Programa el juego de la vida

Marco Jair Mendoza Bernal

April 2025

1. Introducción

El **Juego de la Vida de Conway** es un autómata celular ideado en 1970 por el matemático británico John Horton Conway. Este modelo constituye un sistema dinámico discreto bidimensional que evoluciona según reglas simples, dando lugar a comportamientos sorprendentemente complejos y diversos patrones emergentes. El interés principal del Juego de la Vida radica en su capacidad para simular fenómenos relacionados con sistemas complejos, autoorganización y computación emergente, a pesar de operar sobre reglas deterministas y extremadamente sencillas.

El espacio de simulación en el Juego de la Vida está representado por una malla rectangular infinita o finita compuesta por células que pueden encontrarse en uno de dos estados: *viva* o *muerta*. Cada célula interactúa únicamente con sus ocho vecinas inmediatas (adyacentes y diagonales). En cada iteración del sistema (llamada generación), el estado de cada célula se actualiza simultáneamente de acuerdo con las siguientes reglas clásicas (B3/S23):

1. Una célula **muerta** con exactamente 3 vecinas vivas, nace (se convierte en viva).

2. Una célula **viva** con 2 o 3 vecinas vivas sobrevive a la siguiente generación.

3. En cualquier otro caso, la célula muere por aislamiento (menos de 2 vecinas vivas) o por sobrepoblación (más de 3 vecinas vivas).

Estas reglas elementales permiten la generación de patrones complejos, que pueden ser estáticos (bloques, panales), oscilatorios (faros, péndulos), o incluso capaces de trasladarse a través del espacio celular (planeadores, naves espaciales). La popularidad y el estudio profundo del Juego de la Vida derivan

en gran medida de su aparente simplicidad y la riqueza de los fenómenos que se manifiestan durante su evolución.

La presente práctica tiene por objetivo desarrollar una herramienta computacional que permita explorar y simular el comportamiento del Juego de la Vida mediante la implementación de autómatas celulares bidimensionales.

A continuación, en este reporte se presentan los detalles de la implementación realizada, así como los resultados obtenidos al ejecutar simulaciones con distintas reglas y configuraciones iniciales.

Palabras clave: Autómatas celulares, Juego de la Vida, Conway, simulación computacional, sistemas complejos.

2. Objetivos

El objetivo general de esta práctica es implementar un simulador gráfico del Juego de la Vida de Conway, el cual cumpla con las siguientes especificaciones técnicas detalladas:

- Capacidad para manejar espacios celulares desde 500x500
- Edición directa del espacio celular para agregar o eliminar células.
- Zoom ajustable para visualizar detalles específicos del autómata.
- Guardado y recuperación de configuraciones en archivos externos.
- Personalización visual, incluyendo el cambio dinámico del color de los estados celulares.
- Visualización gráfica continua del conteo poblacional, así como gráficos con densidades logarítmicas.
- Ejecución de simulaciones bajo diferentes reglas (B/S), incluyendo específicamente B3/S23 y B2/S7.
- Selección de condiciones de frontera (nula o toroidal).

Es importante mencionar que desafortunadamente no fue posible cumplir con todos estos requerimientos. De igual forma, en las siguientes secciones se abordan detalladamente los aspectos técnicos, resultados obtenidos, conclusiones del trabajo, así como el código fuente utilizado en la implementación.

3. Desarrollo

En esta sección se describe el proceso de implementación del simulador del Juego de la Vida utilizando tecnologías web, específicamente HTML, JavaScript y las librerías `PIXIJS` y `PIXI-VIEWPORT`. Este simulador permite la visualización y manipulación interactiva de las células en una cuadrícula bidimensional, siguiendo las reglas clásicas del Juego de la Vida de Conway.

3.1. Librerías Utilizadas

Se utilizaron las siguientes librerías para desarrollar la simulación:

- `PIXIJS`: Esta es una librería de gráficos 2D basada en WebGL que permite la creación de aplicaciones interactivas y gráficas de alto rendimiento (en la teoria).
- `PIXI-VIEWPORT`: Esta extensión de `PIXIJS` se usa para crear un sistema de desplazamiento y zoom en el área de simulación, permitiendo que el usuario navegue por la cuadrícula de manera eficiente.

