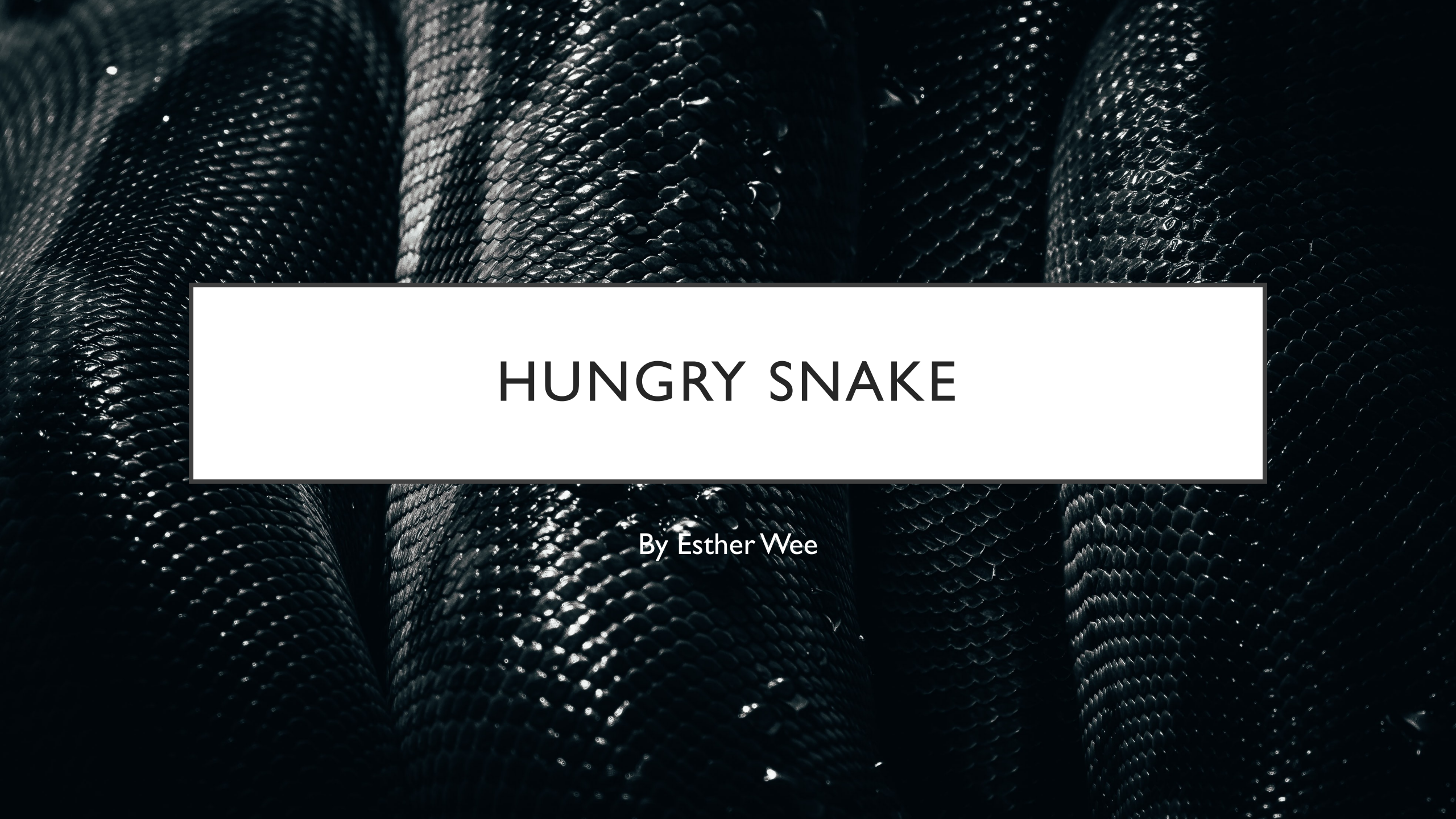
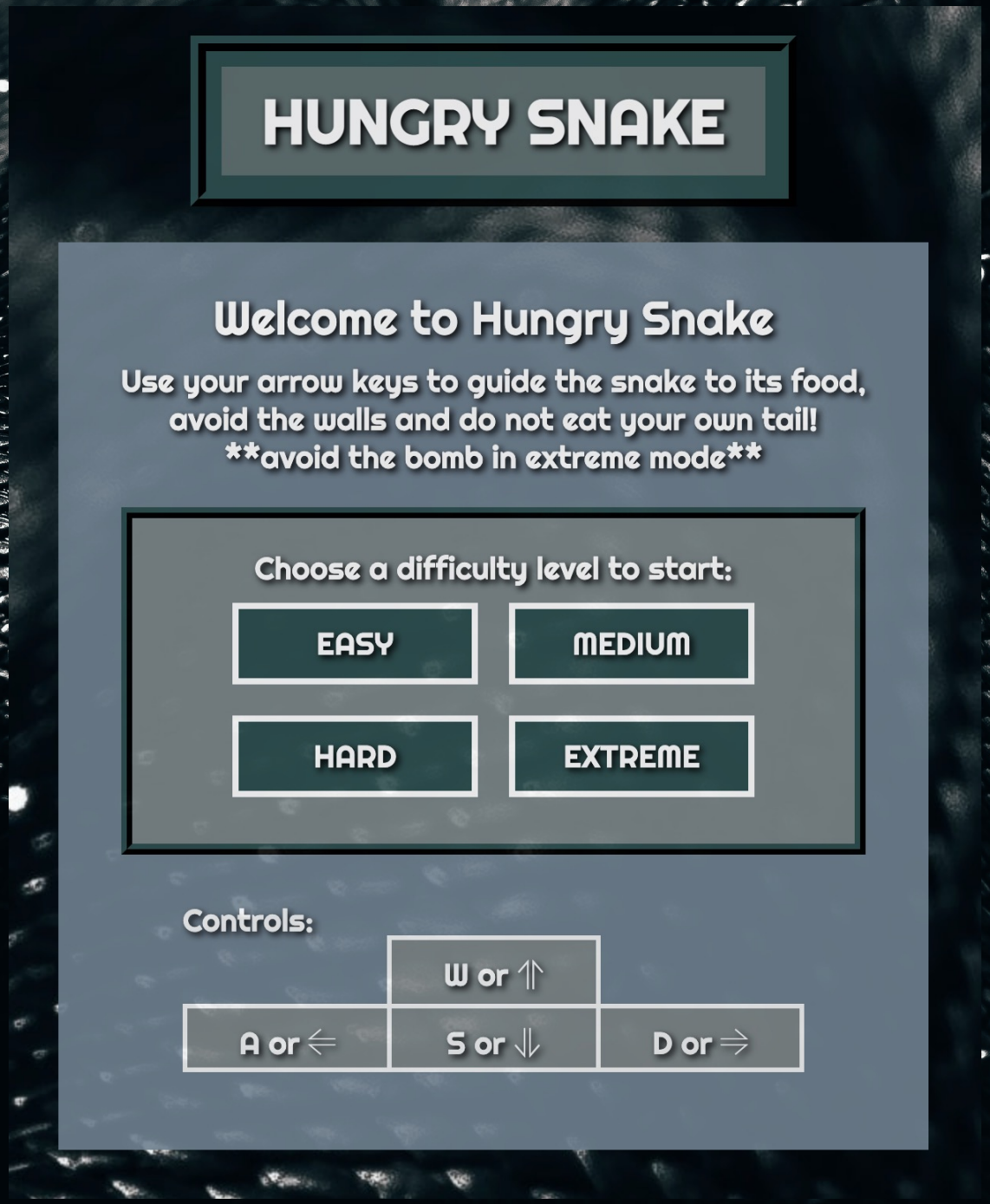


