Conways Game of Life implemented in React by Jonathan Jamel Holloway

The rules of the game that I will implement are as follows:

* If a cell is alive and it has exactly 2 or 3 living neighbors, it stays alive
* If a cell is dead and it has exactly 3 living neighbors, it rises again

The hardest part I believe is the start. How will the data structure that holds the grid be implemented? What type of data structure will it be? When I first thought about this I thought about a double linked list that some how featured a top and a bottom. Because of the rules of the game all of these parts have to be accessible to check for living neighbors. I decided to use an object of arrays.

0: [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],

1: [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],

2: [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],

3: [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],

4: [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],

5: [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],

6: [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],

7: [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],

8: [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],

9: [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],

10: [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],

11: [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],

12: [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],

13: [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],

14: [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],

}

The reason why I chose this is because by incrementing and decrement I can access the left and right as well as the top and the bottom. I started off by creating a grid that would allow me to test how it would look and once I felt it looked good I modified the state so that I could adjust the size of the grid. Now as I continue I have modified some of this to fit a better style of explaining how I implemented though I used some of my original notes as I created the program. Writing down your thoughts is a good way to architect your program. The program will be implemented in React which means we can have a state. To begin this is how my state looks:

state = {

matrix: {},//using this for the grid.

//grid set up

matrixMirror: {},//using for clearing the grid will be set to a all dead cell grid.

matrixUsing: [],//using this to iterate over and show the grid on the screen.

mirror: [], // mirror for clearing the grid.

row\_count: 15, //default amount of rows on the grid can be changed.

col\_count: 15, //default amount of columns on the grid can be changed.

width: "330px", //width of the grid can be changed.

//presets

//cell choices

if\_one\_color: "black",// if one the cell is alive

if\_zero\_color: "white", // if zero the cell is dead

//game instructions

gameRunning: false,//used to show game has started.

generation: 0,// keeps count of generations

gameSpeed: 125 // default game speed can be changed.

};

Now that the matrix (‘grid’) is empty we will have to load the matrix up so that we can display it. For this purpose I created a function called set matrix up.

setMatrixUp = () => {

const matrixUsing = [];

let count = 0;

let beginRow = 0;

let beginColumn = 0;

const matrix = {};

while (beginRow < this.state.row\_count) {

beginColumn = 0;

matrix[beginRow] = [];

while (beginColumn < this.state.col\_count) {

matrix[beginRow].push(0);

beginColumn++;

}

beginRow++;

}

Object.entries(matrix).forEach(entry => {

for (let x of entry[1]) {

const temp\_hash = {

row: 0,

position\_in\_row: 0,

actual\_number: 0,

color: this.state.if\_zero\_color

};

temp\_hash.row = Number(entry[0]);

temp\_hash.position\_in\_row = count % this.state.col\_count;

temp\_hash.actual\_number = count;

temp\_hash.value = x; //just to get rid of the warning.

matrixUsing.push(temp\_hash);

count++;

}

});

const width\_size = 22 \* this.state.col\_count;

const width = `${width\_size}px`;

this.setState({

matrix,

matrixUsing,

width,

matrixMirror: matrix,

mirror: matrixUsing

});

return true; // means its finished.

};

I don’t believe there is much in this function that is difficult to understand but I will recap quickly. I first create nested while loops that will iterate based on the default settings of row count and column count on the state. I create an array and then I push zeros to the array. After this part I create the array, now as I am writing this I consider whether or not I could have created the array in the first two nested loops certainly this would have been more efficient. For the purpose of this article the variable names helps to break down the code a bit better. Object entries takes a object as an argument and then entry (can be whatever you want to call it) is an array of length two. The zero index is the key and the one index is the array. I then call this function in the component Will Mount function.

componentWillMount() {

this.setMatrixUp();

}

From here I created a way to turn the cell on and off.

