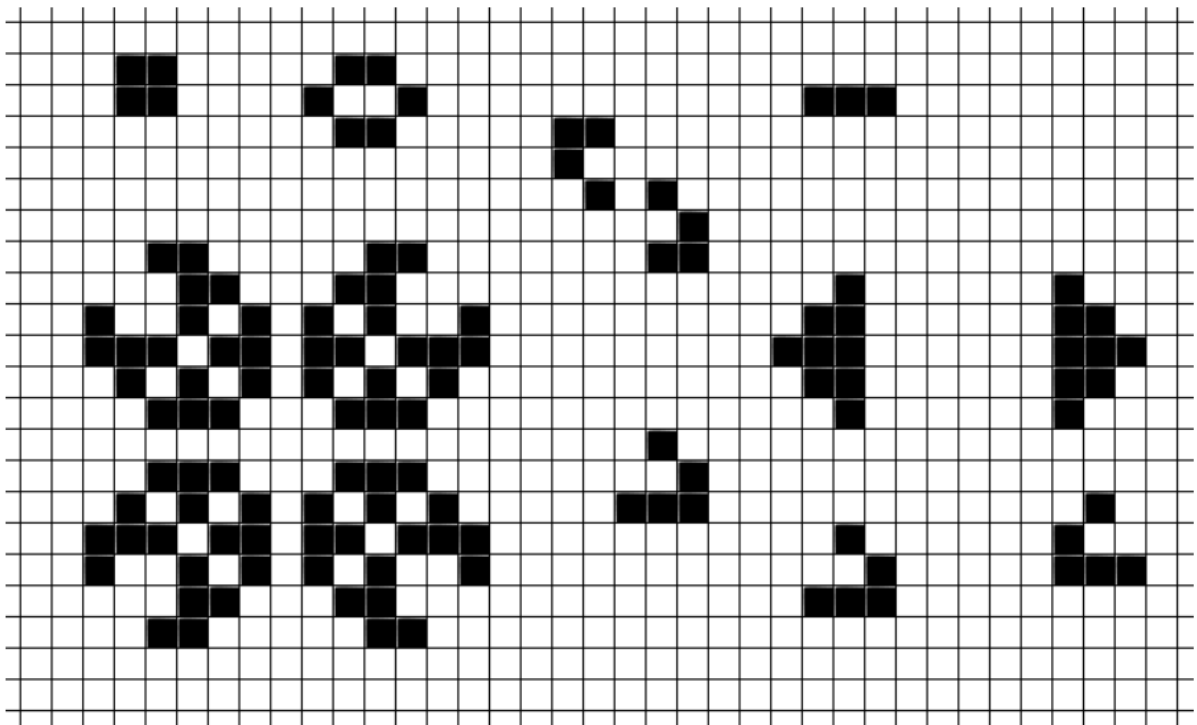


CONWAY'S GAME OF LIFE: AN IN-DEPTH EXPLORATION

Conway's Game of Life is a cellular automaton devised by the mathematician John Conway. It's like a simulation of life where you have a grid of cells, and each cell can be in one of two states: alive or dead. The game is played out in generations, with each generation's state determined by a set of rules.



Initial Setup: You start with an initial configuration of live and dead cells on a grid.

Rules for Life and Death: The next generation is determined by these rules:

1. A live cell with 2 or 3 live neighbors survives to the next generation.
2. A live cell with fewer than 2 live neighbors dies due to underpopulation.
3. A live cell with more than 3 live neighbors dies due to overpopulation.
4. A dead cell with exactly 3 live neighbors becomes alive through reproduction.

Repeating Generations: You keep applying these rules to each cell in the grid to generate the next generation. The pattern can evolve in interesting and complex ways.

Patterns: The game can exhibit various patterns, from static (doesn't change) to oscillators (repeating patterns) to gliders (patterns that move).

Endless Possibilities: One of the fascinating aspects of the Game of Life is that simple initial configurations can lead to incredibly intricate and unpredictable patterns.

GAME FEATURES :

The Grid:

The game is typically played on a 2D grid, where each cell can either be alive (often represented as '1') or dead ('0').

The grid can be finite or infinite, but for practical purposes, most simulations use a finite grid.

