PROCEDURAL CONTENT GENERATION OF AN ARCHIPELAGO VIA CELLULAR AUTOMATA

Marco Cavallari

January 18, 2022

1 Introduction

The porpouse of this project is to use cellular automata approach in order to simulate a 3D archipelago composed by sand, green land, and the sea. The project is about a particular technique called "Procedural Content Generation" (or PCG) which is the "algorithmic creation of game content with limited or indirect user input"[1]. PCG refers to a method that can create game content by itself or together with other people (both players and designers).

2 Foundamental basis

As We already said in the introduction, this project is based on the cellular automata approach. In this section, We will discuss it. A cellular automaton (or Cellular automata) is a discrete computational model first discovered in the 1940s. The basic concept is really simple: it consists of an n-dimensional grid with a number of cells and a set of transition rules. Each cell of the grid can be in one of several (and finite) states. Usually, at t=0 all the cells are initialized with a random state. Then, a new generation (at t=1) is created by applying the transition rules to all cells at once. How? At each step t, every cell determinates its new state based on its own current state and the states of its neighborhood at step t-1. How do We chose the neighbors to consider when We apply the transition rules? The developer can decide by himself. Usually, in two-dimensional space, the two most used types of neighborhoods are Moore neighborhoods and Von Neumann neighborhoods, as illustrated in Figure 1.

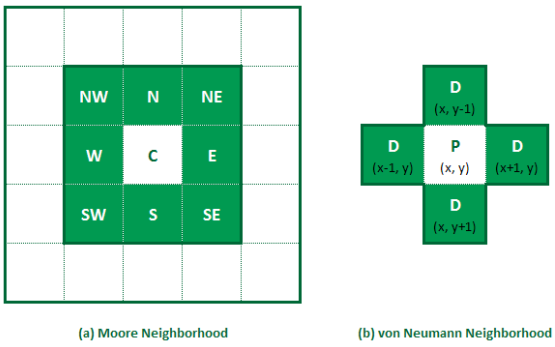

Figure 1: Moore and Von Neuman neighborhoods.

For example, if We consider a 2D grid and the Moore Neighborhood, We could create some rules of the type "if at least 5 of my neighbors are rocks, then I will become a rock" (here We are talking about an easy example of a 2D cave generation showed in class). What will be the result? The result will be that all the rocks cluster together.

3 World and coordinate system

This project is based on the concepts said in the previous section and extends them in a 3D environment where We want to create some archipelago shapes where all the cells can assume one of the following state:

1. Sea,
2. Sand,
3. Green land

Now let's imagine a small 3D world composed of 27 cells (3x3x3) as the one in Figure 2:

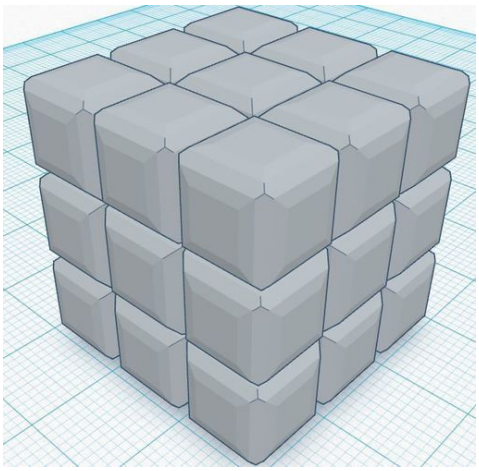

Figure 2: An hypothetical 3x3x3 world of cubes.

And now let's decompose it:

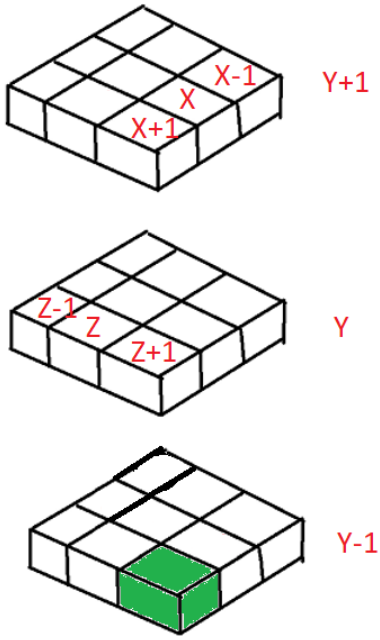

Figure 3: 3x3x3 world of cubes analyzed level per level and cell per cell.