manualTurnOnOrOff = (row, position\_in\_row) => {

//Just add 1 to the % of 2 it will provide 0 or 1. The conditional is already set up on the div to set the div to the correct class based off the value

if (this.state.gameRunning === false) {

const matrix = this.state.matrix;

// matrix[row][position\_in\_row] = matrix[row][position\_in\_row] === 0 ? this.state.if\_one\_color: this.state.if\_zero\_color;

matrix[row][position\_in\_row] = matrix[row][position\_in\_row] === 0 ? 1 : 0;

this.setState({ matrix });

}

};

I also have the reset and generate random functions:

reset = () => {

//resets gameRunning and generation count.

this.setState({ generation: 0, gameRunning: false });

this.updateRowCol();

};

generateRandom = () => {

//generates a random grid to start with.

const keys = Object.keys(this.state.matrix);

const matrix = { ...this.state.matrixMirror }; //creates a copy

for (let i = 0; i < keys.length; i++) {

for (let j = 0; j < matrix[i].length; j++) {

const grabRandom = Math.random();

const convertToNumber = grabRandom \* 2;

const floorNumber = Math.floor(convertToNumber);

if (floorNumber === 1) {

matrix[i][j] = 1;

} else {

matrix[i][j] = 0;

}

}

}

this.setState({ matrix });

this.continueWithGame();

};

The purpose of these functions is to allow for resetting the grid and game parameters. The generate random function allows a user to generate a random grid to start from.

You may notice the update row col function which looks like this:

updateRowCol = () => {

//this function will actually check if the value is acceptable and then make the change.

const rowValue = this.state.row\_count;

const colValue = this.state.col\_count;

if (rowValue < 15 || rowValue > 30) {

alert("Must be a numerical value of at least 15 and less than 30!");

return;

}

if (colValue < 15 || colValue > 30) {

alert("Must be a numerical value of at least 15 and less than 30!");

return;

}

this.setMatrixUp();

};

I have yet to change this but, I first thought about allowing the user to input the information in themselves. However, I changed the way the user changes the columns and row size but implementing a slider. The reason behind this is because the less the user has to input the less likely they are to do something you did not plan for. The rows can be changed and then a button clicked to resetup the matrix. The following functions assist with this:

handleChangeRow = event => {

//Will handle the changing of the rows.

//function allows for user to change the amount of rows that is being used.

this.setState({ [event.target.name]: event.target.value });

};

handleChangeColumn = event => {

//Will handle changing the columns.

//function allows user to change the amount of columns that is being used.

this.setState({ [event.target.name]: event.target.value });

};

I created two functions when in fact I could have used just one. The reason I used two is because I may want to do something different for a row then for a column. However, because of the way I set the state I could get away with using one function. A function however, should serve exactly one purpose so if you did have to do multiple things having a function do multiple things isn’t efficient. A lot of times when I program I plan for multiple things that I may not actually need later but in preparation for a change.

startTheGame = () => {

//Function will start the game.

console.log(this.state.gameRunning);

if (this.state.gameRunning === false) {

// let intervalRef = setInterval(this.runGamne(), 1000); will only run one time. Not a loop.

let intervalRef = setInterval(

() => this.runGamne(),

this.state.gameSpeed

);

this.setState({

gameRunning: true,

intervalRef

});

}

};

stopTheGame = () => {

// //Function will stop the game.

clearInterval(this.state.intervalRef);

this.setState(prevState => ({ gameRunning: !prevState.gameRunning }));

};

The next two functions is start game and stop game which uses setInterval and also clearInterval. Basically start game will call the run game function until the interval gets cleared.

runGamne = () => {

//functionaly for gameplay goes here

const matrix = this.state.matrixUsing.slice();

const state\_matrix = { ...this.state.matrix }; //creates a copy

let i = 0;

for (; i < matrix.length; i++) {

let aliveNeighbors = this.findLiveNeighbors({

row: matrix[i].row,

position\_in\_row: matrix[i].position\_in\_row

});

let current\_cell\_alive =

this.state.matrix[matrix[i].row][matrix[i].position\_in\_row] === 1

? true

: false;

if (current\_cell\_alive) {

if (aliveNeighbors < 2 || aliveNeighbors > 3) {

//kill the cell that is currently alive.