Patterns:

Some common patterns include:


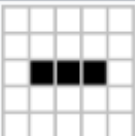
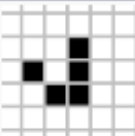
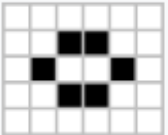
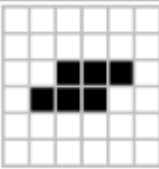
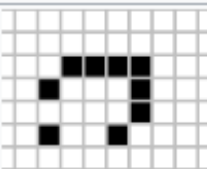
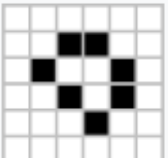
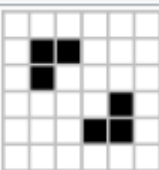
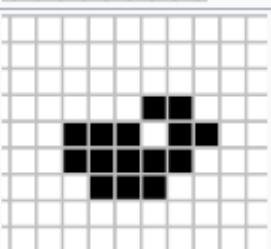
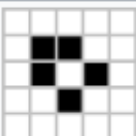
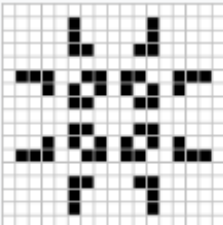
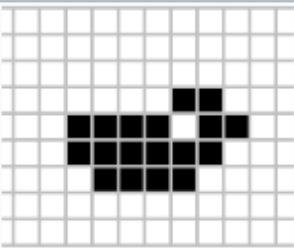
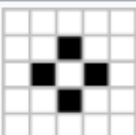
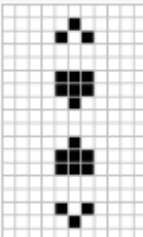
Still lifes: These configurations don't change from one generation to the next. Examples include the block, beehive, and loaf.

Oscillators: These patterns repeat after a certain number of generations. The blinker, toad, and pulsar are classic examples.

Spaceships: These are patterns that move across the grid. The glider is the most famous spaceship.

Gosper's Glider Gun:

One of the most famous patterns in Conway's Game of Life is Gosper's Glider Gun. It's a configuration that constantly emits gliders. Gliders are spaceship patterns that move diagonally across the grid.

Still lifes		Oscillators		Spaceships	
Block		Blinker (period 2)		Glider	
Bee-hive		Toad (period 2)		Light-weight spaceship (LWSS)	
Loaf		Beacon (period 2)		Middle-weight spaceship (MWSS)	
Boat		Pulsar (period 3)		Heavy-weight spaceship (HWSS)	
Tub		Penta-decathlon (period 15)			

Mathematical Properties:

The Game of Life falls under the field of cellular automata, which is a grid of cells governed by a set of rules. It's a Turing complete system, meaning it can simulate any computer program. This property makes it incredibly versatile.

Applications:

While originally a mathematical curiosity, Conway's Game of Life has found applications in various fields, including:

Biology: Modeling population dynamics.

Computer science: Testing and simulating algorithms.

Art: Creating intricate and mesmerizing visual patterns.

Cryptography: Generating random sequences.

Complexity:

Conway's Game of Life can exhibit remarkable complexity from very simple initial configurations. Some patterns can grow indefinitely or create self-replicating structures.

Simulation Software:

Many software tools and programming libraries are available for simulating and visualizing Conway's Game of Life. Some popular ones include Golly, Python libraries like Pygame, and online simulators.

Research and Exploration:

Conway's Game of Life has been the subject of extensive research. Some patterns have been discovered that have remarkable properties, like creating Turing machines within the game.