3.2. Estructura General del Simulador

El simulador utiliza una cuadrícula bidimensional donde cada celda puede estar en uno de dos estados: viva o muerta. El programa sigue las reglas estándar del Juego de la Vida de Conway para la evolución de las células. A continuación, se detallan las funciones clave implementadas en el simulador.

3.3. Inicialización del Simulador

El simulador se inicializa creando un contenedor principal con `PIXIJS` y configurando la cuadrícula de células:

- Se define el tamaño de la célula (`CELL_SIZE`) y el color predeterminado de las células vivas.
- La cuadrícula es representada por un conjunto de objetos `Sprite` de `PIXIJS` que representan las células, y se usa un `Container` para organizarlas y controlarlas.

Además, se establece un sistema de historial que almacena el estado de las células en cada iteración del juego. Esto permite navegar entre las generaciones pasadas y simular el comportamiento de las células a lo largo del tiempo.

3.4. Generación de la Cuadrícula

La función `buildGrid` se encarga de crear la cuadrícula visual, dibujando líneas en una `Graphics` de `PIXIJS` para representar las celdas muertas. Esta cuadrícula es escalable, permitiendo que el usuario ajuste el tamaño de las células mediante un sistema de zoom interactivo.

3.5. Manejo de Celdas

Cada celda es representada como un objeto de tipo `Sprite` de `PIXIJS`. El programa gestiona su estado de visibilidad a lo largo de las iteraciones del juego, mostrándolas como celdas vivas o muertas en función de las reglas del Juego de la Vida.

La interacción con las células se realiza mediante eventos de clic, utilizando el evento `mousedown` en cada celda para alternar su estado. Esta función es implementada en `onTap`, que cambia la visibilidad de la celda al hacer clic en ella.

3.6. Reglas de Evolución

La función `siguienteIteracion` es responsable de calcular la siguiente generación según las reglas de Conway. Para cada celda, la función cuenta el número de vecinos vivos y aplica las siguientes reglas:

- Si una celda está viva y tiene 2 o 3 vecinos vivos, sobrevive.
- Si una celda está muerta y tiene exactamente 3 vecinos vivos, revive.
- En cualquier otro caso, la celda muere (por aislamiento o sobrepoblación).

Estas reglas se implementan dentro de un ciclo que recorre toda la cuadrícula y actualiza el estado de cada celda en función de los vecinos vivos.

3.7. Interactividad y Control del Usuario

El simulador en su fase actual permite que el usuario controle la simulación a través de varios botones que se asignan a funciones específicas, como iniciar una iteración paso a paso, iniciar una simulación continua, limpiar el espacio celular o cambiar el color de las células. Estos botones están vinculados a los siguientes eventos:

- `stepBtn`: Avanza un paso en la simulación.
- `playBtn`: Inicia la simulación continua.
- `stopBtn`: Detiene la simulación continua.
- `clearBtn`: Limpia el espacio de células y reinicia la simulación.
- `colorAlive`: Permite cambiar el color de las células vivas.

4. Imagen de la Aplicación

A continuación se muestra una captura de pantalla de la aplicación web del simulador del Juego de la Vida. En esta imagen se puede observar la cuadrícula bidimensional en la que se simulan las células, así como las herramientas interactivas para manipular el espacio de células y controlar la evolución de la simulación.

