**HO CHI MINH NATIONAL UNIVERSITY – INTERNATIONAL UNIVERSITY**

**SCHOOL OF COMPUTER SCIENCE AND ENGINEERING**

**PROJECT REPORT**

**Course: Algorithms & Data Structures**

**Lecturer: Mrs. Tran Thanh Tung**

**BATTLESHIP**

**Design and Implementation**

--------------------------------------------------------

|  |  |  |
| --- | --- | --- |
|  | **Name** | **ID** |
| 1 | Lê Nhật Trường | ITITIU18257 |
| 2 | Nguyễn Hoàng Long | ITITIU18157 |
| 3 | Nguyễn Nguyên Hưng | ITITIU18182 |

June 07th, 2021

**Table of Contents**

1. **INTRODUCTION 1**
2. Motivation and Abstract 1
3. **DESIGNING A BATTLESHIP GAME 1**

**2.1. Specification of Battleship Game 1**

**2.2. Object – Orientation 2**

**2.3. Model – View – Controller Design Pattern 2**

**2.4. Other Design Pattern 5**

2.4.1. Template 5

2.4.2. Strategy 5

2.4.3. Observer 6

**2.5. Objects 6**

**2.6. Class Design 7**

**2.7. Algorithms 7**

**2.8. Data Structures 9**

2.8.1. The game board 9

2.8.2. The group of ships (the fleet) 10

2.8.3. Coordinates 10

2.8.4. Ship memory 10

**2.9. AI Considerations 11**

2.9.1. Placing ships on the gameboard 11

2.9.2. Intelligent distribution of shots 11

2.9.3. Distribution shots – advanced 13

2.9.4. Other things 13

1. **THE BATTLESHIP CODE 14**
   1. Model 14
   2. View 39
   3. Controller 46
2. **BATTLESHIP GAMEPLAY 47**
3. **CONCLUSION 49**

# **INTRODUCTION:**

**1.1 Motivation and Abstract:**

The purpose of this project is to explore the basics in the data structures and algorithms. The purpose is not to define and develop new and original software or academic knowledge; hence a game of Battleship serves as a case study, for exploring the basics of software design and programming. By using a simple game as a case study, the focus quickly arrives at software design, structure, and code, thus skipping the art of defining all software features from scratch.

# **DESIGNING A BATTLESHIP GAME:**

## **Specification of this Battleship game**

The core part of the Battleship game is the ability for players to fire at the opponent, the game being over when one’s counterpart has no more ships left. Furthermore, the concept of an AI player is perhaps more challenging than the remainder of the game. Also, the concept of intelligent, autonomous systems is a widely used one. It therefore seems relevant to include at least some AI aspects in the game, with regards to AI’s widespread use.

Independently from the number or nature of players and the platform on which the game is implemented, the core part of the game, is the ability to register ships and shots fired on a gaming board. To put it in another way: If we design this core part of the game, we have gone a long way in developing the game, which could then easily be extended with regards to user interface and secondary features.

Based on these thoughts, the overall goal with this student project is to develop a core component of the game, which can represent a Battleship game board, and furthermore, to develop a simplistic user interface, where it is possible for a player to interact with the game board (place shots and score points). Of course, this makes little sense if the game board has no ships on it. Hence an AI player will be partly developed. This AI player will not

be a ‘real’ player, in the sense of being able to fire shots, score points and win games, rather it will an ‘invisible pseudo-player’ that is able to randomly place ships on the game board, for the human player to fire shots at.

## **Object Orientation**

Using the object-oriented logic to develop entities for our program that models the ‘real world’ – the real world in this case being our Battleship game.

One can easily distinguish several abstractions over the game which could be useful. For instance, there are ships, players, shots, boards, score, and a fleet of ships, all of which could (or should) have varying degrees of detail associated with them. For instance, one could consider the concept of a ship. One property of a ship would in the real world be color. Since the color of a ship plays no role in our game, we would probably not include this in our design. On the other hand, if we were to display different ships in different colors (if the ships of one player were to have another color than the counterpart’s) we would need to include color as a property of an object.

Many such decisions would have to be made, and one will have to consider future requirements of the software and the amount of detail needed to create a satisfying design model.

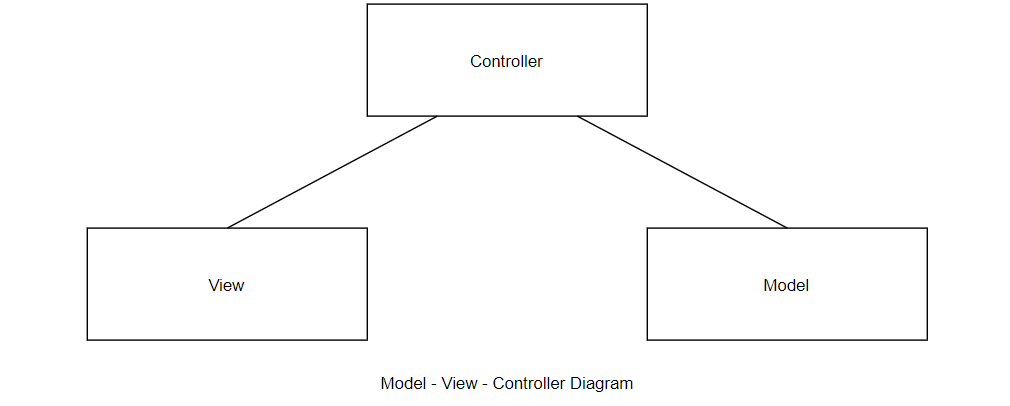
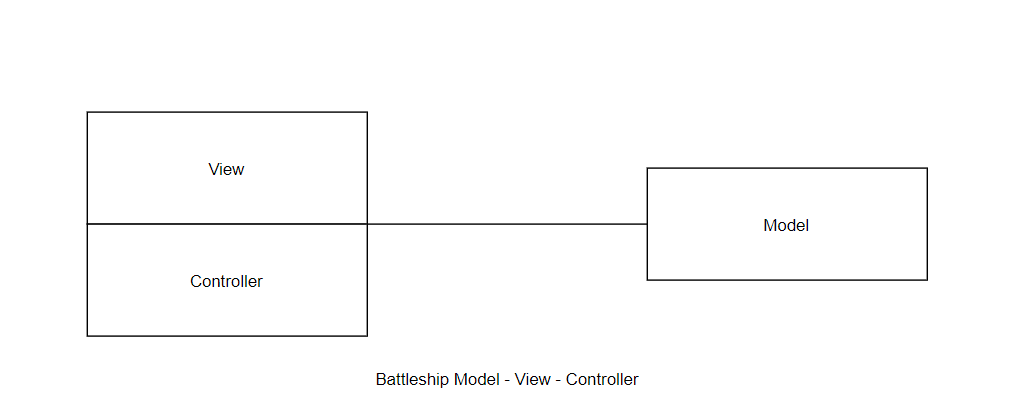
## **Model – View – Controller Design Pattern**

Another perspective on designing software that requires user interaction, is the concept (or design pattern) of Model-View-Controller (MVC), which separates the tasks of the different components of the software in a way such that three general roles can be singled out: The model, the View and

the Controller.