In the previous image It is illustrated the coordinate system used in the project:

1. The "y" is the variable used for iterating on the Height dimension
2. The "z" is the variable used for iterating on the Width dimension
3. The "x" is the variable used for iterating on the Length dimension

According to the coordinate system used, the value of y,x and z of the green cell in Figure 3 will be respectively 0,2 and 2.

4 Implementation

4.1 The variables

Let's introduce the variables used for this project. First of all the user has to choose from the Unity interface some parameters:

1. The three dimensions (so integer variables) of the map:
   (a) Width
   (b) Height
   (c) Length
2. A boolean variable (useRandomSeed)to mark if the user want to use a random seed
3. The user can also choose a fixed seed in order to be able to replicate the experiment with the same results (variable Seed)
4. The materials to associate to the three state:
   (a) Tile_land
   (b) Tile_sea
   (c) Tile_green_land
5. number_of_iteration: A integer variable used by the user to select the number of iteration to run

This is what the Unity interface will look like in Figure 5. In addition to these variables ini-

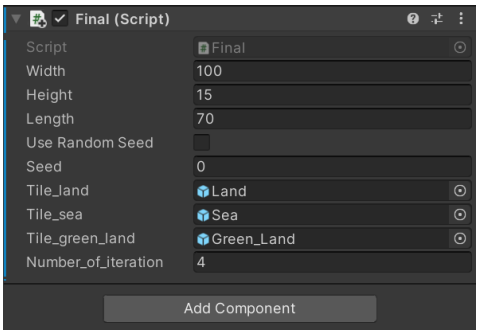

Figure 4: Unity panel where the user choose the configuration of the experiment.

tialized by the user, inside the programs there are three important variables: map, second_map and iteration. "map" and "second_map" are of the same type: they are 3D matrices with dimension [height,length,width], where each cell stores one of three possible values:

1. 0 for water
2. 1 for sand
3. 2 for green land

Why should We use two maps? We need two maps because the smoothing function should, at each iteration, write to a new copy of the map, switching 'source' and 'destination' between each iteration (for this reason I used the variable "iteration"). If I use simply one map as-is, it would re-write to itself at each iteration, effectively 'corrupting' the original before the job is done. Let's see how the algorithm works and then let's also show a simple example of incorrect behavior and a correct one. The algorithm:

1. First at iteration 0 We start filling "map". Each cell will contain one of the three different possible states (1 for sand,2 for green land, 0 for water/air).
2. At iteration 1, for each cell, the transition rules are applied using its own current state and the current state of surrounding cells (the neighbours) stored in "map" and then We put the new value of the considered cell in "second_map".
3. When all cells are processed, second_map will represent the current state for the next iteration.
4. At iteration 2 We will repeat the same passages, but this time We apply the rules on "second_map" and We will store the results (aka the new configuration of each cell) in "map".
5. And so on. At each iteration, We will use one between "map" and "second_map" to perform the evaluation of the rules and the other to store the new results.

Let's see now an easy example of cave generation in 2D. For each cell in the grid "1" means rock, "0" empty space. A cell becomes a rock if at least 3 neighbors out of the 8 around the considered cell are rocks; if they are less than 3 It will become/remain an empty space. In this example I will apply the rules to the center elements (yellow cells in the next table).

NB: In this example, for sake of simplicity, I am considering just the state of the 8 neighbors and not the one of the current cell so in the following matrix the rocks neighbors of X are 3, both if X is 1 or 0.