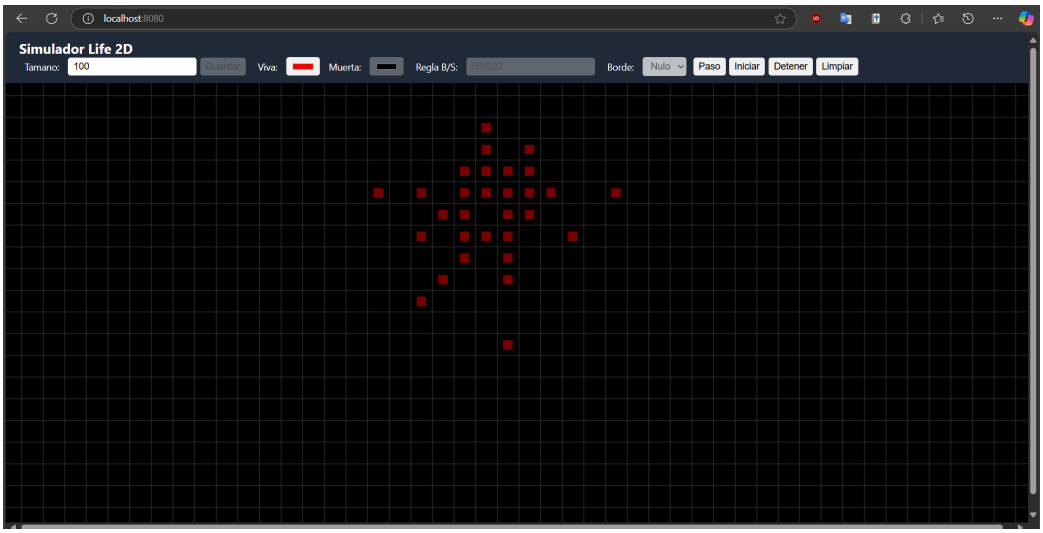

Figura 1: Captura de pantalla del simulador del Juego de la Vida.

5. Código Fuente y Repositorio en GitHub

A continuación se presenta una sección con el código fuente del simulador del Juego de la Vida. El código completo está disponible en mi repositorio de GitHub, lo que permite al lector acceder al proyecto completo y colaborar en su mejora.

