Perfect Maze Algorithm

Code documentation

Afbeelding met tekst

Automatisch gegenereerde beschrijving

Made by Patrick Eikema  
26/06/2022

Inhoud

[Algorithm 3](#_Toc107507637)

[User Interface 9](#_Toc107507638)

[Camera 10](#_Toc107507639)

[Hulp 11](#_Toc107507640)

To use the algorithm, we need a tilemap. Tilemaps have a chunk functionality, which batches each Sprite from the Tilemap into grouped chunks to be rendered. This improves performance a lot.

# Algorithm

To start we need a Cell class. The class has a constructor with 2 parameters. This is to store the row and column number given when the grid is set up. These will be later used to find neighbor cells. This class also has an array of 4 bools for each wall and a visited bool that starts on false. The wall bools are set on false when a neighbor cell visits it. Which wall is set on false is determined by the direction of the neighbor that visits it. The visited bool will be set on true when the walls are decided.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

Now we need a class where the maze logics will be placed. This class is called MazeGenerator.

First we need a list of cells.



The cells need to be placed in a grid. A nested for loop is used to traverse through the rows and columns. A Cell object is made for each cell in the grid. The row and column number is given to it in the parameters.

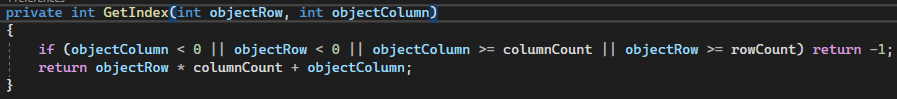
Afbeelding met tekst

Automatisch gegenereerde beschrijving

Now to use the algorithm, we need a current Cell, a next Cell and a stack. The algorithm will look for an unvisited neighbor of the current cell and set it as next. It pushes the current cell to the stack and continues. When no neighbor is found, the current cell is set to the last cell placed on the stack and removes it from the stack. This continues until an unvisited neighbor cell is found again.



Now we need a way to find the index of a cell by using the row and columns of that cell. This is done by a GetIndex function. The function first checks if the row and column given is in bounds of the grid. If it is, it returns -1 to tell. If it is not, it will multiply the given row by the amount of columns in the grid. Now we have the index of the first cell in that row. Now if we add the given column to that number. It returns the index of the cell with that specific row and column.



Now we need a way to find neighbor cells. The function CheckNextNeighbor helps us do that. We need access to the current cell, so it’s given as a parameter. The function has a list of Cells to store the neighbor cells.



To find the indexes of the neighbor cells, we call on the GetIndex function and give the column and rownumber we want to look for.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

Now we need to check if that neighbor is visited and if it is not, we need to add it to the list.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

We want the function to return a random neighbor, but sometimes no unvisited neighbor is found. If none is found, the function returns null. The Random class helps you return random number between two numbers. The number is included and the last is excluded. Afbeelding met tekst

Automatisch gegenereerde beschrijving

Now we have a way to find neighbor cells. The last function needed for the algorithm decides what walls need to be set on false. Two parameters are given. The current cell and the next cell. We need a way to find what direction the next cell is from the current cell. This is done by subtracting both column and row numbers from eachother. Since the walls array is set clockwards, the top wall means walls[0], the right wall means walls[1] and so on.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

Let’s make a function of the way we place the cells in a grid. The list is cleared for re-use. The current cell is set to the first one in the list, to have a way to start.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

Now we can implement the algorithm. The current cell (first one) is set to current. Visited is set to true, so that the algorithm cant traverse back to it. We need a next cell, so we set next to the return of the CheckNextNeighbor(current) function. While a next cell (neighbor) is found, we set next.visited to true. The current cell is pushed to the stack for traversing. The walls need to be set to false of both the current and the next cell, determined by the direction. We set the current cell to the next cell, to continue. Now we check for the next neighbor again. If no unvisited neighbor is found, the while loop will stop, because next will be null.

When the while loop stops, it means that no unvisited neighbor is found. The if statement checks to if it is not the last cell. If it is not, the current is set to the last Cell from the stack and also removed from this stack. Now the algorithm will start over and over until it is the last cell.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

The algorithm now works when placed in a loop. But no tiles are placed yet. We need a way to decide what tiles to place based on the Boolean array named walls. We need to make use of UnityEngine.Tilemaps



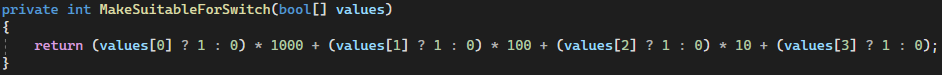
In the class we need access to the tiles and tilemap. This is given in the editor.



Afbeelding met tekst

Automatisch gegenereerde beschrijving

We need a switch for each possible tile, but switches don’t accept arrays. So we need a function to set the bools to binary (integer). If the a value in the array is true, it returns a 1. If it’s false it returns 0. The first value in the array is multiplied by 1000, because we have 4 values. The second is multiplied by 100 and so on. The returns are placed next to eachother. A possible outcome can be 0100. This means the Boolean array was [false,true,false,false]



We need one last function to visualize the maze. This function has a big switch. The returns of the MakeSuitableForSwitch function with the Boolean array of the cell are given as a parameter. Depending on what walls are missing, a tile is placed on the location of the cell’s column and rownumber.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

All functions are now in place, so we need a way to couple them together. The while loop is used so that the maze is generated in one loop and not depending on the update function. It breaks when the algorithm find the last cell. All tiles are placed when the algorithm is finished and decided on all walls.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

# User Interface

The UI class has two functions. One for the generate button and one to change the numbers for the column- and row- slider. It uses objects given in the editor.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

Afbeelding met tekst

Automatisch gegenereerde beschrijving

Two events are made to give to the object listeners.Afbeelding met tekst

Automatisch gegenereerde beschrijving

# Camera

The camera needs to scale to the maze. It also needs to be positioned to the center of the maze. The CameraBehavior class has one function. Since the tilemap starts on x = 0 and y = 0, the position can be halve of the column- and row- amount.

The orthographic size is the camera’s half-size. Since

Afbeelding met tekst, schermafbeelding, scherm, zwart

Automatisch gegenereerde beschrijving

# Hulp

Van de volgende tutorials heb ik gebruik gemaakt:

<https://github.com/mullaney/maze-generator/blob/792c6c33a7b75edbf788e213dcf961ed6b3808d6/sketch.js#L7>

<https://www.youtube.com/watch?v=HyK_Q5rrcr4&ab_channel=TheCodingTrain>