C++ SOURCE CODE:

```
#include <iostream>           // header file
#include <vector>              // vector file for dynamic arrays
using namespace std;
int countLiveNeighbors(const vector<vector<int>>& grid, int x, int y)
// counting the number of live neighbours from the coordinate (x,y) - x,y centre
of the grid
{
    int liveNeighbors = 0;           // it initializes the live neighbours to 0 to
count live neighbours
    for (int dx = -1; dx <= 1; dx++) // two nested for loops for iterating over
each cell over the grid
    {
        for (int dy = -1; dy <= 1; dy++)
        {
            if (dx == 0 && dy == 0) continue; // examining that the cell is
exactly the centre cell or not
            int newX = x + dx;               // for calculating the new x and y
coordinates by increasing the current coordinates with the change in the
coordinates
            int newY = y + dy;

            // this below condition checks that the neighboring cell is in the
boundaries i.e left , right , top ,bottom and also checks whether there are any
alive cells or not
```

```

        if (newX >= 0 && newX < grid.size() && newY >= 0 && newY <
grid[0].size() && grid[newX][newY] == 1)
        {
            liveNeighbors++;           // if there are alive cells then increase
the liveneighbours counter
        }
    }
}
return liveNeighbors;                 // to return the number of live
neighbours
}
int main()
{
    int gridSize;                     // declaring the gridSize variable of type
int

    cout << "Enter the grid size: ";
    cin >> gridSize;                  // inputing the grid size from
user

    vector<vector<int>> grid(gridSize, vector<int>(gridSize, 1));    //
Initialize all cells in the grid as alive (1)

    cout << "Initial Generation:\n";

    // Display the initial grid
    for (int i = 0; i < gridSize; i++)
    {
        for (int j = 0; j < gridSize; j++)
        {
            cout << grid[i][j] << " ";
        }
        cout << endl;
    }

    int generation = 0;
    vector<vector<int>> newGrid(gridSize, vector<int>(gridSize, 0));
    // Create a new grid with all cells initialised to 0 (dead)

    // the below nested loops are used for iterating through each cell of the grid

```

```

for (int i = 0; i < gridSize; i++)
{
    for (int j = 0; j < gridSize; j++)
    {
        int liveNeighbors = countLiveNeighbors(grid, i, j);    // function call
with 3 arguments

        if (liveNeighbors == 3)                                // condition check for alive cells
        {
            newGrid[i][j] = 1; // Cell becomes alive    // changing the cell value
to 1
        }
    }
}

grid = newGrid;                                                // Update the main grid with the new
generation
generation++;
cout << "Final Generation:\n";

// Display the final grid
for (int i = 0; i < gridSize; i++)
{
    for (int j = 0; j < gridSize; j++)
    {
        cout << grid[i][j] << " ";
    }
    cout << endl;
}
cout << "All cells are dead. Game over!" << endl;
return 0;
}

```

UNDERSTANDING OF THE CODE:

countLiveNeighbors Function:

1. This function calculates the number of live neighbors for a given cell in a grid.
2. It takes three arguments: the grid (a 2D vector of integers), and the coordinates x and y of the cell for which you want to count live neighbors.

3. The function initializes a counter, liveNeighbors, to 0. It uses two nested loops to iterate over all cells in the neighborhood of the specified cell (including diagonals).
4. It checks if the examined cell is not the center cell itself ($dx = 0$ and $dy = 0$).
5. For each neighbor, it calculates the new coordinates newX and newY.
6. It then checks if the neighboring cell is within the boundaries of the grid and whether it is alive (1) or not.
7. If the conditions are met, it increments the liveNeighbors counter.
8. Finally, it returns the count of live neighbors for the specified cell.

main Function:

1. This is the main function where the execution of the program starts.
2. It begins by declaring an integer variable gridSize to hold the size of the grid.
3. It prompts the user to enter the grid size and reads the input.
4. It initializes a 2D vector called grid with all cells set to 1, representing an initial generation of all cells being alive.
5. It prints "Initial Generation" and displays the initial state of the grid.

Generating the Next Generation:

1. The code initializes an integer variable generation to 0.
2. It creates a new 2D vector called newGrid with all cells initialized to 0 (dead) to represent the next generation.
3. Two nested loops iterate through each cell of the grid to calculate the number of live neighbors for each cell using the countLiveNeighbors function.
4. If a cell has exactly 3 live neighbors, it is marked as alive (set to 1) in the newGrid, simulating the rules of the Game of Life.
5. After processing all cells, the grid is updated with the newGrid to represent the next generation.
6. The generation is incremented.

Final Display:

1. The code prints "Final Generation" and displays the state of the grid after applying the rules.
2. It prints "All cells are dead. Game over!" to indicate the end of the simulation.

