Hanze University of Applied Sciences

Creative Media & Game Technologies

Elective Artificial Intelligence

Beibei Ji

434608

Contents

Introduction

Goal of the elective Artificial Intelligence, we explored and selected an AI technology to explore its potential with research, create a prototype and document the whole experimentation. I have experimented a few AI technologies but mostly ended up focusing on maze generation and pathfinding.

I have worked mostly on automatization of generating mazes. Therefore, most of the information will be related to it and the process I went through to my latest prototype. Additionally, various algorithm will be discussed since there are many ways to solve my case. Such as:

- Procedural Generation
- Pathfinding algorithms
- Maze generation algorithms
- A* algorithm
- Dijkstra's algorithm
- Breath-first search (BFS)
- Depth-first search (DFS)
- Heuristics
- Object-oriented programming (OOP)
- Unity game engine
- Game development
- Artificial intelligence (AI)

Research

Maze generation can be typically done using procedural generation techniques which means it is created algorithmically instead of manually. It can be created using a set of rules like randomly placing walls and carving out paths. Similarly, such algorithms are possible to use for pathfinding for navigation in-game or NPC behaviour.

Procedural Generation

Procedural generation is a technique used in computer graphics, game design and other areas to generate content algorithmically. In game development, it can be used to create worlds, levels, characters and other content. It can save a lot of time instead of manually creating similar game components, however, it is important to compare the advantages and disadvantages when using procedural generation.

Great examples to look into would be No Man's Sky(Hello Games, 2016) which generates their whole universe including each planets and star system. It saves a lot of space as storing 100 planets would be quite difficult and not pleasant to store in devices. Dwarf Fortress (Adams, 2006) is interesting for its generated world. Every new game would be created with a unique world and its own history and lore. Lastly, Minecraft (Mojang Studios, 2011) is a sandbox game that generates worlds simply with blocks.

Pathfinding Algorithms

Pathfinding algorithms are computational methods used to find the shortest and most efficient path between two points in a graph or network. They are often used in robotics, logistics, planning, and game design. Many games rely on pathfinding to allow NPCs to move around as well navigation for the players, etc.

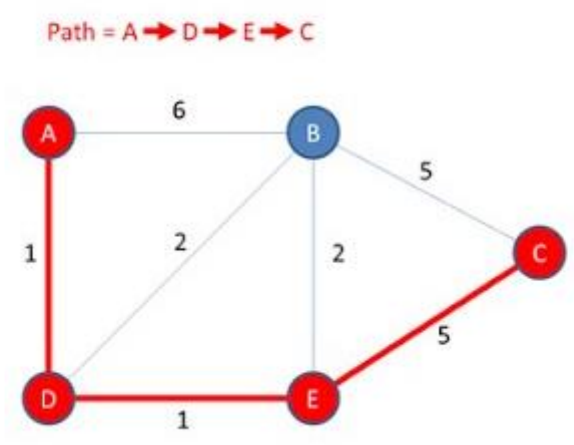
Algorithms

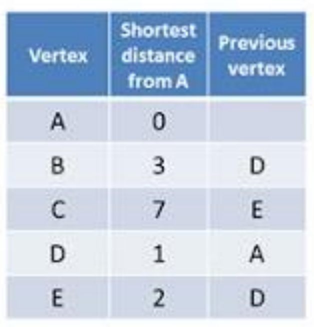
An algorithm is a step-by-step procedure or set of rules designed to solve a specific problem or perform a specific task. They are used in a wide range of applications including AI.

This document will discuss a few of the common algorithms below.

Dijkstra's Algorithm

Djikstra's algorithm is popular in graph theory to find the shortest path between two nodes in a graph. It works by starting at a specified point/cell/node (node will be mostly used from now on) and visit its neighbours and marking the cost to reach them. It selects a neighbour with the lowest cost and the process repeats again until it reaches its goals. The costs can be represented by weight or distance.





Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	D
C	7	E
D	1	A
E	2	D

Heuristic Algorithm

It is a technique to estimate or approximate the right solution. In terms of pathfinding, heuristic uses an estimate of the distance between two points to guide the search for the shortest path. Heuristic function is used to evaluate each node or location in the search space to prioritize the most promising path which reduces the time and effort to find the optimal solution.

