Game 9

A* with Optimisations

A* with Optimisations ~ Introduction (1)

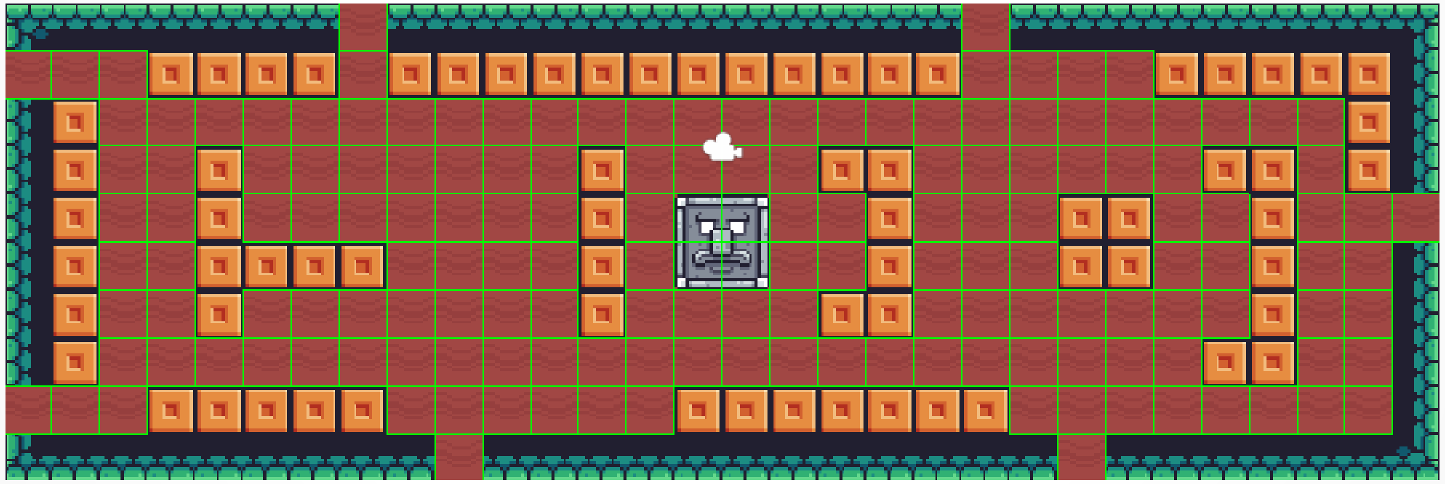
- A* is a searching algorithm used to find the most optimal path from one point to another.
- Considers the cost to reach the node (g(n)) and the cost to reach the goal node from the current node (h(n)). Since we cannot know the exact cost for h(n), we estimate it using some heuristic function.
- A* guarantees optimality given that the heuristic is admissible (never overestimates the cost to reach the goal state) and consistent (the cost of a node never exceeds its successors' cost).

A* with Optimisations ~ Introduction (2)

- We will be considering three heuristic types in this game (related to distance between two game objects):
    - Manhattan
    - Chebyshev
    - Euclidean
- The Jump Point Search algorithm is an improvement on A* search as it considers less nodes when finding a path whilst still preserving its optimality.

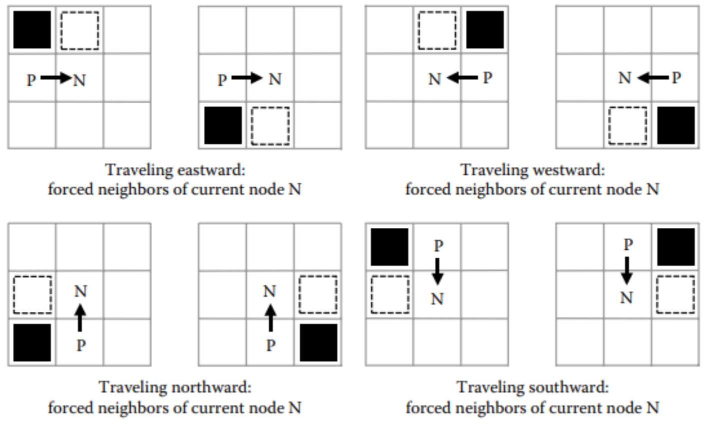
A* with Optimisations ~ AI Explanation (1)

- We first define the search space to consider, in the game's case, it will be a 2D grid over a map, we will also check for any wall obstacles within the grid.



A* with Optimisations ~ AI Explanation (2)

- Following this, we calculate the primary jump points. These are also called forced neighbours, which occur when travelling in cardinal directions. There are eight cases of forced neighbours, these can be seen in the image below.