The Model contains the basic, underlying data access methods and data storage. The View represents the visual user interface, and the Controller is the part of the software that listens for user input and reacts accordingly.

In the original MVC design concept (see figure of MVC model below), all three elements are separated, which typically would be reflected in the final project code. When coding the Battleship game, however the View and Controller will be somewhat integrated in the sense that the use of a MVC design pattern, does not entirely separate the code relating to the View from the code relating to the Controller.

**Data model object containing a list of game objects :**

Offers an interface allowing access, addition, removal, and updating (playing) of games.

Game objects implementing the logic of the game should make up the bulk of the model code:

Contains the state information for the current player’s turn, position of ships in the two grids, and locations that missiles have been launched for each player.

Offers a method to launch a missile from the current player to a location on the grid.

Based on the game state information, the current game phase can be obtained.

The classic game allows the user to place their own ships on the grid before the game begins. To simplify the UI requirements of the assignment, instead write code that creates random, but valid, ship configurations for each player when the game begins. E.g. ships have the possibility of being placed in both columns and rows, and should not overlap or fall off the end of the grid.

Views that offer access into the model. These should be well implemented and function perfectly, but do not need to be very visually appealing. The screens involved are:

Game List Screen: A screen containing a table view that lists games that are in-progress or ended, and that opens a game when its row is tapped. The row should note if the game is in progress or if it has ended, who’s turn it is in that game (or has ended), and how many ships remain un-sunk for each player. Games can be started from this screen by pressing a “new game” button.

Game Screen: A screen showing a grid that contains the locations of the player’s ships and the locations their opponent has launched missiles against them. The screen also needs to show another grid representing the player’s opponent and that lets them launch missiles by tapping a grid cell. This grid should show where they have launched missiles previously, including “hit”, “missed”, and “sunk” information. The screen should not show where the opponent’s ships are, for reasons that are hopefully obvious.

Views should be organized by being added to view controller objects:

When views require information to draw the UI, the view controller should query the model for that information when requested to do so. When the user taps a grid cell, the view controller should ask the model to perform the “launch missile” action (using a delegation pattern or target-action mechanism), rather than the view doing this.

The benefit of using the MVC design pattern is separation of data (the Model) and the View and Controller. This separation makes it easy to update the implementation of the Model, for instance, since the Model and its underlying data structures are independent of the View and the Controller.

We could then easily accommodate changes relating to for example persistent data storage or data structures. It would also be possible to use the Model as a Model for a whole different system, for example if we wanted to use the same model in an online implementation our Battleship game. If

we did not apply the MVC concept to our design, this would present quite a challenge, since reprogramming most of the software for another platform would possibly be required. Thus, the MVC design pattern allows us flexibility, maintainability, and code reusability.

## **Other Design Patterns**

Battle-Ship Game applying some Software Patterns. Battleship is a game composed by two boards with a set of positions and every position has specific coordinates in the board. Each player has a set of ships conformed by 1 Aircraft Carrier, 1 Submarine, 1 Destroyer, 1 Battleship, 1 Patrol Boat. The ships are located randomly in the board of each player occupying specific positions depending on the size of each ships. Once the ships are located, one player specified a set of coordinates related with positions in the board of its opponent to attack its ships. The first player who destroy all the ships of its opponent is the winner. The patterns used in this simple approach were:

* + 1. **Template**

Every turn requires a series of steps, most of those steps are equal even if are executed by human-player or pc-player. Some specific steps require specific implementation for human-player or pc-player due to their behavior. For example, 'make an attack' is a step that pc-player executed based on one strategy, but the human-player executed via pressing one slot in the grid. The idea to implement this pattern is to set and control the steps related with 'take a turn', make the steps common in one specific place and each player implements specific steps depend on its behavior.

* + 1. **Strategy**

The PC - player has to make an attack in the human-player board. It is possible to execute that attack in different ways: Randomly or Systematically. The first option is Randomly, so the PC - player create a shoot randomly and this shoot take the state of Successful or Missed. In the next turn of the PC - player, if the previous turn has the state of Successful, the PC - player change the strategy to Systematically to create the following shoots near the position of the previous shoot. It is possible to create possible shoots in Horizontal, Vertical direction. This change of behavior is carrying on runtime depending on the previous shoots executed by the PC - player.

* + 1. **Observer**

The Ship has state, the state is basically OK, Attacked and Destroyed. This State is updated with every shot that the ship received. When the State is updated in the Ship, the GUI have to update the information that display about the ship to inform to the player which kind of ship has been attacked and how many ships remain in the opponent board. The observer pattern is useful to update information in the GUI based on the different states of the Ships.

## **Objects**

Now, after considering design issues on a somewhat abstract level, we need to focus our attention to some of the details pertaining to the design and coding of the Battleship game. Let us start by

considering the objects and classes needed in our Battleship game and relate these to the MVC concept.

A list of possible objects was suggested above. They were: Ships, players, shots, boards, score, and fleet. Some of the suggested objects seem obvious to include right away.

A ship is an important part of the game since the game cannot be started (or ended) without ships. Also, ships must be placed on a game board, and the game also cannot be played without a game board. This kind of reasoning

also applies to the players. We now have identified three distinct objects to include in our design and program: Ship, Board and Player. Remembering our MVC pattern, the Model element is represented by these three objects, since these three entities also represent the core parts of the Battleship game.

The concept of a fleet was mentioned, and it is another candidate for ‘objectification’. We need some way of tracking the association between ships and the game board, and for now we will decide on an object of type fleet.

Shots, then, does not really qualify as objects since they can be seen as an action and not an entity. If shots fired could have different characteristics, such as velocity, explosive charge and so on, it would probably make more sense to include a shot as an object in our design.

The suggested object, ‘score’, is more a property of a player, than an object. There is going to be no advanced ‘score mechanics’ in this game – it’ll just be a number that changes whenever the human player destroys a ship.

