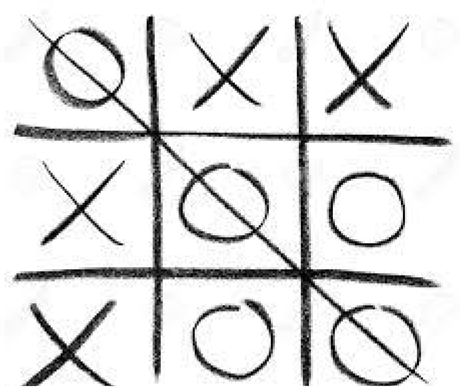
Advanced Data Types: Matrices

Learning Goals

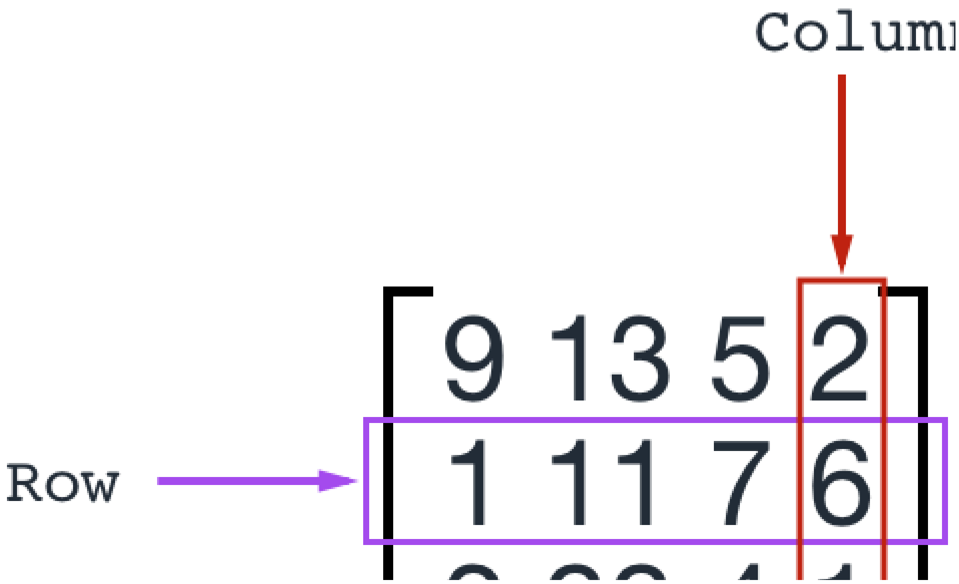
After this lesson you will:

- Know what a matrix is.
- Be able to create a matrix using JavaScript.
- Access particular rows and columns in a matrix.
- Be able to perform some basic matrix functions.

Introduction



Matrices, commonly referred to in the programming world as *Two Dimensional Arrays* is data represented as rows and columns.



They are frequently used data structures for a wide variety of applications including:

