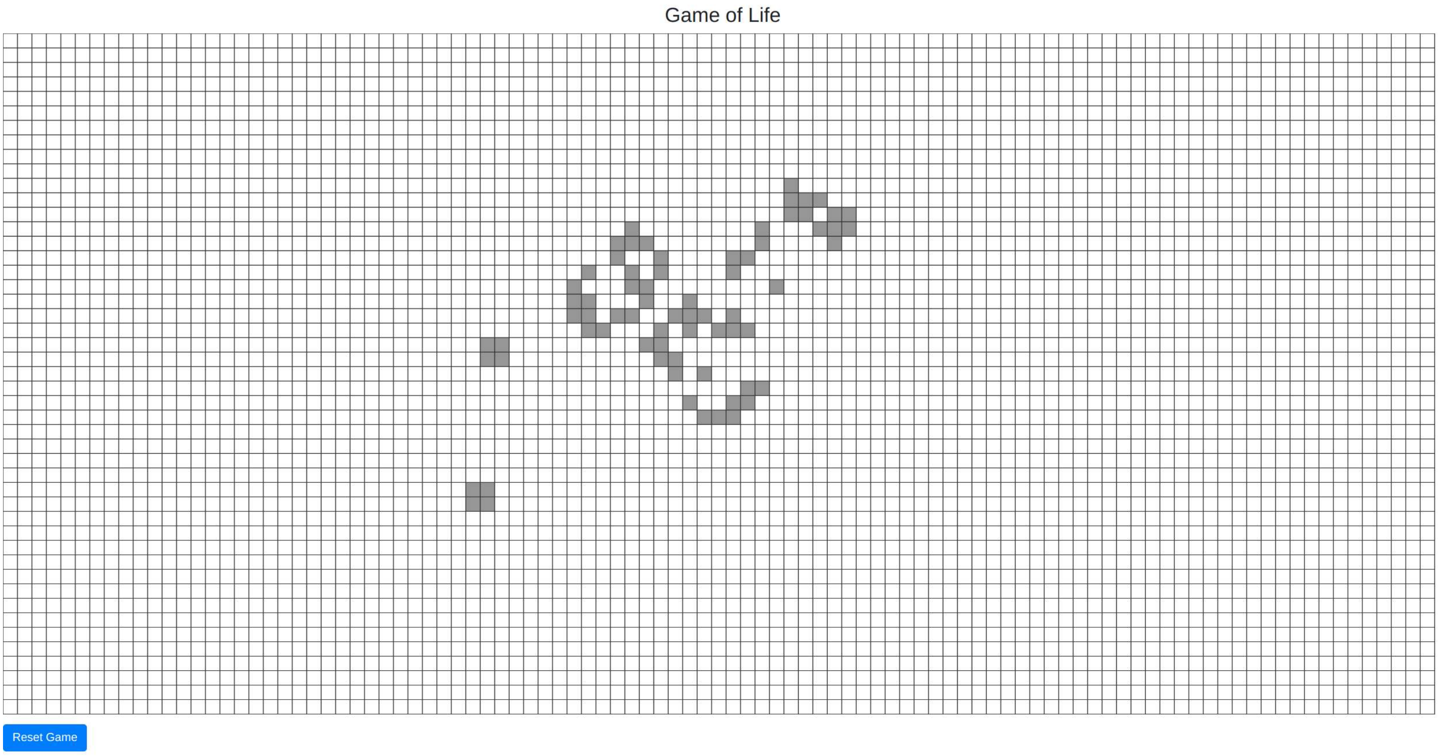
Game of Life Code-Along

We are going to build our game of life step by step today. Here is the resulting product that we are going to build within this section.



