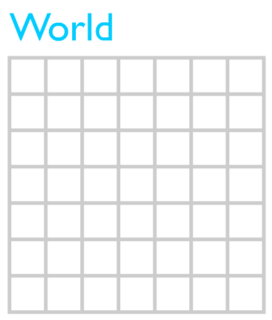
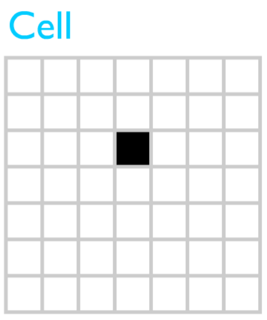
John Conway's Game of Life (recreated in Unity)

Background

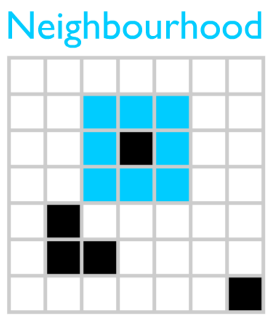
Game of life is one of the most notable examples of cellular automata**. Cellular automata** can appear in variety of different forms, the basic idea – is that - there is a **world** (grid), and each **cell** (agent) of the grid can change **state-** for example, it can be dead or alive. Each cell changes its state based on its relationship with **neighbouring** cells.



World is the grid itself



Cell is a single agent

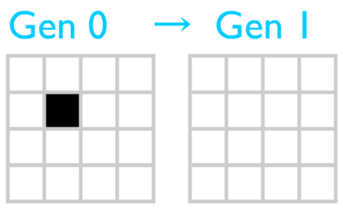


Each cell has 8 neighbours

In most models, each cell has a basic set of **rules**. The rules are used to determine what happens to the cell. Once each cell has calculated result based on those rules, all of the cells are updated all at once per time step (**tick**), producing interesting dynamic results. The initial pattern is known as the **seed** of the system.

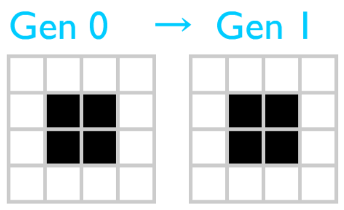
All rules are based on the interaction with 8 neighbouring cells. These are the rules of Conway's Game of Life.

1) Any **live** cell with fewer than 2 live neighbours dies (as if by under population).



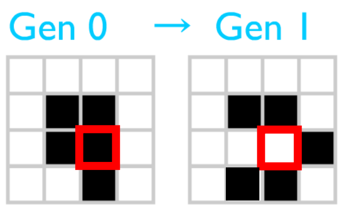
In this example you can see that, the cell did not have any neighbouring cells and it died.

2) Any **live** cell with 2 or 3 neighbours, lives onto the next generation.



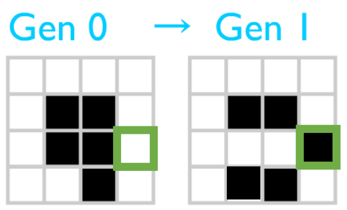
Here you can see that each cell has exactly 3 neighbours, hence they keep each other alive.

3) Any **live** cell with more than 3 neighbours dies (as if by overpopulation).



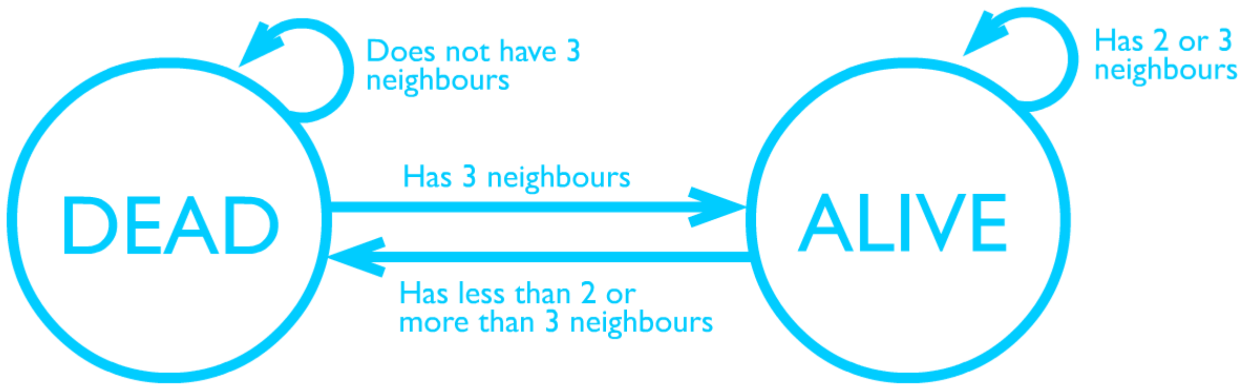
In this example, the highlighted cell has 4 neighbours, hence it dies.

4) Any **dead** cell that has exactly 3 neighbours becomes alive (as if by reproduction).

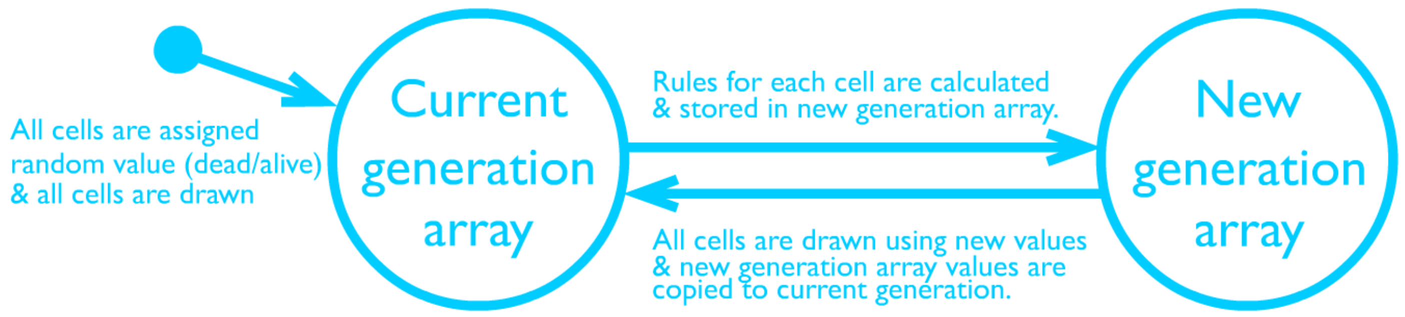


In this example, the dead cell becomes alive as it has exactly 3 neighbours.

The behaviour of each cell can also be explained using **FSM (Finite- State Machines)**.
The cell can be in one of the two states: **Dead** or **Alive**. It can change its state if condition is true.
For example, if the cell is dead and it has 3 neighbours, then in the next generation it will becomes alive.


Does not have 3 neighbours
Has 2 or 3 neighbours
DEAD
Has 3 neighbours
ALIVE
Has less than 2 or more than 3 neighbours

Underneath you can see another state machine describing how 2 arrays are used to store and draw each consecutive generation. This is how it works – current generation array will store states (dead or alive) of each cell, then we will apply rules to each of the cells and store the results in the new generation array. Once all results are calculated, new generation array can be used to draw the next generation. Once the generation is drawn, we can copy all the values from the new generation array and store them in the current generation, and this cycle repeats.


All cells are assigned random value (dead/alive) & all cells are drawn
Current generation array
Rules for each cell are calculated & stored in new generation array.
New generation array
All cells are drawn using new values & new generation array values are copied to current generation.

Effectively, we're juggling 2 arrays: (Firstly Generation 0 is drawn using first array) we use the second array to store new results, once those results are calculated and drawn (generation 1 is displayed), we copy all values from the second array into the first one. The second array can now be used to calculate and draw results for the next generation (generation 2). We repeat the exact same process over and over again to create new generations.

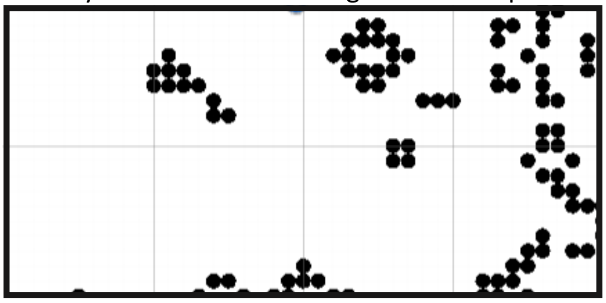
It is important to understand that, all changes (death/ birth) happen all at once – when the new generation is drawn. The discrete moment at which it all happens is called **tick**. Traditionally, this model is implemented using time intervals as ticks, for instance a new generation can be displayed every second. However for debugging purposes you can use a key press to update the next generation.