HUNGRY SNAKE

By Esther Wee



DESCRIPTION

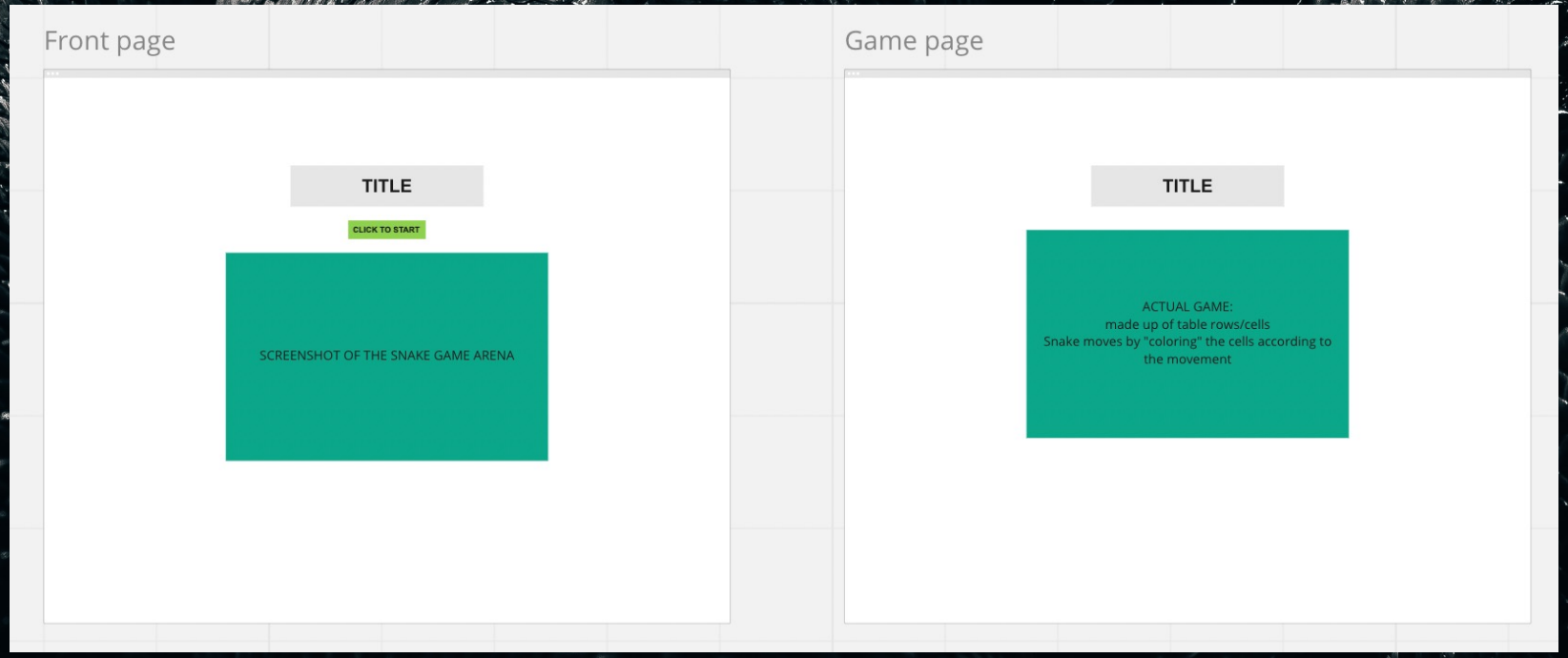
Classic snake game

- Implemented using HTML, CSS and Javascript (Jquery)

- Four difficulty levels with incrementation in speed
  - Extreme mode includes a 'bomb' that the snake has to avoid

APPROACH AND PROCESS

Layout planning via Miro



APPROACH AND PROCESS

Code planning:
1. Storage of data in objects – what do I need to store

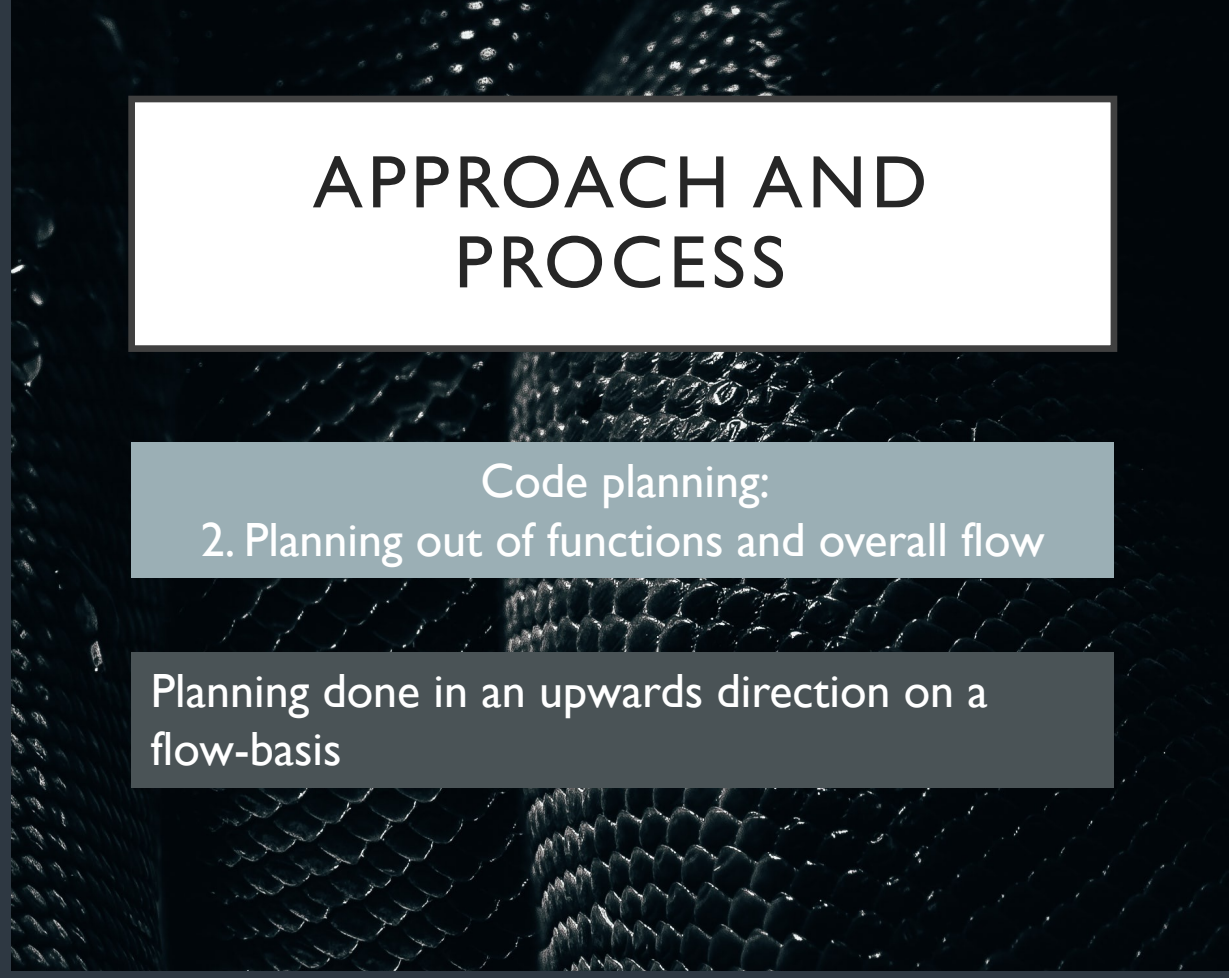
```
//Object: Data storage:
    //table size: rows, columns, no. of cells, cell size
    //snake properties
        //starting snake length
        //starting snake position (based on cell position)
            //(ARRAY of objects) where each object = one cell position
            //eg. array[0] = (head) {row: x, cell: y}
        //snake food count
```

```
const data = {
    arena: {
        boxes: 22,
    },
    snake: [
            {row: 1, cell: 3},
            {row: 1, cell: 2},
            {row: 1, cell: 1},
            {row: 1, cell: 0}
    ],
    direction: "right",
    foodCount: 0,
    food: {
        row: null,
        cell: null,
    },
    bomb: [
        //west bomb
        {row: null, cell: null},
        //east bomb
        {row: null, cell: null},
        //north bomb
        {row: null, cell: null},
        //south bomb
        {row: null, cell: null},
        //middle bomb
        {row: null, cell: null}
],
    speed: null,
    displaySpeed: "",
}

let gameOn;
```