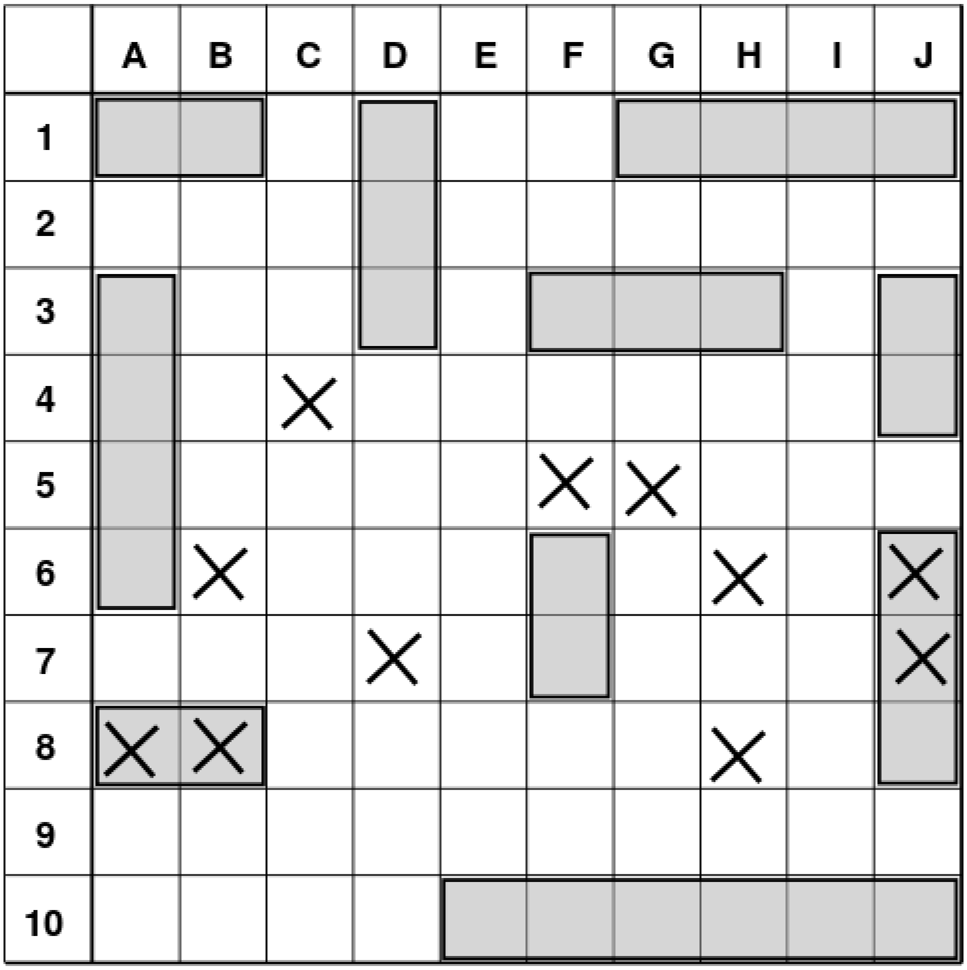
- Representing a grid (Chess Boards, tic-tac-toe, etc).
- 3D Rendering.
- Statistics
- Physics.
- Machine Learning.

  …etc

An array is a one dimensional, linear structure. A matrix's capabilities can reach far beyond that, representing things such as 3D structures and game boards.

While you may not use matrices every day in your development career, they are a handy tool for your toolbelt, and can help you solve problems you would have no idea how to solve otherwise.

In this lesson, we'll create a simple version of the game Battleship.



Battleship is a game where, in most cases two players, guess the locations of opposing player’s ships on a board. This is a great game to demonstrate matrices, because the game logic is simple, and it is *grid based*.

> **Typically battleship has two players. Our version will only have one. Also, boats in Battleship usually take 1 or more spaces. In our version the ship will always take 1 space.**
>
> You can create more advanced versions after finishing the simple version.