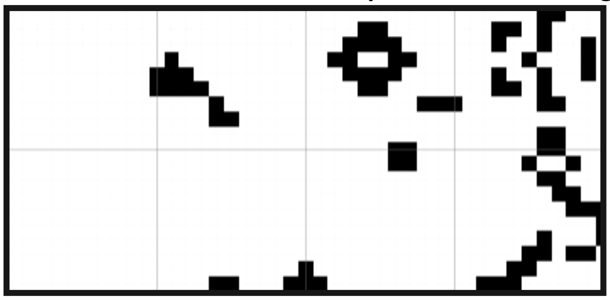
Finally, Game of Life is a **zero-player game** or **no-player game** meaning that once the initial pattern or seed is set the game will play out without any interaction. This game is also an excellent example of AI (Artificial Intelligence) where fixed rules create **emergent behaviour**. The system does no rely on each individual cell (agent) but rather on their interaction between each other. This system produces a variety of unpredictable behaviours, there is no way of telling whether a particular pattern will go onto thousands of generation or halt immediately.

Game of life can be made using different configurations – we will be using classic: **2333.**
This can be decoded – if a cell is alive and it has less than **2** neighbours or more than **3** it will die, and if it's dead and it has no more than **3** and no less than **3** neighbours then it will become alive. Hence **2-3-3-3**.
Reference: https://zhan.itch.io/gol-mixer

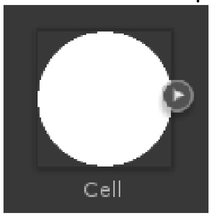
Implementation

Open a new Unity project (2D), I'll be using 2 by 3 layout.
First we will need to create a cell sprite. Traditional models use a square, but I prefer to start with a circle as it has negative space which helps to locate each individual cell. (The shape can be changed at any time)
Below you can see the same generation represented in **circular cells on the left** and **square cells on the right**.

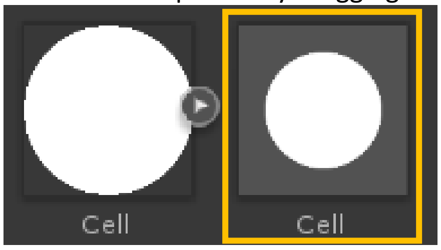




Create a Circle sprite in the project (Create/Sprite/Circle). Name it Cell.

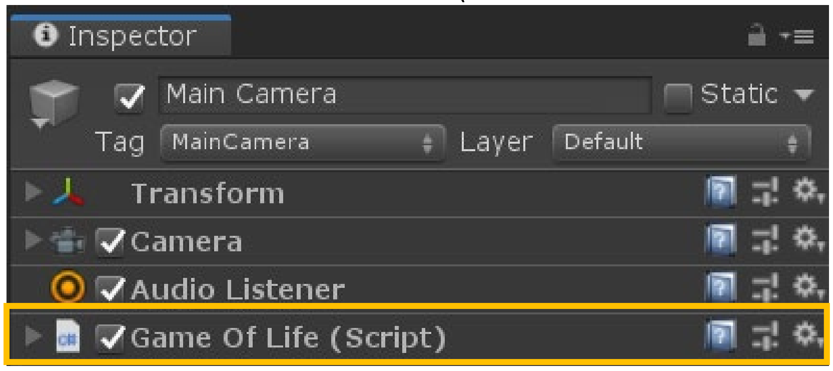


Make it into a prefab by dragging it into the scene and then back into the project. (Delete Cell prefab from the scene)

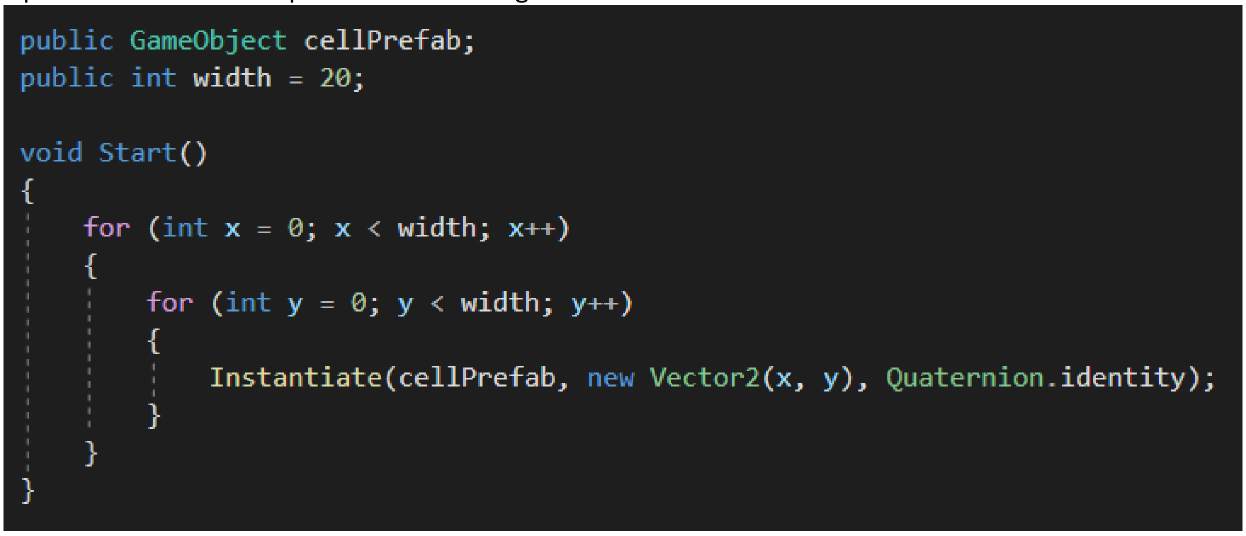


Create a new script call it "**GameOfLife**", there are different ways of structuring classes when creating Game of Life, however we will be writing the entire behaviour in a single class for simplicity sake.

Attach this class to the Main Camera. (All code will be executed from the main camera)



Open “GameOfLife” script and the following.

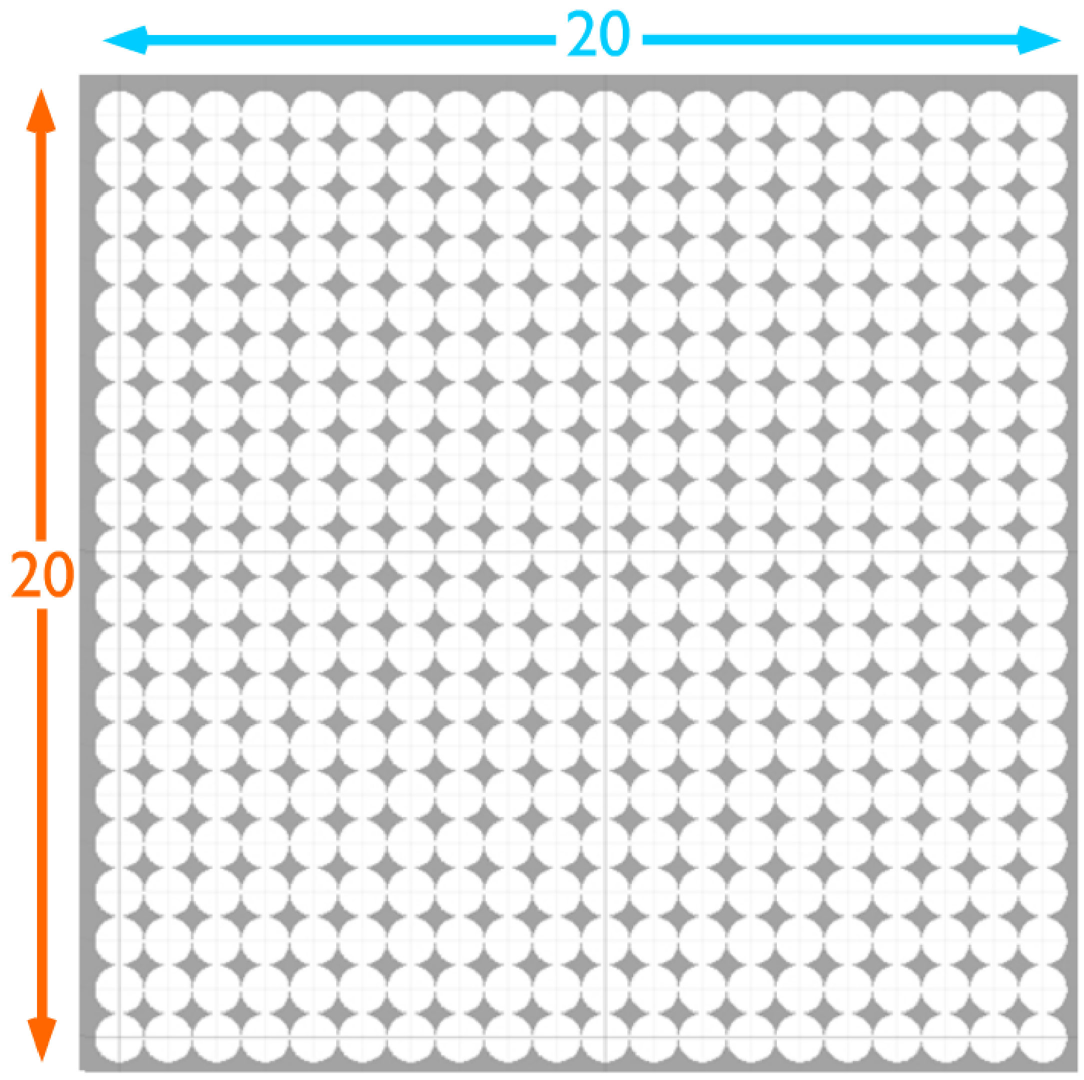


```
public GameObject cellPrefab;
public int width = 20;

void Start()
{
    for (int x = 0; x < width; x++)
    {
        for (int y = 0; y < width; y++)
        {
            Instantiate(cellPrefab, new Vector2(x, y), Quaternion.identity);
        }
    }
}
```

