**Map.cs**

Overview

* The Map class sets up a 2D grid with information of the scene created in Unity. The 2D grid is composed of Cells, which are grouped together to create Zones. The 2D grid is additionally organized into a graph so that Agents can calculate paths through it.

Variables:

* mapWidth: width of the scene created in Unity
* mapHeight: height of the scene created in Unity
* gridWidth: number of Cells in the 2D grid, across
* gridHeight: number of Cells in the 2D grid, up and down
* topLeftPos: position of the top left of the scene in Unity’s x-, y-, z- coordinates
* cellSize: size of the Cells in the 2D grid
* grid: the 2D grid of Cells that cover the entire scene made in Unity
* thresholdGraph:

References:

* ZoneManager (zm): reference to the ZoneManager class
* topLeft: reference to the object in the very top left corner of the scene in Unity
* topRight: reference to the object in the very top right corner of the scene in Unity
* bottomRight: reference to the object in the very bottom right corner of the scene in Unity

Functions:

* Awake
  + Unity’s built-in function that is called in order to initialize the 2D grid the moment the scene is played. Will simply call InitMap
* InitMap
  + Initializes the variables in the Map class as well as the 2D grid. Will simply call the following methods, in order: GetMapDimensions, BuildCellGrid, DefineZones, FindThresholds, AddThresholdsToZones
* GetMapDimensions
  + Determines the entire size of the scene in Unity using its corners
  + Begins by using the Renderers of the corners, topLeft, topRight, and bottomRight, in order to find the bounds of the scene in Unity’s x-, y-, z- coordinates. The following is an example done to find the left boundary of the scene:
    - float approxLeft = (topLeft.bounds.center.x - (topLeft.bounds.size.x / 2));
  + Then uses the bounds of the scene to find the width and the height of the scene. This information is stored in Map’s global variables
* BuildCellGrid
  + Input: cellSize
  + Builds the 2D grid of Cells the cover the entire scene in Unity
  + Begins by calculating how many Cells can fit within the scene
  + Then, loops through the size of the scene and create Cells with information corresponding to each point on the scene, using the following steps:
    - 1. Initialize the grid with gridWidth and gridHeight
    - 2. For (z = 0, z < gridHeight, z++)
      * 3. For (x = 0 , x < gridWidth, x++)
        + 4. Calculate where this next Cell should be with respect to the size of the grid

calc = (cellSize \* x) + (cellSize / 2f), 0, -1 \* ((cellSize \* z) + (cellSize / 2f)))

* + - * + 5. Add this calculation value to topLeftPos so that it’s coordinates will be in the scene in Unity

worldPos = calc + topLeftPos

* + - * + 6. Add this cell to the grid with this information

grid[x,y] = new cell(worldPos)

* + Next, assign neighbors to each Cell in the grid wherever possible.
    - 1. For (z = 0, z < gridHeight, z++)
      * 2. For (x = 0 , x < gridWidth, x++)
        + 3. c = cell at grid[z,x]
        + 4. possibleEdges = new list of Edges
        + 5. Look at the Cells to the left, right, up, down, and diagonal of c
        + for (h = -1, h <= 1, h++)

For (v = -1, v <= 1, v++)

if v = h, then we are actually looking at c

Continue

6. Look at this potential neighbor

checkX = c.gridPositionX + h;

checkZ = c.gridPositionZ + v;

7. Check if this potential neighbor is within the scene and is walkable

If so, this is a neighbor of c. Create an edge to this neighbor and add it to possibleEdges

* + - * + 8. Assign possibleEdges to c
* DefineZones
  + With the list of Zones in the ZoneManager, find all the Cells that are positioned inside each zone and assign them to it.
  + 1. zones = list of Zones in the ZoneManager
  + 2. Foreach z in zones
    - 3. Find the Cells that are at the corners of this zone. The following is an example for the top left Cell:
      * *Cell* topLeftCell = CellFromWorldPos(z.topLeft);
    - 4. Foreach Cell c between these four corners
      * 5. Mark them as belonging to this zone z
      * id = z.zoneID
      * c.zoneID = id
* FindThresholds
  + Within each Zone, find the Thresholds that connect two different zones together by visiting each Cell and checking if they are straddling two zones
  + 1. Create visited = bool[gridHeight, gridWidth] to keep track of all the Cells we have visited
  + 2. Find a random walkable Cell to start at
    - leftBound = 0
    - rightBound = gridWidth
    - bottomBound = gridHeight
    - topBound = 0
    - validStart = false
    - Cell startCell
    - Do
      * Randx = random number between left and right bound
      * Randz = random number between top and bottom bound
      * startCell = grid[randz, randx]
      * If startCell is walkable
        + validStart = true
    - While (!validStart)
  + 3. Search through the grid to find thresholds
    - thresholdsList = FindThresholdSearch(visited, startCell, thresholdList)
  + 4. Build a Threshold graph with all the Thresholds we have found
    - BuildThresholdGraph(thresholdList)
* FindThresholdsSearch
  + Given a visited array and a starting cell traverse through the map in order to find all the valid Thresholds between the zones
  + Input: visited array for traversing, starting cell, list of Thresholds where the Thresholds that are found will be placed
  + Output: list of Thresholds that were found
  + Create a queue to keep track of visited cells
  + 1. Queue s = new Queue of Cells
  + 2. s.enqueue(start)
    - 3. cell = s.dequeue
    - 4. if this cell is visited
      * continue
    - 5. Mark cell as visited
    - Look at all cell’s neighbors to check if it has a threshold
    - 6. Foreach edge e in cell’s edgesToNeighbors
      * 7. c = e.incident (cell’s neighbor)
      * 8. If c.zoneID < cell.zoneID
        + 9. If neither c nor cell are already thresholds
        + 10. Mark c as having a threshold T
        + 11. thresholdList.Add(T)
    - Add the neighbors to the queue so we can continue to traverse the scene for more Thresholds
    - 12. Foreach edge e in cell’s edgesToNeighbors
      * 13. If this neighbor n has not been visited
        + 14. s.enqueue(n)
  + 15. Done! Return thresholdsList
* BuildThresholdGraph
  + Given a list of thresholds, connect all the Thresholds together in a graph and assign the Thresholds to their respective Zones
  + Input: List of Thresholds
  + Look at all the thresholds in the given list, thresholdsList
  + If threshold A and B have zones in common, then connect their Thresholds
  + 1. For (int i = 0, i < thresholdsList.Count, i++)
    - 2. threshold = thresholdsList[i]
    - 3. possibleNeighbors = new list of ThresholdEdges
    - 4. For (int j = 0, j < thresholdsList.Count, j++)
      * 5. If j == i
        + 6. Continue, since we don’t want self-loops
      * 7. Compare the zoneIDs of the Thresholds in order to see if we should connect them
        + int thresholdAID1 = thresholdsList[i].tzoneID; //threshold A's IDs
        + int thresholdAID2 = thresholdsList[i].zoneId;
        + int thresholdBID1 = thresholdsList[j].tzoneID; //threshold B's IDS
        + int thresholdBID2 = thresholdsList[j].zoneId;
      * If they share any common IDs, then B becomes a neighbor of A (A->B)
      * 8. if (thresholdAID1 == thresholdBID1 || thresholdAID1 == thresholdBID2 || thresholdAID2 == thresholdBID1 || thresholdAID2 == thresholdBID2)
        + However we do not want to consider connecting thresholds that are in the same zones and connect because it’s redundant
        + 9. if (thresholdAID1 == thresholdBID1 && thresholdAID2 == thresholdBID2)