ACTUAL CODES:

Data stored in an object

```
// code to update direction that the snake is going, which is stored in the object
    // include check for direction errors
    // e.g. if snake going left but right arrow pressed

// function: update snake
    // calls specific directional movement function
    // based on stored "direction" in the object
    // i.e. if snake direction = right, call moveRight();

//function: showSnake
    // call clear snake function
    // makes snake based on snake properties length + position
    // ? assign class to snake?

// MAIN function to move snake
    // ? use set time out? set interval?
    // if snake doesn't hit wall,
        // call update snake function
        // call show snake function
        // call move snake function again

//function: makeTable
    //makes table using table size in objects
    //appends to div.container

//function: update points
    //calls show points function

//function: show points

//function: showGameArena
    //calls make table function

//function: initgame:
    //calls showGameArena
    //calls showSnake

//function: handle start game button
    //removes startgame button (empty div.container)
    //calls initgame

//function: initializing game
    //make start button (on click activates handle start game function)
```



APPROACH AND PROCESS

Code planning:
2. Planning out of functions and overall flow

Planning done in an upwards direction on a flow-basis

```
//function: to check if food is eaten

//function: to make food appear (only one at any one time)

//function: check if snake hit wall
    // end game if hit

// functions x 4 : moveLeft, moveRight, moveUp, moveDown
    // pop the last tail
    // come up with new snake head position based on direction
    // unshift the new head to start of array
```

ACTUAL CODES:

Section 1: set up front page + handling start of game

```
}

// medium mode chosen
const handleStartMedium = () => {
    $("#rules").remove();
    data.speed = 100;
    data.displaySpeed = "medium";
    initGame(data.speed);
}

// hard mode chosen
const handleStartHard = () => {
    $("#rules").remove();
    data.speed = 50;
    data.displaySpeed = "hard";
    initGame(data.speed);
}

// extreme mode chosen
const handleStartExtreme = () => {
    $("#rules").remove();
    data.speed = 50;
    data.displaySpeed = "extreme";
    initGame(data.speed);
}

const startingPage = () => {
    // create mode choice buttons using divs
    const $speedOption = $("<div>").addClass("speedOption").text("Choose a difficulty level to start:")
    const $easy = $("<div>").addClass("easyButton").text("EASY").on("click", handleStartEasy)
    const $medium = $("<div>").addClass("mediumButton").text("MEDIUM").on("click", handleStartMedium)
    const $hard = $("<div>").addClass("hardButton").text("HARD").on("click", handleStartHard)
    const $extreme = $("<div>").addClass("extremeButton").text("EXTREME").on("click", handleStartExtreme)
    $speedOption.append($easy, $medium, $hard, $extreme)
    // create intro to game
    const $rules = $("<div>").attr("id", "rules")
    $rules.text("Welcome to Hungry Snake")
    const $p =  $("<p>").attr("id", "innerRules").text("Use your arrow keys to guide the snake to its food, avoid the walls a   do not eat your own tail! **avoid the bomb in extreme mode**")
    $rules.append($p, $speedOption)
    $(".container").append($rules);
    // create controls guide
    const $controls = $("<div>").addClass("controls").text("Controls:")
    const $up = $("<div>").attr("id", "up").text("W or ⇑")
    const $down = $("<div>").attr("id", "down").text("S or ⇓")
    const $right = $("<div>").attr("id", "right").text("D or ⇒")
    const $left = $("<div>").attr("id", "left").text("A or ⇐")

    $controls.append($up, $down, $left, $right)
    $($rules).append($controls)
};

const main = () => {
    startingPage();
 }

```

Different handling functions to start game based on mode chosen

- Calls initGame(speed)