Create a Matrix | Battleship Board

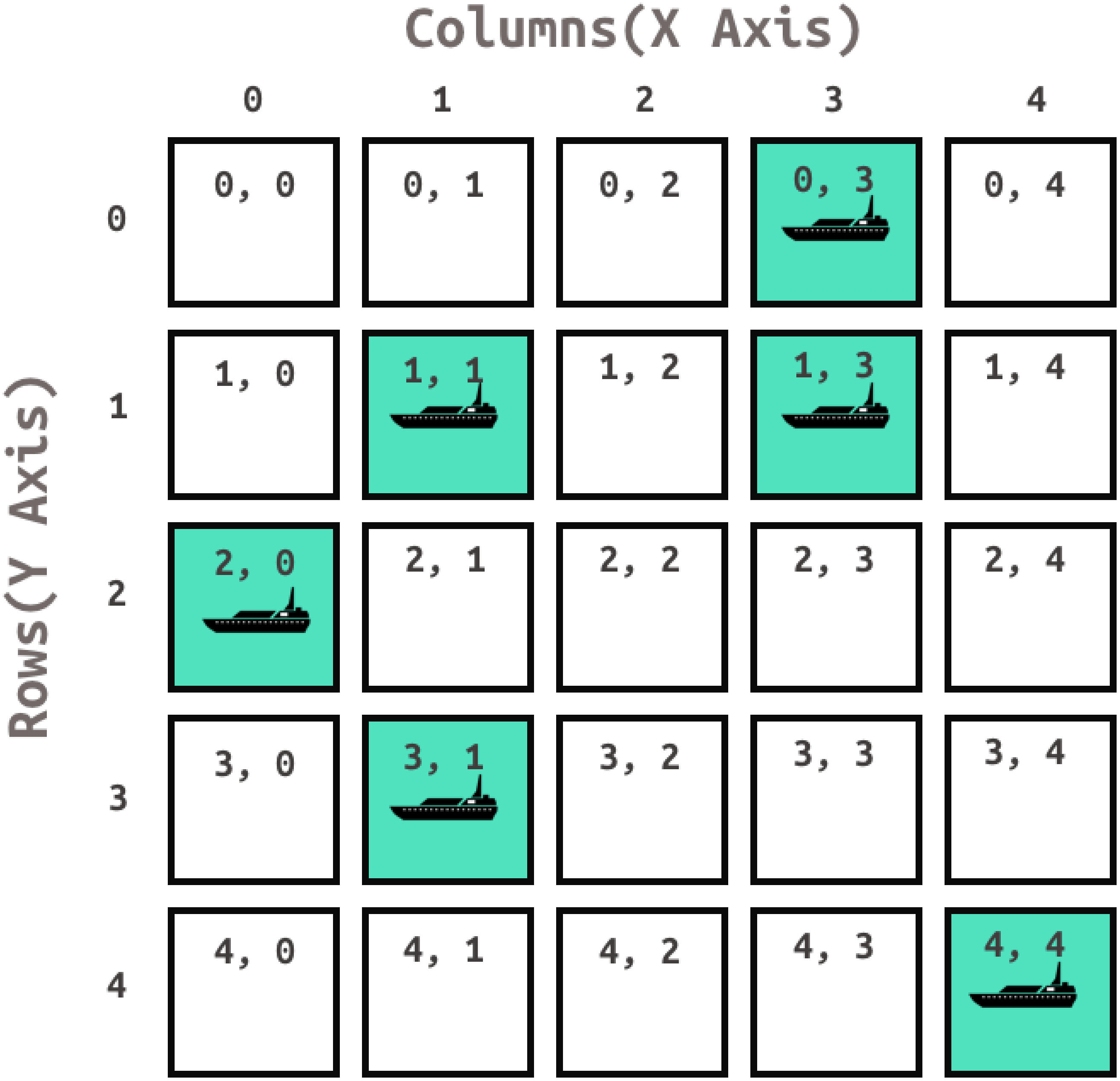
Enough of the theoretical, let’s look at the practical. How do you create a matrix? Despite the complexity of a topic, it’s actually quite easy. We define an *array literal* filled with more *array literals*.

Our Battleship board is going to be defined as a 5x5 matrix, containing an `"S"` for a ship, and `null` for an empty space.

> 💡 `null` is the JavaScript representation of no value. Just as `1` represents the number 1, `"H"` represents the letter H, and `true` represents the boolean value true, `null` is the representation of *nothing*.

```
var board = [
 [null, null, null, "S", null],
 [null, "S", null, "S", null ],
 ["S", null, null, null, null ],
 [null, "S", null, null, null],
 [null, null, null, null, "S"]
];
```

Remember, our an array's index starts at *0*. Let's visualize the matrix (board) that we've just created:



	0	1	2	3	4
0	0, 0	0, 1	0, 2	0, 3 🚢	0, 4
1	1, 0	1, 1 🚢	1, 2	1, 3 🚢	1, 4
2	2, 0 🚢	2, 1	2, 2	2, 3	2, 4
3	3, 0	3, 1 🚢	3, 2	3, 3	3, 4
4	4, 0	4, 1	4, 2	4, 3	4, 4 🚢

> 💡 Matrices can be filled with any kind of data. Numbers, Strings, and even other arrays or objects.