0	0	0
0	X	1
1	0	1

Now let's suppose this starting grid stored in "map":

0	0	1	0	1	1
0	0	0	1	0	0
0	0	1	0	0	0
1	1	0	0	0	1

And now let's analyze two cases:

1. Case 1 (incorrect): I just use the variable "map" and I apply the rules and store the results on the same matrix ("map") . The resulting configuration of "map" will be:

0	0	1	0	1	1
0	0	1	1	1	0
0	1	1	1	1	0
1	1	0	0	0	1

2. Case 2 (correct): I evaluate the rules on "map" and store the results in "second_map" (now empty). This will be the result in "second_map":

0	0	1	0	1	1
0	0	1	1	1	0
0	1	0	0	0	0
1	1	0	0	0	1

As We can see from the example in the orange sub-part of the matrix, the results of the two cases are different. Why? Because in "Case 2" We used "map" (the starting grid before the cases) and We did't update directly it, so when We analyzed the dark-orange cell We saw that in "map" there were just two neighbors that are rocks and that's correct. In "Case 1" We updated "map" so that when We analyzed the dark-orange cell the neighbors that are rocks changed and there were more than 3 rock neighbors. This is wrong.

4.2 Code Structure

4.2.1 Start method

The Start method structure is quite easy and straightforward. Everything starts taking as input the parameter seen in the 4.1 paragraph. Then the variable "map" will be initialized filling it with three possible values for each cell: 0,1, and 2; each one with a probability of 33% to be chosen. What I have just said is not completely true since in some particular location of the grid I will put some default (so not randomly picked) values. This happens when y=0 (It is reasonable to set all elements in y = 0 to 1 since presumably the seabed will be sandy) and when: x=0 or x=length-1 or z=0 or z=width-1 (I want the sea on the boarders of the map) The Figure 5 will show what We have already said.

```
System.Random pseudoRandom = new System.Random(seed.GetHashCode());
for(int y=0;y<height;y++){
    for(int x=0;x<length;x++){
        for(int z=0;z<width;z++){
            if(y==0){
                map[y,x,z]=1;
            }else if(x==length-1 || x==0 || z==0 || z==width-1){
                map[y,x,z]=0;
            }else {
                if(pseudoRandom.Next(0,100)>=1 && pseudoRandom.Next(0,100)<=33){
                    map[y,x,z]=0;
                } else if (pseudoRandom.Next(0,100)>33 && pseudoRandom.Next(0,100)<=66) {
                    map[y,x,z]=1;
                } else{
                    map[y,x,z]=2;
                }
            }
        }
    }
}
```

Figure 5: Part of the start method where We fill "map" with the various elements (0 per sea, 1 per sand and 2 per green_land).

Then, after map has been initialized the SmoothMap method has been applied number_of_iteration times. This method will apply the transition rules (We will see them later) and finally the method Draw will be called and will print out the final archipelago (see Figure 6).

```
  for (int i = 0; i < number_of_iteration; i ++) {
      SmoothMap();
      iteration++;
  }

  Draw();
}//end Start method
```

Figure 6: Ending part of the method Start.

4.2.2 SmoothMap method

This method works as follow: For each cell in the grid (map or second map, it depends by the iteration) the sorrounding_lands variable will be calculated using the method "GetSurroundingLandCount". It will return the sum of the neighbor's value (a value between 0 and 36). Then I apply the transition rules based on sorrounding_lands value:

1. If it is higher than 14 and lower than 21, the considered cell will become 1 (Sand)
2. If it is less or equal than 13 it will become 0 (sea)
3. If it is equal or higher than 25 it will become 2 (green land)
4. If it is equal to 14 it will remain of the same state as before

You now will probably ask how I found these threshold values to generate an archipelago shape. After various attempts with many and many thresholds this was the best. So We can say this values have been empirically chosen. The next figure will show the code to implement what We have just said.

```
void SmoothMap(){
    int c=0;
    for(int y=1;y<height;y++){
        for(int x=0;x<length;x++){
            for(int z=0;z<width;z++){
                int surrounding_lands=GetSurroundingLandCount(y,x,z);
                if (surrounding_lands > 14 && surrounding_lands <21){
                    if(iteration%2==0){
                        second_map[y,x,z]=1;
                    } else{
                        map[y,x,z] = 1;
                    }
                    c++;
                } else if ( surrounding_lands <= 13){
                    if(iteration%2==0){
                        second_map[y,x,z]=0;
                    } else{
                        map[y,x,z] = 0;
                    }
                    c++;
                } else if(surrounding_lands >= 25){
                    if(iteration%2==0){
                        second_map[y,x,z]=2;
                    } else{
                        map[y,x,z] = 2;
                    }
                }

            }
        }
    }
}
```

Figure 7: SmoothMap implementation.