state\_matrix[matrix[i].row][matrix[i].position\_in\_row] = 0;

}

} else {

if (aliveNeighbors === 3) {

//resurrect the currently dead cell.

state\_matrix[matrix[i].row][matrix[i].position\_in\_row] = 1;

}

}

}

this.setState(prevState => ({

matrix: state\_matrix,

generation: prevState.generation + 1

}));

this.continueWithGame();

};

This is the run game function it is also the next generation function. The reason for two is because the run game function will be running until the clear interval. The next generation will only run when the next button is clicked. Therefore run game can just feature a call to the next generation function.

Noticably in the run game and next generation functions there is a find live neighbors function being called.

findLiveNeighbors = position => {

//find neighbors of the live cell to use the rules on.

//position should be an object featuring the row and position in row.

//A neighbor will be to the left to the right up down and diagonal which will be up to the left, up to the right and down to the left down to the right.

let totalAlive = 0;

let lookUp = true;

let lookDown = true;

let lookRight = true;

let lookLeft = true;

//logic for if I should look up or down.

if (position.row === 0) {

lookUp = false;

} else if (position.row === this.state.row\_count - 1) {

lookDown = false;

}

//logic for if I should look right or left.

if (position.position\_in\_row === 0) {

lookLeft = false;

} else if (position.position\_in\_row === this.state.col\_count - 1) {

lookRight = false;

}

if (lookUp) {

if (this.state.matrix[position.row - 1][position.position\_in\_row] === 1) {

totalAlive++;

}

}

if (lookDown) {

if (this.state.matrix[position.row + 1][position.position\_in\_row] === 1) {

totalAlive++;

}

}

if (lookRight) {

if (this.state.matrix[position.row][position.position\_in\_row + 1] === 1) {

//next in line plus one//2

totalAlive++;

}

}

if (lookLeft) {

if (this.state.matrix[position.row][position.position\_in\_row - 1] === 1) {

//4

totalAlive++;

}

}

if (lookUp && lookRight) {

if (

this.state.matrix[position.row - 1][position.position\_in\_row + 1] === 1

) {

//3

totalAlive++;

}

}

if (lookUp && lookLeft) {

//diagonal left up one minus one //5

if (

this.state.matrix[position.row - 1][position.position\_in\_row - 1] === 1

) {

totalAlive++;

}

}

if (lookDown && lookRight) {

//down one to the right (plus one) //7

if (

this.state.matrix[position.row + 1][position.position\_in\_row + 1] === 1

) {

totalAlive++;

}

}

if (lookDown && lookLeft) {

//down one to the left (minus one) //8

if (

this.state.matrix[position.row + 1][position.position\_in\_row - 1] === 1

) {

totalAlive++;

}

}

return totalAlive;

};

Basically, this function looks for the neighbors that are alive surrounding the cell.

Outside of the game there is the ability to change the color of the cells so I have included the following functions:

handleColorChangeIfZero = color => {

this.setState({ if\_zero\_color: color });

};

handleColorChangeIfOne = color => {

this.setState({ if\_one\_color: color });

};

I also have the following function to change grid based off presets presetChange = type => {

const finished = this.setMatrixUp(); //will reset the grid before setting it up.

let row\_index = 0;

let col\_index = 0;

if (finished) {

const matrix = { ...this.state.matrixMirror }; //creates a copy

if (Object.keys(matrix).length) {

switch (type) {

case "Block":

this.setMatrixUp(); //will reset the grid before setting it up.