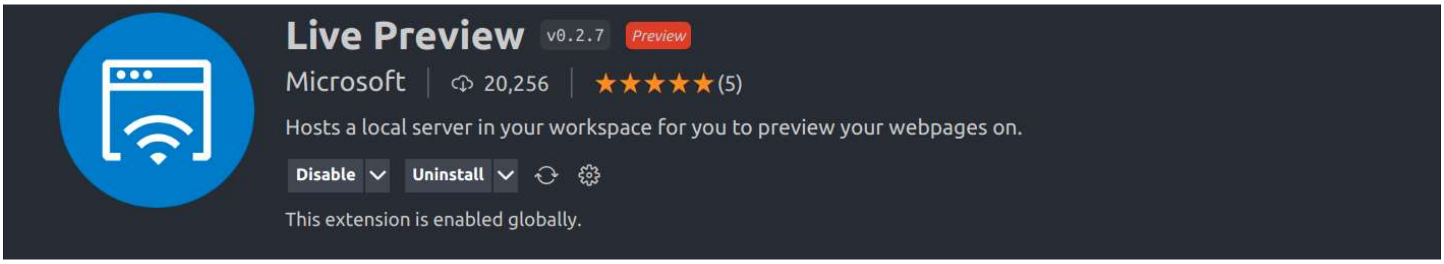
As you can see , we have a large 2-dimensional plane for our `lives` to thrive on. And we have a button called **Reset Game** at the bottom left corner.

Here are the list of features that we want to have in this game :

1. A 2-dimensional plane that allows the lives to reproduce, survive and die based on the rules we mentioned.
2. A **Reset Game** button that clears everything on the plane
3. Allowing user to click and drag to add new lives manually on the lifeless box.

Live Preview Extension

Before you proceed, we would like you to install an VSCode extension called [`Live Preview`].



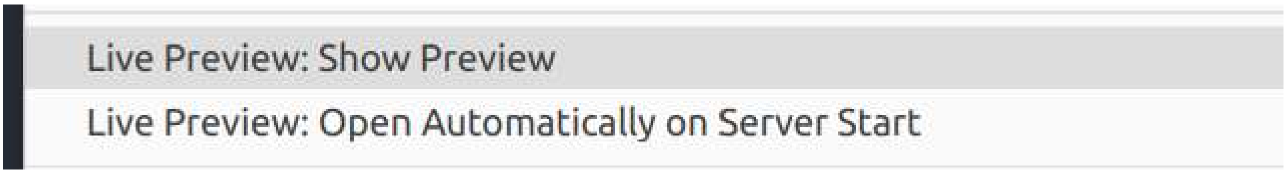
Live Preview v0.2.7 Preview

Microsoft | 20,256 | ★★★★★(5)

Hosts a local server in your workspace for you to preview your webpages on.

This extension is enabled globally.

With this extension, you should be able to serve HTML using the option in the right click menu directly. It makes working with `HTML`,`CSS` and `JavaScript` files much more easier.



State of the Game

Before we start writing anything, we need to consider what would be the state of the game. A state is all of the data that is necessary to represent the current game. We need to get the state of the game right because we would then need to update our user interface using the `state` of the game.

Here is the state of the game of life that we are going to implement.

> 🛠 **Warning:**
>
> Not all codes are require to copy from following sections. Some code sections are only for explains used. Please take care and watch carefully.

```
const unitLength  = 20;
const boxColor    = 150;
const strokeColor = 50;
let columns; /* To be determined by window width */
let rows;    /* To be determined by window height */
let currentBoard;
let nextBoard;
```

Here are the descriptions of all of the states:

1. **unitLength**: The width and height of a box.
2. **boxColor**: The color of the box.
3. **strokeColor**: The color of the stroke of the box.
4. **columns**: Number of columns in our game of life. It is determined by the width of the container and `unitLength`.
5. **rows**: Number of rows in our game of life. It is determined by the height of the container and `unitLength`.
6. **currentBoard**: The states of the board of the current generation.
7. **nextBoard**: The states of the board of the next generation. It is determined by **the current generation**.

