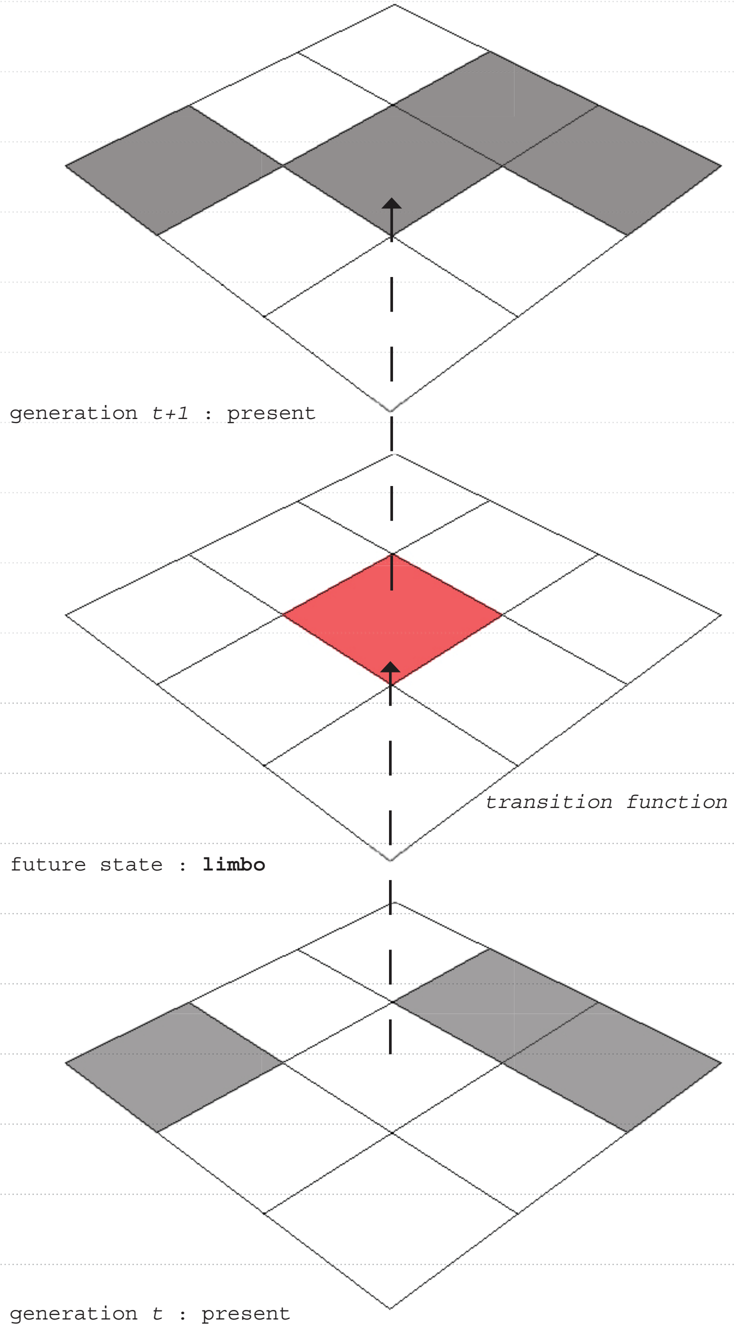


Cellular Automaton - Game of Life

In 1970 the mathematician John Conway introduced through the Scientific American a mathematical game based on the cellular automaton of John von Neumann, that would change the course of artificial intelligence.
The game consists of one type of element only (in NetLogo a patch with only two states) and three very simple rules, which define the state of an element in relation to their eight neighbours (we are going to explore those rules with you during the workshop).
The result of those local rules hinted at a concept by von Neumann called the universal constructor - machine made of any kind of stuff that would be able to compute any state, if morphological, behavioural, economical etc.
Even more daring was the hypothesis that this universal constructor might also generate states or patterns that could reproduce themselves - a phenomenon only attributed to living systems.
The Game of Life, so called by its inventor, although not made of any particulare stuff at all, but relations between virtual patches manages to generate an infinite range of patterns that are not predictable unless one executes an initial constellation of patches. Furthermore, patterns have been discovered - i.e. the R-pentomino or glider - that would after several cycles return their own pattern. Thus, speculations about the universal constructor were reborn.