row\_index =

this.state.row\_count % 2 === 0

? this.state.row\_count / 2

: (this.state.row\_count - 1) / 2;

col\_index =

this.state.col\_count % 2 === 0

? this.state.col\_count / 2

: (this.state.col\_count - 1) / 2;

matrix[row\_index][col\_index] = 1;

matrix[row\_index - 1][col\_index] = 1;

matrix[row\_index - 1][col\_index - 1] = 1;

matrix[row\_index][col\_index - 1] = 1;

this.setState({ matrix }, () => {

this.continueWithGame();

});

break;

case "Beehive":

row\_index =

this.state.row\_count % 2 === 0

? this.state.row\_count / 2

: (this.state.row\_count - 1) / 2;

col\_index =

this.state.col\_count % 2 === 0

? this.state.col\_count / 2

: (this.state.col\_count - 1) / 2;

matrix[row\_index][col\_index] = 1;

matrix[row\_index][col\_index - 1] = 1;

matrix[row\_index - 1][col\_index - 2] = 1;

matrix[row\_index - 2][col\_index] = 1;

matrix[row\_index - 2][col\_index - 1] = 1;

matrix[row\_index - 1][col\_index + 1] = 1;

this.setState({ matrix }, () => {

this.continueWithGame();

});

break;

case "Loaf":

row\_index =

this.state.row\_count % 2 === 0

? this.state.row\_count / 2

: (this.state.row\_count - 1) / 2;

col\_index =

this.state.col\_count % 2 === 0

? this.state.col\_count / 2

: (this.state.col\_count - 1) / 2;

matrix[row\_index][col\_index] = 1;

matrix[row\_index - 1][col\_index] = 1;

matrix[row\_index + 1][col\_index - 1] = 1;

matrix[row\_index][col\_index - 2] = 1;

matrix[row\_index - 1][col\_index - 3] = 1;

matrix[row\_index - 2][col\_index - 2] = 1;

matrix[row\_index - 2][col\_index - 1] = 1;

this.setState({ matrix }, () => {

this.continueWithGame();

});

break;

case "Boat":

row\_index =

this.state.row\_count % 2 === 0

? this.state.row\_count / 2

: (this.state.row\_count - 1) / 2;

col\_index =

this.state.col\_count % 2 === 0

? this.state.col\_count / 2

: (this.state.col\_count - 1) / 2;

matrix[row\_index - 1][col\_index] = 1;

matrix[row\_index - 1][col\_index - 2] = 1;

matrix[row\_index - 2][col\_index - 2] = 1;

matrix[row\_index - 2][col\_index - 1] = 1;

matrix[row\_index][col\_index - 1] = 1;

this.setState({ matrix }, () => {

this.continueWithGame();

});

break;

case "Tub":

row\_index =

this.state.row\_count % 2 === 0

? this.state.row\_count / 2

: (this.state.row\_count - 1) / 2;

col\_index =

this.state.col\_count % 2 === 0

? this.state.col\_count / 2

: (this.state.col\_count - 1) / 2;

matrix[row\_index - 1][col\_index] = 1;

matrix[row\_index - 1][col\_index - 2] = 1;

matrix[row\_index - 2][col\_index - 1] = 1;

matrix[row\_index][col\_index - 1] = 1;

this.setState({ matrix }, () => {

this.continueWithGame();

});

break;

case "Jon":

matrix[2][1] = 1;

matrix[2][2] = 1;

matrix[2][3] = 1;

matrix[3][3] = 1;

matrix[4][3] = 1;

matrix[5][3] = 1;

matrix[6][3] = 1;

matrix[6][2] = 1;

matrix[6][1] = 1;

matrix[6][0] = 1;

matrix[5][0] = 1;

//above is J

matrix[2][6] = 1;

matrix[2][7] = 1;

matrix[2][8] = 1;

matrix[3][6] = 1;

matrix[4][6] = 1;

matrix[5][6] = 1;

matrix[6][6] = 1;

matrix[6][7] = 1;

matrix[6][8] = 1;

matrix[5][8] = 1;

matrix[4][8] = 1;

matrix[3][8] = 1;