Access a *Row*

We can interact with a matrix in the same way we'd interact with any other array, with one caveat: when we access the array one element deep, we will be retrieving an entire array.

For example:

```
var board = [
 [null, null, null, "S", null],
 [null, "S", null, "S", null ],
 ["S", null, null, null, null ],
 [null, "S", null, null, null],
 [null, null, null, null, "S"],
];

var firstRow = board[0];
console.log("First Row: " + firstRow);

var secondRow = board[1];
console.log("Second Row: " + secondRow);

var thirdRow = board[2];
console.log("Third Row: " + thirdRow);
```

Accessing `myMatrix[0]` returns the *first row* containing one battleship.
To access one particular element, *we need to go one level deeper*.

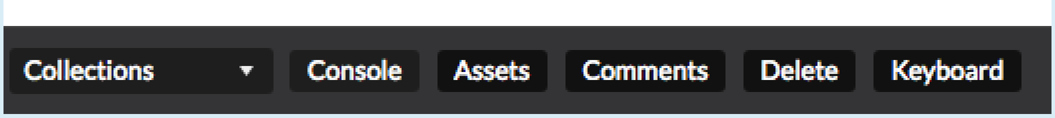
Access a *Column* | Finding Ships

Once we have a single row, we can access a single element (column) inside of it. Since `myMatrix[0]` returns an array, we can use array index notation as we usually would on it:

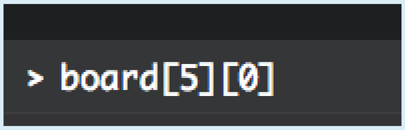
```
var board = [
 [null, null, null, "S", null],
 [null, "S", null, "S", null ],
 ["S", null, null, null, null ],
 [null, "S", null, null, null],
 [null, null, null, null, "S"]
];

var firstRow = board[0];

var emptySpace  = firstRow[0];
var ship = firstRow[3];

console.log("Empty Space: " + emptySpace);
console.log("Ship: " + ship);

// We can also shortcut assigning the row to a variable
console.log("Third row, first col: " + board[2][0]);
```

You may have noticed that the indices we're using to reference elements in the array correspond with the rows and columns in the diagram from the introduction.
Be aware though, if a row doesn’t exist and you try to access a column inside of it, you will get an error:

Open the following pen on codepen, then open the *console*:



When the console is open, enter the following code to see the error:



```
> board[5][0]
```

```
var board = [
 [null, null, null, "S", null],
 [null, "S", null, "S", null ],
 ["S", null, null, null, null ],
 [null, "S", null, null, null],
 [null, null, null, null, "S"],
];
```

