - Instead of choosing the best forward search ($g(S,x)+h(x,G)$) or best backward seach ($g(y,G)+h(S,y)$) it choses a pair of nodes with best:

$$g(S,x)+h(x,y)+g(y,G)$$

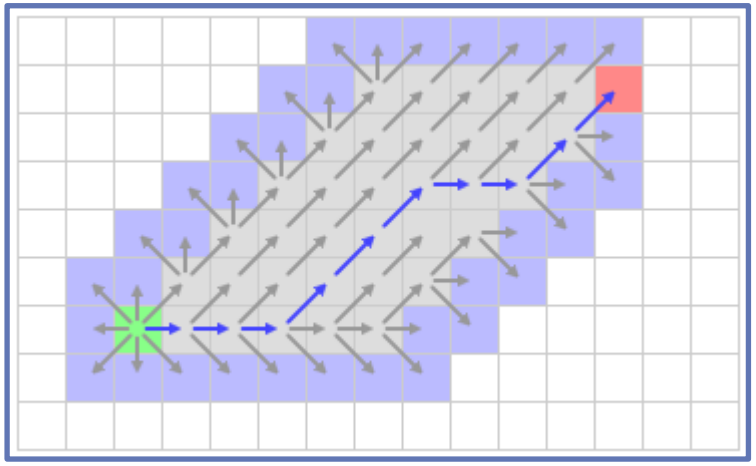
3-A* Variation
Bidirectional Search

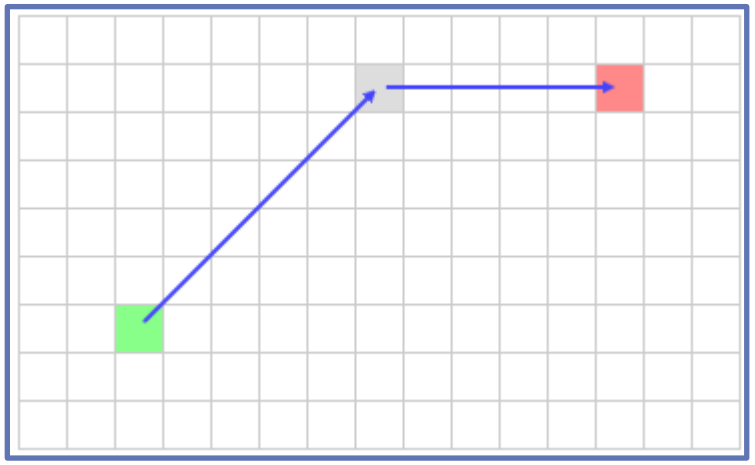
- Don’t spend time on parallelization code if you find problems (threads, jobs, …)
- Use any implementation you can think of:
  - Two search objects, components
  - Two coroutines
  - Etc…
- 2 searches and link paths when they meet

3-A* Variation
Jump Point Search

http://zerowidth.com/2013/05/05/jump-point-search-explained.html

- Goal: reduce the number of nodes
- In square grids with uniform cost, it's a waste to look at all the individual grid cells
- JPS variant of A* that skips ahead to faraway nodes that are visible from current node
- Each step is more expensive, but fewer steps.

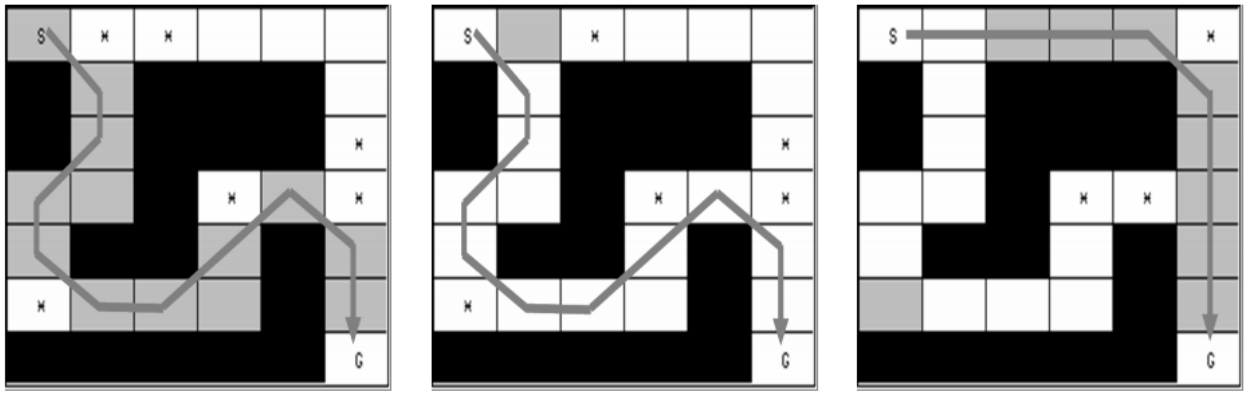




3-A* Variation ARA*

M. Likhachev, G. Gordon, and S. Thrun. ARA*: Anytime A* search with provable bounds on sub-optimality. Proceedings of Conference on Neural Information Processing Systems (NIPS). 2003.

- http://robots.stanford.edu/papers/Likhachev03b.html
- Find a feasible solution quickly
- Continually work on improving it until time runs out (bounded by time → Anytime A* → add maximum computation time parameter)
- Estimate of the total distance from start to goal going through s: $f(s) = g(s) + \varepsilon * h(s)$ $\varepsilon \geq 1$
- Works by executing A* multiple times, starting with a large ε and decreasing ε prior to each execution until $\varepsilon = 1$
- ARA* reuses the results of the previous searches to save computation



$\epsilon = 2.5$ $\epsilon = 1.5$ $\epsilon = 1.0$

4-Pathfinding Integration

- PathManager
  - Compute path to goal from current cell (where the agent is
    - List of cells → waypoints to center of cells
  - Manage waypoints
    - Current waypoint
    - When current waypoint is reach → change to new waypoint
- Simulator
  - Move agents towards current waypoint

5-Agents Density

- Depending on your cell size you can have more than one agent inside it
- Keep track of how many agents are in every cell
  - How? Proper data structure
  - Efficient update
- Use the number of agents in your pathfinding computations to penalize cell with high density of agents in it
  - Heuristic
  - Cost computation
  - The more people in a cell, the higher the cost
- Agents should prefer paths without other agents

6-Triangulated Navigation Mesh

- Extra exercise

- Build Unity's Navmesh
  - Set Geometry to static
  - public static AI.NavMeshTriangulation **CalculateTriangulation**();

CA-MIRI
Computer Animation
2nd Project

Exercise 3 – Pathfinding

Alejandro Beacco Porres

alejandro.beacco@upc.edu