Traveling eastward: forced neighbors of current node N

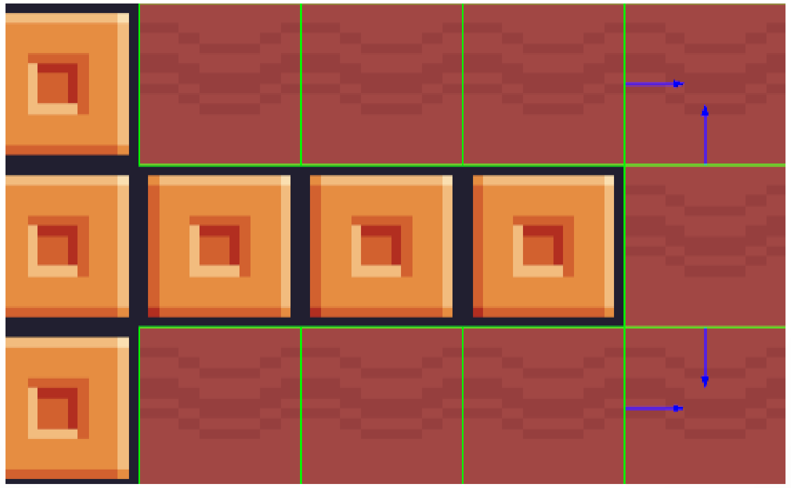
Traveling westward: forced neighbors of current node N

Traveling northward: forced neighbors of current node N

Traveling southward: forced neighbors of current node N

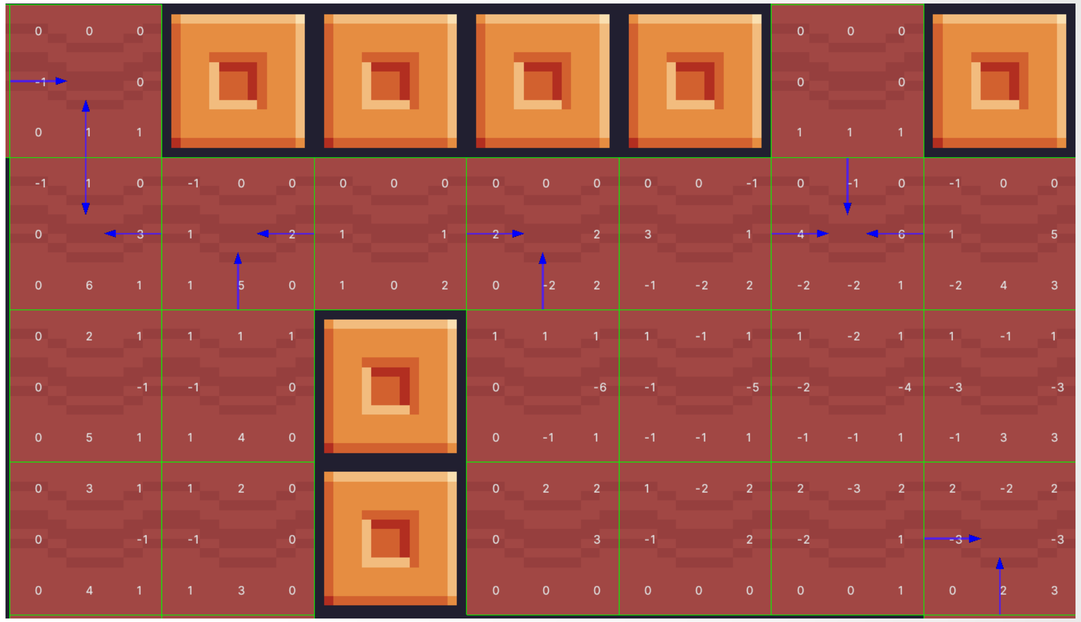
A* with Optimisations ~ AI Explanation (3)

- Note that a node is a primary jump point when moving to it in the direction indicated. For instance, if a primary jump point only has a right direction, if moving to it from the left, it is not considered as a primary jump point.