## **Class Design**

The part of the game, is, as mentioned, above, the objects board, player, and ship. Together these constitute the Model in the MVC design pattern. Each of these objects have several characteristics and must be able to perform certain tasks, such as keeping track of ships on the board, shots fired and the status of each ship on the board, just to mention a few.

A class should represent a single concept. This is also the approach used in identifying the objects of the Battleship game. Following this principle, we can now state three classes for the Model of our game: Board, Ship and Player.

In addition to these classes, we need a class to represent the graphical user interface of the game; hence the class View is created. This class is, in design terms, a representation of the View and Controller parts, from the MVC design pattern. All graphics display and user interaction will lie here.

## **Algorithms**

The first step is a hitting set problem, and many interesting variations are possible. In this paper, however, we consider the second step. The goal of the second step is to sink the ship (which has already been hit once) with a minimal number of misses. During a real game, the position of the new ship can be constrained by the positions of the other ships which have been already discovered and the grid boundaries, but we consider a simpler case. The fleet is composed of only one ship placed on an infinite grid using only integer translations.

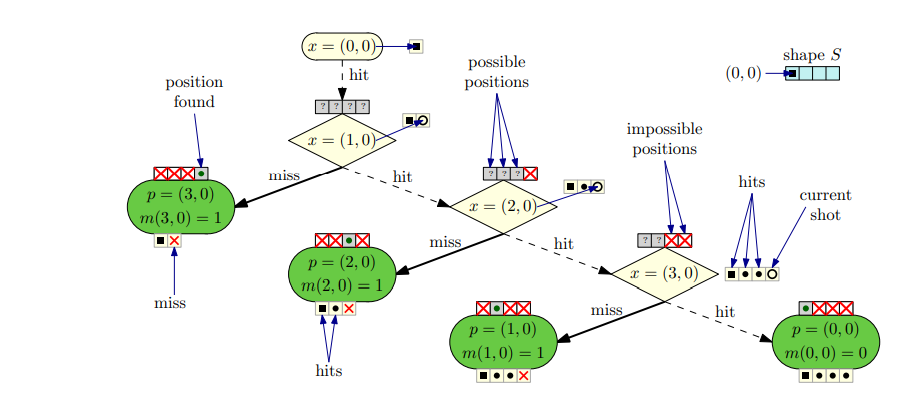
The second modification to the rules that we make is to forbid rotations. This modification simplifies the problem and since there are at most 4 possible rotations, the original problem can be solved by considering each rotation separately.

As a toy example to motivate the problem, we consider the case in which the shape of the ship is a horizontal line segment of length 4. We have already hit enemy’s ship, which is a horizontal line segment of length 4. However, we are clueless about which square of the ship we have hit. In this case, we may progressively shoot to the right of the first hit until a shot missed. At this point, we know the precise location of enemy’s ship and may finish sinking it without missing any additional shot (if we have not already sunk the ship at the fourth shot). In this case, we have a strategy that requires at most 1 missed shot.

Given a shape S, we can model an algorithm to determine the position p by a binary decision tree T. The children of each node correspond to the possible outcomes of the shot: hit or miss. The leaves of the decision tree represent the nodes in which the position p of the ship has been determined (they are not necessarily obtained after a hit). Since each leaf corresponds to a different position p ∈ S of the ship, it follows that there are exactly n leaves.

The efficiency of the algorithm depends on the number of misses in the path going from the root to a leaf corresponding to position p. This number of misses for a position p using tree T is denoted mT (p). We omit the subscript T in mT (S) when the decision tree T is clear from the context. The complexity of the algorithm T is the maximum number of misses in a path going from the root to a leaf, that is maxp ∈ S mT (p). Note that the complexity is generally not equal to the height of the tree.

After the initial shot at x = (0, 0), we shoot at values x = (1, 0), (2, 0), . . . until a miss occurs. This algorithm is modeled by the decision tree represented in figure below. The number of misses m(x) of this algorithm is equal to 0 if p = (0, 0) and 1 otherwise, giving a maximum of 1, which proves c(S) ≤ 1. It is easy to see that for any shape S with |S| > 1, c(S) ≥ 1. Hence, the algorithm is optimal.



## **Data Structures**

* + 1. **The game board**

We need a sensible way to represent the game board, the concept which we developed earlier. There are of course several different ways of representing the game board.

One way is to implement a two-dimensional array, where each array index points to an integer that represents a certain field and its status. A field can have a ship on it or be empty, and it can at the same time be hit or not hit. Besides this information the two-dimensional array contains nothing else if the array stores the status of a field only as data type integer.

This has one advantage. The two-dimensional array is very easy to use, and each element can be accessed simply by using the x and y coordinates of the game board as array index. Furthermore, the efficiency and scalability of this two-dimensional array data structure is very good. It takes constant time to look up any element, independent of the number of elements stored in the array.

The disadvantage of implementing such a data structure is its simplicity. If the two-dimensional array stores only an integer, no other information than field status, can be retrieved. To determine, for example, which ship is placed on a given coordinate, one must then implement this functionality elsewhere; it cannot be read directly from the two - dimensional array, unless we choose an alternative approach by storing not integers in the array, but objects, for example of a custom type ‘Field’. This way we could store the field status, along with any other necessary information, such as a reference to a ship, if any.

For simplicity, we will go along with storing the field status in a two-dimensional array, containing only integers.

* + 1. **The group of ships (the fleet)**

Another central data structure is the group of ships placed on a board. Because we chose to implement the two-dimensional array storing only integers, we need some way of identifying the ship that could potentially lie on a board field. For this purpose, we developed the concept of a fleet.

We have several options when choosing the appropriate data structure. But since the game specification does not require us to allow for a random number of ships, efficiency and scalability is of little importance. Thus, it seems sensible to use an array list in the Board class, for storing the ships associated to the game board. This, then, will be a central data structure for finding ships, when these are hit or sunk.

* + 1. **Coordinates**

Derived from the fact that the game board’s central data structure is a two-dimensional array, where access to a certain board field is achieved through x, y value pairs we need some way to deal with coordinates, that will be more agile than simply using x and y value pairs all the time. Also, we need a more intuitive way than using x and y values to search for ships. Hence, we introduce an object (and a class) of type coordinate. This is more due to practical coding concerns, than other design issues.

* + 1. **Ship memory**

In a real-world concept of a ship, one would expect that any given ship would have knowledge of its position. Deducting from this kind of logic, we should implement some data structure in our Ship class for storing the coordinates of any instance of type Ship. But programming logic can of course be different from real world logic, the game board, for each field on it, is aware of ships.

