Conways Game of Life implemented in React by Jonathan Jamel Holloway

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? My approach is just that and maybe not the best practice however, I decided to use objects. To get to the previous node I could decrement the index as long as the index was not at 0. To get to the next node I could increment as long as the index isn’t the max which is 14 (if I have a 15 by 15 grid). I could use these same techniques to get to the top and bottom locations as long as I use integers as my keys. The reason why I am thinking of a direct access to these other locations is because of the rules.

From <https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life> 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 alive or dead, (or populated and unpopulated, respectively). Every cell interacts with its eight neighbors, 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 neighbors dies, as if by underpopulation.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

I also consider what we need done, initially there has to be a way for a grid cell to be turned on or off and for this to be represented.

In css I immediately create two classes:

.onDiv {

background: yellow;

}

.offDiv {

background: white;

}

For my state I am using arrays inside of a the state like so:

class App extends Component {

state = {

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],

}

render() {

return (

<div className="container">

</div>

);

}

}

export default App;

The next step is to create the grid by looping over the state. This is where I consider there to be a problem. What if I want more items on the state? It might be better if I made the arrays sit inside its own property on the state. So I made the following adjustment:

state = {

matrix: {

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 grid in my opinion just like most of this project could be implemented in many ways however I am going to use div’s initially. Div’s with borders to show that it is a grid that feature a conditional check and a function. First I will implement the grid.

To implement the grid I first run some tests on how to access each value inside of the matrix.

render() {

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

for (let x of entry[1]){

console.log(`|${x}|`);

this.state.count++;

}

});

console.log(`This is the count: ${this.state.count}`);

return (<div className="container">

</div>);

}

}

This successfully logs 15 x 15 which is the count. It also logs |0| 255 times. I now have a way to access each individual spot. However, this is just a test I have not actually created the div’s. I need to create a div that will hold a value the conditional and the function (I will go over what the conditional and function will be once we finish the grid).

setMatrixUp() {

const matrixUsing = [];

let count = 0;

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

for (let x of entry[1]) {

const temp\_hash = { row: 0, position\_in\_row: 0, actual\_number: 0, value : " "};

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

temp\_hash.position\_in\_row = count % 15;

temp\_hash.actual\_number = count;

matrixUsing.push(temp\_hash);

count++;

}

});

this.setState({ matrixUsing });

}

turnOnOrOff = (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

const matrix = this.state.matrix;

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

this.setState({ matrix });

};

I decided to create two functions that allow me to control the implementation of the grid. The reason for doing so involves when I have set up the game board. I will be using the componentWillMount() function and calling the setMatrixUp function.

componentWillMount() {

this.setMatrixUp();

}

What I am doing in the setMatrixUP function is I create a empty array “matrixUsing” I then use Object.entries to iterate over the keys and values of the this.state.matrix object. I would suggest looking up Object.entries if you do not know what it does however entry (you can call it something else if you like) will basically be an array with a length of 2. The first index is the key and the second index is the value. Inside of the forEach loop I have a nested for loop where I iterate over each item in entry[1] which is the actual array. From here I add a row and position in row as well as actual\_number and a value. Currently the value has no purpose and may get taken out but as I was architecting this out I felt I may need this at some point if I do need I just add a line of code, if I don’t I’ll just get rid of a line of code. What is important is the temp\_hash object placement, had I created temp\_hash outside of the loop I would get invalid data, the data would point to the same address in memory and give me the same results many times. Creating it inside of the loop tells JavaScript I need a new address in memory for each iteration.

Each cell needs the ability to turn on or off so for this I have used a arrow function. What is good about an arrow function is that it doesn’t require binding. I use the arrow function because the turnOnOrOff function will be called when the user clicks on a cell in the grid. The function is simple in that it uses a conditional if the matrix row position in the row is 0 make it 1 else make it 0. I am using 0 as an off switch and 1 as a on switch (this may change I may reverse it) however, 0 currently means the cell is white and 1 means the cell is black.

To implement the grid on the screen I have done the following:

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

return (

<div className="container">

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

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

<div

key={id}

onClick={() => this.turnOnOrOff(hash.row, hash.position\_in\_row)}

className={

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

? "offDiv gridDiv"

: "onDiv gridDiv"

}

>

{hash.position\_in\_row}

</div>

))}

<div>

<div>

<button>Start</button>{" "}

</div>

<div>

<button>Stop</button>

</div>

<div>

<button>Pause</button>

</div>

<div>

<button>Clear</button>

</div>

<div>

<button>Presets</button>{" "}

</div>

</div>

</div>

My CSS code currently looks like this:

.container {

width: 330px;

border: 3px solid blue;

display: flex;

flex-wrap: wrap;

}

.titleApp {

width: 100%;

text-align: center;

color: blue;

}

.gridDiv {

width: 20px;

border : 1px solid black;