A* with Optimisations ~ AI Explanation (4)

- Following this, we calculate the straight and diagonal jump points as well as the wall distances.
  - Straight Jump Points - We assign the cardinal values of a cell to the numbers of cells needed to run into a primary jump point for that cardinal direction of travel.
  - Diagonal Jump Points – We assign the diagonal values of a cell to the shortest distance between the current cell and a cell in which a diagonal direction of travel will reach either a primary jump point or a straight jump point that is traveling in its cardinal directions (Ex: If travelling to top right, cardinal directions are up and right).
  - Wall Distances - If a direction has a value of zero, we assign it the negative distance between it and the nearest wall of that direction

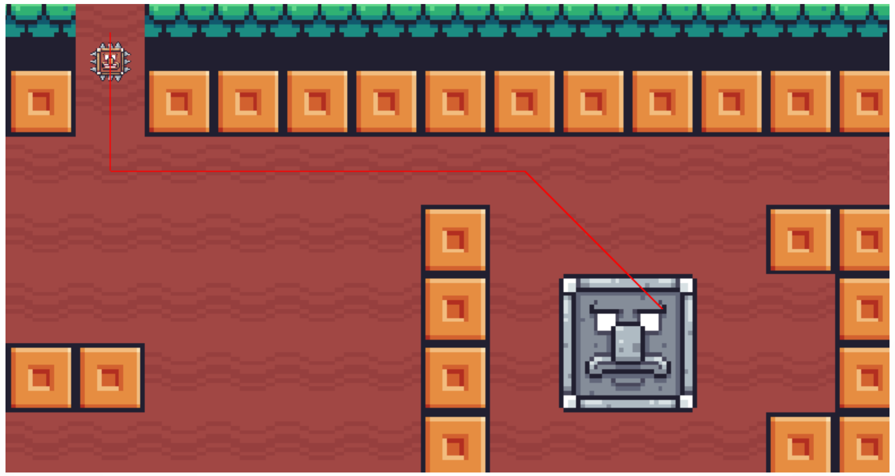


A* with Optimisations ~ AI Explanation (5)

- After generating the jump point search grid, we start the A* search. We will be using two lists, the open list to hold the unexplored nodes and the closed list to hold the explored ones.
- The algorithm can be described using the following pseudocode:
  - Add the starting node to the open list.
  - Loop while the node is not empty and take the node from the open list with the smallest $f(n)$, recall that $f(n) = g(n) + h(n)$.
    - If the node is the goal node, the path is complete.
    - Else
      - Move the node to the closed list
      - Check if any jump points within the node's reach have a lower $g(n)$ than the current node. If they do, add them to the open list.
      - For each neighbouring node that is not in the open list, closed list, or is an obstacle, add it to the open list.

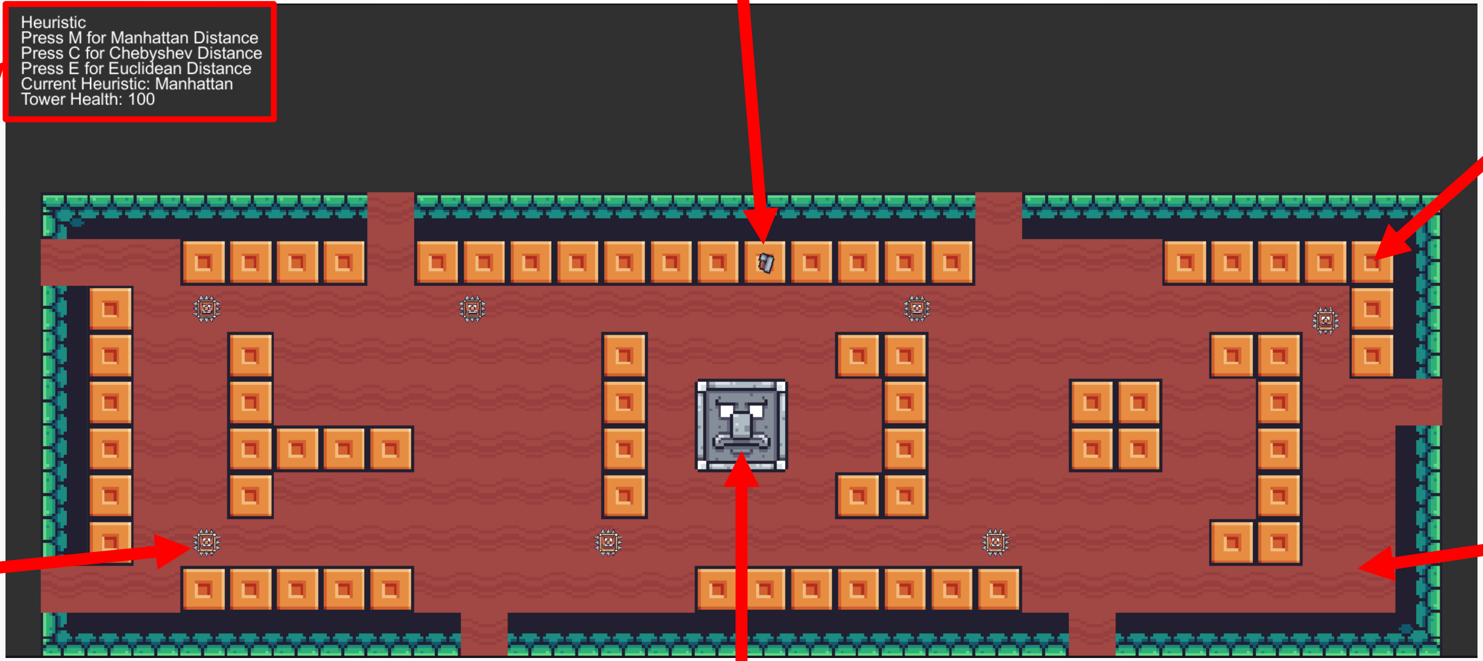
A* with Optimisations ~ AI Explanation (6)