Setup Function

Let's write the `setup` function for the initialization.

```
function setup(){
    /* Set the canvas to be under the element #canvas*/
```

```
    const canvas = createCanvas(windowWidth, windowHeight - 100);
    canvas.parent(document.querySelector('#canvas'));

    /*Calculate the number of columns and rows */
    columns = floor(width  / unitLength);
    rows    = floor(height / unitLength);

    /*Making both currentBoard and nextBoard 2-dimensional matrix that has (columns * rows
    currentBoard = [];
    nextBoard = [];
    for (let i = 0; i < columns; i++) {
        currentBoard[i] = [];
        nextBoard[i] = []
    }
    // Now both currentBoard and nextBoard are array of array of undefined values.
    init();  // Set the initial values of the currentBoard and nextBoard
}
```

Let's look at some of the `magic variables` here. The `magic variables` include `windowWidth`, `windowHeight`, `width` and `height`. They are all provided by p5.js to make our life easier.

- `windowWidth` and `windowHeight` are the width and height of the viewport.
- `width` and `height` are the width and height of the canvas element.

```
const canvas = createCanvas(windowWidth, windowHeight - 100);
canvas.parent(document.querySelector('#canvas'));
```

We are calling `createCanvas()` with `windowWidth` and `windowHeight - 100` to make a canvas that is as wide as the screen but `100 px` shorter than the height. We then use `.parent()` to insert our canvas element to the element with id `canvas`.

```
/*Calculate the number of columns and rows */
columns = floor(width  / unitLength);
rows    = floor(height / unitLength);
```

We can then calculate the columns and rows using the `width`,`height` and the `unitLength`. We need to use the `floor` function because there is no guarantee that the quotients would be an integer.

After that , we can simply initialize `currentBoard` and `nextBoard` to be an array of array. Then we run another function called `init` to initialize all the boxes' value to 0.

```
currentBoard = [];
nextBoard = [];
for (let i = 0; i < columns; i++) {
    currentBoard[i] = [];
```

```
    nextBoard[i] = []
}
// Now both currentBoard and nextBoard are array of array of undefined values.
init();  // Set the initial values of the currentBoard and nextBoard
```

Init function

Let's finish the `init()` function , the function is simple. We just need loop over both `currentBoard` and `nextBoard` to set all of the boxes' value to `0`.

```
/**
* Initialize/reset the board state
*/
function init() {
    for (let i = 0; i < columns; i++) {
        for (let j = 0; j < rows; j++) {
            currentBoard[i][j] = 0;
            nextBoard[i][j] = 0;
        }
    }
}
```

Upon loading the page, every box in the board are `0` now.

We can also use random input, for example we can use the random function to randomize initial state of `currentBoard`.

```
// let someVariables = <condictions> : <when_true> : <when_false>;
currentBoard[i][j] = random() > 0.8 ? 1 : 0; // one line if
nextBoard[i][j] = 0;
```

Draw Function

As mentioned before, the `draw()` function is being run for every single frame. Therefore, we need to draw the state of the current generation to the canvas inside `draw()` function.

```
function draw() {
    background(255);
    generate();
    for (let i = 0; i < columns; i++) {
        for (let j = 0; j < rows; j++) {
            if (currentBoard[i][j] == 1){
                fill(boxColor);
            } else {
```

```
                fill(255);
            }
            stroke(strokeColor);
            rect(i * unitLength, j * unitLength, unitLength, unitLength);
        }
    }
}
```

```
background(255);
generate();
```

In the first line, we set the background to white ((255,255,255) is the RGB code of white) with the function `background()`. Then we call the function `generate()` which calculates the next generation with current generation.

Within the nested for-loop, you can see we are checking if the `currentBoard[i][j] == 1`. It means that we are checking if the box has life. If `true`, then we set the filling color to the `boxColor` , else we set it to `white`. The stroke is set to `strokeColor`. Then we can call the `rect` function which conveniently use the configuration we just set (filling color is `boxColor` and stroke color is `strokeColor`) to make a `rect`. The parameters `i * unitLength, j * unitLength` sets the position of the top left corner of the rectangle and the parameters `unitLength, unitLength` set the size of the rectangle.