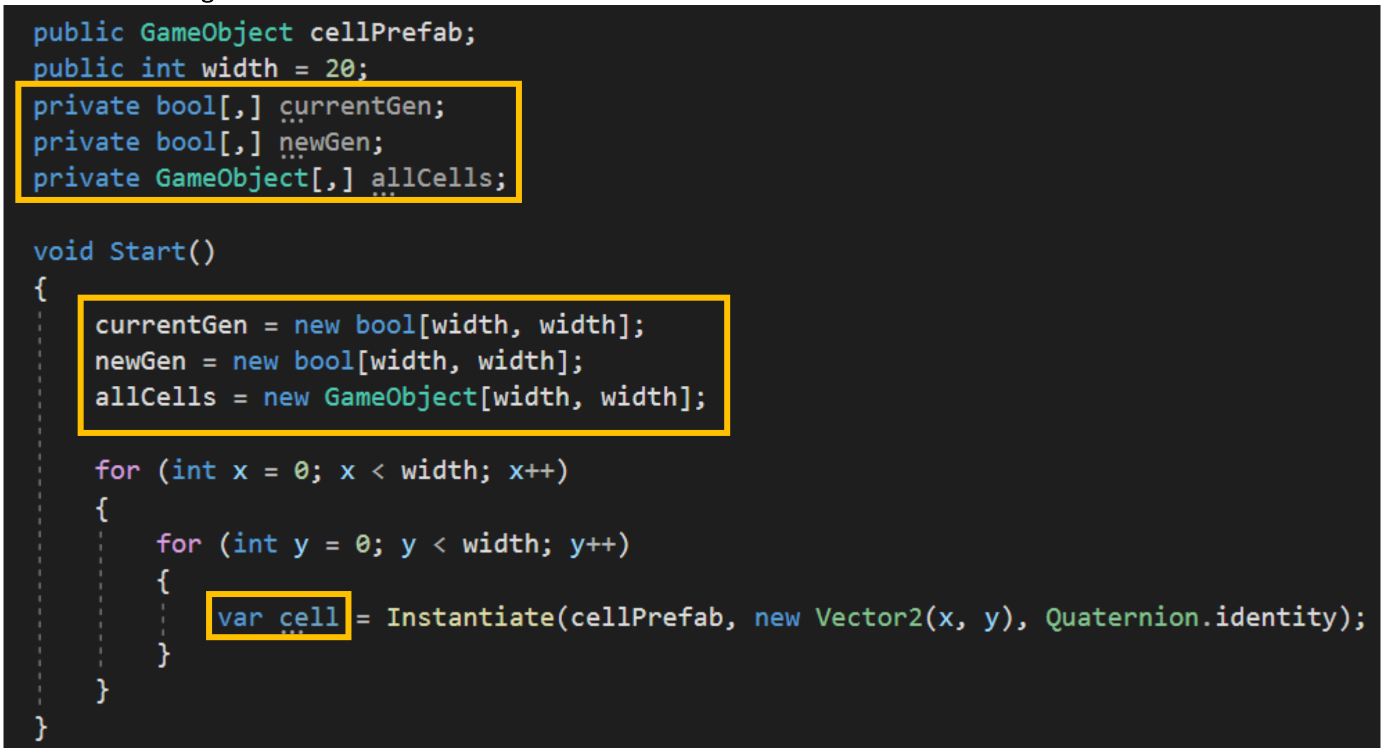
cellPrefab – is a circle prefab that is used to represent a single cell.
width – will serve as width and length of the grid, hence we will be generating a square grid.
Set default value to **20**, hence our original grid will have 400 cells (20x20).

In the Start function I’m using, 2 loops to generate a grid. Notice how x and y values are used for positions.

If you save and run your project you should generate a grid that looks like this.



Add the following code:

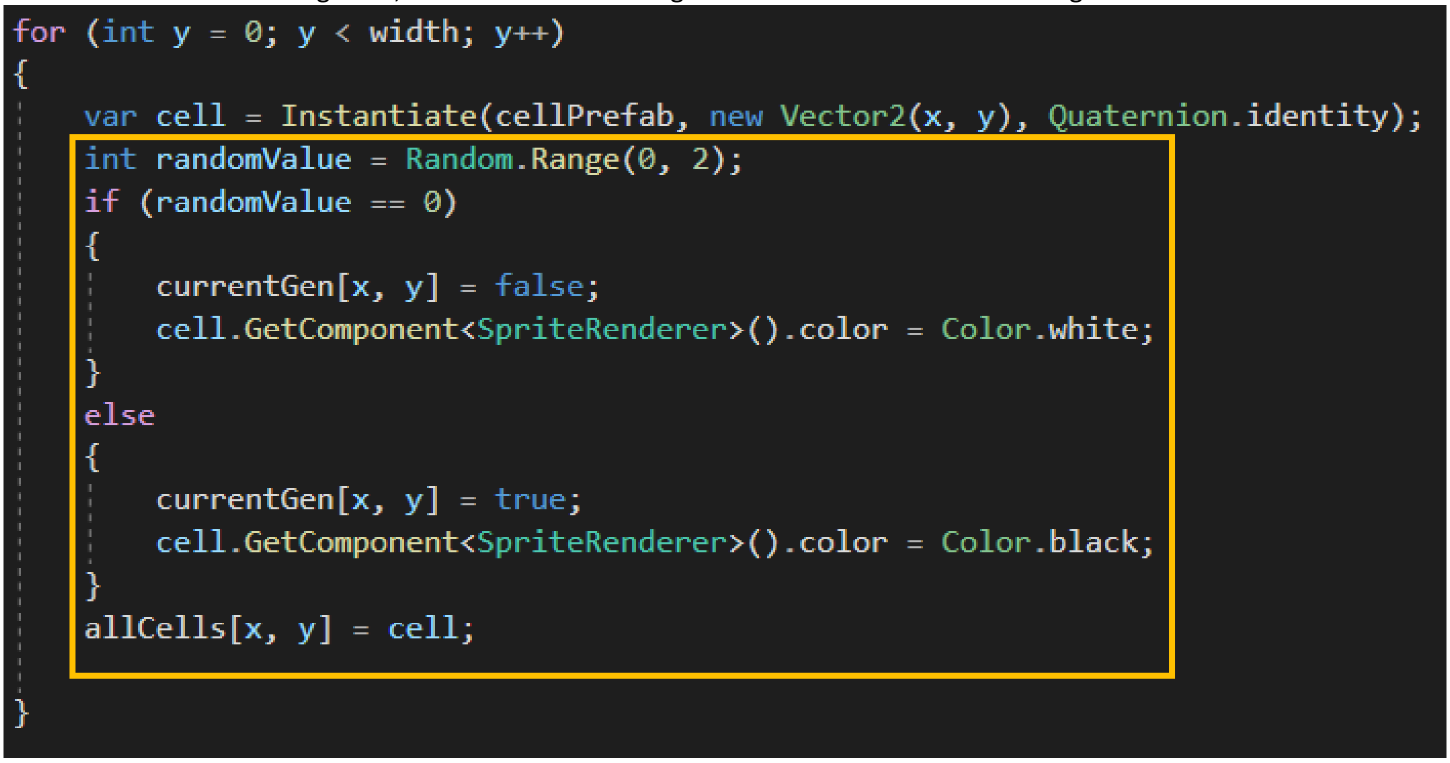


```
public GameObject cellPrefab;
public int width = 20;
private bool[,] currentGen;
private bool[,] newGen;
private GameObject[,] allCells;

void Start()
{
    currentGen = new bool[width, width];
    newGen = new bool[width, width];
    allCells = new GameObject[width, width];

    for (int x = 0; x < width; x++)
    {
        for (int y = 0; y < width; y++)
        {
            var cell = Instantiate(cellPrefab, new Vector2(x, y), Quaternion.identity);
        }
    }
}
```