- Once the path has been generated, the object (in our game's case, the enemy) will move from one node to another until it reaches the goal node.
  - Note: Red line represents the path the enemy will take.



A* with Optimisations ~ Mini-Game Implementation (1)

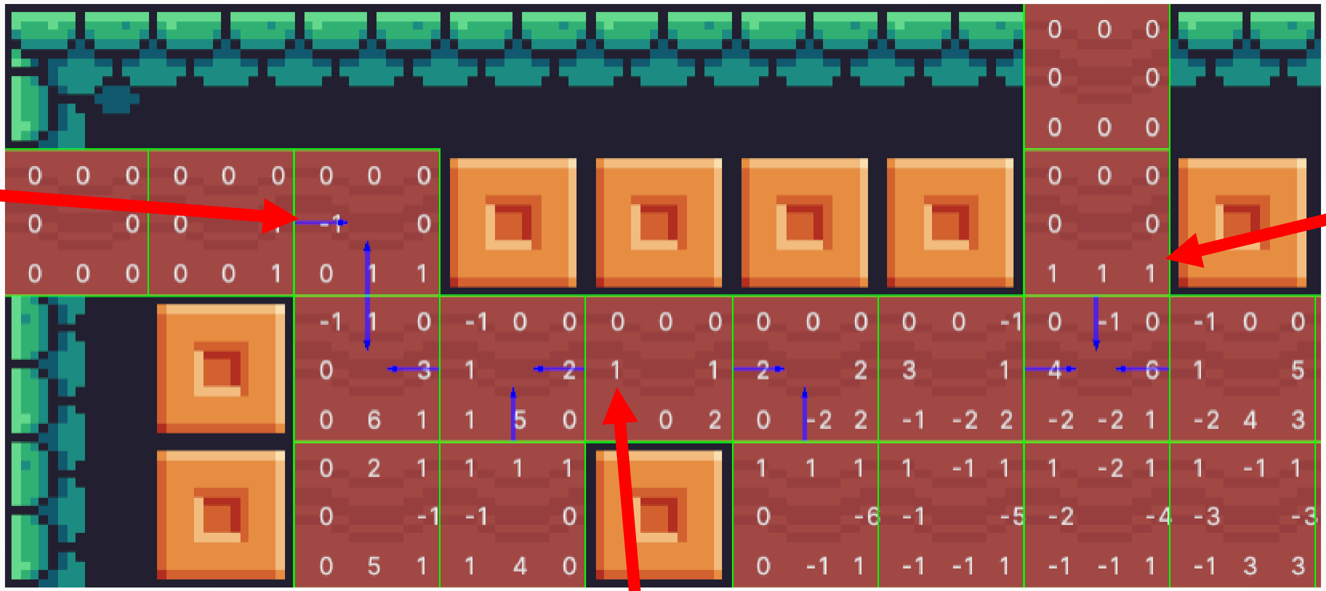
Playable Area:



Heuristic
Press M for Manhattan Distance
Press C for Chebyshev Distance
Press E for Euclidean Distance
Current Heuristic: Manhattan
Tower Health: 100

A* with Optimisations ~ Mini-Game Implementation (2)