//above is o

matrix[2][11] = 1;

matrix[2][12] = 1;

matrix[2][13] = 1;

matrix[3][11] = 1;

matrix[4][11] = 1;

matrix[5][11] = 1;

matrix[6][11] = 1;

matrix[3][13] = 1;

matrix[4][13] = 1;

matrix[5][13] = 1;

matrix[6][13] = 1;

this.setState({ matrix }, () => {

this.continueWithGame();

});

break;

case "Blinker 1":

row\_index =

this.state.row\_count % 2 === 0

? this.state.row\_count / 2

: (this.state.row\_count - 1) / 2;

col\_index =

this.state.col\_count % 2 === 0

? this.state.col\_count / 2

: (this.state.col\_count - 1) / 2;

matrix[row\_index][col\_index] = 1;

matrix[row\_index - 1][col\_index] = 1;

matrix[row\_index + 1][col\_index] = 1;

this.setState({ matrix }, () => {

this.continueWithGame();

});

break;

case "Blinker 2":

row\_index =

this.state.row\_count % 2 === 0

? this.state.row\_count / 2

: (this.state.row\_count - 1) / 2;

col\_index =

this.state.col\_count % 2 === 0

? this.state.col\_count / 2

: (this.state.col\_count - 1) / 2;

matrix[row\_index][col\_index] = 1;

matrix[row\_index][col\_index - 1] = 1;

matrix[row\_index][col\_index + 1] = 1;

this.setState({ matrix }, () => {

this.continueWithGame();

});

break;

default:

console.log("That type doesn't exist"); //only for react warning purposes this won't actually hit.

}

}

}

};

There is also the ability to change the speed of the game from the default. In the start game function I use the state parameter game speed is the time interval.

decrementSpeed = () => {

if (this.state.gameSpeed > 125) {

this.setState(prevState => ({ gameSpeed: prevState.gameSpeed - 1 }));

}

};

incrementSpeed = () => {

if (this.state.gameSpeed < 2000) {

this.setState(prevState => ({ gameSpeed: prevState.gameSpeed + 1 }));

}

};

The speed choice ranges from 125 to 2000 milliseconds.

render() {

const matrix = this.state.matrixUsing.slice();

return (

<div className="container" style={{ width: this.state.width }}>

<h2 className="titleApp">Conway's Game of Life</h2>

<p>

Rules: • If a cell is alive and it has exactly 2 or 3 living

neighbors, it stays alive. • If a cell is dead and it has exactly 3

living neighbors, it rises again.

</p>

<h4 className="titleApp">Generation : {this.state.generation}</h4>

<div className="topButtons">

<div>

<button onClick={this.startTheGame}>Start</button>{" "}

</div>

<div>

<button onClick={this.stopTheGame}>Stop</button>

</div>

<div>

<button onClick={this.reset}>Clear</button>

</div>

<div>

<button onClick={this.generateRandom}>Random</button>

</div>

<div>

<button onClick={this.nextGeneration}>Next</button>

</div>

<br />

<br />

<br />

</div>

{matrix.map((hash, id) => (

<div

key={id}

onClick={() =>

this.manualTurnOnOrOff(hash.row, hash.position\_in\_row)

}

style={{

background:

this.state.matrix[hash.row][hash.position\_in\_row] === 0

? this.state.if\_zero\_color

: this.state.if\_one\_color,

color:

this.state.matrix[hash.row][hash.position\_in\_row] === 0

? this.state.if\_zero\_color

: this.state.if\_one\_color

}}

className={

this.state.matrix[hash.row][hash.position\_in\_row] === 0

? "offDiv gridDiv"

: "onDiv gridDiv"