Generate function

Generate function contains the core business logic of game of life. It basically calculates the next generation solely with the information of the current generation. Let's look at the implementation of the `generate()` function first.

```
function generate() {
    //Loop over every single box on the board
    for (let x = 0; x < columns; x++) {
        for (let y = 0; y < rows; y++) {
            // Count all living members in the Moore neighborhood(8 boxes surrounding)
            let neighbors = 0;
            for (let i of [-1, 0, 1]) {
                for (let j of [-1, 0, 1]) {
                    if( i == 0 && j == 0 ){
                        // the cell itself is not its own neighbor
                        continue;
                    }
                    // The modulo operator is crucial for wrapping on the edge
                    neighbors += currentBoard[(x + i + columns) % columns][(y + j + rows)
                }
            }
```

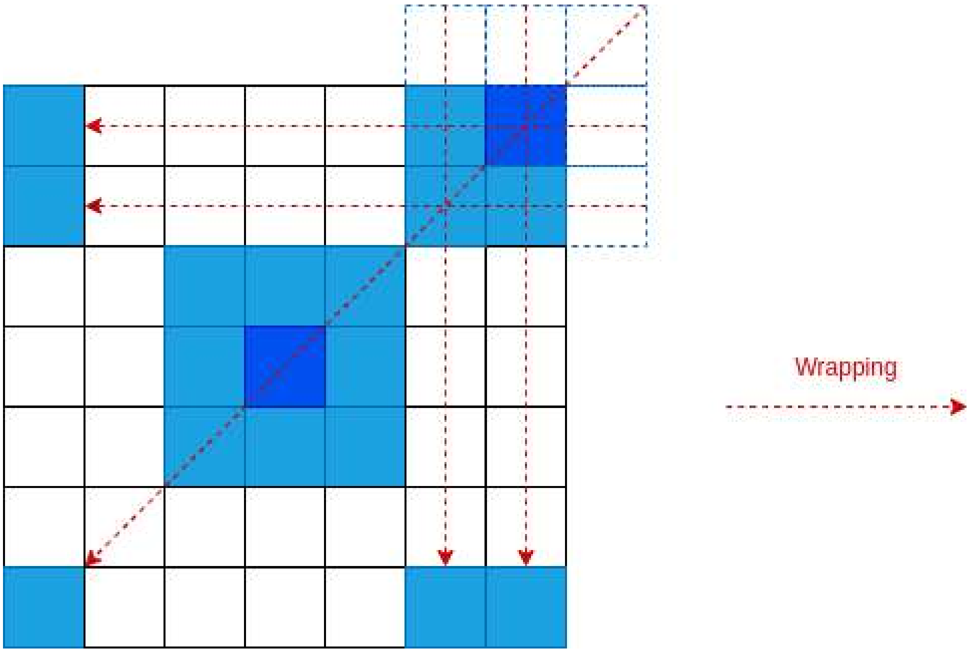
```
            // Rules of Life
            if (currentBoard[x][y] == 1 && neighbors < 2) {
                // Die of Loneliness
                nextBoard[x][y] = 0;
            } else if (currentBoard[x][y] == 1 && neighbors > 3) {
                // Die of Overpopulation
                nextBoard[x][y] = 0;
            } else if (currentBoard[x][y] == 0 && neighbors == 3) {
                // New life due to Reproduction
                nextBoard[x][y] = 1;
            } else {
                // Stasis
                nextBoard[x][y] = currentBoard[x][y];
            }
        }
    }

    // Swap the nextBoard to be the current Board
    [currentBoard, nextBoard] = [nextBoard, currentBoard];
}
```