OUTPUT:

```
Enter the grid size: 5
Initial Generation:
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
Final Generation:
1 0 0 0 1
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
1 0 0 0 1
All cells are dead. Game over!
```

**\\ THIS C++ CODE JUST GENRATES THE ARRAY AND
FULLFILS THE LOGIC OF THE CONWAY'S GAME OF
LIFE NOW THE NEXT JAVASCRIPT CODE WILL GIVE
THIS C++ CODE THE INTERFACE.**

JAVASCRIPT SOURCE CODE

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <style>
    body {
      margin: 0;
      padding: 0;
      display: flex;
      flex-direction: column;
      align-items: center;
      justify-content: flex-start;
      height: 100vh;
      background-color: #f0f0f0;
    }
    h1 {
      color: #333;
    }
    .grid-container {
```



```

    display: grid;
    grid-template-columns: repeat(20, 20px);
    gap: 1px;
    justify-content: center;
  }
  .cell {
    width: 20px;
    height: 20px;
    background-color: blue;
    border: 1px solid #eee;
  }
  .cell.dead {
    background-color: red;
  }

  button {
    margin: 10px;
    padding: 5px 10px;
    background-color: #333;
    color: #fff;
    border: none;
    cursor: pointer;
  }
</style>
</head>
<body>
  <h1>Conway's Game of Life</h1>
  <div class="grid-container">
    <!-- This div will contain the grid -->
  </div>
  <button id="start-button">Start</button>
  <button id="stop-button">Stop</button>
  <script>
    const gridSize = 20;
    const cellSize = 20;
    let grid = new Array(gridSize).fill(null).map(() => new
Array(gridSize).fill(0));
    let intervalId = null;

    function createGrid() {

```

```

const gridContainer = document.querySelector('.grid-container');
gridContainer.innerHTML = "";

for (let i = 0; i < gridSize; i++) {
  for (let j = 0; j < gridSize; j++) {
    const cell = document.createElement('div');
    cell.className = 'cell';
    cell.style.width = `${cellSize}px`;
    cell.style.height = `${cellSize}px`;
    cell.addEventListener('click', () => toggleCellState(i, j));
    gridContainer.appendChild(cell);
  }
}

// Initialize the grid with all cells alive
for (let i = 0; i < gridSize; i++) {
  for (let j = 0; j < gridSize; j++)
  {
    grid[i][j] = 1;
  }
}

updateGrid();
}

function updateGrid() {
  const cells = document.querySelectorAll('.cell');
  cells.forEach((cell, index) => {
    const row = Math.floor(index / gridSize);
    const col = index % gridSize;
    if (grid[row][col] === 0) {
      cell.classList.add('dead');
    } else {
      cell.classList.remove('dead');
    }
  });
}

function toggleCellState(row, col) {
  grid[row][col] = 1 - grid[row][col];
}

```

```

        updateGrid();
    }

    function simulateGeneration() {
        if (!intervalId) {
            intervalId = setInterval(() => {
                for (let i = 0; i < gridSize; i++) {
                    for (let j = 0; j < gridSize; j++) {
                        if ((i === 0 || i === gridSize - 1) && (j === 0 || j === gridSize
- 1))
                            {
                                grid[i][j] = 1;
                            } else {
                                grid[i][j] = 0;
                            }
                        }
                    }
                }
                updateGrid();
            }, 1000);
        }
    }

    document.getElementById('start-button').addEventListener('click', () => {
        simulateGeneration();
    });

    document.getElementById('stop-button').addEventListener('click', () => {
        if (intervalId) {
            clearInterval(intervalId);
            intervalId = null;
        }
    });

