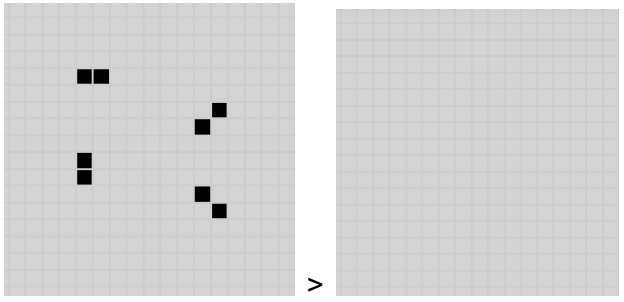
Game of Life

References
https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life
https://danhough.com/blog/building-game-of-life-typescript/

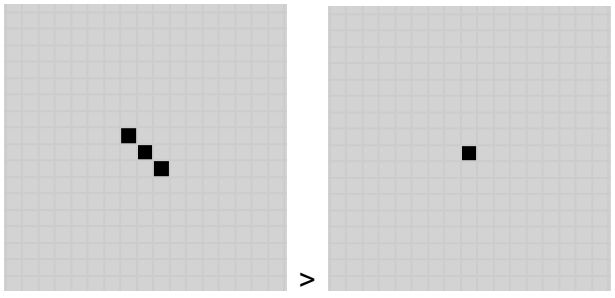
Understanding the problem according to Wikipedia

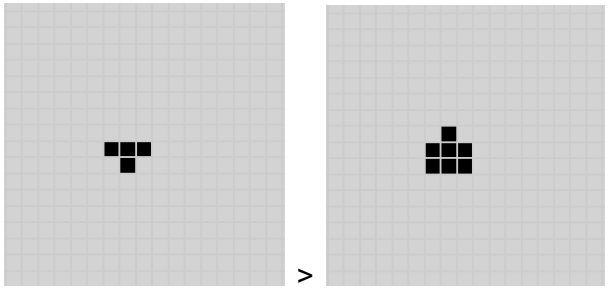
The universe of the Game of Life is an infinite, two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, live or dead (or populated and unpopulated, respectively). Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, 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 underpopulation.

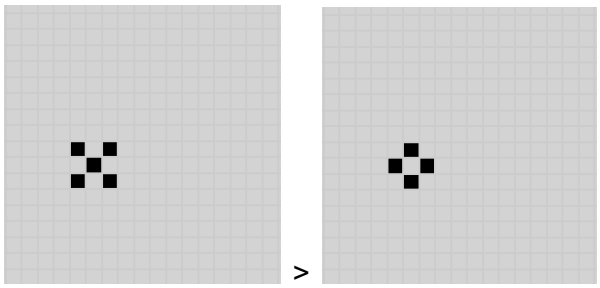


2. Any live cell with two or three live neighbours lives on to the next generation.

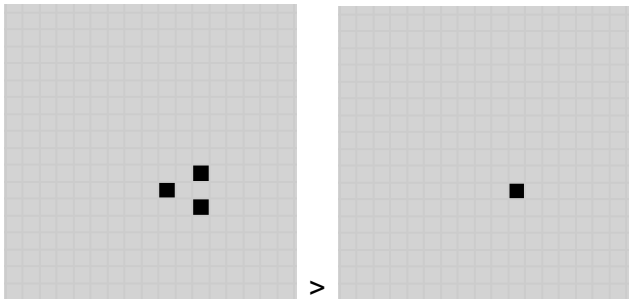




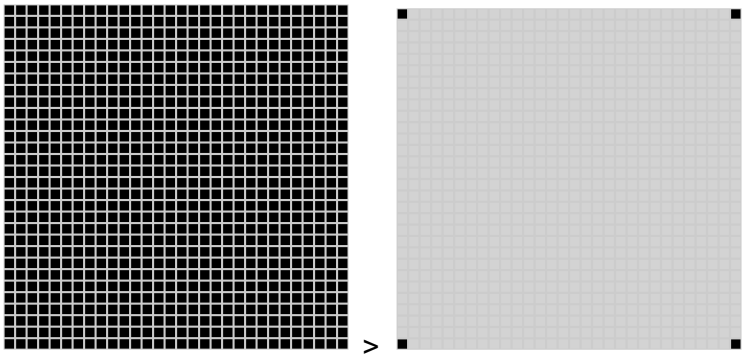
3. Any live cell with more than three live neighbours dies, as if by overpopulation.

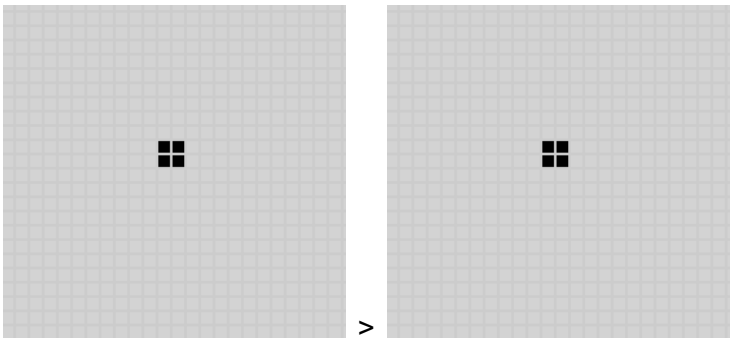


4. Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.