startingPage function: set up the front page

```
                $snakeID.addClass("snakeBody1")
            }
        }
    }
    // check if snake head hits any part of the snake body
    checkHitOwnTail();
    // only in extreme mode: check if snake head hits the bomb
    if (data.displaySpeed === "extreme") {
        checkBomb();
    }
}

const initGame = (speed) => {
    makePoints();
    makeTable();

    // stores set Interval as a global variable so that we can stopInterval to end the game later
    gameOn = setInterval(showSnake, speed);
    makeFood();
    if (data.displaySpeed === "extreme") {
        makeBomb();
    }
}

// create game play area for the snake
const makeTable = () => {
    const $table = $("<div>").addClass("table")
    const $arena = $("<div>").addClass("arena")
    for (let i = 0; i < data.arena.boxes; i++) {
        for (let j = 0; j < data.arena.boxes; j++) {
            const $td = $("<div>").attr("id", `row${i}cell${j}`);
            const gr = i + 1;
            const gc = j + 1
            $td.css("grid-row", `${gr}`);
            $td.css("grid-column", `${gc}`)
            $td.addClass("cell")
            $arena.append($td)
        }
    }
    $table.append($arena)
    $(".container").append($table)
}

// display points on screen
const makePoints = () => {
    const $div = $("<div>").addClass("points").text(`SCORE: ${data.foodCount}`);
    $(".container").append($div);
}

// easy mode chosen
const handleStartEasy = () => {
    $("#rules").remove();
    data.speed = 150;
    data.displaySpeed = "easy";
    initGame(data.speed);
```

ACTUAL CODES:

Section 2: Set up game page + initiate game

initGame: initates main game

- Calls **makePoints** and **makeTable**
- Uses **setInterval** to recurringly call **showSnake** (main game function) at set intervals (speed)
- Calls **makeFood** and if in extreme mode, **makeBomb**

makeTable: makes game arena
- Initially done via tables
- Switched to grids due to css/layout formatting issues

makePoints: shows current score while in game

```
        if (data.direction === "left") {
            return;
        } else {
            data.direction = "right"
        }
    }
})

// update snake body array
const updateSnake = () => {
    // check food
    // if snake did not eat food, pop the tail.
    // Else, don't pop the tail (tail extends if food eaten)
    if (!checkFood()) {
        data.snake.pop()
    }

    if (data.direction === "right") {
        data.snake.unshift({row: data.snake[0].row, cell: data.snake[0].cell + 1});
    } else if (data.direction === "down") {
        data.snake.unshift({row: data.snake[0].row + 1, cell: data.snake[0].cell});
    } else if (data.direction === "up") {
        data.snake.unshift({row: data.snake[0].row - 1, cell: data.snake[0].cell});
    } else if (data.direction === "left") {
        data.snake.unshift({row: data.snake[0].row, cell: data.snake[0].cell - 1});
    }
}

// displays snake on screen
const showSnake = () => {
    updateSnake();
    checkOutOfBoundary();
    // clear previous snake from screen
    $(".snakeBody1").removeClass("snakeBody1")
    $(".snakeBody2").removeClass("snakeBody2")
    $(".snakeHead").removeClass("snakeHead");
    // print snake based on updated snake body array
    for (let i = 0; i < data.snake.length; i++) {
        const $snakeID = $(`#row${data.snake[i].row}cell${data.snake[i].cell}`);
        if (i === 0) {
            $snakeID.addClass("snakeHead");
        } else if (i > 0) {
            if (i % 2 === 0) {
                $snakeID.addClass("snakeBody2")
            } else {
                $snakeID.addClass("snakeBody1")
            }
        }
    }
    // check if snake head hits any part of the snake body
    checkHitOwnTail();
    // only in extreme mode: check if snake head hits the bomb
    if (data.displaySpeed === "extreme") {
        checkBomb();
    }
```

ACTUAL CODES:

Section 3: main functions to make snake move

updateSnake:

- Pops the tail if food not eaten
- Unshifts new snake head position to start of array → new head position generated based on direction snake is going

showSnake: main game function

- Calls **updateSnake**: update snake positions array
- Calls **checkOutOfBoundary**: check if snake goes out of the walls
- Clears current snake by removing the snake class from the current positions
- Loop through updated snake position array → generates the individual grid position's IDs and assigns the snakeHead and snakeBody classes to display snake

ACTUAL CODES:

Section 3: main functions to make snake move

```
$(document).on("keydown", (event) => {
    if (event.key === "ArrowDown" || event.key === "Down" || event.key === "s" || event.key === "S") {
        if (data.direction === "up") {
            return;
        } else {
            data.direction = "down";
        }
    } else if (event.key === "ArrowUp" || event.key === "Up" || event.key === "w" || event.key === "W") {
        if (data.direction === "down") {
            return;
        } else {
            data.direction = "up";
        }
    } else if (event.key === "ArrowLeft" || event.key === "Left" || event.key === "a" || event.key === "A") {
        if (data.direction === "right") {
            return;
        } else {
            data.direction = "left"
        }
    } else if (event.key === "ArrowRight" || event.key === "Right" || event.key === "d" || event.key === "D") {
        if (data.direction === "left") {
            return;
        } else {
            data.direction = "right"
        }
    }
})

// update snake body array
const updateSnake = () => {
    // check food
    // if snake did not eat food, pop the tail.
    // Else, don't pop the tail (tail extends if food eaten)
    if (!checkFood()) {
        data.snake.pop()
    }

    if (data.direction === "right") {
        data.snake.unshift({row: data.snake[0].row, cell: data.snake[0].cell + 1});
    } else if (data.direction === "down") {
        data.snake.unshift({row: data.snake[0].row + 1, cell: data.snake[0].cell});
    } else if (data.direction === "up") {
        data.snake.unshift({row: data.snake[0].row - 1, cell: data.snake[0].cell});
    } else if (data.direction === "left") {
        data.snake.unshift({row: data.snake[0].row, cell: data.snake[0].cell - 1});
    }
}

// displays snake on screen
const showSnake = () => {
    updateSnake();
    checkOutOfBoundary();
    // clear previous snake from screen
    $(".snakeBody1").removeClass("snakeBody1")
    $(".snakeBody2").removeClass("snakeBody2")
```