}

.onDiv {

background: black;

color: black;

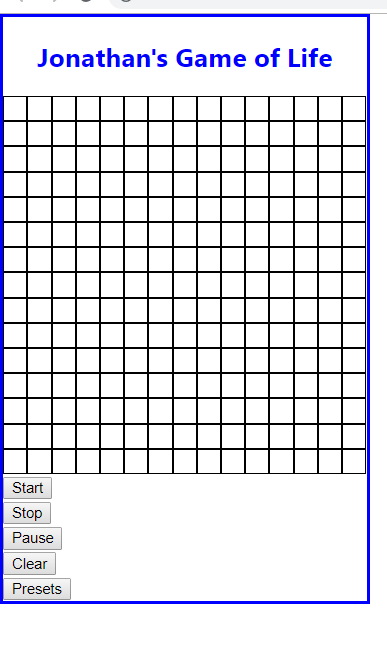
}

.offDiv {

background:white;

color: white;

}

All of this code produces the following application on the screen: 

Of course while it looks fairly decent enough it is brainless and does nothing. As I think about this application I begin to consider the presets may be smart to implement to begin to make sure I have decent enough logic to render the correct cells black and white on the on the screen. A start pointing would be nice the ability to stop, pause and continue as well as clear the screen. As you can see I have implemented these buttons to begin.

Thinking about the architecture of the application and the needs the following functions have been created with nothing inside of them but just a note that we will do something with them.

// Game functions being declared below this line.

handleChangeRow = (event) => {

//Will handle the changing of the rows.

}

handleChangeColumn = (event) => {

//Will handle changing the columns.

}

startTheGame = () => {

//Function will start the game.

}

stopTheGame = () => {

//Function will stop the game.

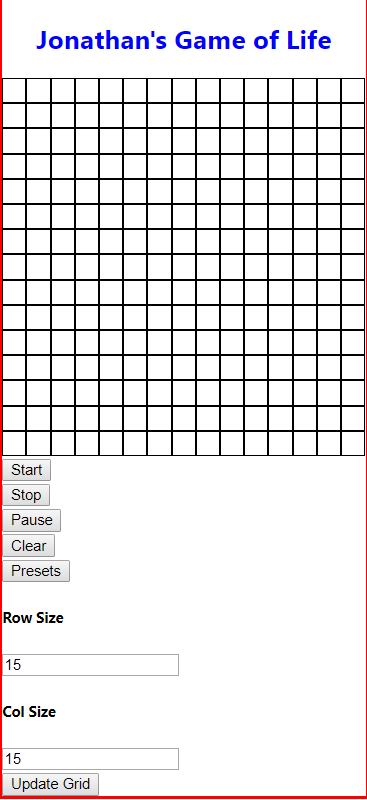
}

unPauseGame = () => {

//This function will unpause the game running where it left off.

//Will also pause the game if it is currently running.

}



This is what my game currently looks like after implementing one additional feature. The user can now select the number of rows and columns they desire and the grid will update. Of course there are limitations. Currently I don’t allow for a grid less than 15 or greater than 30 for either row or column. I am doing this with conditionals but am thinking about ways to avoid user error completely. Now to show the code and explain what I have done :

class App extends Component {

state = {

matrix: {

},

matrixUsing: [],

row\_count: 15,//default

col\_count: 15,//default

width: "330px",

};

componentWillMount() {

this.setMatrixUp();

}

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, value : " "};

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

temp\_hash.position\_in\_row = count % 15;

temp\_hash.actual\_number = count;

matrixUsing.push(temp\_hash);

count++;

}

});

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

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

this.setState({ matrix, matrixUsing, width });

}

turnOnOrOff = (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

const matrix = this.state.matrix;

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

this.setState({ matrix });

};

// Game functions being declared below this line.

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});

}

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();

}

startTheGame = () => {

//Function will start the game.

}

stopTheGame = () => {

//Function will stop the game.

}

unPauseGame = () => {

//This function will unpause the game running where it left off.

//This will pause the game if it is currently running.

}

render() {

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

return (

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

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

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

<div

key={id}

onClick={() => this.turnOnOrOff(hash.row, hash.position\_in\_row)}

className={

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

? "offDiv gridDiv"

: "onDiv gridDiv"

}

>

{hash.position\_in\_row}

</div>

))}

<div>

<div>

<button>Start</button>{" "}

</div>

<div>

<button>Stop</button>

</div>

<div>

<button>Pause</button>

</div>

<div>

<button>Clear</button>

</div>

<div>

<button>Presets</button>{" "}

</div>

<div>

<h5>Row Size</h5>

<input type="text" name = "row\_count" value = {this.state.row\_count} onChange = {this.handleChangeRow}/>

<h5>Col Size</h5>

<input type="text" name ="col\_count" value = {this.state.col\_count} onChange = {this.handleChangeColumn}/>

<br/>

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

</div>

</div>

</div>

);

}

}

export default App;

You will notice many changes one of those changes being in the setMatrixUp function. Here I have allowed for the creation of the matrix instead of just using the default matrix in the state. My thinking here was to get something implemented then adjust as we go along. Some of the most complex applications known to start off with scribble on a napkin! The only difference in my approach is I decided to scribble on the screen. Get something implemented to lead me down where my logic is leading me to. The only draw back to this approach you may find yourself updating your code a lot however if you look at the differences when I first started I left room for changes, small ones that could produce a greater affect. By using the matrix that was on the state already all I had to do is build a matrix then change the variable name and all of the code I created before still works. Slow but steady. In the event I am wrong I can go back to the drawing board.