Since a Ship can only have five or less coordinates (the maximum length of a ship is five, this is equivalent to five coordinates) efficiency and scalability is of little importance, and therefore it will suffice to use an array list for storing coordinates.

## **AI Considerations**

As described earlier, it was decided to add an AI Player class to the game design. This is a subclass of the Player class, and it currently should inherit characteristics such as player name and score, along with trivial methods to set and get player score and player name and so on.

All these characteristics apply to the artificial player as well as the human player, since, to add to the excitement of the game, it could be a good idea to display both computer score and human player score.

* + 1. **Placing ships on the game board**

For the computer to be able to place ships on the board, so that the human player can start shooting and sinking ships. This functionality requirement leads us to the design of an AI Player object and class. And the approach taken in designing this software suggests that AI related functionality should be considered part of the AI Player subclass, since certain operations pertain only to the machine player.

In future versions of the game, to allow the human player to use parts of the AI functionality. If the game really should be expanded, one could imagine a ‘quick game’ menu option, where the human player is then able to utilize the automatic placing of ships on the game board, simply to get started playing right away, not spending any time placing ships manually on the board.

* + 1. **Intelligent distribution of shots**

But the placing of ships is only one aspect of the AI functionality that a game of Battleship could have. Like a human player, the machine player would also have to fire shots at the human player’s game board.

The firing of shots is straightforward if not considering tactics or some sort of intelligence added to the action. A random set of coordinates within the borders of the game board could easily be generated, and then the shot could be placed on the board.

This would be a very basic implementation of shooting capabilities, but it would probably not feel very authentic to play against such a machine – it would simply be too easy to win. Therefore, it would make sense to adopt some systematic approach to firing shots.

One thing is the distribution of shots on the game board, another is machine player memory and playing behavior. The machine player must remember the previous shot, but it must also remember the result of the previous shot.

When a shot is fired and a ship is hit for the first time, we will know as humans, that there must be more of that ship on at least one of the neighboring game board fields, depending on the number and type of ships left on the game board. The machine player should therefore be able to simulate this kind of inductive behavior.

After a successful first shot the machine player should then consecutively try each of the direct neighbors to the field in question. When another shot becomes successful, the machine player should then be able to induct, that it now knows the orientation of the ship it is firing at. In other words, if two shots are successful, and they are oriented horizontally, the machine player should not fire shots to either the above or below of the previously hit fields, but only to the right and/or left of the previously hit fields.

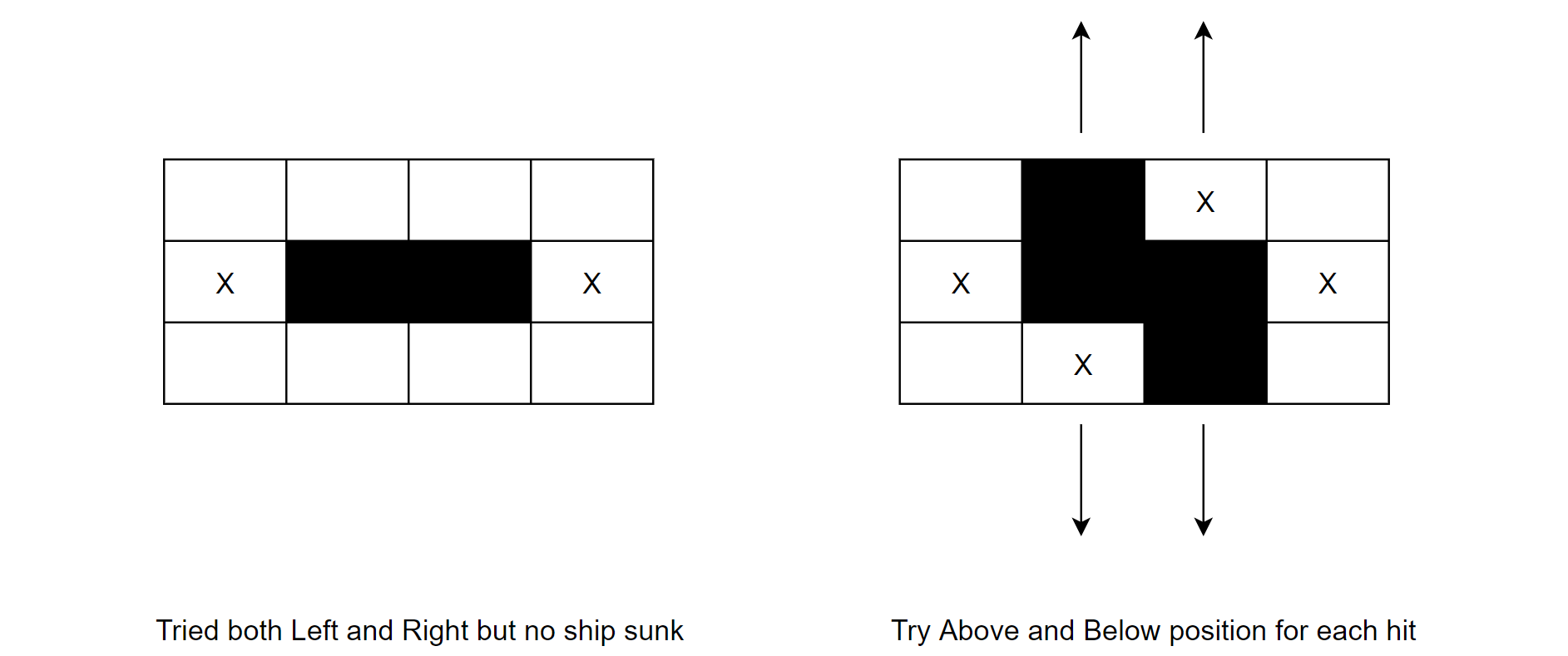
Hence, after having had a second successful shot on one of the neighboring game board fields, the machine player should continue firing in the direction suggested by the successful second shot.

The machine player should then continue placing shots in this direction, until the ship is sunk, or an unsuccessful shot occurs. If an unsuccessful shot occurs, and the ship is still not sunk, the machine player should continue firing in the opposite direction from where the first successful shot occurred, until the ship is sunk.

* + 1. **Distribution shots – advanced**

The paragraph above merely suggests a basic outline of how a reasonable level of intelligence could be applied to the machine player. It does not consider how the machine player should react if an unsuccessful shot occurs for the second time, working in the opposite direction from the direction initially taken.