A* Algorithm

A-star is an algorithm that uses both the cost to reach a node and the estimated cost to reach the goal from that node to determine the best path to take. It's an extension of Djikstra's algorithm, which only considers the cost to reach a node.

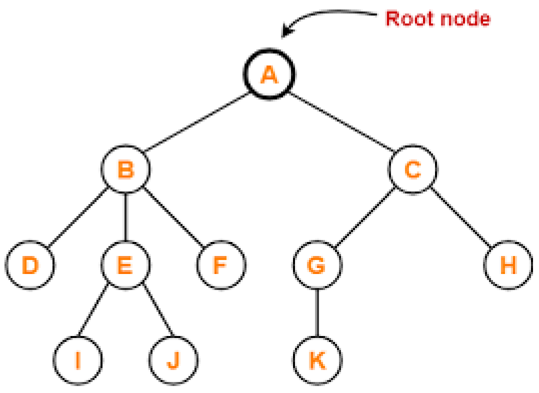
It works by maintaining a priority queue of nodes to visit next, sorted by the sum of the cost to reach a node and the estimated cost to reach the goal. It choose the node with the lowest total cost and examines its neighbours and updating their cost estimates to find a cheaper path. This process continues until the goal is reached or there are no more nodes to check.

The estimated cost to reach the goal is calculated using a heuristic function that provides an estimate of the distance between a node and the goal. With this function, it is highly unlikely to overestimate the actual cost to reach the goal.

Breath-First Search (BFS)

It’s a graph traversal algorithm that starts at a particular node(usually the root note aka the very beginning, you can imagine a tree that’s spreading out into a triangle) and explores all the neighbouring nodes at the current depth level before moving on to nodes at the next depth level.

BFS is often used for finding the shortest path between two nodes in an unweighted graph as it explores nodes in order of their distance from the starting node.



Depth-First Search (DFS)

DFS is a graph traversal algorithm that explores as far as possible along each branch before backtracking. Unlike BFS, DFS checks all the neighbouring nodes by going through the levels along a branch and then going back to check another neighbouring node with a branch. It could be explained that DFS explores vertically, while BFS mentioned above explores horizontally.

Prototype

I focused on being able to generate a maze first. I created a 3D Unity project and started to create a script to code. You can easily access it by right-clicking on the 'project' section and then create>C# script.

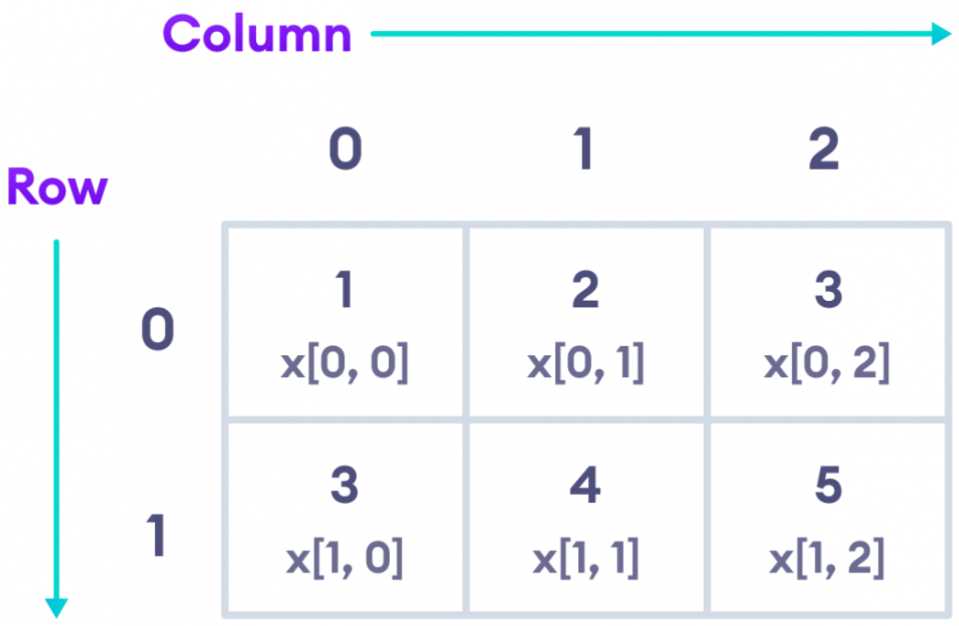
To create the maze, I have used a recursive back-tracker. Similarly, to Depth-First Search, it explores the nodes by going as deep as possible along each branch and then backtracking until it doesn't find any unvisited nodes. With this function when generating a maze, it creates only one possible way out.