currentGen and **newGen** are multidimensional Boolean arrays. They will be used to store values about cells. **currentGen** will be used to store current data about cells and **newGen** array will be used to store new data. **allCells** GameObject array will be used for storing/drawing sprites.

In the **Start** function we populate all arrays with **width** by **width** variable – in this case all arrays store 20 x 20 values.

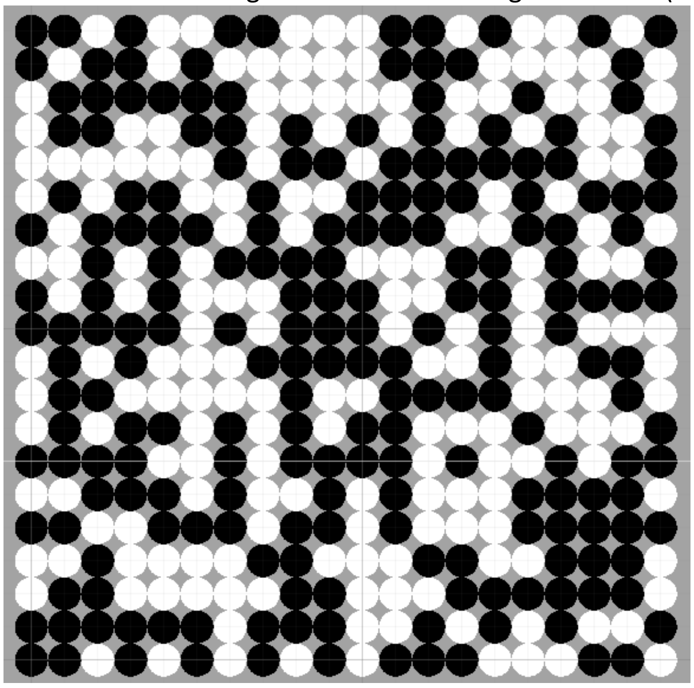
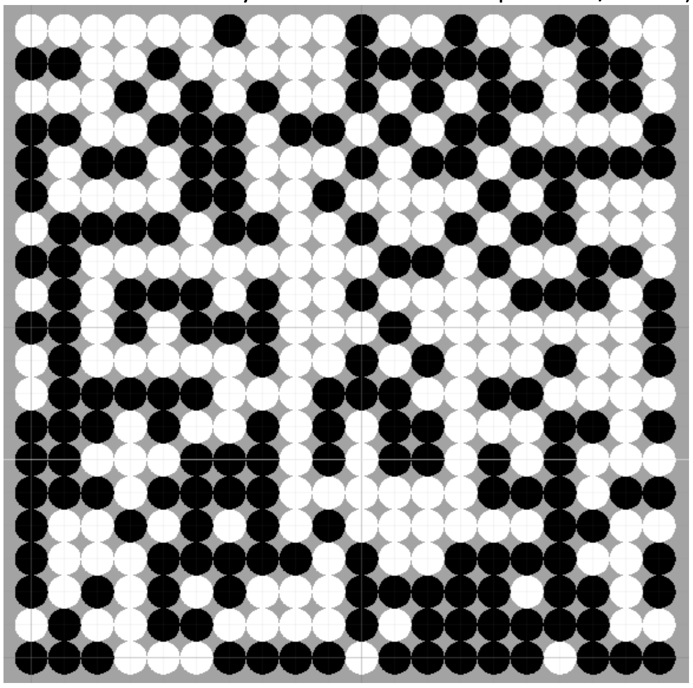
Now let's add the following code, which will allow us to generate random initial seed using random function.



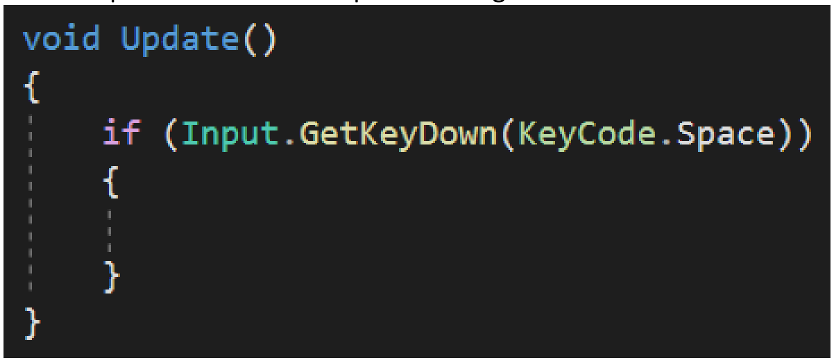
```
for (int y = 0; y < width; y++)
{
    var cell = Instantiate(cellPrefab, new Vector2(x, y), Quaternion.identity);
    int randomValue = Random.Range(0, 2);
    if (randomValue == 0)
    {
        currentGen[x, y] = false;
        cell.GetComponent<SpriteRenderer>().color = Color.white;
    }
    else
    {
        currentGen[x, y] = true;
        cell.GetComponent<SpriteRenderer>().color = Color.black;
    }
    allCells[x, y] = cell;

}
```

Firstly, for each cell we generate a random value between 0 and 1. If the value is 0 then that cell is in the current gen is set to false (dead) and the colour of that cell is white, otherwise the cell is set to true (alive) and the colour of that set to black. Finally, cell is added to the allCells array.
Save and run your project.

You should see something that resembled a game of Reversi/Othello. This is the initial **seed** / **generation 0**, the initial seed will be used to generate consecutive generations. (Run it a few times and you will see different patterns/ seeds)