Edge cases





Observations

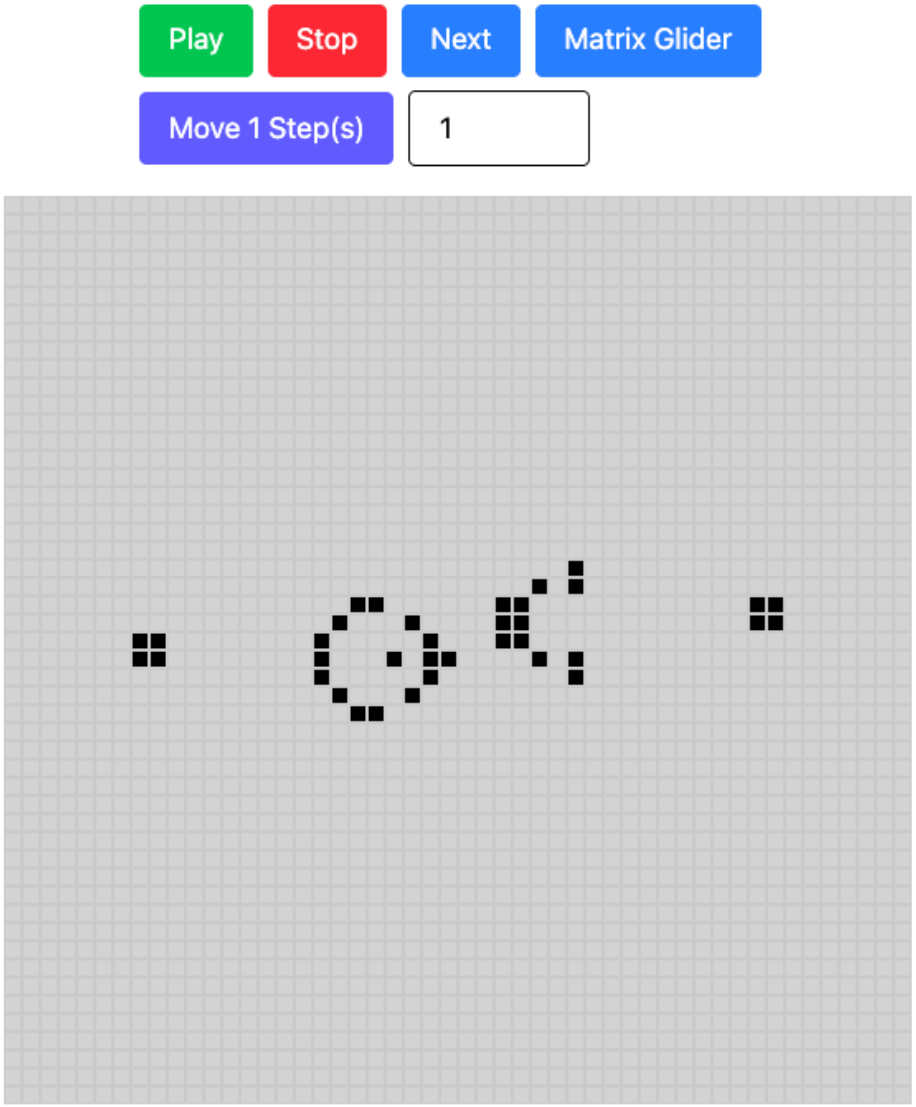
- Infinite universe: Assuming the game takes place in an infinite universe but with limited computing resources, I've implemented a representation of the universe as a matrix with a dimension of X (you can set any value). While there are techniques available to simulate an infinite universe, such as dynamically expanding the universe when boundaries are reached, I chose not to implement this approach for this test.
- Performance: To improve performance, I decided to use **react-window** lib. It is a lightweight React library for efficiently rendering large lists and grids by windowing or virtualization. Instead of rendering all items in a long list or large grid (which can be very slow and consume lots of

memory), react-window only renders the visible items in the viewport plus a small buffer, dramatically improving performance.

- Given an matrix of RxC = N cells, the algorithm is O(N) for space, as we need a secondary matrix to store the new cells value, therefore O(2*N) = O(N).
- For execution time, it is also O(N), as we need to visit all cells to count its neighbors. To find the neighbors, it's straight O(1) as we are calculating the neighbors indexes using an array of fixed directions:
  (top-left, top, top-right, left, right, bottom-left, bottom, bottom-right)

```
[
    [-1, -1], [-1, 0], [-1, 1],
    [0, -1], [0, 1],
    [1, -1], [1, 0], [1, 1],
]
```

UI and controls



Play - run the next gen every 300ms
Stop - stop the generation
Next - run the next generation
Move X steps - run the next X generations
Matrix Glider - fill matrix with a seed example