For this function, I mainly followed a tutorial on YouTube by javidx9 but it was not made in Unity.

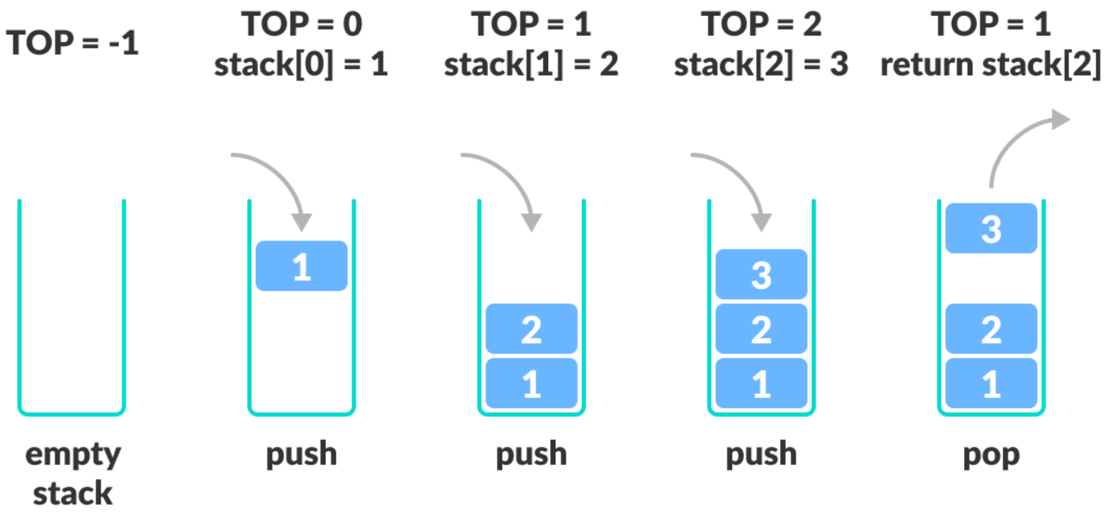
Here is the link: https://www.youtube.com/watch?v=Y37-gB83HKE

Regardless, main steps will be explained here as well.

Firstly, a maze is a 2-dimensional as it has an x or y axis. Therefore, we would need a grid to have some determined size for that I created a 2-dimensional array. It can be described as a list that has two values of information in each cell, the first number representing the 'x' and second number representing the 'y'. These would give us the location inside the grid.