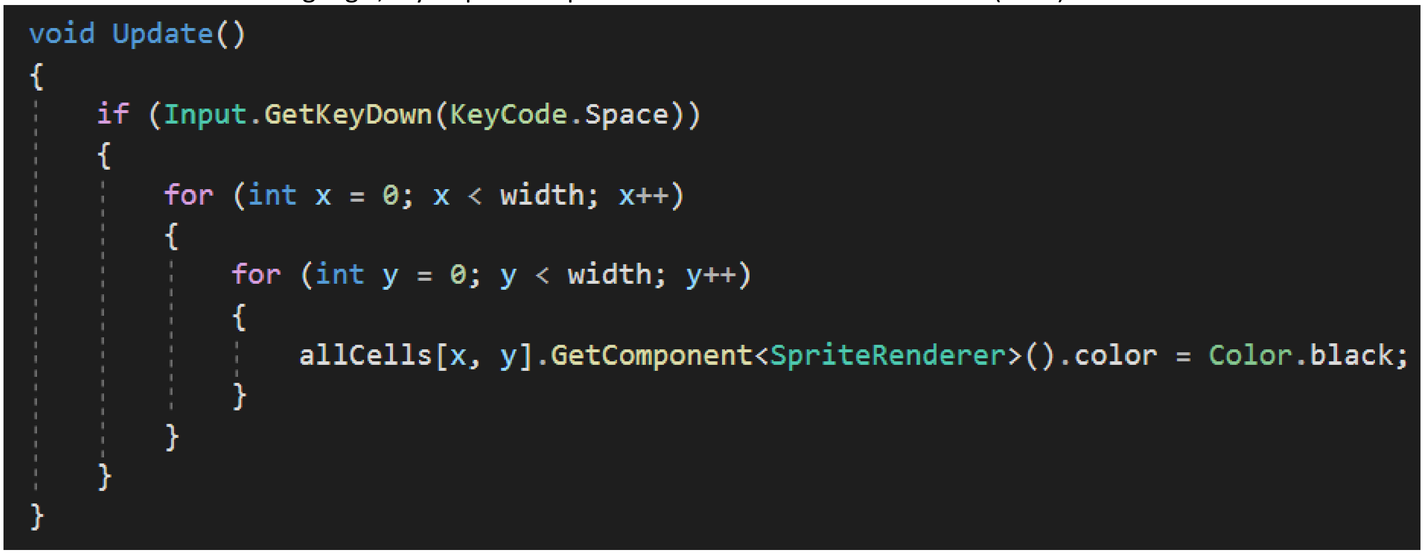



Below in the Update function add the following code. For now we will use space key to iterate through generation at our own pace. Later we will update each generation based on time intervals.



```
void Update()
{
    if (Input.GetKeyDown(KeyCode.Space))
    {

    }
}
```

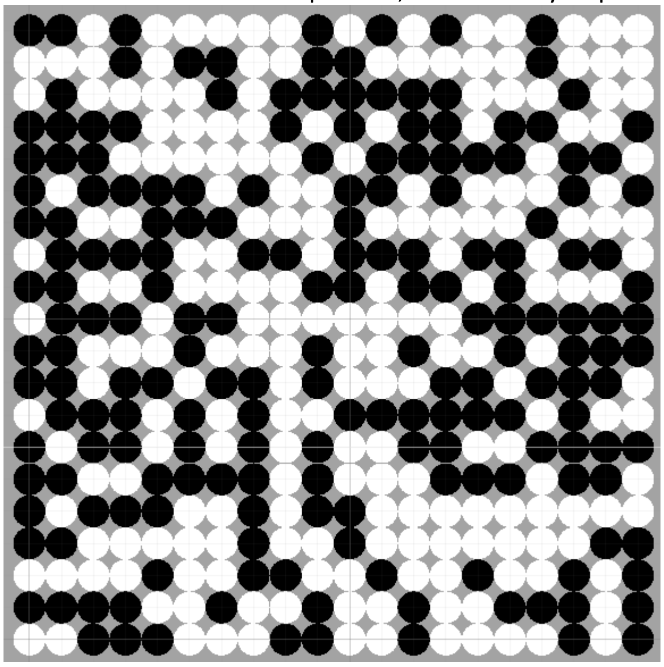
For now add the following logic, if you press a spacebar each cell should turn black (alive).

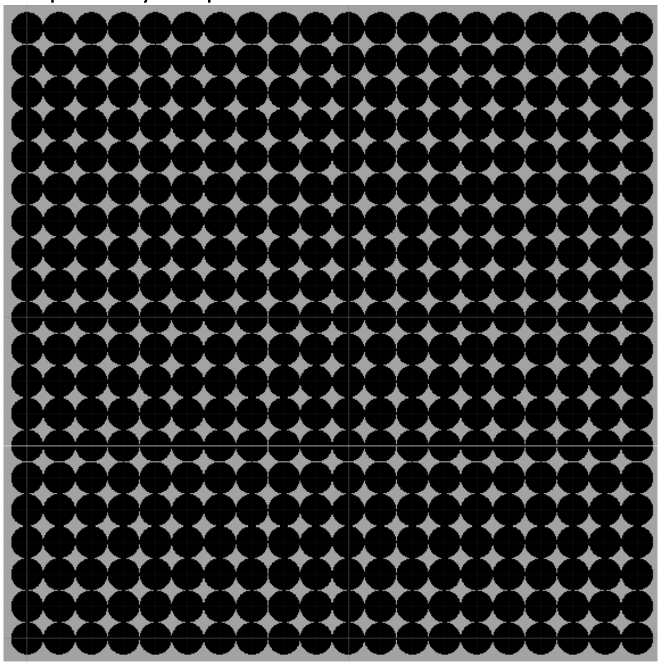


```
void Update()
{
    if (Input.GetKeyDown(KeyCode.Space))
    {
        for (int x = 0; x < width; x++)
        {
            for (int y = 0; y < width; y++)
            {
                allCells[x, y].GetComponent<SpriteRenderer>().color = Color.black;
            }
        }
    }
}
```

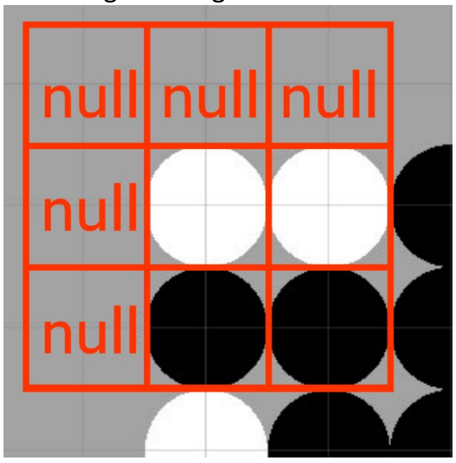
Save and run your project.

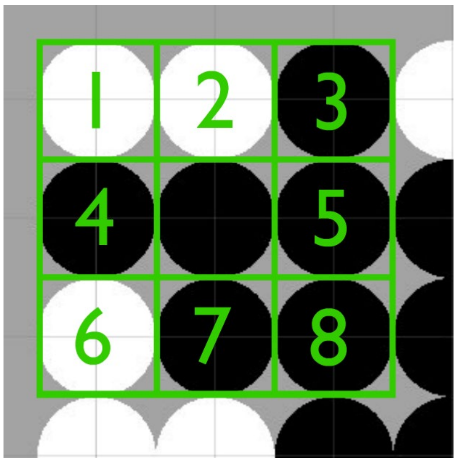
You should see the initial pattern, then when you press the space key all sprites should turn black as shown below.



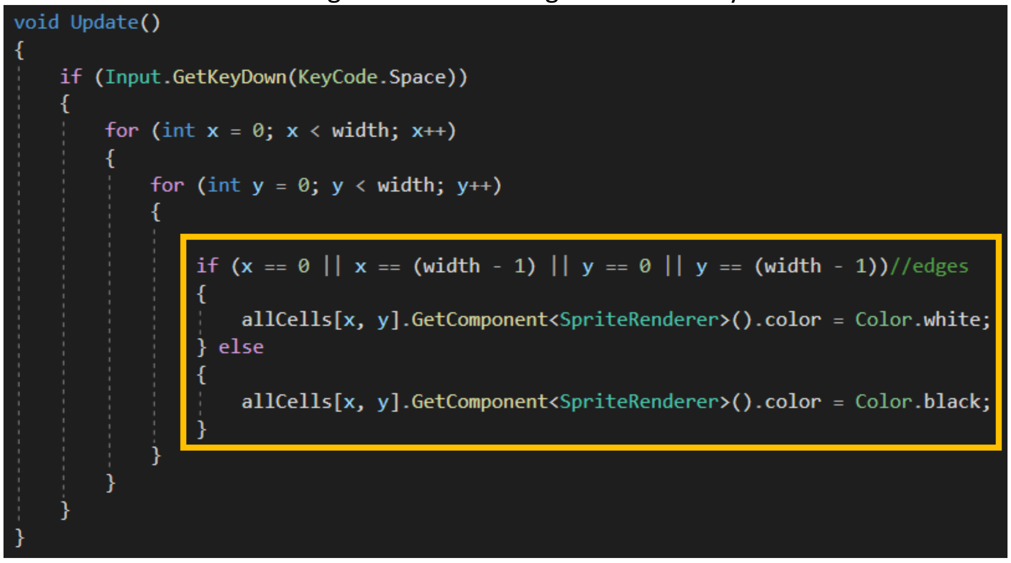


Just before we move on to the logic of cells, there is a little discrepancy we need to take into consideration.
Any cell on the border cannot calculate its neighbours as they do not exist. Only cells within the border can access their neighbouring cells.