4.2.3 GetSurroundingLandCount method

This method, as It can be seen in figure 8, takes as input the 3 coordinate of the cell of which We want to calculate the surrounding_lands value. In this project I didn't use all the 26 surrounding cells (9 on the higher level, 9 on the lower and the 8 at the same one) but just the 9 under the considered cell and the 8 surrounding it at the same height (plus the state of the considered cell). This method will return a value landCount that will be an integer value from 0 to 36(min=0*18 & max=2*18).

```
int GetSurroundingLandCount(int gridY,int gridX,int gridZ){
    int landCount=0;
    for(int y=gridY-1;y<=gridY;y++){
        for(int x=gridX-1;x<=gridX+1;x++){
            for(int z=gridZ-1;z<=gridZ+1;z++){
                if (y>=0 && y<height && x>=0 && x<length && z>= 0 &&  z< width) {
                    if(iteration%2==0){
                    landCount += map[y,x,z];
                    } else{
                        landCount += second_map[y,x,z];
                    }
                }
            }
        }
    }
    return landCount;
}
```

Figure 8: GetSurroundingLandCount implementation.

4.2.4 Draw method

This method, (Figure 9) as the others, iterate over the three dimensions and, depending on the iteration, instantiate cubes according to the state of the considered cell:

1. If map[y,x,z] == 0 (or second_map[y,x,z]==0, depending by the iteration) then I instantiate a "Sea cube" (Tile_sea)
2. If map[y,x,z] == 1 (or second_map[y,x,z]==1, depending by the iteration) then I instantiate a "Sand cube" (Tile_sand)
3. If map[y,x,z] == 2 (or second_map[y,x,z]==2, depending by the iteration) then I instantiate a "Green land cube" (Tile_green_land)

```
void Draw(){
    if (map != null && second_map!=null) {
        for(int y=0;y<height;y++){
            for (int x = 0; x < length; x ++) {
                for (int z = 0; z < width; z ++) {
                    if(iteration%2!=0){
                        //Debug.Log("second");
                        if(second_map[y,x,z]==1){
                            Instantiate(Tile_sand, new Vector3(x,y,z), Quaternion.identity);
                        } else  if(second_map[y,x,z]==0) {
                            if(y<(height/2)+1){
                                Instantiate(Tile_sea, new Vector3(x,y,z), Quaternion.identity);
                            }
                        }   else if(second_map[y,x,z]==2){
                            Instantiate(Tile_green_land, new Vector3(x,y,z), Quaternion.identity);
                        }
                    } else{
                        //Debug.Log("map");
                        if(map[y,x,z]==1){
                            Instantiate(Tile_sand, new Vector3(x,y,z), Quaternion.identity);
                        } else if(map[y,x,z]==0){
                            if(y<(height/2)+1){
                                Instantiate(Tile_sea, new Vector3(x,y,z), Quaternion.identity);
                            }
                        }   else if(map[y,x,z]==2){
                            Instantiate(Tile_green_land, new Vector3(x,y,z), Quaternion.identity);
                        }
                    }
                }
            }
        }
    }
}
```

Figure 9: Draw implementation.

5 Software version used

1. Version used of Untiy: 2020.3.19f1 (LTS)
2. Version used of Visual Sudio: 16.11

6 Results achieved

In this final section I will show some of the archipelagos generated by my code.