On keydown events using arrow keys and WASD as controls

- Checks for errors e.g. snake going left but right arrow pressed
- Updates direction stored in object

ACTUAL CODES:

Section 4: making and checking of bombs

```
const makeBomb = () => {
    // clear previous bomb
    $(".bomb").removeClass("bomb")

    // generate coordinates for west(left-most) bomb
    let row = Math.floor(Math.random() * (data.arena.boxes - 2)) + 1;
    let cell = Math.floor(Math.random() * (data.arena.boxes - 2));

    //check if any of the five bombs coincides with food or snake class
    while (checkForClash(row, cell)) {
        row = Math.floor(Math.random() * data.arena.boxes);
        cell = Math.floor(Math.random() * data.arena.boxes);
    }

    //store the final bomb coordinates in the data
    storeBombCoordinates(row, cell);
    //assign bomb classes according to the finalized bomb coordinates
    finalizeBomb();
}

const storeBombCoordinates = (r, c) => {
    // 0: west bomb/left-most bomb
    data.bomb[0].row = r
    data.bomb[0].cell = c;
    // 1: east bomb/right-most bomb
    data.bomb[1].row = r;
    data.bomb[1].cell = c + 2;
    // 2: north bomb/top-most bomb
    data.bomb[2].row = r - 1;
    data.bomb[2].cell = c + 1;
    // 3: south bomb/bottom bomb
    data.bomb[3].row = r + 1;
    data.bomb[3].cell = c + 1;
    // 4: center bomb
    data.bomb[4].row = r;
    data.bomb[4].cell = c + 1;
}

//check if any of the five bombs coincides with food or snake class
const checkForClash = (r, c) => {
    storeBombCoordinates(r, c);
    for (let i = 0; i < data.bomb.length; i++) {
        const $bomb = $(`#row${data.bomb[i].row}cell${data.bomb[i].cell}`)
        if ($bomb.hasClass("snakeBody1")
        || $bomb.hasClass("snakeBody2")
        || $bomb.hasClass("snakeHead")
        || $bomb.hasClass("food")) {
            return true;
        }
    }
    return false;
}

//assign bomb classes according to the finalized bomb coordinates
const finalizeBomb = () => {
```

makeBomb: main function to make bomb

- Generates coordinates
- Calls **checkForClash**: check if bomb coordinates is already occupied by snake or food → regenerate coordinates if clash
- Calls **storeBombCoordinates**: stores the five different bomb squares' coordinates in the object
- Calls **finalizeBomb**: run through the bomb array, to identify the grid's ID & assign bomb class to display bomb

checkBomb: check if snake head hits bomb → if yes, stops interval and calls endgame

```
229
230 //assign bomb classes according to the finalized bomb coordinates
231 const finalizeBomb = () => {
232     for (let i = 0; i < data.bomb.length; i++) {
233         const $bombCell = $(`#row${data.bomb[i].row}cell${data.bomb[i].cell}`)
234         $bombCell.addClass("bomb")
235     }
236 }
237
238 // check if snake head hits bomb
239 const checkBomb = () => {
240     for (let i = 0; i < data.bomb.length; i++) {
241         if (data.snake[0].row === data.bomb[i].row && data.snake[0].cell === data.bomb[i].cell) {
242             $(".bomb").removeClass("bomb")
243             clearInterval(gameOn)
244             endGame();
245         }
246     }
247 }
```

```
119
120 const checkFoodBombClash = (r, c) => {
121     const $food = $(`#row${r}cell${c}`);
122     if ($food.hasClass("snakeBody1")
123     || $food.hasClass("snakeBody2")
124     || $food.hasClass("snakeHead")
125     || $food.hasClass("bomb")) {
126         return true
127     } else if ((data.bomb[0].row - 1 === r && data.bomb[0].cell === c)
128         || (data.bomb[0].row + 1 === r && data.bomb[0].cell === c)
129         || (data.bomb[1].row - 1 === r && data.bomb[1].cell === c)
130         || (data.bomb[1].row + 1 === r && data.bomb[1].cell === c)) {
131             return true;
132     } else {
133         return false
134     }
135 }
136
137 // to generate random food box
138 // if food box coincides with snake body, regenerate the food box
139 const makeFood = () => {
140     let row = Math.floor(Math.random() * data.arena.boxes);
141     let cell = Math.floor(Math.random() * data.arena.boxes);
142
143     while (checkFoodBombClash(row, cell)) {
144         row = Math.floor(Math.random() * data.arena.boxes);
145         cell = Math.floor(Math.random() * data.arena.boxes);
146     }
147
148     const $foodCell = $(`#row${row}cell${cell}`)
149     $foodCell.addClass("food");
150     data.food.row = row;
151     data.food.cell = cell;
152     // random RGB color for food box
153     const red = Math.floor(Math.random() * 256);
154     const blue = Math.floor(Math.random() * 256);
155     const green = Math.floor(Math.random() * 256);
156     $(".food").css("background-color", `rgb(${red}, ${green}, ${blue})`).css("border-color", "white");
157 }
158
159 // to check if snake ate the food
160 const checkFood = () => {
161     if (data.snake[0].row === data.food.row && data.snake[0].cell === data.food.cell) {
162         // to clear CSS from the current food box
163         $(".food").removeClass("food").css("background-color","").css("border-color", "");
164         data.foodCount++;
165         if (data.displaySpeed === "extreme" && data.foodCount % 5 === 0) {
166             makeBomb();
167         }
168         updatePoints();
169         makeFood();
170         return true;
171     } else {
172         return false;
173     }
```