Primary Jump Points

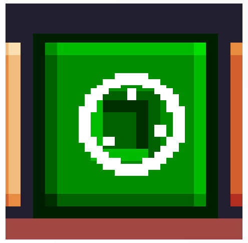


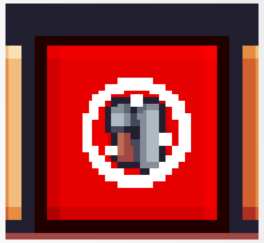
Diagonal Jump Points

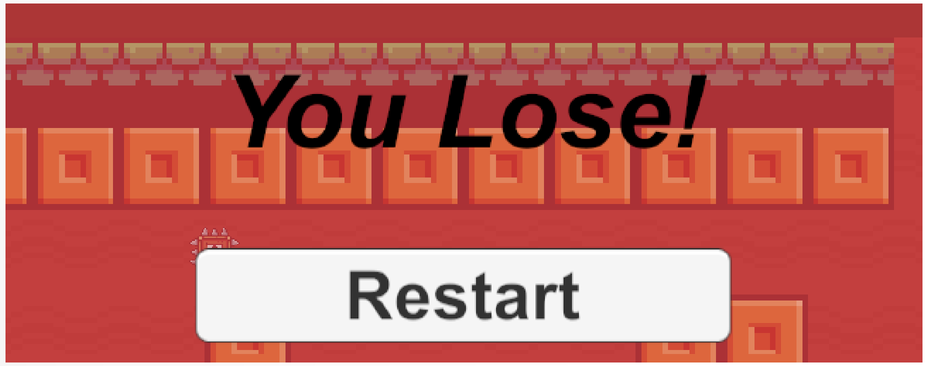
Straight Jump Points

A* with Optimisations ~ Mini-Game Implementation (3)

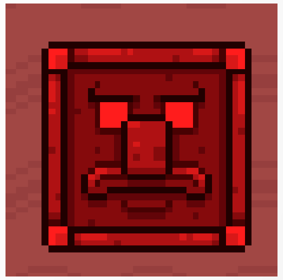
- This game is a simple top-down tower-defence game, where the player places weapons to stop enemies from attacking the tower. Every time an enemy collides, with the tower, its health drops. Once its health reaches zero, a game over screen is displayed with an option to restart.
- Squares are highlighted green when they are available to place a tower and the cursor is hovering on them. If they are unavailable, they are highlighted in red.
- The tower's colour gradually changes to red as its health drops.


Available Tile


Unavailable Tile



Game Over Screen


Damaged Tower

A* with Optimisations ~ Mini-Game Implementation (4)

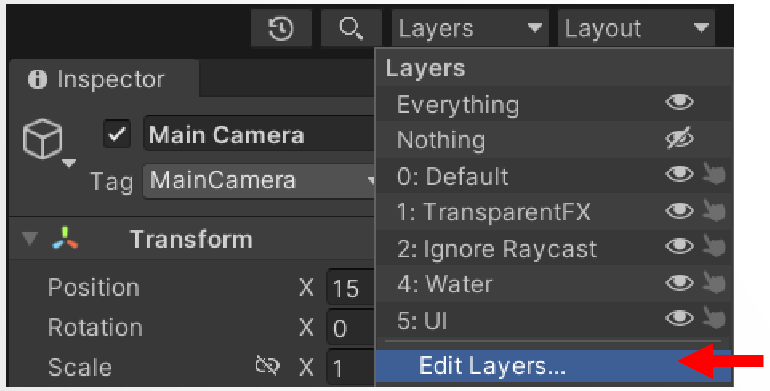
- Implementation of the Mini Game was inspired from [1-4], and sprites used to create the game were used from [5].
- The scripts related to A* and JPS found in the Scripts sub-directory include the:
  - Cell - This script is being used as a data structure, to initialise each individual cell which compose the field grid.
  - FlowFieldGrid – This script is responsible for generating the jump point search grid.
  - Pathfinding – This script performs A* pathfinding, aided by JPS, and moves the game object through the path once it is found.
  - PathGrid - This script is being used as a data structure when finding a path, it is composed of path nodes.
  - PathNode - This script is being used as a data structure when finding a path. Each path node is associated with a cell in the FlowFieldGrid.
  - SetHeuristic – This script is used to change between one heuristic to another, depending on user input.
  - Vectors - This script is being borrowed from Game 4.

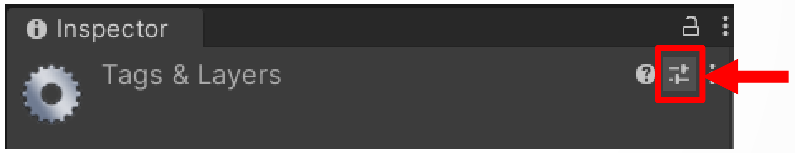
A* with Optimisations ~ Mini-Game Implementation (5)