Again , we are going to loop over the every single box in the board. Inside the for-loop , we need to first count the neighbors of each box.

```
// Count all living members in the Moore neighborhood(8 boxes surrounding)
let neighbors = 0;
for (let i of [-1, 0, 1]) {
    for (let j of [-1, 0, 1]) {
        if (i == 0 && j == 0) {
            // the cell itself is not its own neighbor
            continue;
        }
        // The modulo operator is crucial for wrapping on the edge
        neighbors += currentBoard[(x + i + columns) % columns][(y + j + rows) % rows];
    }
}
```

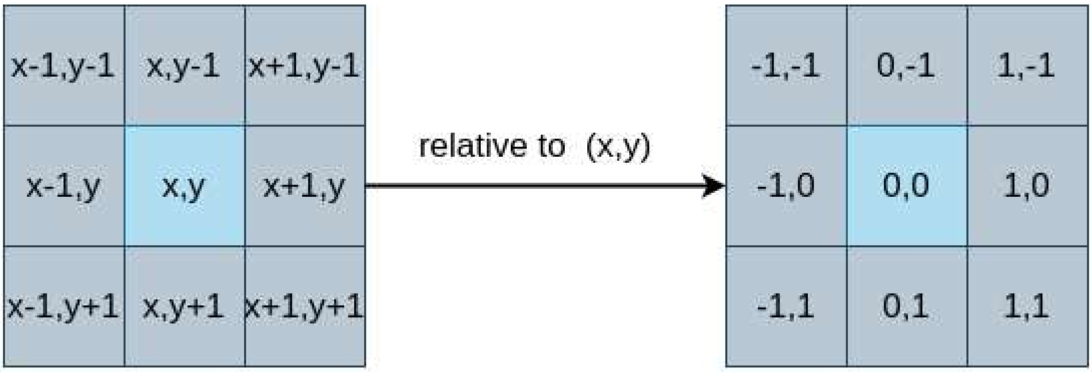
As you can we loop over all of the Moore neighborhoods. Except when `i == 0 && j == 0` (which is basically the box itself). We add the number of the neighbors' value. Since `0` represent lifeless, we will add `1` to `neighbors` every box with life. Note that we are using `(x + i + columns) % columns` and similar code in rows case. Because we don't want our lives hit the edge of our board, we would like them to wrap to the other side of the board.

The following diagram shows you how wrapping works in our board.



How wrapping actually works here ?

How neighbours are calculated ?