If this situation occurs it must be because there are several ships adjacent to one another. Then the machine player, to behave human-like, should also be able to induct this, and behave like the basic outline described above suggests, for each of the possible ships adjacent to each other. This would certainly require a more capable machine player, since it must not only remember the last successful shot, but it must also be able to pursue the sinking of all the ships in adjacency to each other, and finally it must return to the strategic firing of shots, remembering where the next shot should be played, according to whatever strategy has been laid out for firing shots.



* + 1. **Other things**

Some final thoughts about a machine player suggest certain variations in its behavior. If the machine always placed its ships in the same way, the game would soon become trivial. Thus, we need variations in ship placement. To keep the game interesting and more difficult to predict, also after the first couple of games played, variations should be introduced in the way the machine player fires shots, i.e., it should not always start firing at the same field, and then distribute the shots identically around the game board. This would make it easy for the human player to place her ships in a way that avoids detection.

# **THE BATTLESHIP CODE**

## **Model**

1. **AI Object**

// AI Object

// Optimal battleship-playing AI

// Constructor

function AI(gameObject) {

    this.gameObject = gameObject;

    this.virtualGrid = new Grid(Game.size);

    this.virtualFleet = new Fleet(this.virtualGrid, CONST.VIRTUAL\_PLAYER);

    this.probGrid = []; // Probability Grid

    this.initProbs();

    this.updateProbs();

}

AI.PROB\_WEIGHT = 5000; // arbitrarily big number

// how much weight to give to the opening book's high probability cells

AI.OPEN\_HIGH\_MIN = 20;

AI.OPEN\_HIGH\_MAX = 30;

// how much weight to give to the opening book's medium probability cells

AI.OPEN\_MED\_MIN = 15;

AI.OPEN\_MED\_MAX = 25;

// how much weight to give to the opening book's low probability cells

AI.OPEN\_LOW\_MIN = 10;

AI.OPEN\_LOW\_MAX = 20;

// Amount of randomness when selecting between cells of equal probability

AI.RANDOMNESS = 0.1;

// AI's opening book.

// This is the pattern of the first cells for the AI to target

AI.OPENINGS = [

    {'x': 7, 'y': 3, 'weight': getRandom(AI.OPEN\_LOW\_MIN, AI.OPEN\_LOW\_MAX)},

    {'x': 6, 'y': 2, 'weight': getRandom(AI.OPEN\_LOW\_MIN, AI.OPEN\_LOW\_MAX)},

    {'x': 3, 'y': 7, 'weight': getRandom(AI.OPEN\_LOW\_MIN, AI.OPEN\_LOW\_MAX)},

    {'x': 2, 'y': 6, 'weight': getRandom(AI.OPEN\_LOW\_MIN, AI.OPEN\_LOW\_MAX)},

    {'x': 6, 'y': 6, 'weight': getRandom(AI.OPEN\_LOW\_MIN, AI.OPEN\_LOW\_MAX)},

    {'x': 3, 'y': 3, 'weight': getRandom(AI.OPEN\_LOW\_MIN, AI.OPEN\_LOW\_MAX)},

    {'x': 5, 'y': 5, 'weight': getRandom(AI.OPEN\_LOW\_MIN, AI.OPEN\_LOW\_MAX)},

    {'x': 4, 'y': 4, 'weight': getRandom(AI.OPEN\_LOW\_MIN, AI.OPEN\_LOW\_MAX)},

    {'x': 0, 'y': 8, 'weight': getRandom(AI.OPEN\_MED\_MIN, AI.OPEN\_MED\_MAX)},

    {'x': 1, 'y': 9, 'weight': getRandom(AI.OPEN\_HIGH\_MIN, AI.OPEN\_HIGH\_MAX)},

    {'x': 8, 'y': 0, 'weight': getRandom(AI.OPEN\_MED\_MIN, AI.OPEN\_MED\_MAX)},

    {'x': 9, 'y': 1, 'weight': getRandom(AI.OPEN\_HIGH\_MIN, AI.OPEN\_HIGH\_MAX)},

    {'x': 9, 'y': 9, 'weight': getRandom(AI.OPEN\_HIGH\_MIN, AI.OPEN\_HIGH\_MAX)},

    {'x': 0, 'y': 0, 'weight': getRandom(AI.OPEN\_HIGH\_MIN, AI.OPEN\_HIGH\_MAX)}

];

// Scouts the grid based on max probability, and shoots at the cell

// that has the highest probability of containing a ship

AI.prototype.shoot = function() {

    var maxProbability = 0;

    var maxProbCoords;

    var maxProbs = [];

    for (var i = 0; i < AI.OPENINGS.length; i++) {

        var cell = AI.OPENINGS[i];

        if (this.probGrid[cell.x][cell.y] !== 0) {

            this.probGrid[cell.x][cell.y] += cell.weight;

        }

    }

    for (var x = 0; x < Game.size; x++) {

        for (var y = 0; y < Game.size; y++) {

            if (this.probGrid[x][y] > maxProbability) {

                maxProbability = this.probGrid[x][y];

                maxProbs = [{'x': x, 'y': y}]; // Replace the array

            } else if (this.probGrid[x][y] === maxProbability) {

                maxProbs.push({'x': x, 'y': y});

            }

        }

    }

    maxProbCoords = Math.random() < AI.RANDOMNESS ?

    maxProbs[Math.floor(Math.random() \* maxProbs.length)] :

    maxProbs[0];

    var result = this.gameObject.shoot(maxProbCoords.x, maxProbCoords.y, CONST.HUMAN\_PLAYER);

    // If the game ends, the next lines need to be skipped.

    if (Game.gameOver) {

        Game.gameOver = false;

        return;

    }

    this.virtualGrid.cells[maxProbCoords.x][maxProbCoords.y] = result;

    // If you hit a ship, check to make sure if you've sunk it.

    if (result === CONST.TYPE\_HIT) {

        var humanShip = this.findHumanShip(maxProbCoords.x, maxProbCoords.y);

        if (humanShip.isSunk()) {

            // Remove any ships from the roster that have been sunk

            var shipTypes = [];

            for (var k = 0; k < this.virtualFleet.fleetRoster.length; k++) {

                shipTypes.push(this.virtualFleet.fleetRoster[k].type);

            }

            var index = shipTypes.indexOf(humanShip.type);

            this.virtualFleet.fleetRoster.splice(index, 1);

            // Update the virtual grid with the sunk ship's cells

            var shipCells = humanShip.getAllShipCells();

            for (var \_i = 0; \_i < shipCells.length; \_i++) {

                this.virtualGrid.cells[shipCells[\_i].x][shipCells[\_i].y] = CONST.TYPE\_SUNK;

            }

        }

    }

    // Update probability grid after each shot

    this.updateProbs();