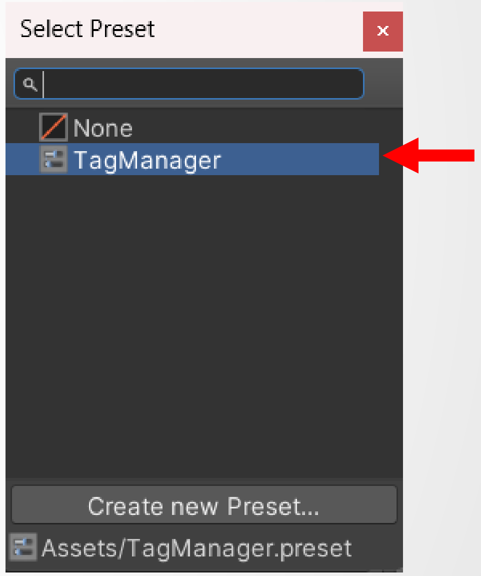
- The other scripts found in the Scripts sub-directory include the:
  - AttackEnemy – Applied to weapons so that they look at and destroy enemies after a set amount of time.
  - CursorScript – This script manages movement related to the cursor.
  - PlaceTower – This script is responsible for placing weapons where available and highlighting the tiles appropriately.
  - RestartGame – This script is used to restart the current scene.
  - Spawner – This script instantiates enemies every set amount of time.
  - TowerHealth – This script manages the tower's health and displays the canvas to restart the game once the health variable reaches zero.

A* with Optimisations ~ Mini-Game Implementation (6)

- **How to Set Up Package:**
  - To initially set up the layers in the project, go in the edit layers section found on top of the inspector tab. Following this, click on the slider icon and double click on TagManager to import it. This import has to be done for both the exercise and solution packages due to how Unity exports packages.

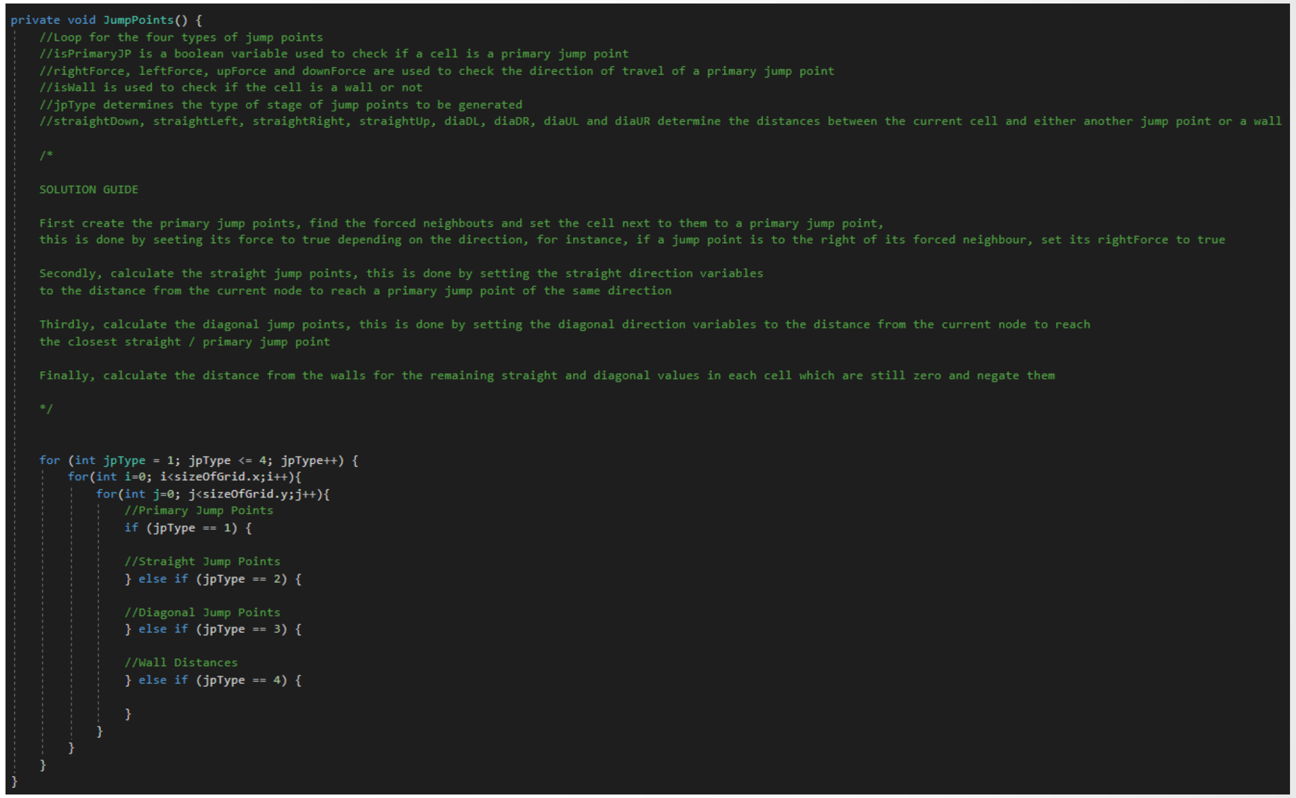






A* with Optimisations ~ Exercise (1)