The Cellular Automaton (CA) we are looking at today is essentially exactly the same in VBA as the one we looked at within NetLogo. Again we will be looking at the Game of Life by John Conway in order to demonstrate the characteristics of a CA.

'Limbo-World'

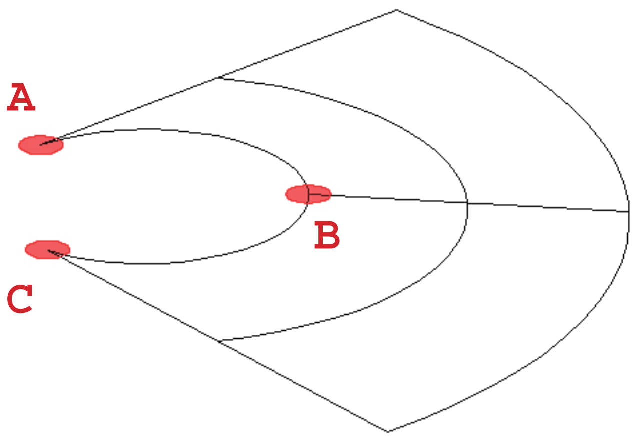
The salient difference between the NetLogo and the VBA code is the explicity with which the VBA code is written. In NetLogo ideas and concepts like the 'meta-world' or 'limbo-world' are taken for granted and don't show, just as loops didn't show up.
To refresh your memory: the 'limbo-world' is a parallel array which stores the observations of the present situation without translating them during the observation process. Thus, the future state of the automaton is stored in a parallel matrix of cells which reflect the present relationships between the cells.
When all the present relationships have been evaluated and strored in the 'limbo-world', the 'present world' is swapped with the 'limbo-world', making the 'present' the 'limbo' world and vice versa.
This delaying of translation of a reading of a situation helps to circumvent the problem of the lack of real parallel computation . Since computers can only evaluate sequentially, the 'real' picture of the CA would be distorted if each cell would be updated right after it had been evaluated. The delaying of the updating and the reading of a situation in a *quasi-frozen* state ensures a fake simultaneity!

Topology vs Topography

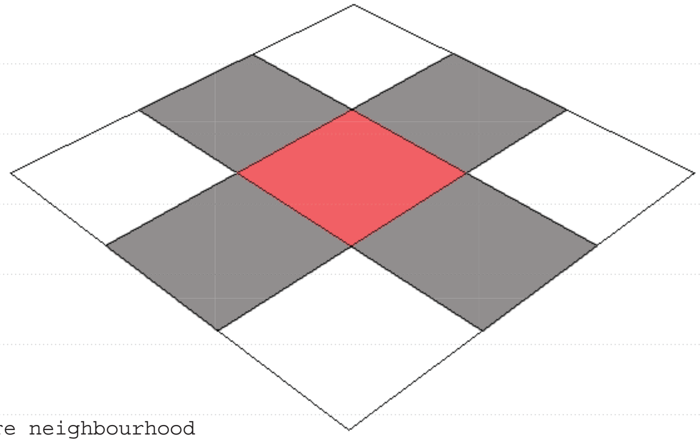
When one is talking about explicit visual qualities of a space or surface desribed through geometric semiology, one generally refers to a topography - the description of the appearance of a form.
On the other hand, when one is trying to describe an implicit, generally non-visible structure of a space, surface or geometry, one generally refers to a topology - the description of the structure of a form.

All calculations for a CA that is fixed on a orthogonal grid are topological, since other Euclidean qualities like distance, volume, surface are neglectible. Once a CA runs off the grid, like a voronoi diagram, geometric features can be taken into account. Thus, topography as well as topology form the basis for calculations.