1. If you would like to replicate the experiment on Figure 10, the parameters are:
   (a) Width=80
   (b) Height=15
   (c) Length=100
   (d) Use Random Seed NOT checked
   (e) Seed=AI4VG
   (f) number of iteration=4

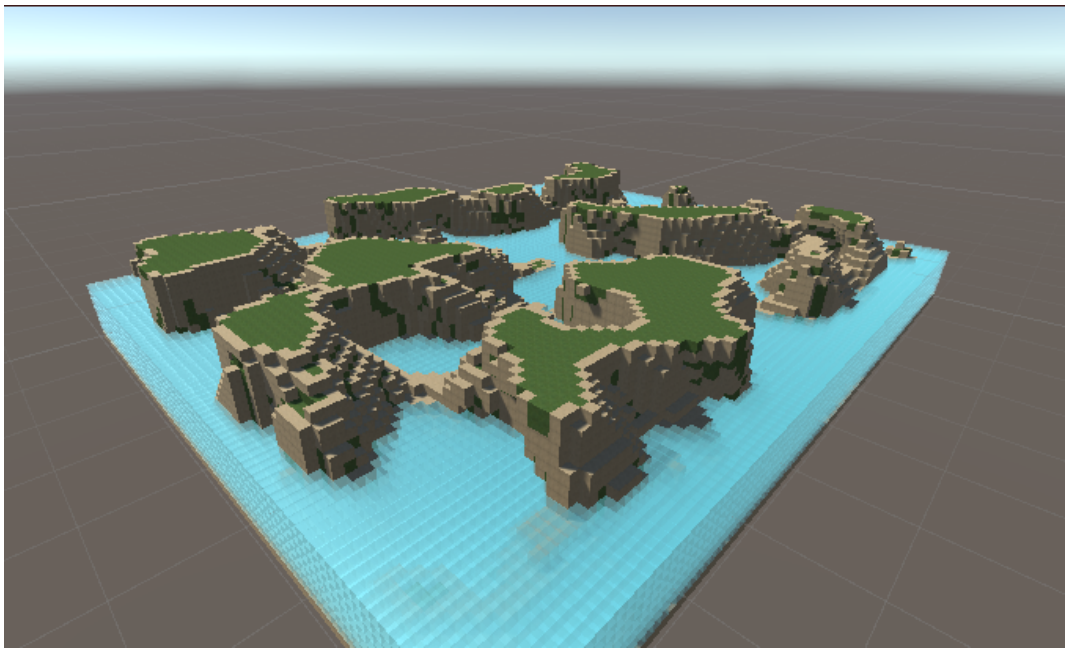

Figure 10: Experiment with fixed seed

2. In Figure 11 the same experiment as before is reported but this time with no instantiation of the sea cubes (in order to show You the rugged coastlines, step cliffs and jagged seabeds).