- Replace the dummy program with the functionality mentioned earlier.
- Solution guide provided in the next few slides.



```
private void JumpPoints() {
    //Loop for the four types of jump points
    //isPrimaryJP is a boolean variable used to check if a cell is a primary jump point
    //rightForce, leftForce, upForce and downForce are used to check the direction of travel of a primary jump point
    //isWall is used to check if the cell is a wall or not
    //jpType determines the type of stage of jump points to be generated
    //straightDown, straightLeft, straightRight, straightUp, diaDL, diaDR, diaUL and diaUR determine the distances between the current cell and either another jump point or a wall

    /*

    SOLUTION GUIDE

    First create the primary jump points, find the forced neighbouts and set the cell next to them to a primary jump point,
    this is done by seeting its force to true depending on the direction, for instance, if a jump point is to the right of its forced neighbour, set its rightForce to true

    Secondly, calculate the straight jump points, this is done by setting the straight direction variables
    to the distance from the current node to reach a primary jump point of the same direction

    Thirdly, calculate the diagonal jump points, this is done by setting the diagonal direction variables to the distance from the current node to reach
    the closest straight / primary jump point

    Finally, calculate the distance from the walls for the remaining straight and diagonal values in each cell which are still zero and negate them

    */


    for (int jpType = 1; jpType <= 4; jpType++) {
        for(int i=0; i<sizeOfGrid.x;i++){
            for(int j=0; j<sizeOfGrid.y;j++){
                //Primary Jump Points
                if (jpType == 1) {

                //Straight Jump Points
                } else if (jpType == 2) {

                //Diagonal Jump Points
                } else if (jpType == 3) {

                //Wall Distances
                } else if (jpType == 4) {

                }
            }
        }
    }
}
```

A* with Optimisations ~ Exercise (2)

Now it's your turn to Code ! – Let's implement the A* and JPS algorithms ☺

Open the FlowFieldGrid script and implement the following code in the JumpPoints() function:

1. If the jpType is 1 (The following is the pseudocode for two cases, in total there are eight cases, the other cases are for the other cases of forced neighbours as seen in slide 5):
   - If i and j are within the grid
     - If the cell at i and j+1 is a wall, and the cell at i and j is not a wall
       - If the cell at i+1 and j+1 is not a wall and the cell at i+1 and j is not a wall
         - Set the cell at i+1 and j to a primary jump point and set its rightForce to true
       - If the cell at i-1 and j+1 is not a wall and the cell at i-1 and j is not a wall
         - Set the cell at i-1 and j to a primary jump point and set its leftForce to true

A* with Optimisations ~ Exercise (3)

2. If the jpType is 2 (The following is the pseudocode for one case, in total there are four cases):
   - If i and j are within the grid
     - Loop from i+1 to the sizeOfGrid.x – 1, using variable x
       - If the cell at k and j is a wall, break out of the loop
       - If the cell at k and j is a primary jump point and has a rightForce, set the cell at i and j's straightRight to Math.Abs(i-k)

A* with Optimisations ~ Exercise (4)

3. If the jpType is 3 (The following is the pseudocode for one case, in total there are four cases):
   - If i and j are within the grid
     - Set k to 1
     - Loop while i+j and j+k are within the grid, and the cell at i+k and j+k is not a wall
       - If the cell at i+k and j+k has a straightUp value greater than zero, set the cell at i and j's diaUR to k and break out of the loop.
       - If the cell at i+k and j+k has a straightRight value greater than zero, set the cell at i and j's diaUR to k and break out of the loop.
       - Increment k by one.

A* with Optimisations ~ Exercise (5)

Open the PathFinding script and implement the following code:

4. Create a function FindPath which returns a list of type PathNode and takes integer parameters startX, startY, endX and endY.
   - Create two PathNodes: startNode and endNode and set them to the nodeGrid's gridArray at the passed parameters locations.
   - Set the openList to a list containing the startNode and set the closedList to an empty list.
   - Loop through evert PathNode in the nodeGrid's gridArray and set its gCost to the maximum integer value, call the CalculateFCost() function on it, and set its cameFromNode to null.

A* with Optimisations ~ Exercise (6)

Open the PathFinding script and implement the following code:

5. In the FindPath() function:
    - Set the startNode's gCost to zero and set its hCost to the returned value of calling the CalculateDistanceCost() function with startNode and endNode as passed parameters. Then, call the CalculateFCost() function on the startNode.
    - Loop while the openList is not empty
        - Get the node with lowest fcost via the function GetLowestFCostNode(), if it is the endNode return the list returned by calling the CalculatePath() function on the endNode.
        - Remove the current node from the open list and add it to the closed list.