};

// Update the probability grid

AI.prototype.updateProbs = function() {

    var roster = this.virtualFleet.fleetRoster;

    var coords;

    this.resetProbs();

    for (var k = 0; k < roster.length; k++) {

        for (var x = 0; x < Game.size; x++) {

            for (var y = 0; y < Game.size; y++) {

                if (roster[k].isLegal(x, y, Ship.DIRECTION\_VERTICAL)) {

                    roster[k].create(x, y, Ship.DIRECTION\_VERTICAL, true);

                    coords = roster[k].getAllShipCells();

                    if (this.passesThroughHitCell(coords)) {

                        for (var i = 0; i < coords.length; i++) {

                            this.probGrid[coords[i].x][coords[i].y] += AI.PROB\_WEIGHT \* this.numHitCellsCovered(coords);

                        }

                    } else {

                        for (var \_i = 0; \_i < coords.length; \_i++) {

                            this.probGrid[coords[\_i].x][coords[\_i].y]++;

                        }

                    }

                }

                if (roster[k].isLegal(x, y, Ship.DIRECTION\_HORIZONTAL)) {

                    roster[k].create(x, y, Ship.DIRECTION\_HORIZONTAL, true);

                    coords = roster[k].getAllShipCells();

                    if (this.passesThroughHitCell(coords)) {

                        for (var j = 0; j < coords.length; j++) {

                            this.probGrid[coords[j].x][coords[j].y] += AI.PROB\_WEIGHT \* this.numHitCellsCovered(coords);

                        }

                    } else {

                        for (var \_j = 0; \_j < coords.length; \_j++) {

                            this.probGrid[coords[\_j].x][coords[\_j].y]++;

                        }

                    }

                }

                // Set hit cells to probability zero so the AI doesn't

                // target cells that are already hit

                if (this.virtualGrid.cells[x][y] === CONST.TYPE\_HIT) {

                    this.probGrid[x][y] = 0;

                }

            }

        }

    }

};

// Initializes the probability grid for targeting

AI.prototype.initProbs = function() {

    for (var x = 0; x < Game.size; x++) {

        var row = [];

        this.probGrid[x] = row;

        for (var y = 0; y < Game.size; y++) {

            row.push(0);

        }

    }

};

// Resets the probability grid to all 0.

AI.prototype.resetProbs = function() {

    for (var x = 0; x < Game.size; x++) {

        for (var y = 0; y < Game.size; y++) {

            this.probGrid[x][y] = 0;

        }

    }

};

AI.prototype.metagame = function() {

    // Inputs:

    // Proximity of hit cells to edge

    // Proximity of hit cells to each other

    // Edit the probability grid by multiplying each cell with a new probability weight (e.g. 0.4, or 3).

    // Set this as a CONST and make 1-CONST the inverse for decreasing, or 2\*CONST for increasing

};

// Finds a human ship by coordinates

// Returns Ship

AI.prototype.findHumanShip = function(x, y) {

    return this.gameObject.humanFleet.findShipByCoords(x, y);

};

// Checks whether or not a given ship's cells passes through

// any cell that is hit.

// Returns boolean

AI.prototype.passesThroughHitCell = function(shipCells) {

    for (var i = 0; i < shipCells.length; i++) {

        if (this.virtualGrid.cells[shipCells[i].x][shipCells[i].y] === CONST.TYPE\_HIT) {

            return true;

        }

    }

    return false;

};

// Gives the number of hit cells the ships passes through. The more

// cells this is, the more probable the ship exists in those coordinates

// Returns int

AI.prototype.numHitCellsCovered = function(shipCells) {

    var cells = 0;

    for (var i = 0; i < shipCells.length; i++) {

        if (this.virtualGrid.cells[shipCells[i].x][shipCells[i].y] === CONST.TYPE\_HIT) {

            cells++;

        }

    }

    return cells;

};

1. **Fleet object**

// Places ships randomly on the board

// TODO: Avoid placing ships too close to each other

Fleet.prototype.placeShipsRandomly = function() {

    var shipCoords;

    for (var i = 0; i < this.fleetRoster.length; i++) {

        var illegalPlacement = true;

        // Prevents the random placement of already placed ships

        if(this.player === CONST.HUMAN\_PLAYER && Game.usedShips[i] === CONST.USED) {

            continue;

        }

        while (illegalPlacement) {

            var randomX = Math.floor(Game.size \* Math.random());

            var randomY = Math.floor(Game.size \* Math.random());

            var randomDirection = Math.floor(2\*Math.random());

            if (this.fleetRoster[i].isLegal(randomX, randomY, randomDirection)) {

                this.fleetRoster[i].create(randomX, randomY, randomDirection, false);

                shipCoords = this.fleetRoster[i].getAllShipCells();

                illegalPlacement = false;

            } else {

                continue;

            }

        }

        if (this.player === CONST.HUMAN\_PLAYER && Game.usedShips[i] !== CONST.USED) {

            for (var j = 0; j < shipCoords.length; j++) {

                this.playerGrid.updateCell(shipCoords[j].x, shipCoords[j].y, 'ship', this.player);

                Game.usedShips[i] = CONST.USED;

            }

        }

    }

};

// Finds a ship by location

// Returns the ship object located at (x, y)

// If no ship exists at (x, y), this returns null instead

Fleet.prototype.findShipByCoords = function(x, y) {

    for (var i = 0; i < this.fleetRoster.length; i++) {

        var currentShip = this.fleetRoster[i];

        if (currentShip.direction === Ship.DIRECTION\_VERTICAL) {

            if (y === currentShip.yPosition && x >= currentShip.xPosition && x < currentShip.xPosition + currentShip.shipLength) {

                return currentShip;

            } else {

                continue;

            }

        } else {

            if (x === currentShip.xPosition && y >= currentShip.yPosition && y < currentShip.yPosition + currentShip.shipLength) {

                return currentShip;

            } else {

                continue;

            }

        }

    }

    return null;

};

// Finds a ship by its type

// Param shipType is a string

// Returns the ship object that is of type shipType

// If no ship exists, this returns null.