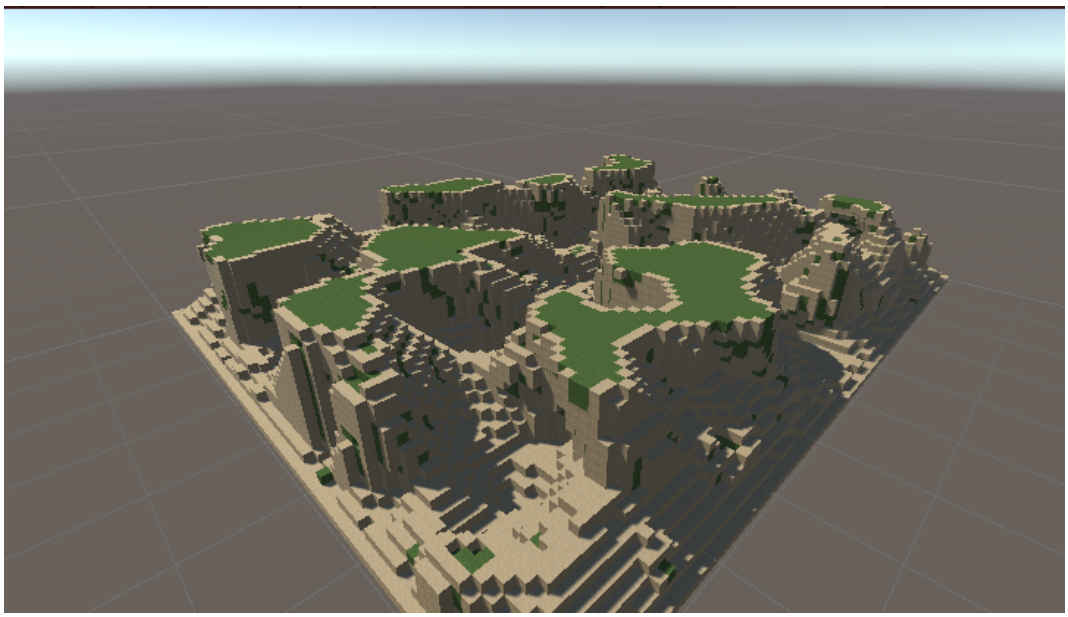

Figure 11: Experiment with the same fixed seed as before but without sea.

3. In Figure 12 I reported an experiment with the following parameters:
   (a) Width=100
   (b) Height=20
   (c) Length=120
   (d) Use Random Seed checked (the random seed generated in this case was gbNZo4ga)
   (e) number of iteration=6

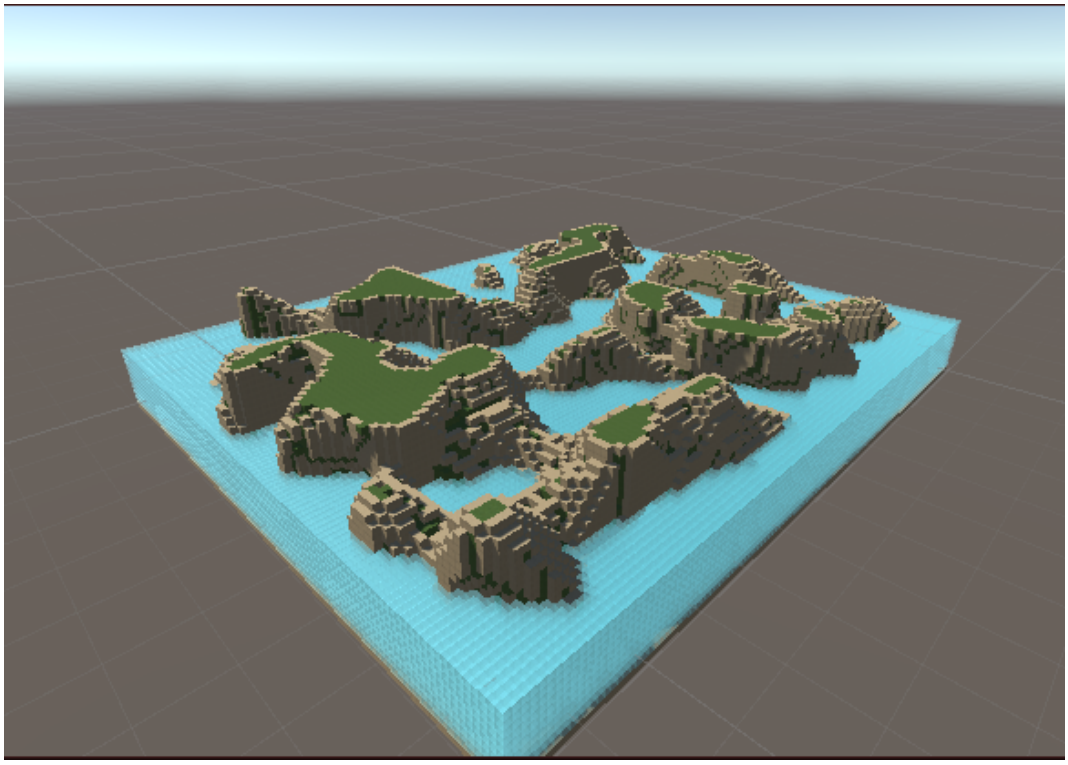

Figure 12: Experiment with random seed.

7 Conclusion and final thoughts

The result achieved are acceptable but far from being good: The shapes resemble some sort of islands but are not so realistic. The project could be improved, for example We could use some better and more complex transition rules in order to achieve more detailed results.

References

[1] Procedural Content Generation in Games by N. Shaker, J. Togelius, M. J. Nelson