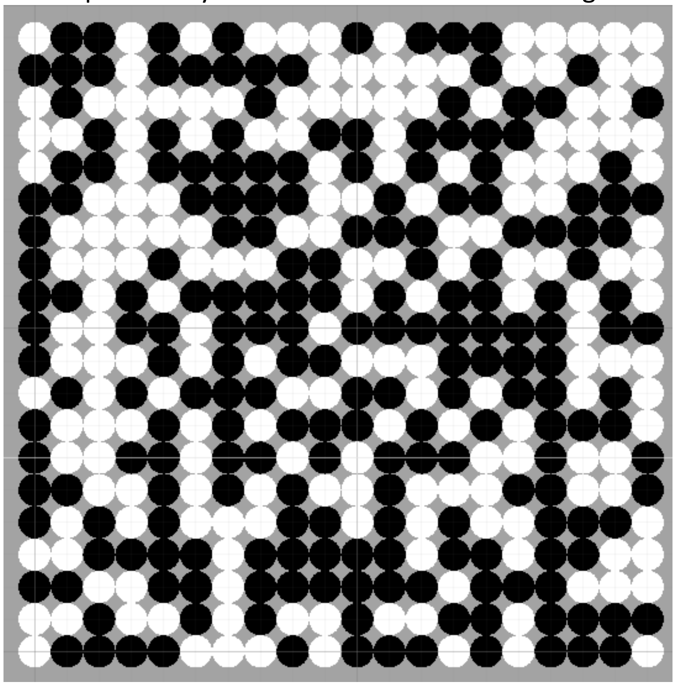
A short term solution is to ignore all of the edge cells and only work with the internal cells.

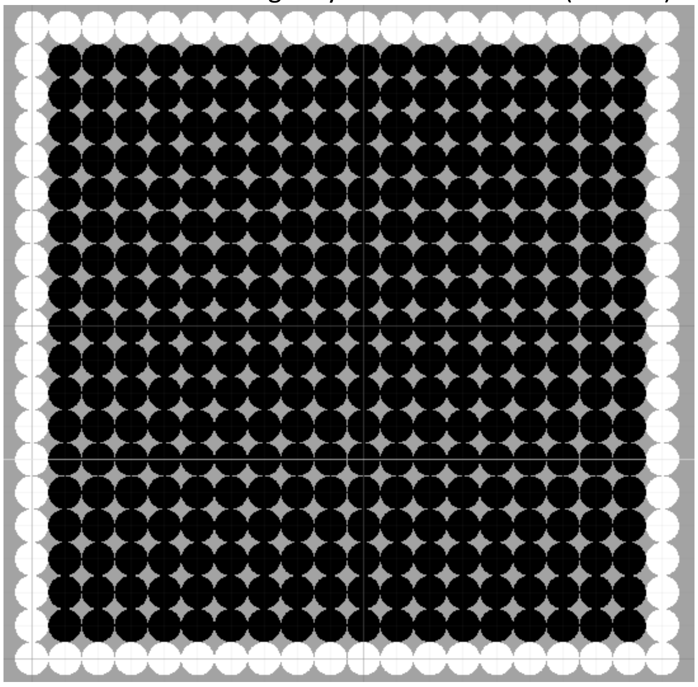


```
void Update()
{
    if (Input.GetKeyDown(KeyCode.Space))
    {
        for (int x = 0; x < width; x++)
        {
            for (int y = 0; y < width; y++)
            {
                if (x == 0 || x == (width - 1) || y == 0 || y == (width - 1))//edges
                {
                    allCells[x, y].GetComponent<SpriteRenderer>().color = Color.white;
                } else
                {
                    allCells[x, y].GetComponent<SpriteRenderer>().color = Color.black;
                }
            }
        }
    }
}
```

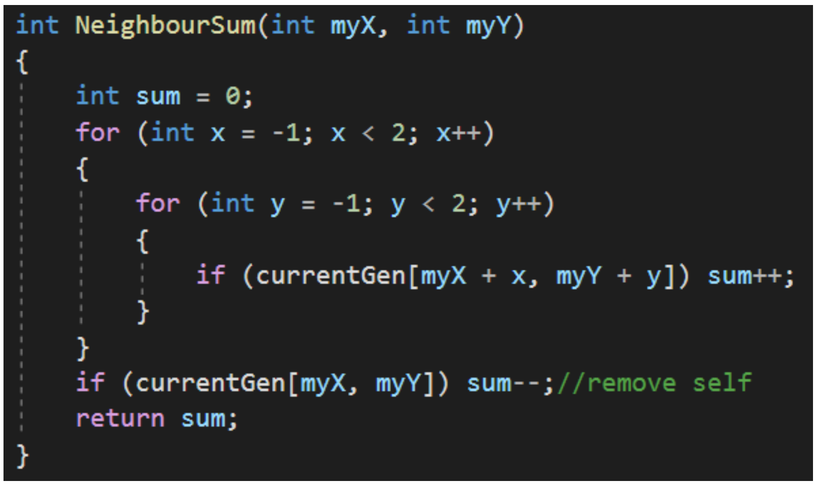
This code can be used to separate edges from the internal cells.
Save and run your project.

Press space and you should see the outline of edges. For now we will be working only with internal cells (in black).

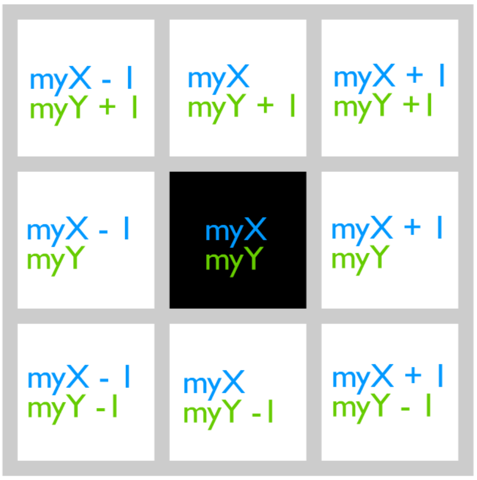




For the next part we are going to implement a function that will output a number of live neighbours surrounding a particular cell. Examine this function carefully. First of all, it starts with **int** and not with **void –** this means that this function can be used as an integer. In other words it returns an **integer** value, a **void** function does return anything, hence **void**. This function takes in 2 integer values **myX** and **myY**, neighbours will be located based on these values. Then we run 2 for loops locate all neighbours, and if their value is set to true then the sum will go up by 1 value. Please use image on the right to understand how the values are located.

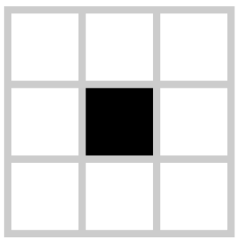


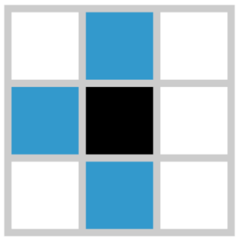
```
int NeighbourSum(int myX, int myY)
{
    int sum = 0;
    for (int x = -1; x < 2; x++)
    {
        for (int y = -1; y < 2; y++)
        {
            if (currentGen[myX + x, myY + y]) sum++;
        }
    }
    if (currentGen[myX, myY]) sum--;//remove self
    return sum;
}
```

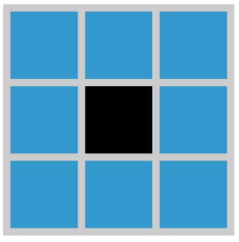


Finally, the cell checks if it is alive, and if so it removes 1 values from the sum. As calculation of neighbours does not include the cell inside. The last line of the function is **return sum;** meaning that it will output that value when the function is complete.