    createGrid();
</script>
</body>
</html>

```

UNDERSTANDING OF THE CODE:

HTML Structure:

1. The HTML file defines the basic structure of the web page.
2. It includes the necessary metadata, such as character set and viewport settings.
3. It links an external stylesheet to style the page.

Styling:

1. The included CSS within the <style> tags defines the styling for the page.
2. It sets up the body to be a flex container, centering the content vertically and horizontally.
3. Styles for the grid, cells, and buttons are defined.
4. Live cells are displayed in blue, while dead cells are displayed in red.

JavaScript Implementation:

1. The JavaScript code within the <script> tags handles the functionality of the Game of Life simulation.

Grid Initialization:

1. It sets the size of the grid to 20x20 cells, and each cell has a fixed size of 20x20 pixels.
2. It initializes a 2D array called grid to represent the grid. Initially, all cells are set to 0, representing dead cells.

createGrid Function:

1. This function dynamically creates the grid by generating HTML div elements for each cell.
2. It attaches a click event listener to each cell to toggle its state (alive or dead) when clicked.
3. Initially, it sets all cells to be alive (1) and updates the grid display.

updateGrid Function:

1. This function updates the visual representation of the grid based on the data in the grid array.
2. It adds the 'dead' class to cells with a value of 0, giving them a red background, and removes the class for live cells.

toggleCellState Function:

1. This function is called when a cell is clicked, toggling its state between alive (1) and dead (0).
2. It updates the grid array and calls updateGrid to reflect the change visually.