Fleet.prototype.findShipByType = function(shipType) {

    for (var i = 0; i < this.fleetRoster.length; i++) {

        if (this.fleetRoster[i].type === shipType) {

            return this.fleetRoster[i];

        }

    }

    return null;

};

// Checks to see if all ships have been sunk

// Returns boolean

Fleet.prototype.allShipsSunk = function() {

    for (var i = 0; i < this.fleetRoster.length; i++) {

        // If one or more ships are not sunk, then the sentence "all ships are sunk" is false.

        if (this.fleetRoster[i].sunk === false) {

            return false;

        }

    }

    return true;

};

1. **Game object**

// Game manager object

// Constructor

function Game(size) {

    Game.size = size;

    this.shotsTaken = 0;

    this.createGrid();

    this.init();

}

Game.size = 10; // Default grid size is 10x10

Game.gameOver = false;

// Checks if the game is won, and if it is, re-initializes the game

Game.prototype.checkIfWon = function() {

    if (this.computerFleet.allShipsSunk()) {

        alert('Congratulations, you win!');

        Game.gameOver = true;

        Game.stats.wonGame();

        Game.stats.syncStats();

        Game.stats.updateStatsSidebar();

        this.showRestartSidebar();

    } else if (this.humanFleet.allShipsSunk()) {

        alert('Yarr! The computer sank all your ships. Try again.');

        Game.gameOver = true;

        Game.stats.lostGame();

        Game.stats.syncStats();

        Game.stats.updateStatsSidebar();

        this.showRestartSidebar();

    }

};

// Shoots at the target player on the grid.

// Returns {int} Constants.TYPE: What the shot uncovered

Game.prototype.shoot = function(x, y, targetPlayer) {

    var targetGrid;

    var targetFleet;

    if (targetPlayer === CONST.HUMAN\_PLAYER) {

        targetGrid = this.humanGrid;

        targetFleet = this.humanFleet;

    } else if (targetPlayer === CONST.COMPUTER\_PLAYER) {

        targetGrid = this.computerGrid;

        targetFleet = this.computerFleet;

    } else {

        // Should never be called

        console.log("There was an error trying to find the correct player to target");

    }

    if (targetGrid.isDamagedShip(x, y)) {

        return null;

    } else if (targetGrid.isMiss(x, y)) {

        return null;

    } else if (targetGrid.isUndamagedShip(x, y)) {

        // update the board/grid

        targetGrid.updateCell(x, y, 'hit', targetPlayer);

        // IMPORTANT: This function needs to be called \_after\_ updating the cell to a 'hit',

        // because it overrides the CSS class to 'sunk' if we find that the ship was sunk

        targetFleet.findShipByCoords(x, y).incrementDamage(); // increase the damage

        this.checkIfWon();

        return CONST.TYPE\_HIT;

    } else {

        targetGrid.updateCell(x, y, 'miss', targetPlayer);

        this.checkIfWon();

        return CONST.TYPE\_MISS;

    }

};

// Creates click event listeners on each one of the 100 grid cells

Game.prototype.shootListener = function(e) {

    var self = e.target.self;

    // Extract coordinates from event listener

    var x = parseInt(e.target.getAttribute('data-x'), 10);

    var y = parseInt(e.target.getAttribute('data-y'), 10);

    var result = null;

    if (self.readyToPlay) {

        result = self.shoot(x, y, CONST.COMPUTER\_PLAYER);

        // Remove the tutorial arrow

        if (gameTutorial.showTutorial) {

            gameTutorial.nextStep();

        }

    }

    if (result !== null && !Game.gameOver) {

        Game.stats.incrementShots();

        if (result === CONST.TYPE\_HIT) {

            Game.stats.hitShot();

        }

        // The AI shoots iff the player clicks on a cell that he/she hasn't

        // already clicked on yet

        self.robot.shoot();

    } else {

        Game.gameOver = false;

    }

};

// Creates click event listeners on each of the ship names in the roster

Game.prototype.rosterListener = function(e) {

    var self = e.target.self;

    // Remove all classes of 'placing' from the fleet roster first

    var roster = document.querySelectorAll('.fleet-roster li');

    for (var i = 0; i < roster.length; i++) {

        var classes = roster[i].getAttribute('class') || '';

        classes = classes.replace('placing', '');

        roster[i].setAttribute('class', classes);

    }

    // Move the highlight to the next step

    if (gameTutorial.currentStep === 1) {

        gameTutorial.nextStep();

    }

    // Set the class of the target ship to 'placing'

    Game.placeShipType = e.target.getAttribute('id');

    document.getElementById(Game.placeShipType).setAttribute('class', 'placing');

    Game.placeShipDirection = parseInt(document.getElementById('rotate-button').getAttribute('data-direction'), 10);

    self.placingOnGrid = true;

};

// Creates click event listeners on the human player's grid to handle

// ship placement after the user has selected a ship name

Game.prototype.placementListener = function(e) {

    var self = e.target.self;

    if (self.placingOnGrid) {

        // Extract coordinates from event listener

        var x = parseInt(e.target.getAttribute('data-x'), 10);

        var y = parseInt(e.target.getAttribute('data-y'), 10);

        // Don't screw up the direction if the user tries to place again.

        var successful = self.humanFleet.placeShip(x, y, Game.placeShipDirection, Game.placeShipType);

        if (successful) {

            // Done placing this ship

            self.endPlacing(Game.placeShipType);

            // Remove the helper arrow

            if (gameTutorial.currentStep === 2) {

                gameTutorial.nextStep();

            }

            self.placingOnGrid = false;

            if (self.areAllShipsPlaced()) {

                var el = document.getElementById('rotate-button');

                el.addEventListener(transitionEndEventName(),(function(){

                    el.setAttribute('class', 'hidden');

                    if (gameTutorial.showTutorial) {

                        document.getElementById('start-game').setAttribute('class', 'highlight');

                    } else {

                        document.getElementById('start-game').removeAttribute('class');

                    }

                }),false);

                el.setAttribute('class', 'invisible');

            }

        }

    }

};

// Creates mouseover event listeners that handles mouseover on the

// human player's grid to draw a phantom ship implying that the user

// is allowed to place a ship there

Game.prototype.placementMouseover = function(e) {

    var self = e.target.self;

    if (self.placingOnGrid) {

        var x = parseInt(e.target.getAttribute('data-x'), 10);

        var y = parseInt(e.target.getAttribute('data-y'), 10);

        var classes;

        var fleetRoster = self.humanFleet.fleetRoster;

        for (var i = 0; i < fleetRoster.length; i++) {

            var shipType = fleetRoster[i].type;

            if (Game.placeShipType === shipType && fleetRoster[i].isLegal(x, y, Game.placeShipDirection)) {

                // Virtual ship

                fleetRoster[i].create(x, y, Game.placeShipDirection, true);

                Game.placeShipCoords = fleetRoster[i].getAllShipCells();

                for (var j = 0; j < Game.placeShipCoords.length; j++) {

                    var el = document.querySelector('.grid-cell-' + Game.placeShipCoords[j].x + '-' + Game.placeShipCoords[j].y);

                    classes = el.getAttribute('class');