10. continue

* + - * + 11. Add a weight to the edges. (This is optional)
        + 12. if possibleNeighbors does not contain this neighbor

possibleNeighbors.Add(neighbor)

* + - We’re done looking at this Threshold! Add all the found neighbors to this Threshold
    - 13. threshold.TAssignNeighbors(possibleNeighbors)
  + 14. Finished looking at all thresholds! Assign the graph we’ve created to thresholdGraph
* AddThresholdsToZone
  + With the thresholdGraph, assign the appropriate Thresholds to the Zones the belong to
  + 1. thresholdList = thresholdGraph
  + Iterate through the list to find the zones the Thresholds belong to
  + 2. Foreach Threshold t in thresholdList
    - 3. Zone z = ZoneManager.GetZone(t.zoneID)
    - 4. z.addThresholdToZone(t)
    - 5. z = ZoneManager.GetZone(t.tzoneID)
    - 6. z.AddThresholdToZone(t)
* CellFromWorldPos
  + Given some world position, give the Cell that is at that position
  + Input: Vector3 of the desired cell
  + Output: Cell at the given Vector3 position
  + Calculate how far worldPos is in Unity’s scene. This is done by converting the worldPos into a percentage that represents how far left and down the position is in the 2D grid.
  + 1. Convert worldPos’s x-coordinates into a percentage by calculating (worldPos’s distance from the leftmost side of the scene / total distance of the scene)
  + 2. Do the same as above, but for worldPos’s z-coordinate
  + 3. Clamp both percentages to make sure that they are not over 100%
  + 4. Find the indices of the Cell in the grid. This is done by multiplying the percentages by the height and width of the grid - 1
    - int x = (Mathf.RoundToInt(gridWidth \* percentX)) - 1;
    - int z = (Mathf.RoundToInt(gridHeight \* percentZ)) - 1;
  + 5. If x < 0
    - 6. x = 0
  + 7. If z < 0
    - 8. z = 0
  + 9. Return grid[z,x]
* CellFromThreshold
  + Give the Cell that is positioned at the same place as the current Threshold position
  + Input: Threshold
  + Output: Cell at that Threshold
  + Search for the Cell with the same z-coordinate as the Threshold threshold using binary search
  + 1. Xpos = threshold.worldPosition.x
  + 2. Zpos = threshold.worldPosition.z
  + 3. x, z = 0
  + 4. t = 0, b = grid.GetLength(0)-1
  + 5. while(t <= b)
    - 6. m = (int)Mathf.Floor((t + b) / 2);
    - 7. if (Mathf.Approximately(grid[m, 0].worldPosition.z, zpos))
      * 8. z = m; break;
    - 9. if (grid[m, 0].worldPosition.z > zpos)
      * 10. t = m + 1;
    - 11. Else
      * 12. b = m - 1;
  + Now search for the Cell with the same x-position, using binary search just like above
  + 13. l = 0, r = grid.GetLength(1) - 1
  + 14. while (l <= r)
    - 15. m = (int)Mathf.Floor((l + r) / 2);
    - 16. if (Mathf.Approximately(grid[z, m].worldPosition.x, xpos))
      * 17. x = m, break
    - 18. if (grid[z, m].worldPosition.x < xpos)
      * 19. 1 = m + 1
  + 20. Done! Return grid[z,x]
* GetZone
  + Give the zone that has the specified zoneID
  + Input: zoneID
  + Output: Zone
  + 1. Return zm.zones[zoneID]
* OnDrawGizmos
  + Only used for debugging
  + Draws the 2D grid in Unity’s Scene view tab
* CreateImage
  + Only used for debugging
  + Color zones, thresholds, and obstacles on an EMGU image and export the image to a folder outside the Unity project folder