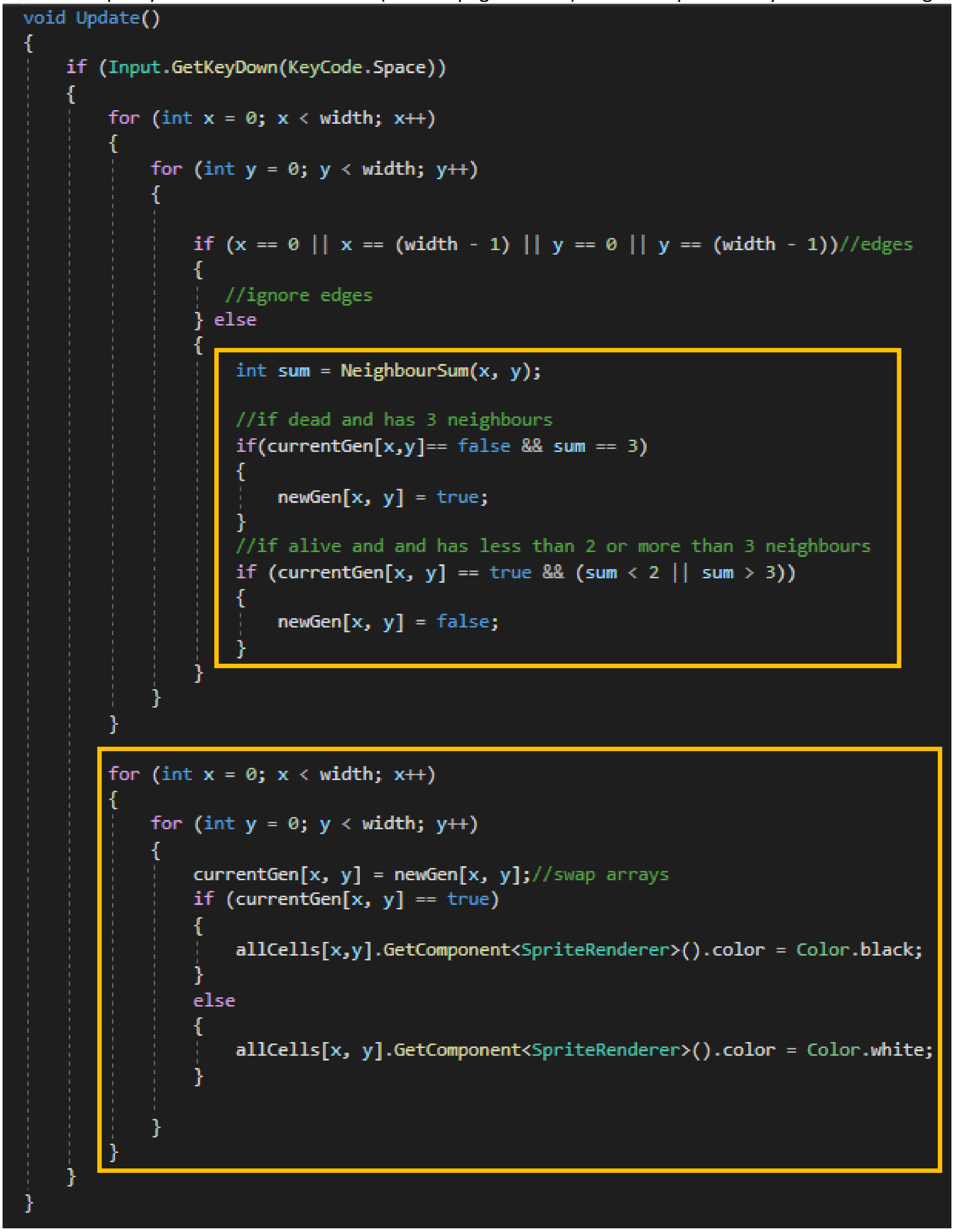
Use the image below for a reference how the neighbours are summed up.







For the last part you need to add the rules (refer to page 1 and 2) and then update arrays and draw new generation.

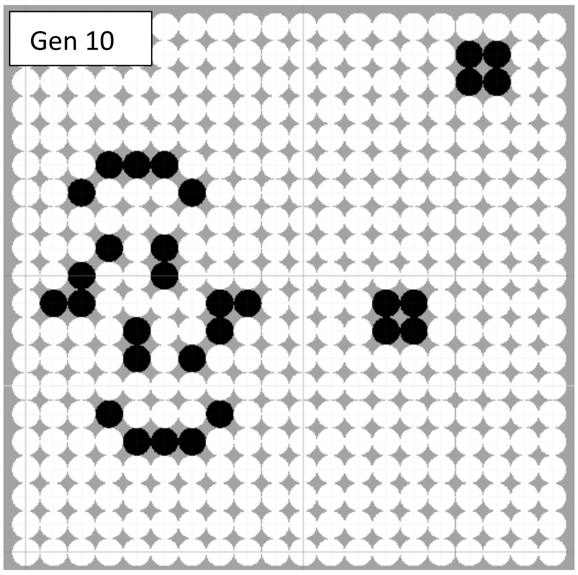


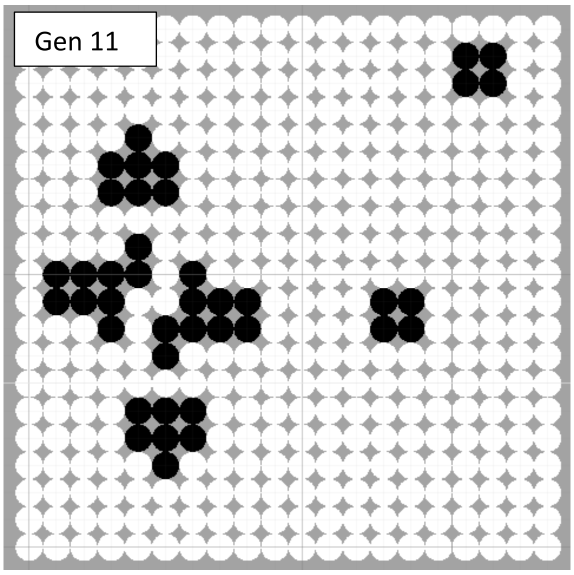
```
void Update()
{
    if (Input.GetKeyDown(KeyCode.Space))
    {
        for (int x = 0; x < width; x++)
        {
            for (int y = 0; y < width; y++)
            {

                if (x == 0 || x == (width - 1) || y == 0 || y == (width - 1))//edges
                {
                   //ignore edges
                } else
                {
                    int sum = NeighbourSum(x, y);

                    //if dead and has 3 neighbours
                    if(currentGen[x,y]== false && sum == 3)
                    {
                        newGen[x, y] = true;
                    }
                    //if alive and and has less than 2 or more than 3 neighbours
                    if (currentGen[x, y] == true && (sum < 2 || sum > 3))
                    {
                        newGen[x, y] = false;
                    }
                }
            }
        }

        for (int x = 0; x < width; x++)
        {
            for (int y = 0; y < width; y++)
            {
                currentGen[x, y] = newGen[x, y];//swap arrays
                if (currentGen[x, y] == true)
                {
                    allCells[x,y].GetComponent<SpriteRenderer>().color = Color.black;
                }
                else
                {
                    allCells[x, y].GetComponent<SpriteRenderer>().color = Color.white;
                }

            }
        }
    }
}
```

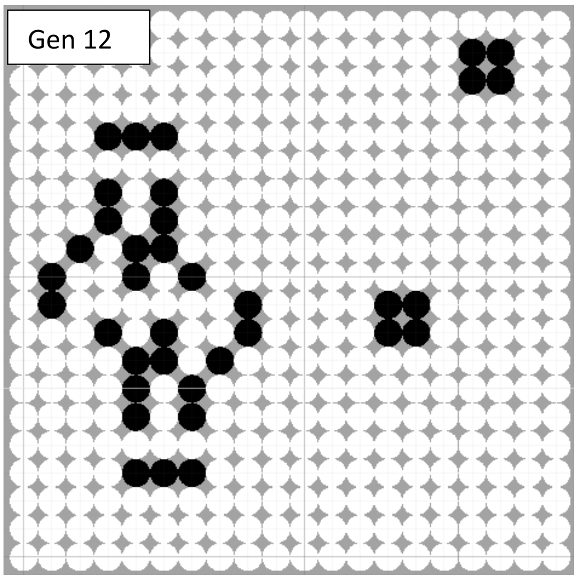
In the top code block, I'm storing a **NeighbourSum** output in a **sum** variable. And then I apply the rules of game of life, however they are condensed into 2 if statements. Read comments to better understand how it works.
In the bottom code block, I'm assigning each value of the **newGen** array into **currentGen** array, then I use those values to turn the cell dead(white) or alive(black).