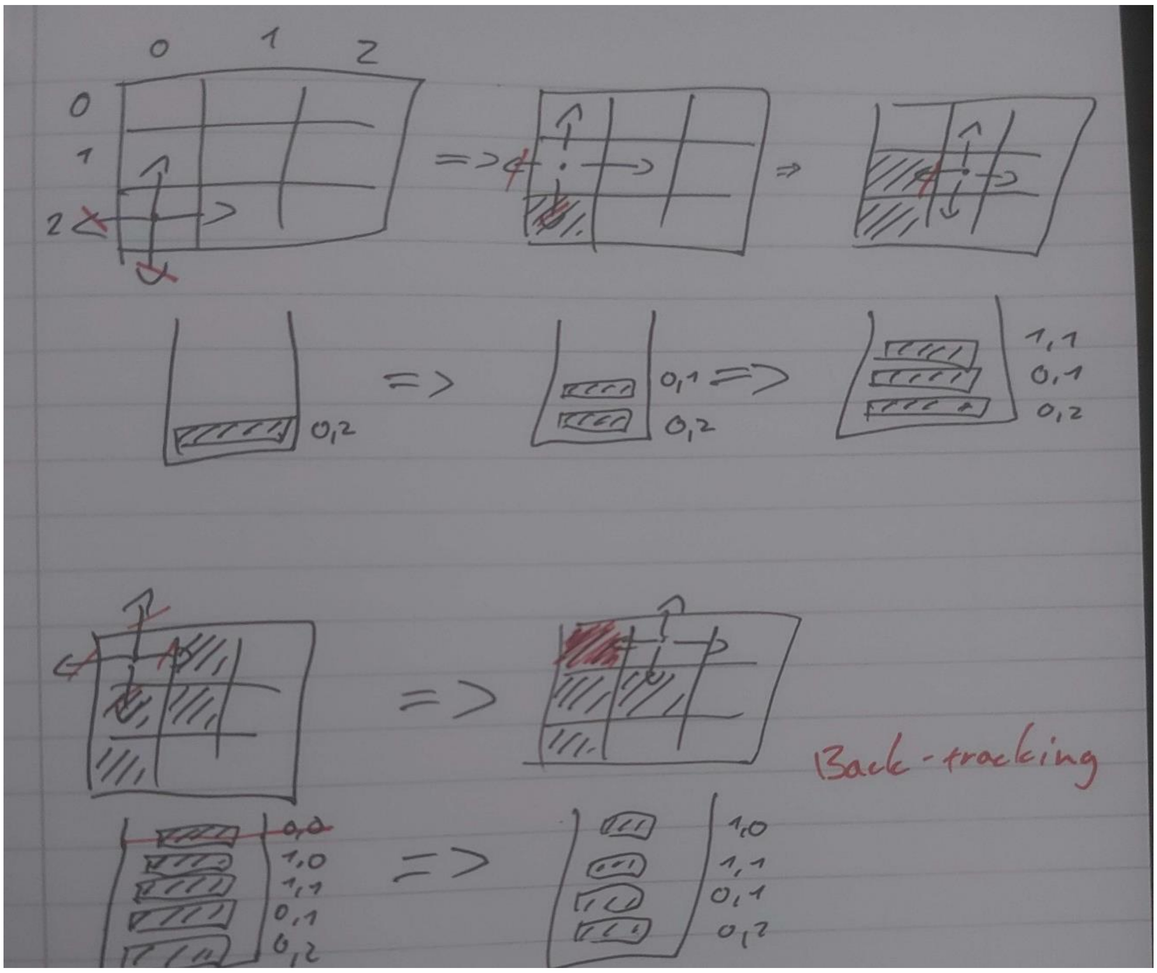


To track the path I used a Stack that is a data structure which stores a collection of elements and allows operations like adding values and taking them out in the principle "last in, first out" also known as LIFO. This means that the last element will be the first one to be removed. For the code, I used the Stack to overview which points belong to our one 'path'. It will add locations from the 2D array and when it hits an end, it will back-track by taking each location from the last visited until we find a point where we can change direction.



The function how it would explore is by checking each neighbouring side (North, East, South, and West) to check if they were not visited yet. In case there are more than 1 unvisited sides, it would randomly choose one and repeat the action. If all sides are visited, the stack will take out the last location it went to, meaning we would start the same action with a location that we were at right before the current one.

For a visualization:



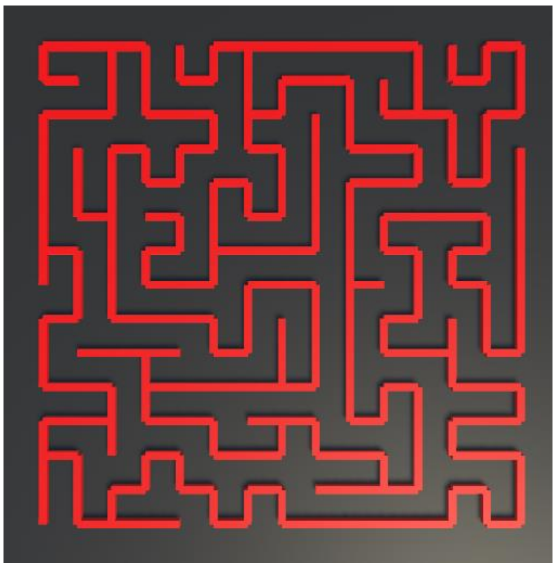
And code (MazeCreation.cs):

```
void Update()
        {
            //maze algorithm
            if(m_visitedCells<mazeHeight*mazeWidth)
            {
                //in case there are no cells in stack
                if(myPairStack.Count==0)
                    {
                        CreateAnotherFrontier();
                    }

                var neighbours = new List<int>();
                //create a set of unvisited neighbours(current cell/node)
                var _top = new
int[]{myPairStack.Peek().x,myPairStack.Peek().y};

                SearchNeighbours(_top[0],_top[1],neighbours);

                //are the neighbours available?
                if(neighbours.Count > 0)
                {
                    //randomly choose next cell
                    int _nextDir =
UnityEngine.Random.Range(0,neighbours.Count);

                    //create a path between the previous and new cell
                    switch(neighbours[_nextDir])
                    {
                        case 0:
                                if(m_Maze[_top[0],_top[1]-1] == 0)
                                {
                                    m_Maze[_top[0],_top[1]-1] = 1;

                                    Instantiate(block,new
Vector3(myPairStack.Peek().x*sizeMultipler,0,(myPairStack.Peek().y-
0.5f)*sizeMultipler), Quaternion.Euler(0,90,0));
                                    m_visitedCells++;
                                }
                                myPairStack.Push(new Pair<int,
int>(_top[0],_top[1]-1));
                            break;
                        case 1:
                                if(m_Maze[_top[0]+1,_top[1]] == 0)
                                {
                                    m_Maze[_top[0]+1,_top[1]] =1;
                                    Instantiate(block,new
Vector3((myPairStack.Peek().x+0.5f)*sizeMultipler,0,myPairStack.Peek().y*sizeM
ultipler), Quaternion.identity);
                                    m_visitedCells++;
                                }
                                myPairStack.Push(new Pair<int,
int>(1+_top[0],_top[1]));
                            break;
                        case 2:
                                if(m_Maze[_top[0],_top[1]+1]==0){
                                    m_Maze[_top[0],_top[1]+1] = 1;

                                    Instantiate(block,new
Vector3(sizeMultipler*myPairStack.Peek().x,0,(myPairStack.Peek().y+0.5f)*sizeM
ultipler), Quaternion.Euler(0,90,0));
                                    m_visitedCells++;
                                }
                                myPairStack.Push(new Pair<int,
int>(_top[0],_top[1]+1));
                            break;
                        case 3:
                                if(m_Maze[_top[0]-1,_top[1]] ==0)
                                {
                                    m_Maze[_top[0]-1,_top[1]] = 1;

                                    Instantiate(block,new
Vector3((myPairStack.Peek().x-
0.5f)*sizeMultipler,0,myPairStack.Peek().y*sizeMultipler),
Quaternion.identity);
                                    m_visitedCells++;
                                }
                                myPairStack.Push(new Pair<int, int>(-
1+_top[0],_top[1]));
                            break;
                    }

                }else{
                    myPairStack.Pop();
                    //no available neighbours so were going back
                }
            }
        }
```
The code will constantly activate if the number of visited cells/nodes is less then the total amount of cells/nodes on the grid that we created.

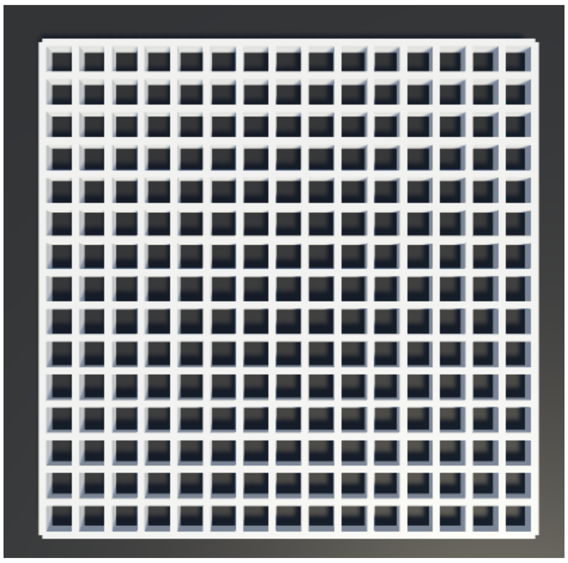
For specifically searching for unvisited neighbours of the current cell/node, I created a separate function that uses numbers and adds them to a small list that is stored in the Update function shown above:

```
        void SearchNeighbours(int a, int b, List<int> neighbours)
        {
            var _top = new int[]{a,b};
                //North
                if(b>0 && m_Maze[a,b-1]==0)
                {
                    neighbours.Add(0);
                }
                //East
                if(a<mazeWidth-1 && m_Maze[a+1,b]==0)
                {
                    neighbours.Add(1);
                }
                //South
                if(b<mazeHeight-1 && m_Maze[a,b+1]==0)
                {
                    neighbours.Add(2);
                }
                //West
                if(a>0 && m_Maze[a-1,b]==0)
                {
                    neighbours.Add(3);
                }
        }
```