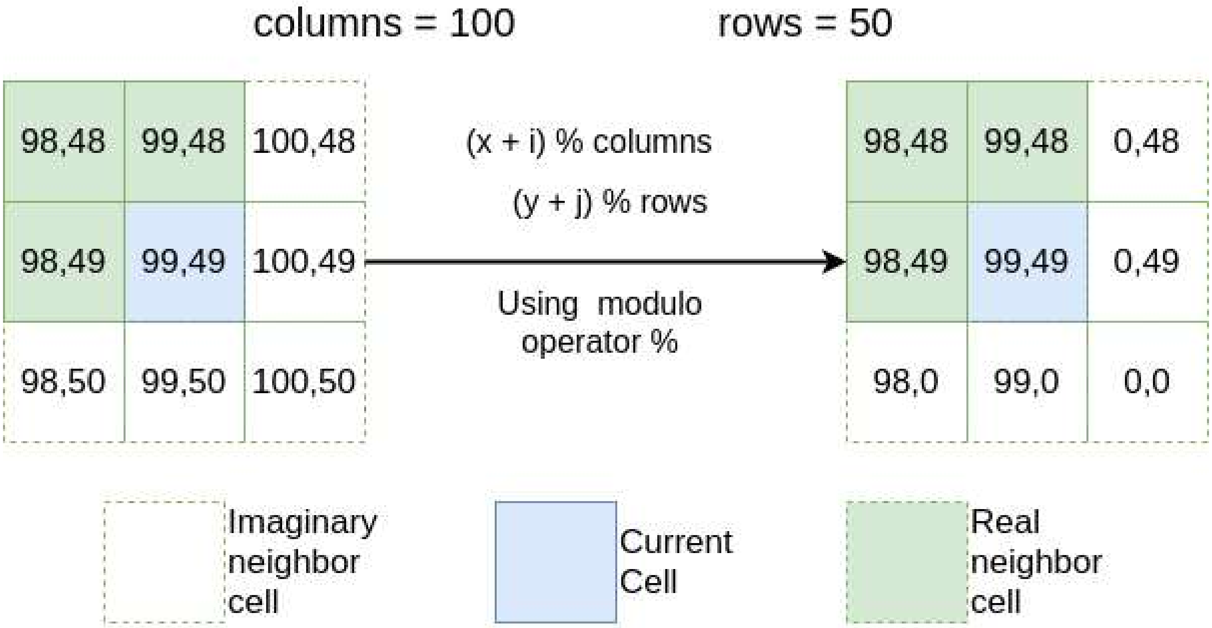


x-1,y-1	x,y-1	x+1,y-1
x-1,y	x,y	x+1,y
x-1,y+1	x,y+1	x+1,y+1

relative to (x,y)

-1,-1	0,-1	1,-1
-1,0	0,0	1,0
-1,1	0,1	1,1

Wrapping at the corners



columns = 100 rows = 50

98,48	99,48	100,48
98,49	99,49	100,49
98,50	99,50	100,50

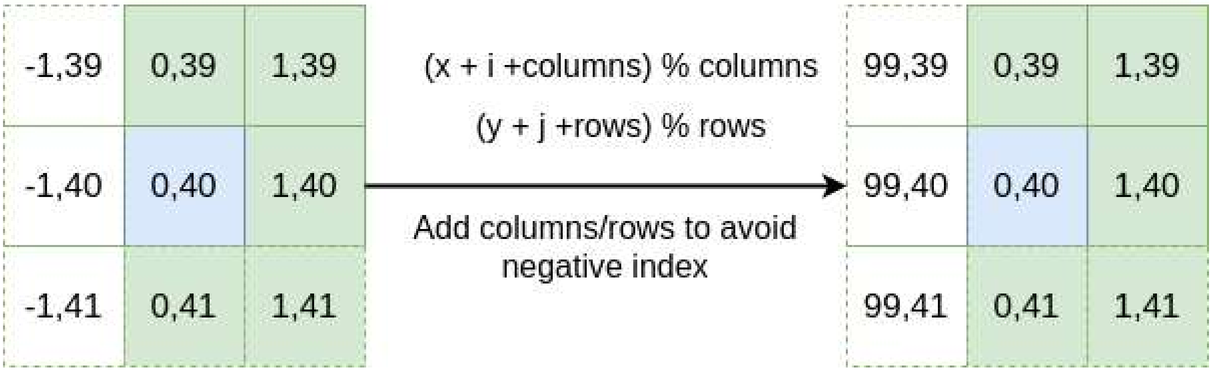
(x + i) % columns

(y + j) % rows

Using modulo operator %

98,48	99,48	0,48
98,49	99,49	0,49
98,0	99,0	0,0

Wrapping at the "0-side" edge



-1,39	0,39	1,39
-1,40	0,40	1,40
-1,41	0,41	1,41

(x + i +columns) % columns

(y + j +rows) % rows

Add columns/rows to avoid negative index

99,39	0,39	1,39
99,40	0,40	1,40
99,41	0,41	1,41

As you can see , calculating the neighbors of a cell involves counting all of the neighbors around the current cell. That is what the nested for-loop here is doing.

```
for (let i of [-1, 0, 1]) {
    for (let j of [-1, 0, 1]) {
        if (i == 0 && j == 0) {
            // the cell itself is not its own neighbor
            continue;
        }
        // rest of the code.
    }
}
```

The `continue` here is for skipping the `0,0` since it is essentially the element itself.

It becomes a problem since we are having a trouble at the edge.We may have our *array out of bound*.

