MAZE GENERATOR

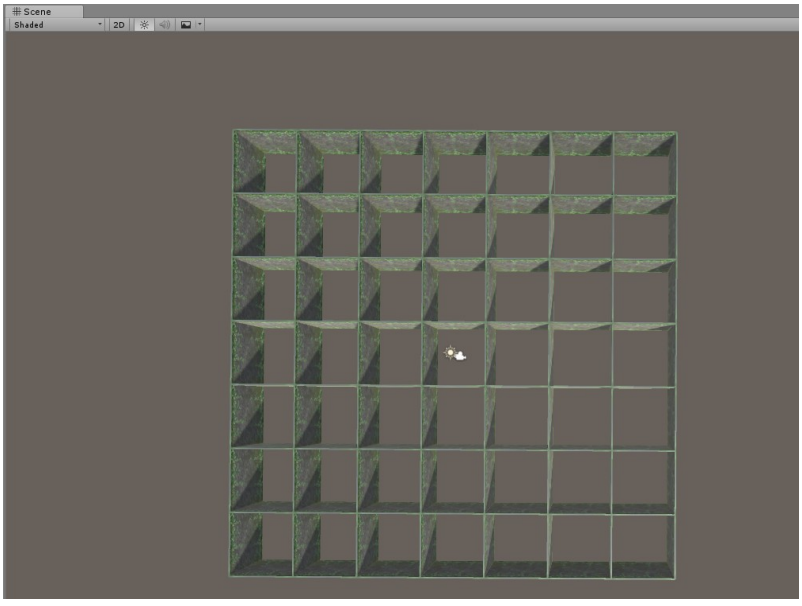
PSEUDOCODE

1) Scegli a random la cella iniziale, rendila “cella corrente” e contrassegnala come visitata
2) Trova i suoi vicini che non sono stati visitati
3) Scegli a random un suo vicino “non visitato”
4) Rendi il vicino come “cella corrente”
5) Distruggi il muro che separano queste due celle
6) Torna al punto 2 fino a quando tutte le celle non sono state visitate

L'algoritmo si basa su una matrice (max X max) scelta dall'utente. Il primo passo è creare i muri del labirinto. Con questa funzione creeremo la matrice.

```
void createWalls(GameObject allParent)
{
    Maze = new GameObject("Maze");

    //X
    for (int i = 0; i < ySize; i++)
    {
        for (int j = 0; j <= xSize; j++)
        {
            GameObject wallX = Instantiate(wall,
                                    new Vector3(initialPos.x + (j * sizeWall), 0, initialPos.z + (i * sizeWall)),
                                    Quaternion.identity) as GameObject;
            wallX.transform.parent = Maze.transform;

        }
    }


    //Y
    for (int i = 0; i <= ySize; i++)
    {
        for (int j = 0; j < xSize; j++)
        {
            GameObject wallY = Instantiate(wall,
                                    new Vector3(initialPos.x + (j * sizeWall) + middle, 0, initialPos.z + (i * sizeWall) - middle),
                                    Quaternion.Euler(0, 90, 0)) as GameObject;
            wallY.transform.parent = Maze.transform;
        }
    }
    Maze.transform.parent = allParent.transform;
}
```

RISULTATO



Come seconda cosa bisogna creare il pavimento per ogni cella

```
void createFloor(GameObject allParent)
{
    Floor = new GameObject("Floor"); //group all the Quad to keep everything in order

    for (int i = 0; i < ySize; i++)
    {
        for (int j = 0; j < xSize; j++)
        {
            var insFloor = Instantiate(floor, new Vector3((j * sizeWall) + middle, -middle, i * sizeWall), Quaternion.Euler(90, 0, 0)) as GameObject;

            insFloor.transform.parent = Floor.transform; //group all the Quad to keep everything in order
            indexCell = (i * xSize) + j; //I transform the array into an array
            insFloor.transform.name = indexCell.ToString();
        }
    }
    Floor.transform.parent = allParent.transform;
}
```

La variabile “middle” serve per allineare il pavimento in base alla grandezza dei muri scelto dempre dall'utente

Una volta creati gli elementi principali bisogna definire il termine “cella”. La cella è una classe che rappresenta ogni quadrato della matrice di muri. Questa classe comprende :

- muro EST
- muro OVEST
- muro NORD
- muro SUD
- pavimento
- booleana se è stata visitata

```
[SerializeField]
12 riferimenti
public class Cell
{
    public bool isVisited;
    public GameObject north;
    public GameObject south;
    public GameObject east;
    public GameObject west;
    public GameObject floor;
    public int current; //Debug
    public Vector3 cellPosition; //Debug


}
```