simulateGeneration Function:

1. This function simulates one generation of the Game of Life.
2. It uses a timer to periodically update the grid.

3. Two nested loops iterate through each cell of the grid to calculate the number of live neighbors for each cell using the countLiveNeighbors function.
4. If a cell has exactly 3 live neighbors, it is marked as alive (set to 1) in the newGrid, simulating the rules of the Game of Life.
5. After processing all cells, the grid is updated with the newGrid to represent the next generation.

Start and Stop Buttons:

1. There are two buttons on the web page, labeled "Start" and "Stop."
2. The "Start" button triggers the simulateGeneration function when clicked, initiating the simulation.
3. The "Stop" button allows you to stop the simulation.

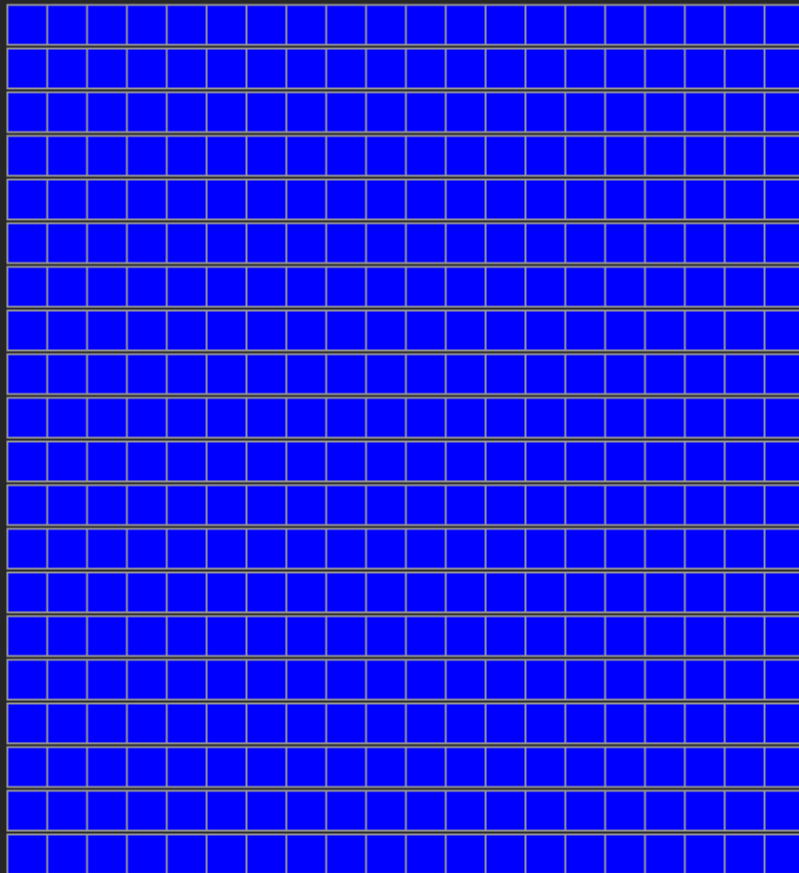
Initialization:

1. The createGrid function is called to set up the grid and initialize it with all cells alive.
2. Event listeners are attached to the buttons to enable starting and stopping the simulation.

OUTPUT:

BEFORE GENERATION – INITIAL STATE:

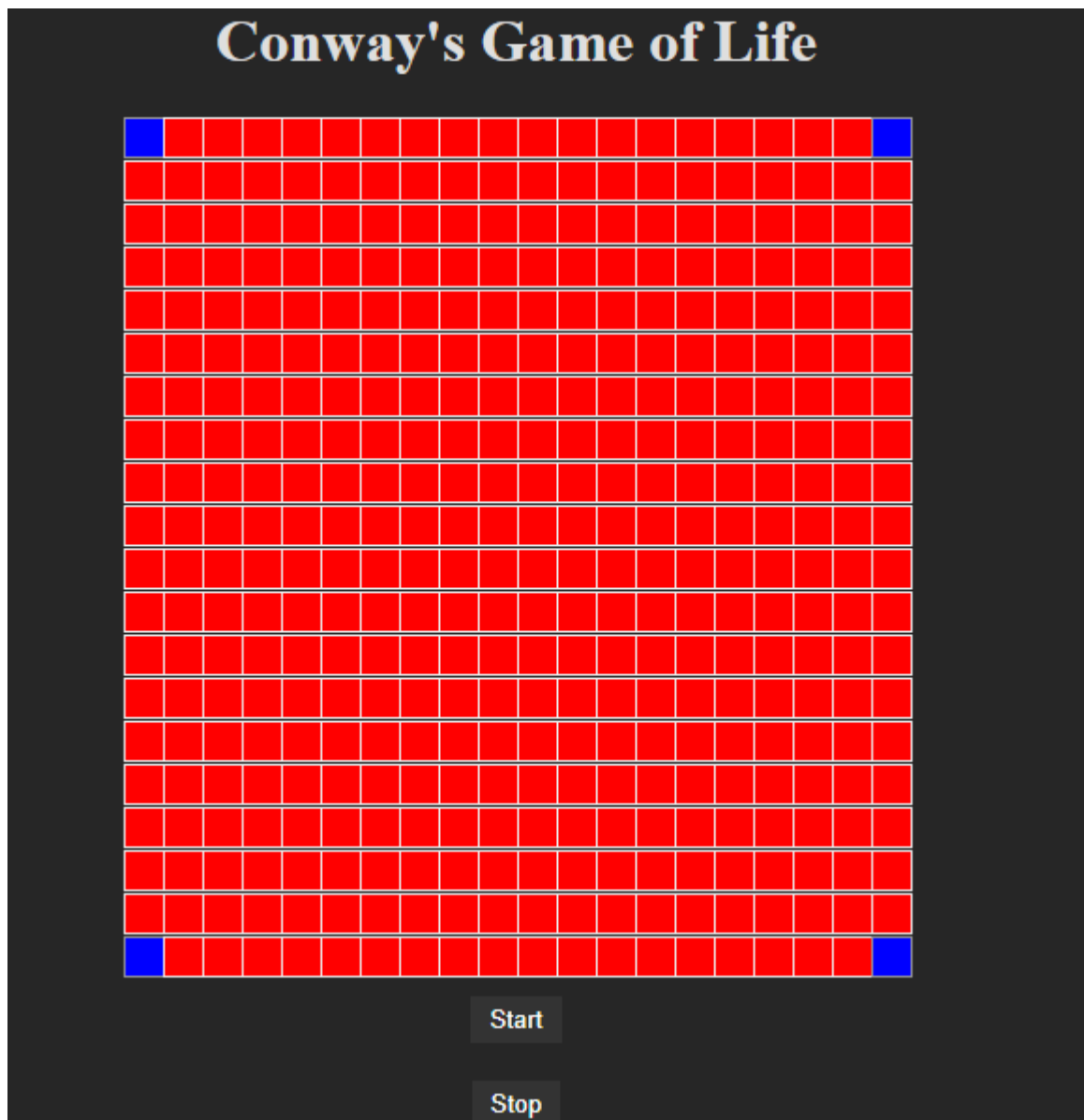
Conway's Game of Life



Start

Stop

AFTER GENERATION – FINAL STATE:



\\ THIS JAVASCRIPT CODE GENRATES THE INTERFACE AND FULLFILS THE LOGIC OF THE CONWAY’S GAME OF LIFE . THUS COMPLETING THE PRACTICAL IMPLEMENTATION OF CONWAY’S GAME OF LIFE.

CONCLUSION

Conway's Game of Life is a remarkable example of a simple concept leading to profound complexity. Its influence extends beyond mathematics and computer science into art and practical applications. This report only scratches the surface of its potential and the fascinating patterns it can produce. It continues to captivate minds and inspire creativity.