Another change or rather addition is the updateRowCol, I added this and a button to use it with because I didn’t want to perform conditional checks on the changing for the input. Again I am thinking of a way I can handle this more intuitively maybe a drop box or a slider where you can only make the choices designed to be chosen. I have put my width for the container on the div itself and added a new property to the state. I did this because of course as the number of cells changes per col\_count so will the layout of the width of the grid is not updated. Currently the size of each cell is 20 however because of the borders it takes 22px for each cell so I am using simple math to calculate the width 22 \* col\_count. The height will not matter so the row\_count \* by a number is not needed. The next feature I would like to work on is setting some color changes. The user should be able to choose the color they desire.

The changes I made include the following parts of code. The first part is from the set matrix up function:

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 % 15;

temp\_hash.actual\_number = count;

matrixUsing.push(temp\_hash);

count++;

}

});

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

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

this.setState({ matrix, matrixUsing, width });

}

turnOnOrOff = (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

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 });

};

What I did was remove the value and add a color column that will set the column to the zero color. This was added to the state:

if\_one\_color : "black",

if\_zero\_color: "white",

In the turn on or off function I attempted to change the color there but this did not work because I don’t have this functionality within the cell itself. To make up for this I then made use of the matrix map functionality that I use to show the grid.

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

<div

key={id}

onClick={() => this.turnOnOrOff(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>

What I am doing here is checking the value of the row and position of that row and setting the color to the zero if it is equal to zero and then setting the color to the one if it is not equal to zero. This was a lot simpler (to me) than actually adding a property to the cell, why not make use of what the cell already has. I have also added the ability to change the color of the cells based off this same feature.

handleColorChangeIfZero = (color) => {

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

}

handleColorChangeIfOne = (color) => {

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

}

<h5>Color 0: {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>Color 1: {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>

The first part is the function logic and the second part is the options. Currently its just a bunch of colors next to each other and needs a bit more styling to look better but once I click on one of them it changes the colors. This is the only function I have that doesn’t completely reset the board. Remember we will have a pause function if I un-pause the board cannot reset. I have added the following to the state to help with that :

gameRunning: false,

If this is true we will do something a bit different either creating a new function or modifying the first one. So this is two of my required 3 features of the game my next feature may have to be implemented with the game so I am moving forward to the game logic.

startTheGame = () => {

//Function will start the game.

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

this.setState({

gameRunning: true,

}, () => {

this.intervalRef = setInterval(() => this.runGame(), 10);

})

}

}

stopTheGame = () => {

//Function will stop the game.

this.setState({

gameRunning: false

}, () => {

if(this.intervalRef){

clearInterval(this.intervalRef);

}

})

}

Here I have to functions set up for the game. We are either going to set the game running to true or false, start the intervalRef or clear it. Pretty simple. The runGame function is empty for now but I will go over this part soon.

Before diving into the logic and creating more functions a reminder of the rules seems wise:

1. Any live cell with fewer than two live neighbors dies, as if by underpopulation.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction

Along with these rules we don’t want to allow a user to be able to click on a cell during gameplay. This is actually very simple how I am going to take care of this. First, I am going to change the name of turn on or off and name it manually turn on or off. I will then add in a conditional check for if game running is false then I will do everything I normally do. Lastly, I will have to change the function name on the grid cell implementation. I have decided to keep the turn on or off function for now and I created a second manual turn on or off. The logic behind turning on or off is find I just want to set limits for when a user can do it hence the name.

**GAME LOGIC**

To make sure we are “playing” by the rules I will create two functions: find dead neighbors and find live neighbors these two functions will locate the neighbors of a dead or live cell. These functions will also feature one argument which you can call anything, but I will call position. Position will be an object featuring row and position in row as arguments. 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. Remember how the data structure is set up we will have to perform checks for where we are like if we are at row 14 there is no bottom neighbors, same with index 14 in the row it won’t have right side neighbors, neither will index 0 have a top or left side neighbors. Now starting off looking at our gird before anything is clicked 0 represents a dead cell. In binary 0 means off so 1 will mean alive and black by default unless the colors are changed. The two functions will calculate the total alive around them and return this number.