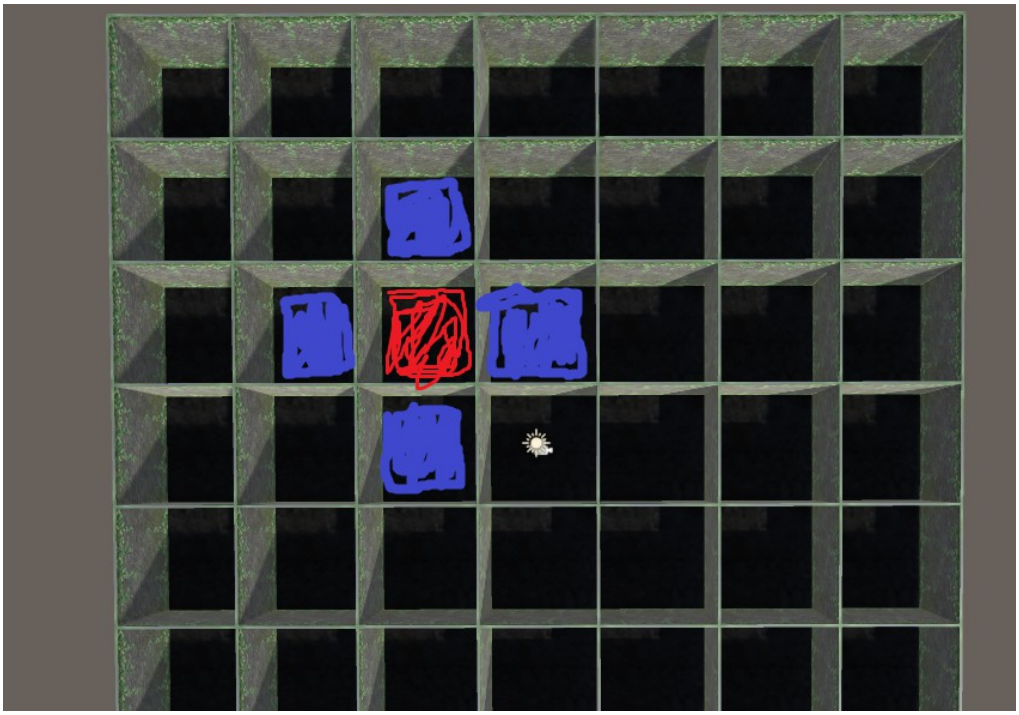
Una volta stabilita la classe bisogna creare effettivamente la cella.

```
void createCell()
{
    walls = new GameObject[Maze.transform.childCount];
    floors = new GameObject[xSize * ySize];


    int south = (xSize + 1) * ySize;
    int north = ((xSize + 1) * ySize) + xSize;

    for (int i = 0; i < Maze.transform.childCount; i++)
    {
        walls[i] = Maze.transform.GetChild(i).gameObject;
    }


    for (int i = 0; i < xSize * ySize; i++)
    {
        floors[i] = Floor.transform.GetChild(i).gameObject;
    }

    for (int i = 0; i < xSize * ySize; i++)
    {

        cells.Add(new Cell());
        currentRow = (i / xSize); //find current row for every cell


        cells[i].south = walls[i + south];
        cells[i].north = walls[i + north];
        cells[i].east = walls[(currentRow + i) + 1];
        cells[i].west = walls[currentRow + i];
        cells[i].floor = floors[i];
        cells[i].current = i;


    }


    mazeCreation(cells);



}
```

Con questo metodo abbiamo creato la cella. Ora ogni cella ha i propri muri, pavimenti ecc... Ogni cella ha dei vicini. Il seguente metodo mostra come trovare i vicini di ogni cella considerando anche le celle ai margini del labirinto.

```
void AddNears(Cell currentCell)
{


    int currentRow = (currentCell.current / xSize) + 1;
    int eastLimited = currentCell.current % xSize;
    int westLimited = eastLimited;
    int southLimited = xSize - 1;
    int northLimited = currentCell.current / xSize;



    //East
    if (eastLimited != xSize - 1)
    {
        if (cells[currentCell.current + 1].isVisited == false)
            neighboringCell.Add(cells[currentCell.current + 1]);

    }
    //West
    if (westLimited != 0)
    {
        if (cells[currentCell.current - 1].isVisited == false)
            neighboringCell.Add(cells[currentCell.current - 1]);

    }
    //South
    if (currentCell.current > southLimited)
    {
        if (cells[currentCell.current - xSize].isVisited == false)
            neighboringCell.Add(cells[currentCell.current - xSize]);

    }
    //North
    if (northLimited != ySize - 1)
    {
        if (cells[currentCell.current + xSize].isVisited == false)
            neighboringCell.Add(cells[currentCell.current + xSize]);

    }
}
```

Ora tutte le celle hanno i propri vicini.
ESEMPIO: cella rossa = cella corrente. Celle blu = vicini della cella rossa



il passo successivo è quello di creare il metodo per distrugere i muri tra la cella corrente e il suo vicino scelto a random

```
void breakWall(Cell curr, Cell choose)
{
    //East
    if (choose.current == curr.current + 1)
    {
        Destroy(curr.east);
        currentCell = cells[currentCell.current + 1];


    }

    //West
    if (choose.current == curr.current - 1)
    {
        Destroy(curr.west);
        currentCell = cells[currentCell.current - 1];


    }
    //South

    if (choose.current == curr.current - xSize)
    {
        Destroy(curr.south);
        currentCell = cells[currentCell.current - xSize];



    }
    //North
    if (choose.current == curr.current + xSize)
    {
        Destroy(curr.north);
        currentCell = cells[currentCell.current + xSize];

    }



}
```

Una volta creati tutti i metodi essenziali, si scrive l'algoritmo

```
void mazeCreation(List<Cell> allCells)
{

    int visitedCell = 0;


    int randomNeighboring;
    int random = Random.Range(0, allCells.Count); //I choose a cell at random
    currentCell = allCells[random]; //the choice cell becomes the current cell



    while (visitedCell < allCells.Count) //I run the cycle for all the cells of the matrix
    {
        AddNears(currentCell); //add to the selection cell neighboring cells
         randomNeighboring = Random.Range(0, neighboringCell.Count); //I choose a neighboring cell at random



        if (neighboringCell.Any()) //if there are cells neighboring the current cell
        {
            lastCellVisited.Add(currentCell); //becomes visited

            breakWall(currentCell, neighboringCell[randomNeighboring]); //Break the wall

            neighboringCell.Clear(); //Reset the list of neighboring cells
            visitedCell++;
            currentCell.isVisited = true;


        }

        else
        {
            //else I choose a already been visited cell
            random = Random.Range(0, lastCellVisited.Count);
            currentCell = lastCellVisited[random];

        }

    }

    isGenerate = true;

}
```

Risultato finale di una matrice 20X20