Despite already having a script and instantiating objects to show how the path is made, we didn't make the walls yet. Hence, it'll look something like this:



For creating the grid, I made another script called 'GritCreation.cs' that would take the variables from the previous code to get the exact size as the MazeCreation.cs has.



The code creates each side separately so the walls could be destroyed.

```
        void Start()
        {
            wall = gameObject.GetComponent<MazeCreation>().wall;
            mazeHeight = gameObject.GetComponent<MazeCreation>().mazeHeight;
            mazeWidth = gameObject.GetComponent<MazeCreation>().mazeWidth;
            sizeMultipler =
gameObject.GetComponent<MazeCreation>().sizeMultipler;
            m_Maze = new int[mazeWidth,mazeHeight];

            for(int i=0;i<mazeWidth;i++)
            {
                for(int j=0;j<mazeHeight;j++)
                {
                    Instantiate(wall,new Vector3(i*sizeMultipler,0,(j-
0.5f)*sizeMultipler),Quaternion.identity);
                    Instantiate(wall,new
Vector3((i+0.5f)*sizeMultipler,0,j*sizeMultipler),Quaternion.Euler(new
Vector3(0,90,0)));
                }
            }
            for(int i=0;i<mazeHeight;i++)
            {
                Instantiate(wall,new Vector3(-
0.5f*sizeMultipler,0,i*sizeMultipler),Quaternion.Euler(new Vector3(0,90,0)));
            }
            for(int i=0;i<mazeWidth;i++)
            {
                Instantiate(wall,new Vector3(i*sizeMultipler,0,(mazeHeight-
0.5f)*sizeMultipler),Quaternion.identity);
            }

        }
```

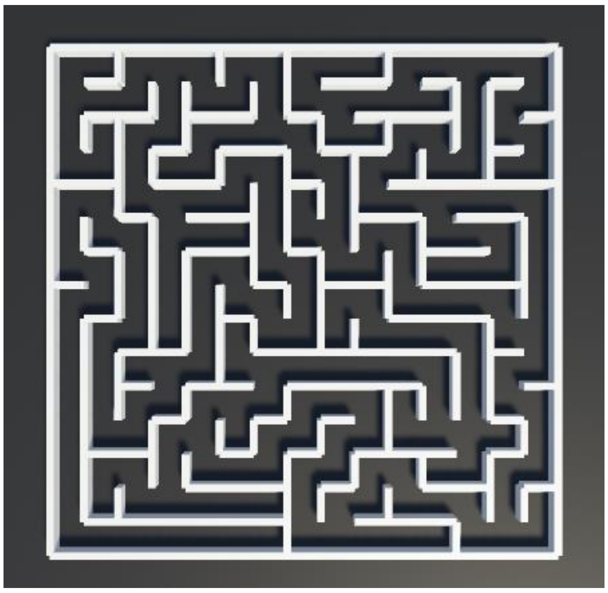
However, the object that destroys those walls is being created by the first code that tries to make a path. Since the maze has cells with the size 1x1, I create the object by 0.5 towards the side the code chose to go to before the function starts again. Hence, when it touches a ‘wall’ it would destroy the wall and itself since it won’t make the connection again anymore(DestroyWall.cs).

```
    void OnTriggerEnter(Collider other)
    {
        if(other.tag=="wall")
        {
            Destroy(other.gameObject);
            Destroy(gameObject);
        }
    }
```

That object is made as a prefab and a trigger which means that it can pass through other objects without affecting them. To make a prefab, create a folder ‘Prefabs’ in the project and put a 3D object inside the folder with the following traits on it:

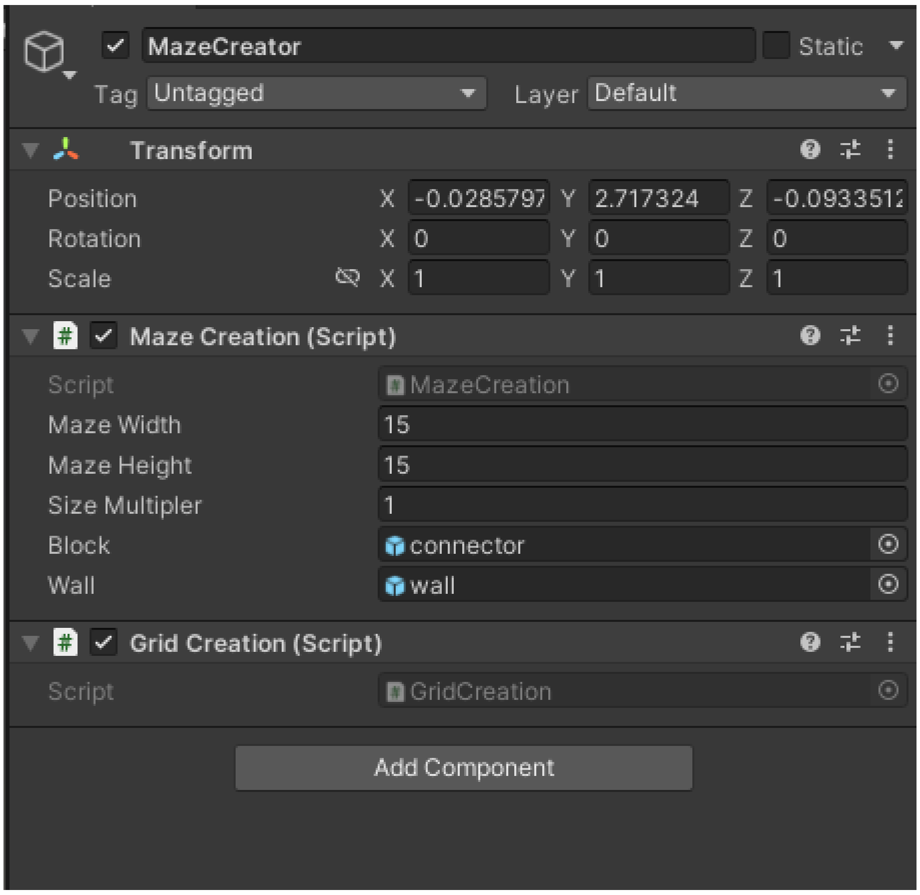
- RigidBody
- Collider → check the trigger box
- DestroyWall.cs

And this is the result:



Maze in the Unity Project

In the Hierarchy, there is an object called ‘MazeCreator’, click on that object and then in the Inspector you have two codes ‘MazeCreator.cs’ and ‘GritCreator.cs’. You can freely change the size by changing the number in the ‘Maze Width’ and ‘Maze Height’ row. Additionally, you can also change the number on ‘Size Multiplier’ but that is at your own risk.


MazeCreator
Static
Tag Untagged
Layer Default

Transform
Position X -0.0285797 Y 2.717324 Z -0.0933512
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Maze Creation (Script)
Script MazeCreation
Maze Width 15
Maze Height 15
Size Multipler 1
Block connector
Wall wall

Grid Creation (Script)
Script GridCreation

Add Component

In case you won't be able to find the script in the project, you can right click on their names and click on 'Edit script'.

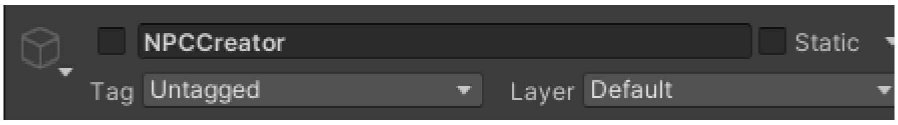
NPC Behaviour

The NPC behaviour can have the same principle as the maze creation above, but with a bit different conditions to check whether there is a wall etc. I have only created a script where they would go forward and change direction to left or right when they bump into a wall. Unfortunately, it doesn't work much as it was intended.