5.1. Código Fuente

El código fuente implementado para la simulación del Juego de la Vida es el siguiente. Este código ha sido organizado en varias funciones que permiten la creación, actualización y visualización de las células, así como la interacción con el usuario:

```
import { Application, Container, Graphics, Sprite, Texture,Rectangle } from "pix
```

```
import { Viewport } from 'pixi-viewport'

////////////////////////////////////////////////////////////////////////////
//  Variables globales

const CELL_SIZE = 10;

let root = document.getElementById("simulador");
let simulador;
let grid;
let cells;
let conCells;
let btnCells;
let tam;
let col=0xFFFFFF;

// Iteraciones de la cuadricula, indicador del paso actual
let history=[];
let ind = 0;


let cellGraph = new Graphics()
                  .setFillStyle({ color: 'white',alpha:.4 })
                  .rect(0, 0, 1, 1)
                  .fill()
                  .stroke({ color: 0xffffff, pixelLine: true });
let cellContext;

// Textura de las celulas pre-creada para aumentar eficiencia
const baseTexture = Texture.WHITE;

////////////////////////////////////////////////////////////////////////////
//  Funciones para el simulador (acciones, funcionalidades)

// onClick de cada celula
function onTap(){
  history[history.length-1][this.i][this.j]=cells[this.i][this.j].visible=!cells
}

function buildGrid(graphics, cols, rows){
  graphics.clear();
```

```
  for (let i = 0; i < cols + 1; i++)
  {
      graphics
          .moveTo(i * (CELL_SIZE + 1), 0)
          .lineTo(i * (CELL_SIZE + 1), (CELL_SIZE + 1) * rows);
  }

  for (let i = 0; i < rows + 1; i++)
  {
      graphics
          .moveTo(0, i * (CELL_SIZE + 1))
          .lineTo((CELL_SIZE + 1)*cols, i * (CELL_SIZE + 1));
  }
  graphics.stroke({ color: 0xffffff, pixelLine: true, width: 1,alpha:.2 });

  // graphics.cacheAsTexture(false);
  // graphics.cacheAsTexture(true);

  return graphics;
}

function buildCells(conCells,cols,rows){
  for (let i = 0; i < cells.length; i++) {
    for (let j = 0; j < cells[i].length; j++) {
      cells[i][j].destroy();
      btnCells[i][j].destroy();
    }
  }

  history = [];
  ind = 0;

  cells = new Array(cols);
  btnCells = new Array(cols);
  let act = new Array(cols);
  for(let i = 0;i<cols;i++){
    cells[i] = new Array(rows);
    btnCells[i] = new Array(rows);
    act[i] = new Array(rows);

    for(let j=0;j<rows;j++){
```

```
      act[i][j] = 0;
      let cell = new Sprite(baseTexture);
      let cell2 = new Sprite();
      cell.alive = 1;

      cell.x=i * (CELL_SIZE + 1)+3;
      cell.y=j * (CELL_SIZE + 1)+3;

      cell.width = 5;
      cell.height = 5;
      cell.alpha = .5;
      cell2.alpha=0;
      cell.tint = (col);
      cell.visible=false;


      cell2.cursor='pointer';
      cell2.eventMode = 'static';

      cell2.x=cell.x-3;
      cell2.y=cell.y-3;
      cell2.width=cell2.height = CELL_SIZE;
      cell2.i=i;
      cell2.j=j;

      cell2.on('mousedown',onTap);

      conCells.addChild(cell,cell2);
      cells[i][j]=cell;
      btnCells[i][j]=cell2;

    }

  }

  history.push(act);

}

// Aparecer o desaparecer celulas de acuerdo a la iteracion actual
function actualizarCelulas(){
```

```
  for(let i = 0;i<tam;i++){
    for(let j=0;j<tam;j++){
        cells[i][j].visible=history[ind][i][j];
    }
  }
}

// Cambiar color de las celulas
function cambiarColor(color){
  for(let i = 0;i<tam;i++){
    for(let j=0;j<tam;j++){
        cells[i][j].tint=color;
    }
  }
}

// Generar siguiente iteracion
function siguienteIteracion(history, tam) {
  // Obtener la última matriz de la historia (la matriz actual)
  const matriz = history[history.length - 1];

  let siguienteMatriz = Array.from({ length: tam }, () => Array(tam).fill(0));

  // Función para contar los vecinos vivos de una celda
  function contarVecinosVivos(fila, col) {
      let contador = 0;
      // Verificar las 8 direcciones alrededor de la celda (esquinas incluidas)
      for (let i = -1; i <= 1; i++) {
          for (let j = -1; j <= 1; j++) {
              if (i === 0 && j === 0) continue; // No contar la propia celda
              const nuevaFila = fila + i;
              const nuevaCol = col + j;
              if (nuevaFila >= 0 && nuevaFila < tam && nuevaCol >= 0 && nuevaCol
                  if (matriz[nuevaCol][nuevaFila] === true) {
                      contador++;
                  }
              }
          }
      }
      return contador;
  }
```

```
  // Recorrer cada celda de la matriz actual y calcular su siguiente estado
  for (let col = 0; col < tam; col++) {
    for (let fila = 0; fila < tam; fila++) {
          const vecinosVivos = contarVecinosVivos(fila, col);

          // Si la celda es viva
          if (matriz[col][fila] === 1) {
              // La celda sobrevive si tiene 2 o 3 vecinos vivos
              if (vecinosVivos === 2 || vecinosVivos === 3) {
                  siguienteMatriz[col][fila] = true;
              } else {
                  siguienteMatriz[col][fila] = false;
              }
          }
          // Si la celda está muerta
          else {
              // La celda revive si tiene exactamente 3 vecinos vivos
              if (vecinosVivos === 3) {
                  siguienteMatriz[col][fila] = true;
              }
          }
      }
  }

  history.push(siguienteMatriz);

  ind++;
  return siguienteMatriz;
}

function reiniciarSimulador(){
  buildGrid(grid, tam, tam);
  buildCells(conCells,tam,tam);

  simulador.x=root.clientWidth/2;
  simulador.y=root.clientHeight/2;

  simulador.pivot.x= simulador.width/2;
  simulador.pivot.y= simulador.height/2;
}
```

```
////////////////////////////////////////////////////////////////////////////
//  Inicializacion del simulador

(async () => {
    const app = new Application();
    globalThis.__PIXI_APP__ = app;

    await app.init({ resizeTo: root, preference:"webgpu" });
    app.ticker.maxFPS = 20;
    app.ticker.minFPS = 1;

    // create viewport
  const viewport = new Viewport({
    screenWidth: app.canvas.width,
    screenHeight: app.canvas.height,
    worldWidth: 1000,
    worldHeight: 1000,
    stopPropagation:true,
    passiveWheel:false,
    events: app.renderer.events, // the interaction module is important for whee
  });

// add the viewport to the stage
app.stage.addChild(viewport);

// activate plugins
viewport.drag().wheel();

  simulador = new Container();
  let gridContainer = new Container({isRenderGroup:true});
  tam = 100;
  grid = buildGrid(new Graphics(),100,100);
  gridContainer.addChild(grid);
  grid.eventMode = "static";
  grid.hitArea = new Rectangle(0, 0, grid.width,grid.height);
  grid.on("pointerdown",(event)=>{
    let local = grid.toLocal(event);
    console.log(local.x,local.y);
  });
```

```
  simulador.addChild(gridContainer);
  viewport.addChild|(simulador);

  simulador.x=root.clientWidth/2;
  simulador.y=root.clientHeight/2;

  simulador.pivot.x= simulador.width/2;
  simulador.pivot.y= simulador.height/2;


  conCells = new Container();
  simulador.addChild(conCells);

  cellContext = app.renderer.generateTexture(cellGraph);

  cells = [];
  buildCells(conCells,100,100);

// add a red box
  viewport.addChild(simulador);

  root.appendChild(app.canvas);
})();


////////////////////////////////////////////////////////////////////////////
//  Listeners

document.getElementById("num").addEventListener('change', (e) => {
  tam = parseInt(e.target.value);;

  reiniciarSimulador();
});


document.getElementById("colorAlive").addEventListener("change", function() {
  console.log("Color cambiado.");
  let nuevo = parseInt(this.value.substring(1),16);
```

```
  cambiarColor(nuevo);
  col=nuevo;

});

// Paso único
document.getElementById("stepBtn").addEventListener("click", function() {
  console.log("Iteración paso a paso ejecutada.");
  siguienteIteracion(history,tam);

  actualizarCelulas();

});

let intervalId = null;

// Iniciar simulación continua
document.getElementById("playBtn").addEventListener("click", function() {
  console.log("Simulación iniciada.");
  if (!intervalId) { // Si no se está ejecutando ya
    intervalId = setInterval(() => {
      siguienteIteracion(history,tam);
      actualizarCelulas();
    }, 200); // 200 ms por generación
    console.log("Simulación iniciada.");
  }
});

// Detener simulación
document.getElementById("stopBtn").addEventListener("click", function() {
  console.log("Simulación detenida.");
  if (intervalId) {
    clearInterval(intervalId);
    intervalId = null;
    console.log("Simulación detenida.");
  }
});

// Limpiar espacio celular
document.getElementById("clearBtn").addEventListener("click", function() {
  console.log("Espacio limpiado.");
```

```
  reiniciarSimulador();
});

// Guardar configuración
document.getElementById("saveBtn").addEventListener("click", function() {
  console.log("Guardando configuración.");
});

// Cargar archivo
/*document.getElementById("load").addEventListener("change", function(event) {
  const file = event.target.files[0];
  if (file) {
    console.log("Archivo seleccionado:", file.name);
  }
});*/
```