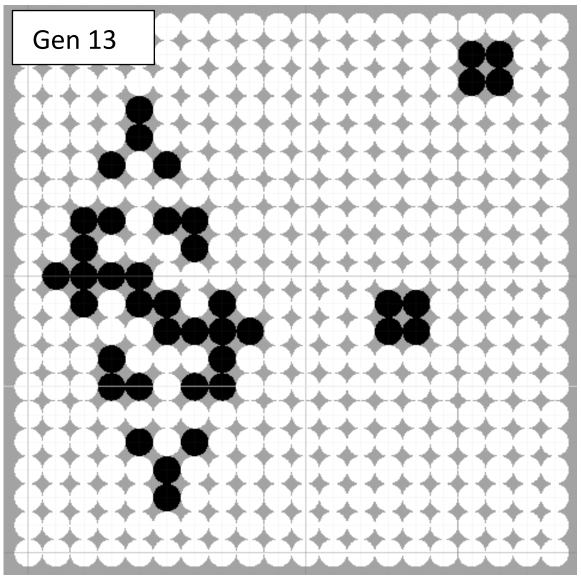
Save and Run your project, press space to go through generations. (Keep in mind some seeds may die very fast)

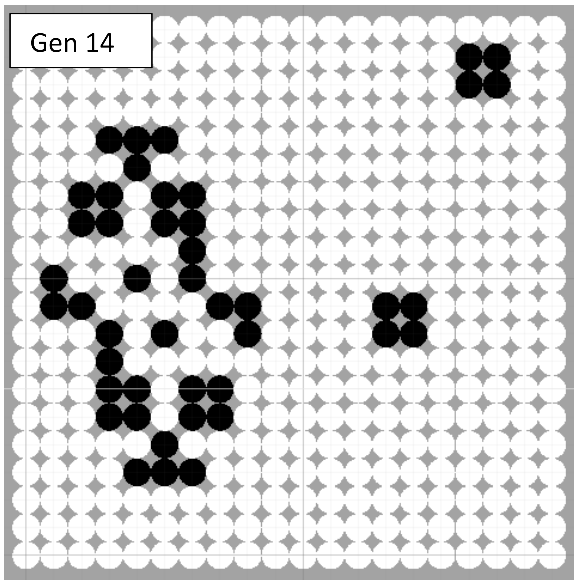
Here is an example of a seed stepping through the generations.

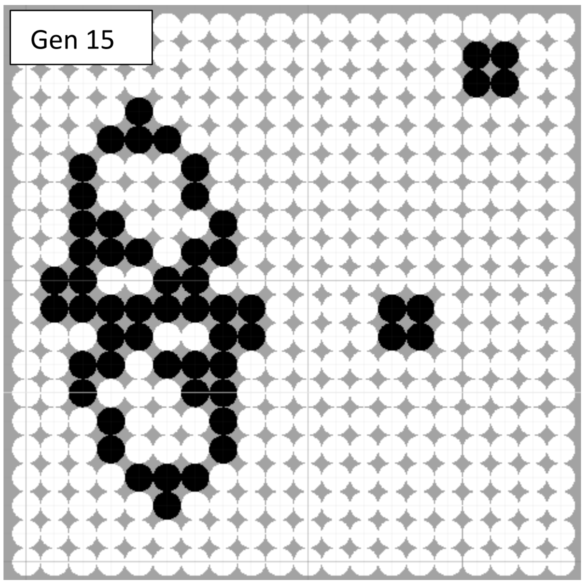




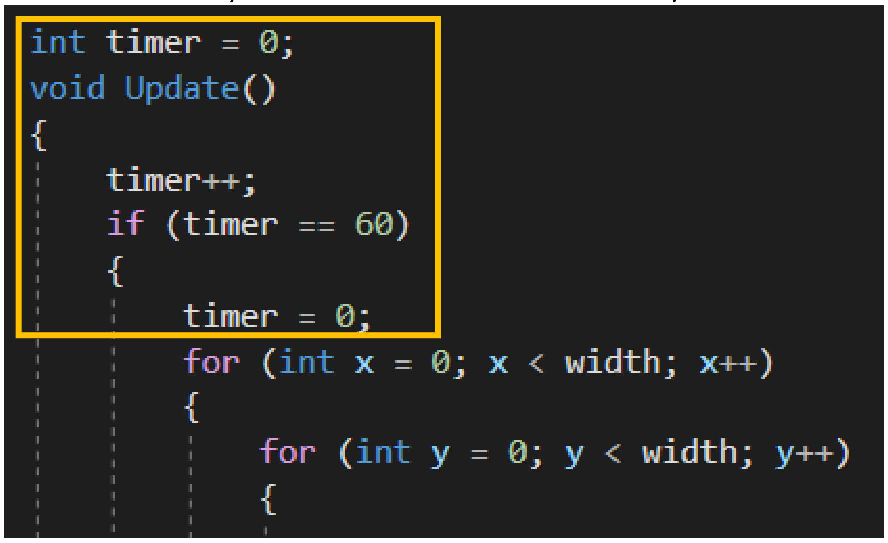








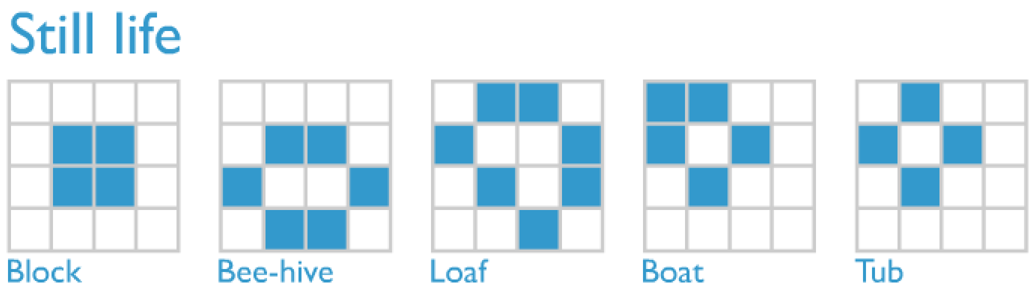
This is optional, you can change your code so that the new generation gets update per time interval, in this example 60 frames. This way the simulation will run automatically.

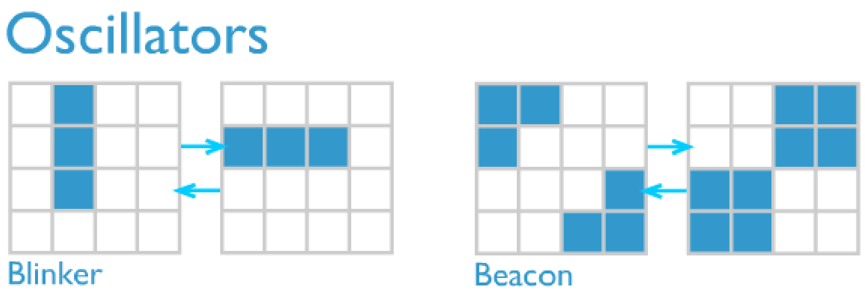


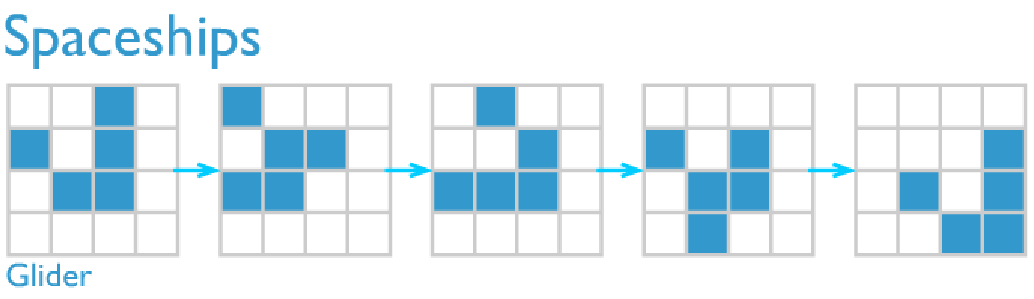
```
int timer = 0;
void Update()
{
    timer++;
    if (timer == 60)
    {
        timer = 0;
        for (int x = 0; x < width; x++)
        {
            for (int y = 0; y < width; y++)
            {
```

Finally, there is a definitive way to find out if your simulation is running correctly. Specific rules govern this simulation, hence there is a tendency for particular patterns to emerge.

Here is a list of some of the most common patterns (there are many more out there!).
Still life – these pattern sustain themselves in the same location over generations.
Oscillators – as the name implies oscillate in the same position.
Spaceships – are a cluster of cells that move into a direction.
(There are many other formations, machines that create machines and so on… experiment and discover others!)


Still life
Block
Bee-hive
Loaf
Boat
Tub


Oscillators
Blinker
Beacon


Spaceships
Glider