Introduction

As per Wikipedia “The **Game of Life**, also known simply as **Life**, is a cellular automaton devised by the British mathematician John Horton Conway in 1970.” It was described in Martin Gardner's Mathematical Games column in Scientific American that year.

This is a zero-player **game**, meaning that its evolution is determined by its initial state, requiring no further input. One interacts with the **Game** **of** **Life** by creating an initial configuration and observing how it evolves. In other words, there are no players, and no winning or losing. Once the "pieces" are placed in the starting position, the rules determine everything that happens later.

The universe of the **Game** **of** **Life** is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, live or dead. Every cell interacts with its eight neighbours, which are the cells that are directly horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

1. Any live cell with fewer than two live neighbours dies, as if by loneliness.
2. Any live cell with more than three live neighbours dies, as if by overcrowding.
3. Any live cell with two or three live neighbours lives, unchanged, to the next generation.
4. Any dead cell with exactly three live neighbours comes to **life**.

The initial pattern constitutes the '*seed*' of the system.

The first generation is created by applying the above rules.

Simultaneously to every cell in the seed — births and deaths happen simultaneously, and the discrete moment at which this happens is sometimes called a tick.

Problem Definition

The inputs below represent the cells in the universe as X or ̶

X is an alive cell.

̶ is a dead cell or no cell.

The below inputs provide the provide pattern or initial cells in the universe. The output is the state of the system in the next tick (one run of the application of all the rules), represented in the same format.

**Block pattern**

|  |  |
| --- | --- |
| **Input A** | **Output A** |
|  |  |

**Boat pattern**

|  |  |
| --- | --- |
| **Input B** | **Output B** |
|  |  |

**Blinker pattern**

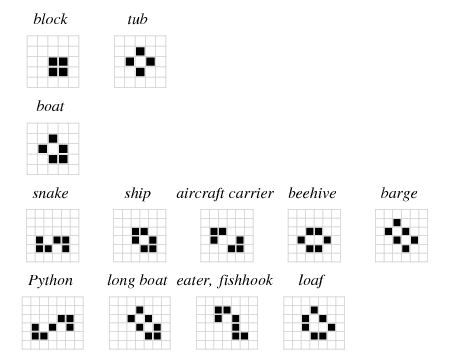
|  |  |
| --- | --- |
| **Input C** | **Output C** |
|  |  |

**Toad pattern**

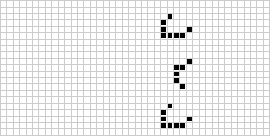
|  |  |
| --- | --- |
| **Input D** | **Output D** |
|  |  |

It is important to note that in Output D, the two new rows have been due to Rule #4 i.e. Any dead cell with exactly three live neighbours comes to **life**. Thus, the next state will include two new auto grown rows.

A pattern which does not change from one generation to the next is known as a still life, and is said to have period 1.



Patterns that cycle through a set of configurations are called oscillators.

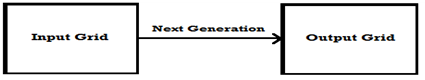


Solution - Approach

**Approach for Handling Grid Generations**

As a good idea I will keep separate Grid in initial generation and next Generation. In the beginning, I shall have two grids say Input Grid and Output Grid. Input Grid is the initial state of **Game** **of** **Life** to start with. Output grid will contain the next generation of Input Grid. I shall need to apply rules for each cell in Input Grid and get the Cell's next generation in Output Grid. In case, where Row or Column growth is required then these will be added to Output Grid.

Please note that the state of Input grid will not be updated to get the next generation, so there is no run-time consideration of cell state changes. In other words, consider Input Grid as grid with freeze cells and Output Grid will change until all cells in Input Grid is evaluated. Please see figure below:



Once a generation is complete, I shall do a swapping of Output Grid to Input Grid and the process will continue.

**Approach for two dimensional matrix**

Also I shall need to implement a two dimensional matrix having every cell containing one of two boolean values; live or dead. This two dimensional structure should be able to grow in either side e.g. new row could be added to top as well as bottom and new column can be added before first column and last column. It gives a sense of a list as we need not refer a cell from its index but need to enumerate from first item till the last element.

It is clear that I am moving into the utilizing list collection classes in C#. I have a custom class for cell having a boolean property as Isalive. This will make the implementation more extensible. Further, to hold a list of cell a custom object called Row which contains a list of cells. Therefore, the grid will contain a list of such rows, which in turn contains a list of Columns. Also, if any other behavior is required to be added at the cell level then it can be easily done in the Cell class.

**Approach for simultaneous update to all cells**

This is very important point on simultaneously updating all cells in the **Game** **of** **Life** Grid. At first sight this seems trivial. Let us analyze the below questions before moving on:

Update all cells at the same time:

It does not require CPU to execute cell update operation at the same time, but it requires that all tasks are done in parallel in different threads as cell's next generation is not dependent upon next generation of other cells. Here I need to implement threading to apply simultaneous update to all cells.

Update to a cell dependent on other cells updates:

Cell's next generation is not dependent upon next generation of other cells. This is true, we always need to read Input Grid (which is freeze) and stuff the result to Output Grid cell. So Output Grid cell state is independent of other cell state in Output Grid. However, in some situations the OutputGrid can grow; a new row or column can be added to Output Grid. This will cause issues in parallel operations.

Glossary

**Grid**: This class is a dynamic two dimensional data structure which holds matrix of cells. It consists of generic list of rows which in turn contains list of cells.

**Row**: This class is a generic list of cells. User can add a cell at the end of cell list or insert a cell at specified index.

**Cells**: This class holds the actual representation of cell data in a Boolean property IsAlive which is true for live and false for dead cells.

**Coordinates**: This is a struct to hold x and y co-ordinates of the grid.

**Rule**: This class separates the algorithm of transitioning grid to the next generation. It exposes ChangeCellsState and ChangeGridState methods to apply changes on grid. It has following static methods.

**AccessibleCell**: This is a specialized class to make traversal of adjacent cells from a specified cell easier. It keeps reachable adjacent cell locations for unique cell types on any size of grid. Cell types are distinct cell locations which share similar adjacent cells

**GridHelper**: This is a static helper class to perform operations like displaying the grid and Copy source grid to destination.

References

* [**http://en.wikipedia.org/wiki/Conway%27s\_Game\_of\_Life**](http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life)
* [**http://www.math.com/students/wonders/life/life.html**](http://www.math.com/students/wonders/life/life.html)
* [**http://www.emergentuniverse.org/#/life**](http://www.emergentuniverse.org/#/life )