5.2. Repositorio en GitHub

El código completo, así como los archivos adicionales relacionados con el proyecto, se pueden encontrar en el siguiente repositorio de GitHub:

`https://github.com/MarcoMendoza1/Cellular-automaton`

6. Conclusiones

La implementación del Juego de la Vida en una aplicación web interactiva permite explorar la evolución de sistemas complejos de manera eficiente y accesible. Gracias a la flexibilidad de `PIXIJS` y `PIXI-VIEWPORT`, la simulación es escalable y visualmente atractiva, permitiendo un análisis detallado del comportamiento de las células bajo diferentes condiciones iniciales y reglas de evolución.

Además, la interactividad del simulador hace que el usuario pueda experimentar directamente con diferentes configuraciones y observar en tiempo real cómo las reglas del Juego de la Vida generan patrones complejos y fascinantes.

7. Referencias

1. Gardner, M. (1971). *Mathematical games: on cellular automata, self-reproduction, the Garden of Eden and the game "Life"*. Scientific Ame-

rican, 224, 112-117.

2. Adamatzky, A. (Ed.). (2010). *Game of Life Cellular Automata*. London: Springer.

3. Martínez, G. J., Adamatzky, A., & McIntosh, H. V. (2010). *Localization dynamics in a binary two-dimensional cellular automaton: the Diffusion Rule*. Journal of Cellular Automata, 5(4-5), 289-313.

4. Martínez, G. J., Adamatzky, A., & Seck-Tuoh-Mora, J. C. (2022). *Some Notes About the Game of Life Cellular Automaton*. En: The Mathematical Artist (pp. 93-104). Springer.

5. The Game of Life Sites. Disponible en: `https://www.comunidad.escom.ipn.mx/genaro/Ce` (Consultado en Abril, 2024).