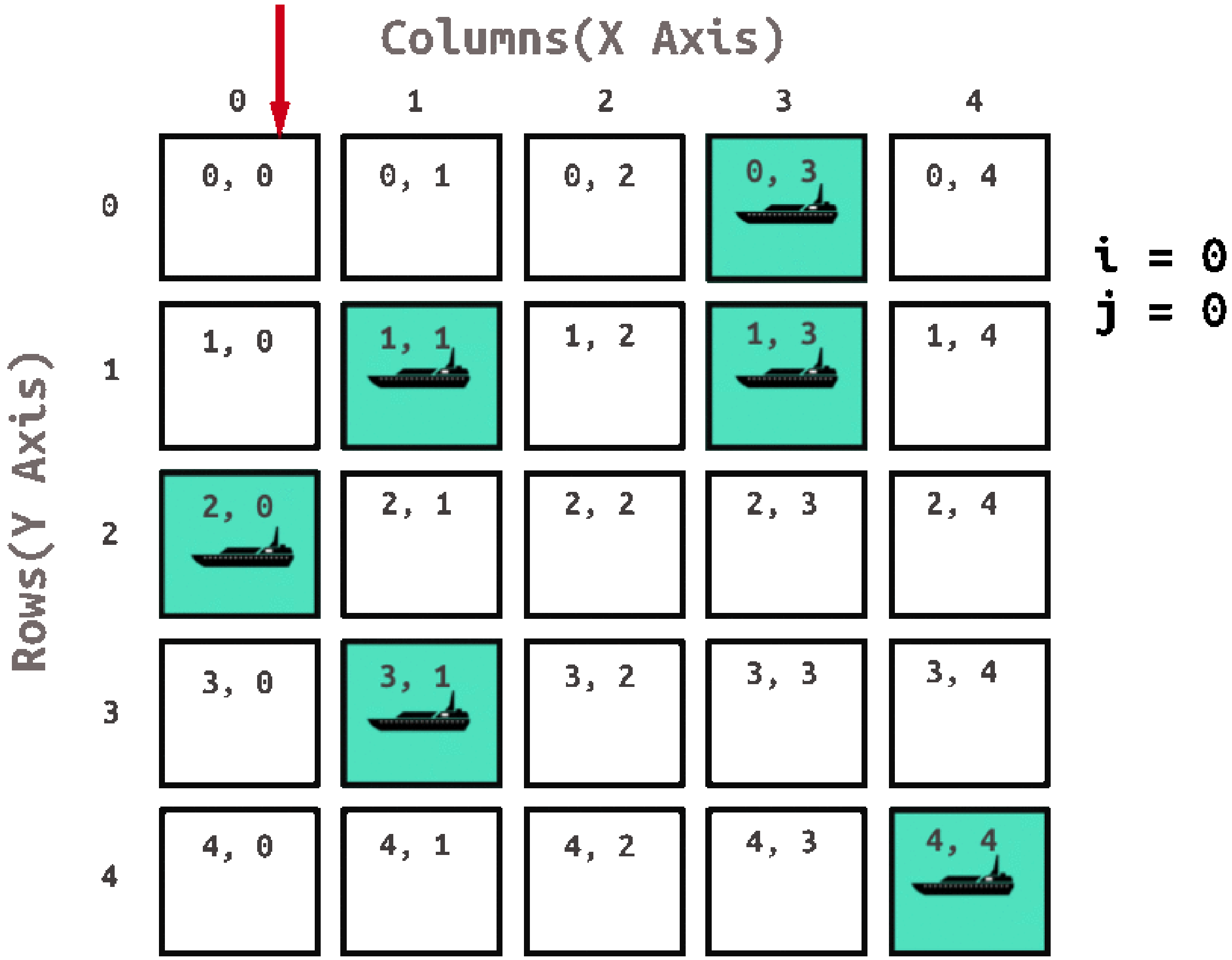
Matrix Iteration

Matrix iteration is a bit more complex than one dimensional array iteration. Since each first level element in the matrix is a row, we have to iterate over the row using another loop:

```
var board = [
 [null, null, null, "S", null],
 [null, "S", null, "S", null ],
 ["S", null, null, null, null ],
 [null, "S", null, null, null],
 [null, null, null, null, "S"],
];

for (var i = 0; i < board.length; i++){
 // A single row, such as board[0], board[1], board[2]
 var row = board[i];
 // Loop over each element in the row
 // We use "j" because "i" is already being used.
 // What would happen if we used i in this loop instead? Try it.
 for (var j = 0; j < row.length; j++){
   var column = row[j];
   // If the column is a ship, log the coords
   if (column === "S"){
     console.log("Ship Found at: " + i + ", " + j);
   }
   // Instead of using variables, you could reference: board[i][j]
 }
}
```

This may be your first time using nested loops.

Consider that the inner loop(`j`) will run for as many columns there are in a row. The outer loop(`i`) will run for as many rows there are in the matrix.