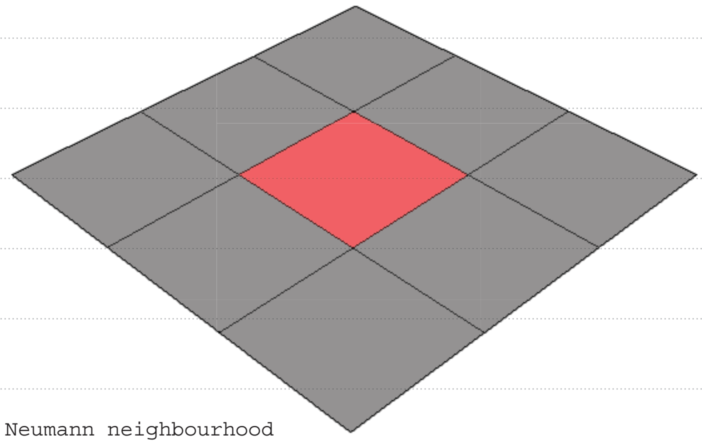
In the example given below, topographically speaking, vertex A looks closer in distance to vertex C, making C geometrically the neighbour of A.
But topologically speaking, according to the structure of the surface mesh, vertex B is the closest neighbour to vertex A as well as C.



10 09 08 07 06 05 04 03 02 01



Moore neighbourhood



van Neumann neighbourhood

Neighbour Count

In the function counthem(), we count up the states of all the topologically neighbouring cells from the perspective of one single cell at a time. There are two differenct types of neighbourhoods as shown in the diagram on the left.

In order to count up the immediate topological neighbours, we need to loop through the two or three dimensional array that contains all the cells and collect the necessary information. Any given cell has an array index position. In 2d for example:

grid*(i, j)*

Topologically, the neighbours are the ones which are exactly one in array index away from the given cell in either direction, *(i-1)* to *(i+1)* and *(j-1)* to *(j+1)*. In the code that is expressed through the nested loops

```
For i = rowpos - 1 To rowpos + 1
        For j = colpos - 1 To colpos + 1
                neighs = neighs + grid(i, j).state
        next j
next i
```

Below you will find a diagram of the array ind-eces:

i - 1 j + 1	i j + 1	i + 1 j + 1
i - 1 j	ME	i + 1 j
i - 1 j - 1	i j - 1	i + 1 j - 1

array indeces in van Neumann neighbourhood

10 09 08 07 06 05 04 03 02 01

When we have counted all the states of all the cells around a given cell (**ME** in the diagram above) and added them up, we still have to subtract our own state from the sum:

```
countthem = neighs - grid(rowpos, colpos, levelpos).state
```

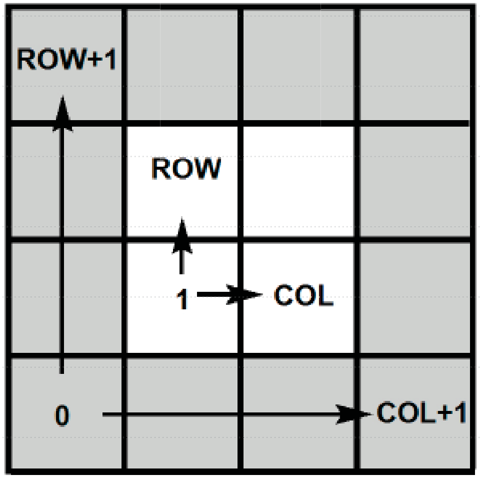
Now the function *countthem()* returns the sum of all the state values from the topological neighbourhood of a given cell to the sub procedure *iterate()*. In *iterate()* follow the transition rules that emulate the logic of the life game discussed in NetLogo.

Edge Condition

What happens if a cell is at the edge of the automaton and needs to calculate a neighbour that doesn't exist? There are three common solutions to that problem:

1 > test the array location of the cell before calculation and tell it explicitly not to search in certain non-existing neighbourhood array positions. Rigorous but complicated solution.
2 >create a ring of dead cells around the automaton, whose state can be interrogated but which never count their neighbours themselves
3 > wrap the right edge to the left and the top edge to the bottom, thus creating a seemingly infinite universe.

Today we introduce solution 2 where we set up one more row of cells on either edge in Y and one more column on either side in X. See the diagram opposite:



In the sample code we seed all cells from ***0*** to ***COL+1*** and ***ROW+1*** as dead cells initially:

```
For i = 0 To row + 1
        For j = 0 To col + 1
                grid(i, j).state = dead
                limbo(i, j) = dead
        Next j
 Next i
```

Whereas, when we loop through the array positions in the sub procedure *iterate()*, we only loop from ***1*** to ***COL*** and ***ROW***:

```
For i = 1 To row
        For j = 1 To col
                ...
        next j
next i
```

Thus, we make sure that we don't jump over the edge and get a Run-Time Error: Out of Range.