ACTUAL CODES:

Section 5: making and checking of food

makeFood: generates food grid

- Calls **checkFoodBombClash**: checks if food grid is already occupied by snake or bomb OR is in the corner of the bomb → regenerate food coordinate if true
- Retrieve grid ID via coordinates, adds food class and stores the coordinates into object
- Generate random RGB for the food grid

checkFood: checks if snake head eats food

- If true, update score in object
- If score is divisible by 5 in extreme mode,
  - call **makeBomb** to generate new bomb
  - calls **updatePoints** to display new points and **makeFood** again to generate new food.

ACTUAL CODES:

Section 6: checking of other end game scenarios

```
            {row: 1, cell: 3},
            {row: 1, cell: 2},
            {row: 1, cell: 1},
            {row: 1, cell: 0}
        ];
        data.direction = "right";
        data.foodCount = 0;
        initGame(data.speed);
    })
    // to refresh page & re-select speed
    $returnHome.on("click", () => {
        location.reload()
    })
}

// to check if snake went out of game boundary: if yes, clear interval and end game
const checkOutOfBoundary = () => {
    const rightWallCell = data.arena.boxes - 1;
    const leftWallCell = 0;
    const topWallRow = 0;
    const bottomWallRow = data.arena.boxes -1;
    if (data.snake[0].cell > rightWallCell
        || data.snake[0].cell < leftWallCell
        || data.snake[0].row < topWallRow
        || data.snake[0].row > bottomWallRow) {

        clearInterval(gameOn)
        endGame();
    } else {
        return false;
    }
}

// to check if snake hits own tail: if yes, clear interval and end game
const checkHitOwnTail = () => {
    if ($(".snakeHead").hasClass("snakeBody1") || $(".snakeHead").hasClass("snakeBody2")) {
        clearInterval(gameOn)
        endGame();
    }
}

const checkFoodBombClash = (r, c) => {
    const $food = $(`#row${r}cell${c}`);
    if ($food.hasClass("snakeBody1")
    || $food.hasClass("snakeBody2")
    || $food.hasClass("snakeHead")
    || $food.hasClass("bomb")) {
        return true
    } else if ((data.bomb[0].row - 1 === r && data.bomb[0].cell === c)
        || (data.bomb[0].row + 1 === r && data.bomb[0].cell === c)
        || (data.bomb[1].row - 1 === r && data.bomb[1].cell === c)
        || (data.bomb[1].row + 1 === r && data.bomb[1].cell === c)) {
            return true;
    } else {
        return false
```

checkOutOfBoundary: checks if snake goes out of boundary

- if true: clear interval and calls endGame

checkHitOwnTail: checks if snake goes out of boundary

- if true: clear interval and calls endGame

ACTUAL CODES:

Section 7: set up end game page + localStorage of high scores

```
// to check and update localStorage of highscores for current speed mode
const checkHighScore = (speed) => {
    let previousScores = window.localStorage.getItem(`${speed}`)
    const updatedScores = [data.foodCount];
    if (previousScores !== null) {
        updatedScores.push(previousScores);
        updatedScores.sort((a,b) => {
            return b-a;
        })
    }
    const highest = updatedScores[0]
    window.localStorage.setItem(`${speed}`, highest);
    return highest;
}

const endGame = () => {
    $(".table").remove()
    $(".points").remove()
    endingPage();
}

const endingPage = () => {
    const $end = $("<div>").addClass("end");
    const $lose = $("<div>").attr("id", "lose").text("GAME OVER")
    const $again = $("<div>").attr("id", "again").text("Replay")
    const $score = $("<div>").attr("id", "score").text(`You fed the snake ${data.foodCount} times in ${data.displaySpeed} mode`)
    const $returnHome = $("<div>").attr("id", "returnHome").text("Click to re-select difficulty")
    $lose.append($score, $again, $returnHome)
    $end.append($lose)
    $(".container").append($end)

    // to display personal highscore for current speed mode
    const $scoreBoard = $("<div>").text(`Your ${data.displaySpeed} mode personal highscore:`).addClass("scoreBoard")
    const highestScore = checkHighScore(data.displaySpeed);
    const $highScore = $("<div>").text(highestScore);
    $scoreBoard.append($highScore);
    $lose.append($scoreBoard)

    // to replay in the same speed mode
    $again.on("click", () => {
        $(".end").remove();
        data.snake = [
            {row: 1, cell: 3},
            {row: 1, cell: 2},
            {row: 1, cell: 1},
            {row: 1, cell: 0}
        ];
        data.direction = "right";
        data.foodCount = 0;
        initGame(data.speed);
    })
    // to refresh page & re-select speed
    $returnHome.on("click", () => {
        location.reload()
```

checkHighScore(speed):