	0	1	2	3	4
0	0, 0	0, 1	0, 2	0, 3 🚢	0, 4
1	1, 0	1, 1 🚢	1, 2	1, 3 🚢	1, 4
2	2, 0 🚢	2, 1	2, 2	2, 3	2, 4
3	3, 0	3, 1 🚢	3, 2	3, 3	3, 4
4	4, 0	4, 1	4, 2	4, 3	4, 4 🚢

Modifying Matrices | Playing Battleship

Let’s add some code to our Matrix’s codepen to play the game of battleship. Our rules will be simple:

- Randomly guess a square on the grid
  - If the square is a ship, the ship has “sunk”. Replace it with null.
  - If the square is an empty space, do nothing.

You will have 10 turns to find the ships.

Turns

```
var board = [
 [null, null, null, "S", null],
 [null, "S", null, "S", null ],
 ["S", null, null, null, null ],
 [null, "S", null, null, null],
 [null, null, null, null, "S"],
];


for (var i = 0; i < 10; i++){

}
```

We will simply create a loop that will iterate 10 times to take turns:

Random Guesses

Random Index

We need to pick a random square on the board. We can do this with JavaScript's `Math.random()` function.

To pick a random square without going out of bounds, we have to select a *row* with an index between 0 and 4, and a *column* with an index between 0 and 4.

`Math.random()` in JavaScript returns a random number between 0 and 1. To get a random number between 0 and 4 (whole numbers), we will take `Math.random()`, multiply it by 5, and then round it down.

We're selecting 0 through 4 because Our grid is **5x5** with a 0 index.

```
Math.floor(Math.random()  * 5)
```

Let's create a function to give us a whole number between 0 and 4:

```
var board = [
 [null, null, null, "X", null],
 [null, "X", null, "X", null ],
 ["X", null, null, null, null ],
 [null, "X", null, null, null],
 [null, null, null, null, "X"],
];
```

```
for (var i = 0; i < 10; i++){

}

function getRandomNum(){
 return Math.floor(Math.random() * 5);
}

console.log(getRandomNum());
console.log(getRandomNum());
console.log(getRandomNum());
console.log(getRandomNum());
console.log(getRandomNum());
console.log(getRandomNum());
console.log(getRandomNum());
console.log(getRandomNum());
```

Random Square

As discussed previously, we can access an element in the matrix by referencing the row and column:

```
board[0][1]
board[1][2]
board[1][3]
// etc
```

In our for loop, we should use our random number function to pick a random number for the row and column to select a square:

```
var board = [
 [null, null, null, "S", null],
 [null, "S", null, "S", null ],
 ["S", null, null, null, null ],
 [null, "S", null, null, null],
 [null, null, null, null, "S"],
];

for (var i = 0; i < 10; i++){
 var row = getRandomNum();
 var column = getRandomNum();
 console.log("Row " + row + " " + "Column " + column);
 var randomSquare = board[row][column];
 console.log(randomSquare);
}

function getRandomNum(){
 return Math.floor(Math.random() * 5);
}
```

Update the Board

As stated previously, if we guess a square that contains a ship, then we should replace that square with `null`, because we have sunk the ship.

```
var board = [
 [null, null, null, "S", null],
 [null, "S", null, "S", null ],
 ["S", null, null, null, null ],
 [null, "S", null, null, null],
 [null, null, null, null, "S"],
];

for (var i = 0; i < 10; i++){
 var row = getRandomNum();
 var column = getRandomNum();

 var randomSquare = board[row][column];

 if (randomSquare === "S"){
   console.log("Hit on: " + row + ", " + column);
   board[row][column] = null;
 }
}

function getRandomNum(){
 return Math.floor(Math.random() * 5);
}
```

If no hits show up, try refreshing the page, it just means we were unlucky that round.

And that’s it! Using matrices and some logic, we’ve created a pretty simple game that we could add plenty of features to.

Summary

In this lesson we learned exclusively about matrices.

The topics we covered included how to create a matrix, how to access a row and column inside of a matrix, and how to iterate over a matrix.

In addition, we built a small battleship game to demonstrate the usefulness of matrices and to more easily visualize how to work with matrices.

Matrices are a pretty complex topic, and it may make your stomach turn to look at the wikipedia page for them, but you don’t have to be a math wizard to get value out of using them!