}

>

{hash.position\_in\_row}

</div>

))}

<div>

<h5>Presets</h5>

<div className="presetsDiv">

<p onClick={() => this.presetChange("Block")}>Block</p>

<p onClick={() => this.presetChange("Beehive")}>Beehive</p>

<p onClick={() => this.presetChange("Loaf")}>Loaf</p>

<p onClick={() => this.presetChange("Boat")}>Boat</p>

<p onClick={() => this.presetChange("Tub")}>Tub</p>

<p onClick={() => this.presetChange("Jon")}>Jon</p>

</div>

<h5>Current Speed {this.state.gameSpeed}</h5>

<div className="presetsDiv">

<button onClick={this.decrementSpeed}>-</button>

<p>{this.state.gameSpeed}</p>

<button onClick={this.incrementSpeed}>+</button>

</div>

<div className="slidecontainer">

<h5>Row Count {this.state.row\_count}</h5>

<input

type="range"

min="15"

max="30"

value={this.state.row\_count}

className="slider"

id="myRange"

name="row\_count"

onChange={this.handleChangeRow}

/>

<h5>Col Count {this.state.col\_count}</h5>

<input

type="range"

min="15"

max="30"

value={this.state.col\_count}

className="slider"

id="myRange"

name="col\_count"

onChange={this.handleChangeColumn}

/>

<button onClick={this.updateRowCol}>Update Grid</button>

<br />

<h5> Dead Color: {this.state.if\_zero\_color}</h5>

<div className="colorChoices">

<div

className="divcolor white"

onClick={() => this.handleColorChangeIfZero("white")}

>

w

</div>

<div

className="divcolor black"

onClick={() => this.handleColorChangeIfZero("black")}

>

b

</div>

<div

className="divcolor blue"

onClick={() => this.handleColorChangeIfZero("blue")}

>

bl

</div>

<div

className="divcolor green"

onClick={() => this.handleColorChangeIfZero("green")}

>

g

</div>

<div

className="divcolor orange"

onClick={() => this.handleColorChangeIfZero("orange")}

>

o

</div>

<div

className="divcolor purple"

onClick={() => this.handleColorChangeIfZero("purple")}

>

pu

</div>

<div

className="divcolor pink"

onClick={() => this.handleColorChangeIfZero("pink")}

>

pk

</div>

<div

className="divcolor yellow"

onClick={() => this.handleColorChangeIfZero("yellow")}

>

y

</div>

</div>

<h5> Alive Color: {this.state.if\_one\_color}</h5>

<div className="colorChoices">

<div

className="divcolor white"

onClick={() => this.handleColorChangeIfOne("white")}

>

w

</div>

<div

className="divcolor black"

onClick={() => this.handleColorChangeIfOne("black")}

>

b

</div>

<div

className="divcolor blue"

onClick={() => this.handleColorChangeIfOne("blue")}

>

bl

</div>

<div

className="divcolor green"

onClick={() => this.handleColorChangeIfOne("green")}

>

g

</div>

<div

className="divcolor orange"

onClick={() => this.handleColorChangeIfOne("orange")}

>

o

</div>

<div

className="divcolor purple"

onClick={() => this.handleColorChangeIfOne("purple")}

>

p

</div>

<div

className="divcolor pink"

onClick={() => this.handleColorChangeIfOne("pink")}

>

pk

</div>

<div

className="divcolor yellow"

onClick={() => this.handleColorChangeIfOne("yellow")}

>

y

</div>

</div>

</div>

<br />

<br />

<br />

<br />

<br />

</div>

</div>

);

}

}

The last part is displaying everything to the screen using conditionals to make the choice for colors and class names for different styling.

In conclusion this was a fun project once I got everything implemented, there was tough parts to getting this project done. I enjoyed mostly about this project that I was able to implement it without using the canvas. If you look up this program I didn’t find many implementations without using the canvas. I also thought that this project really showcases what a coder can do. It is field with algorithms and opportunities to show your skills. More importantly writing this blog helped see where I could have cleaned up my code and improved. I am always trying to improve and I know this may not have been the best practices but mastery comes through the practice.