- Retrieves stored highscore under the current speed mode from localStorage
- Adds current score and previous highscore to array → sorts in decreasing order
- Stores updated high score back in local storage and returns the high score

endGame: initiates end of game

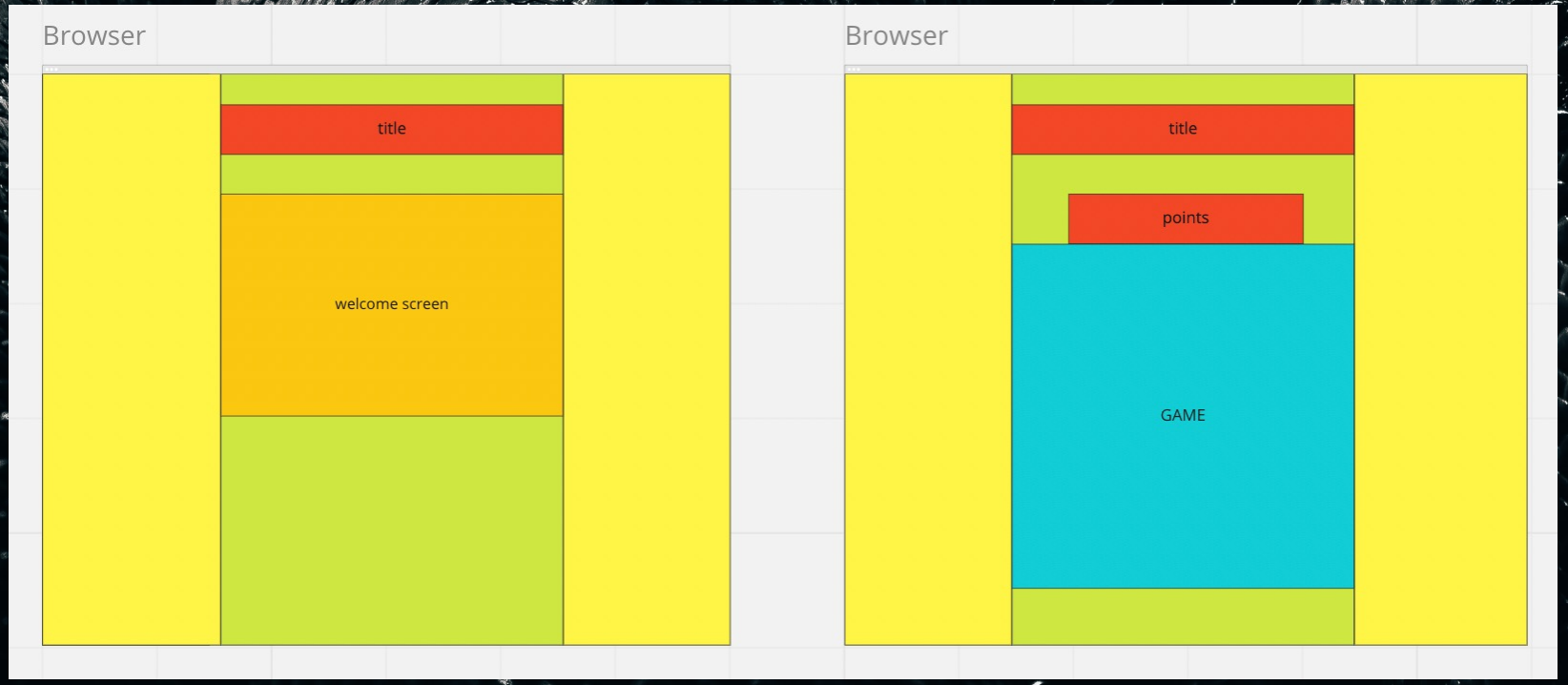
- Removes game arena and points display
- Calls **endingPage**

endingPage: sets up end of game page

- Display score using return value of **checkHighScore(speed)**
- Button to replay at same difficulty
  - Removes ending page, resets snake position array, direction & score
  - Calls **initGame** again at same speed
- Button to refresh page to re-select difficulty
- Display high score stored in localStorage based on current speed mode

APPROACH AND PROCESS

Layout: initially haphazardly done → switched to grids → planning via Miro



LEARNING POINTS

What went well:

- **Proper planning based on flow of functions**
  - This helped in knowing what functions I needed and what each function needs to do
- Ensuring that each function has a **specific role** to play
  - Helps to later identify which function was causing problems if there are errors
- Even when unsure of how to code a certain part, just got to start somewhere and slowly figure it out from there through the errors instead of having a mindset of having to get it right from the start

LEARNING POINTS

What could be done better:

- Need to think more logically/thoroughly in terms of **ordering of functions**
  - Snake was unable to lengthen when eating food or ending game when hitting tail because of this
  - Only resolved when I decided to "rewrite" and "re-order" from scratch in a new js file.

- Need to weigh and decide more carefully on **what structure to use** for the main game as switching from tables to grids after I completed my codes was extremely painful