There are two cases we need to cater:

- cell at the corners
- cell at the '0-sided' edge

To cater for the cells at the corners, we need to use the module operator `%` to limit our index between `0`(inclusive) and `columns/rows`(exclusive).

To cater for the cells at the 0-sided, we need to use add `columns/rows` to the index to make them positive before using the modulo operator`%`.

> 📃 **Hints:**
>
> We can add `columns/rows` to the index as we wish because "-1 % 7" is the same as "6 % 7" which is also the same as "13 % 7".

Implementing the rules of game of life

The remaining code is basically the rules of the `Game of Life`.

```
// Rules of Life
if (currentBoard[x][y] == 1 && neighbors < 2) {
    // Die of Loneliness
    nextBoard[x][y] = 0;
} else if (currentBoard[x][y] == 1 && neighbors > 3) {
    // Die of Overpopulation
    nextBoard[x][y] = 0;
} else if (currentBoard[x][y] == 0 && neighbors == 3) {
    // New life due to Reproduction
    nextBoard[x][y] = 1;
} else {
```

```
        // Stasis
        nextBoard[x][y] = currentBoard[x][y];
    }
```

At the end, we need to swap `currentBoard` and `nextBoard` . Making the calculated next generation to be the current generation.

```
// Swap the nextBoard to be the current Board
[currentBoard, nextBoard] = [nextBoard, currentBoard];
```

This is it. You should have a working `Game of Life` with random lifes appearing everything you load the page.

Mouse Interaction

It would not feel complete if the users cannot interact with our board. Let's add some mouse events handler. Luckily, `p5.js` already provides useful function like `mouseDragged()`, `mousePressed()` and `mouseReleased()` for us to implement event handler. As their name suggested, they are invoked when the mouse is dragged , pressed and released.

Let's look at how we implement it.

```
/**
 * When mouse is dragged
 */
function mouseDragged() {
    /**
     * If the mouse coordinate is outside the board
     */
    if (mouseX > unitLength * columns || mouseY > unitLength * rows) {
        return;
    }
    const x = Math.floor(mouseX / unitLength);
    const y = Math.floor(mouseY / unitLength);
    currentBoard[x][y] = 1;
    fill(boxColor);
    stroke(strokeColor);
    rect(x * unitLength, y * unitLength, unitLength, unitLength);
}

/**
 * When mouse is pressed
 */
function mousePressed() {
    noLoop();
    mouseDragged();
```

```
}

/**
 * When mouse is released
 */
function mouseReleased() {
    loop();
}
```

As you can see, we run `noLoop()` for `mousePressed()`. It means that we want `p5.js` to stop running `draw()` whenever our mouse is pressed. We also resume the loop of running `draw()` when the mouse is released. So the game pauses when the user pressed on the canvas and resume when the mouse is released. We also reuse `mouseDragged` in `mousePressed` function.

```
function mouseDragged() {
    /**
     * If the mouse coordinate is outside the board
     */
    if (mouseX > unitLength * columns || mouseY > unitLength * rows) {
        return;
    }
    const x = Math.floor(mouseX / unitLength);
    const y = Math.floor(mouseY / unitLength);
    currentBoard[x][y] = 1;
    fill(boxColor);
    stroke(strokeColor);
    rect(x * unitLength, y * unitLength, unitLength, unitLength);
}
```

In our `mouseDragged` function , we first check if the coordinate of the cursor (`mouseX` and `mouseY`) is out of the board(Remember the `floor` function?). Then we calculate which box our cursor is currently above and we set the box to have a life. Then we paint the box directly because we are now pausing the `draw()` function.

Now you can add new life by pressing and dragging on the canvas!

Reset Button

How about the reset button at the bottom left corner. We can easily make the `reset` behavior with a simple event handler and a call to `init()`.

```
document.querySelector('#reset-game')
    .addEventListener('click', function() {
        init();
    });
```

> 🛠 **Warning:**
>
> You cannot use mouseClicked or mousePressed here. Because the reset button is outside the canvas. So p5.js does not manage this button.