A* with Optimisations ~ Exercise (7)

Open the PathFinding script and implement the following code:

6. In the FindPath() function's while loop:
    - Declare a new PathNode called tempNode
    - If the current node's cell has a straightDown value greater than zero, set a tempTentativeGCost to the current node's gCost added with the return of the CalculateDistanceCost() function with the currentNode and the primary jump point node it is pointing to as parameters. Set tempNode to the primary jump point.
        - If the tempTentativeGCost is smaller than the tempNode's gCost, set the tempNode's cameFromNode to the current node, its gCost to the tempTentativeCost, its hCost to the return of the CalculateDistanceCost() function with tempNode and endNode as passed parameters, and call the CalculateFCost() function on it. If the tempNode is not in the openList, add it.
    - Repeat this functionality for the other seven directions.

A* with Optimisations ~ Exercise (8)

Open the PathFinding script and implement the following code:

7. In the FindPath() function's while loop:
   - Loop through each pathnode in the returned list when calling the GetNeighbours function with the currentNode as a passed parameter.
     - If the closedList already contains the node, continue to the next iteration.
     - If the node is a wall, add it to the closed list and continue to the next iteration.
     - Set tentativeGCost to the current node's gCost added with the return of calling the CalculateDistanceCost() function with the currentNode and the neighbourNode.
     - If the tentativeGCost is smaller than the neighbour node's gCost
     - Set the neighbour node's cameFromNode to the current node, its gCost to the tentativeGCost, its hCost to the return of the CalculateDistanceCost() function with the neighbour node and endNode as passed parameters, and call the CalculateFCost() function on it. If the tempNode is not in the openList, add it.

A* with Optimisations ~ Exercise (9)

Open the PathFinding script and implement the following code:

8. Create another function called FindPath() which returns a list of Vector3.
    - Set startX, startY, endX and endY to the current game object and the target's game object's cell position appropriately.
    - Call the FindPath() function with startX, startY, endX and endY as passed parameters and save the returned list.
    - If the list is null, return null, else create a new list called vectorPath and for each PathNode in the list, add its Vector3 position into vectorPath. Set pathFound to true and return vectorPath.

A* with Optimisations ~ Exercise (10)

Open the PathFinding script and implement the following code:

9. In the Start() function, after setting the nodeGrid, call the FindPath() function and save it in pathToFollow. Set startPosition to the current position's cell's grid index vector.
10. In the Update() function, remove the transform.position line of code and add the following:
    - If pathFound is true
        - If the current game object's position is equal to the pathToFollow at waypointCounter's Vector3, increment waypointCounter by one. Else, set temp to the Vector3 returned by calling MoveTowards() with the current game object's position and the pathToFollow at waypointCounter. Set transform.position to the Vector3 calculated.

A* with Optimisations ~ Conclusion

- A* algorithm has a guaranteed optimal path given the admissible heuristics, thus it provides a realistic and intelligent pathfinding experience for enemies.
- However, in a large search space, A* is computationally expensive. It requires memory for both a very large open list as well as a closed list.
- To ease the amount of memory required and reduce the computation time, we use Jump Point Search to skip through open areas in a grid

A* with Optimisations ~ References

[1] - S. Rabin, "JPS+ An Extreme A* Speed Optimization for Static Uniform Cost Grids," *Game AI Pro*. [Online]. Available: http://www.gameaipro.com/GameAIPro2/GameAIPro2_Chapter14_JPS_Plus_An_Extreme_A_Star_Speed_Optimization_for_Static_Uniform_Cost_Grids.pdf. [Accessed: 26-Mar-2023].

[2] - "JPs+: Over 100x faster than A*," *GDC Vault*. [Online]. Available: https://www.gdcvault.com/play/1022094/JPS-Over-100x-Faster-than. [Accessed: 26-Mar-2023].

[3] – "Trgrote, "Trgrote/JPs-Unity: Interactive JPS Search Algorithim, using Steve Rabin's algorithim," *GitHub*. [Online]. Available: https://github.com/trgrote/JPS-Unity. [Accessed: 26-Mar-2023].

[4] - Prof. A. Dingli, ICS2211: "LEVEL 3 PATHFINDING" [Online]. Available: https://www.um.edu.mt/vle/pluginfile.php/1108327/mod_resource/content/1/Level3_PathFinding.pdf [Accessed: 18-Mar-2023]

[5] - Pixel Frog, "Unity Asset Store: Pixel Adventure 1" 2019 [Online]. Available: https://assetstore.unity.com/packages/2d/characters/pixel-adventure-1-155360 [Accessed: 18-Mar-2023]