The code(NPCSearch.cs):

```
       void Update()
        {
            if(action == sides.moving){
                transform.Translate(Vector3.forward * speed * Time.deltaTime);
            }else{
                gameObject.transform.position +=
Vector3.back*speed*Time.deltaTime;
            }

        }
        void OnCollisionEnter(Collision collision){
            if(collision.gameObject.CompareTag("wall"))
            {
                Debug.Log("FSDGSDG");
                action = sides.searching;
            }
        }
        void OnCollisionExit(Collision collision)
        {
                int rnd = UnityEngine.Random.Range(0,1);
                switch (rnd){
                    case 0:
                    gameObject.transform.Rotate(0,90,0);
                    //gameObject.transform.rotation = Quaternion.Euler(new
Vector3(0,gameObject.transform.rotation.y+90,0));
                    break;
                    case 1:
                    gameObject.transform.Rotate(0,-90,0);
                    //gameObject.transform.rotation = Quaternion.Euler(new
Vector3(0,gameObject.transform.rotation.y-90,0));
                    break;
                }
                action=sides.moving;
}
```

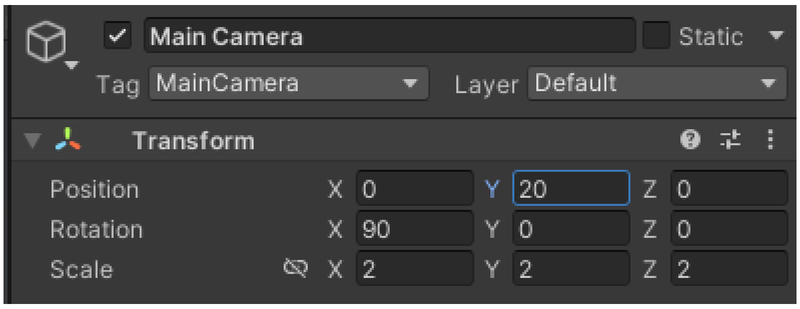
In the project, there is also an NPCCreator that spawns such described NPCs randomly on the grid. If you would like to see try it just check the box right next to the name in the inspector on the left.


NPCCreator Static
Tag Untagged Layer Default

Bonus Object

I also have added a simple spider model acting as 'player' so you would be able to move around as well. You can just find the spider in Hierarchy and check it just like NPCCreator.

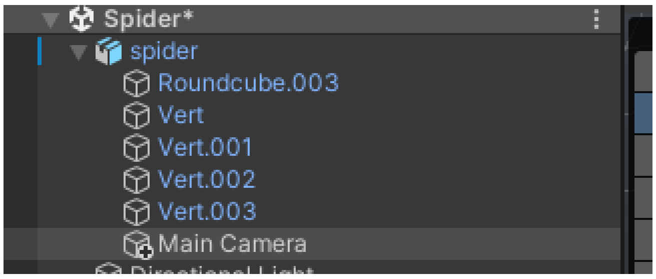
If you would like to have the camera follow the spider, you can set the camera coordinates as shown below:


✓ Main Camera Static
Tag MainCamera Layer Default

Transform

Position	X	0	Y	20	Z	0
Rotation	X	90	Y	0	Z	0
Scale	X	2	Y	2	Z	2

And put it in the spider object.


Spider*
spider
Roundcube.003
Vert
Vert.001
Vert.002
Vert.003
Main Camera

References

Youtube tutorial for recursive backtracking. https://www.youtube.com/watch?v=Y37-gB83HKE

https://media.pragprog.com/titles/jbmaze/first.pdf

https://www.redblobgames.com/pathfinding/a-star/introduction.html

Images:

https://www.youtube.com/watch?v=pVfj6mxhdMw

https://www.programiz.com/csharp-programming/multidimensional-arrays

https://www.gatevidyalay.com/tree-data-structure-tree-terminology/

Game References:

No Man's Sky. (2016). [Microsoft Windows, PlayStation 4]. Hello Games.

Minecraft. (2011). [Microsoft Windows, macOS, Linux, Android, iOS, Xbox 360, Xbox One, PlayStation 3, PlayStation 4, PlayStation Vita, Wii U, Nintendo Switch]. Mojang Studios.

Tarn Adams and Zach Adams. (2006). Dwarf Fortress. [Computer game]. Bay 12 Games.