                    // Check if the substring ' grid-ship' already exists to avoid adding it twice

                    if (classes.indexOf(' grid-ship') < 0) {

                        classes += ' grid-ship';

                        el.setAttribute('class', classes);

                    }

                }

            }

        }

    }

};

// Creates mouseout event listeners that un-draws the phantom ship

// on the human player's grid as the user hovers over a different cell

Game.prototype.placementMouseout = function(e) {

    var self = e.target.self;

    if (self.placingOnGrid) {

        for (var j = 0; j < Game.placeShipCoords.length; j++) {

            var el = document.querySelector('.grid-cell-' + Game.placeShipCoords[j].x + '-' + Game.placeShipCoords[j].y);

            classes = el.getAttribute('class');

            // Check if the substring ' grid-ship' already exists to avoid adding it twice

            if (classes.indexOf(' grid-ship') > -1) {

                classes = classes.replace(' grid-ship', '');

                el.setAttribute('class', classes);

            }

        }

    }

};

// Click handler for the Rotate Ship button

Game.prototype.toggleRotation = function(e) {

    // Toggle rotation direction

    var direction = parseInt(e.target.getAttribute('data-direction'), 10);

    if (direction === Ship.DIRECTION\_VERTICAL) {

        e.target.setAttribute('data-direction', '1');

        Game.placeShipDirection = Ship.DIRECTION\_HORIZONTAL;

    } else if (direction === Ship.DIRECTION\_HORIZONTAL) {

        e.target.setAttribute('data-direction', '0');

        Game.placeShipDirection = Ship.DIRECTION\_VERTICAL;

    }

};

// Click handler for the Start Game button

Game.prototype.startGame = function(e) {

    var self = e.target.self;

    var el = document.getElementById('roster-sidebar');

    var fn = function() {el.setAttribute('class', 'hidden');};

    el.addEventListener(transitionEndEventName(),fn,false);

    el.setAttribute('class', 'invisible');

    self.readyToPlay = true;

    // Advanced the tutorial step

    if (gameTutorial.currentStep === 3) {

        gameTutorial.nextStep();

    }

    el.removeEventListener(transitionEndEventName(),fn,false);

};

// Click handler for Restart Game button

Game.prototype.restartGame = function(e) {

    e.target.removeEventListener(e.type, arguments.callee);

    var self = e.target.self;

    document.getElementById('restart-sidebar').setAttribute('class', 'hidden');

    self.resetFogOfWar();

    self.init();

};

// Debugging function used to place all ships and just start

Game.prototype.placeRandomly = function(e){

    e.target.removeEventListener(e.type, arguments.callee);

    e.target.self.humanFleet.placeShipsRandomly();

    e.target.self.readyToPlay = true;

    document.getElementById('roster-sidebar').setAttribute('class', 'hidden');

    this.setAttribute('class', 'hidden');

};

// Ends placing the current ship

Game.prototype.endPlacing = function(shipType) {

    document.getElementById(shipType).setAttribute('class', 'placed');

    // Mark the ship as 'used'

    Game.usedShips[CONST.AVAILABLE\_SHIPS.indexOf(shipType)] = CONST.USED;

    // Wipe out the variable when you're done with it

    Game.placeShipDirection = null;

    Game.placeShipType = '';

    Game.placeShipCoords = [];

};

// Checks whether or not all ships are done placing

// Returns boolean

Game.prototype.areAllShipsPlaced = function() {

    var playerRoster = document.querySelectorAll('.fleet-roster li');

    for (var i = 0; i < playerRoster.length; i++) {

        if (playerRoster[i].getAttribute('class') === 'placed') {

            continue;

        } else {

            return false;

        }

    }

    // Reset temporary variables

    Game.placeShipDirection = 0;

    Game.placeShipType = '';

    Game.placeShipCoords = [];

    return true;

};

// Resets the fog of war

Game.prototype.resetFogOfWar = function() {

    for (var i = 0; i < Game.size; i++) {

        for (var j = 0; j < Game.size; j++) {

            this.humanGrid.updateCell(i, j, 'empty', CONST.HUMAN\_PLAYER);

            this.computerGrid.updateCell(i, j, 'empty', CONST.COMPUTER\_PLAYER);

        }

    }

    // Reset all values to indicate the ships are ready to be placed again

    Game.usedShips = Game.usedShips.map(function(){return CONST.UNUSED;});

};

// Resets CSS styling of the sidebar

Game.prototype.resetRosterSidebar = function() {

    var els = document.querySelector('.fleet-roster').querySelectorAll('li');

    for (var i = 0; i < els.length; i++) {

        els[i].removeAttribute('class');

    }

    if (gameTutorial.showTutorial) {

        gameTutorial.nextStep();

    } else {

        document.getElementById('roster-sidebar').removeAttribute('class');

    }

    document.getElementById('rotate-button').removeAttribute('class');

    document.getElementById('start-game').setAttribute('class', 'hidden');

    if (DEBUG\_MODE) {

        document.getElementById('place-randomly').removeAttribute('class');

    }

};

Game.prototype.showRestartSidebar = function() {

    var sidebar = document.getElementById('restart-sidebar');

    sidebar.setAttribute('class', 'highlight');

    // Deregister listeners

    var computerCells = document.querySelector('.computer-player').childNodes;

    for (var j = 0; j < computerCells.length; j++) {

        computerCells[j].removeEventListener('click', this.shootListener, false);

    }

    var playerRoster = document.querySelector('.fleet-roster').querySelectorAll('li');

    for (var i = 0; i < playerRoster.length; i++) {

        playerRoster[i].removeEventListener('click', this.rosterListener, false);

    }

    var restartButton = document.getElementById('restart-game');

    restartButton.addEventListener('click', this.restartGame, false);

    restartButton.self = this;

};

// Generates the HTML divs for the grid for both players

Game.prototype.createGrid = function() {

    var gridDiv = document.querySelectorAll('.grid');

    for (var grid = 0; grid < gridDiv.length; grid++) {

        gridDiv[grid].removeChild(gridDiv[grid].querySelector('.no-js'));

        for (var i = 0; i < Game.size; i++) {

            for (var j = 0; j < Game.size; j++) {

                var el = document.createElement('div');

                el.setAttribute('data-x', i);

                el.setAttribute('data-y', j);

                el.setAttribute('class', 'grid-cell grid-cell-' + i + '-' + j);

                gridDiv[grid].appendChild(el);

            }

        }

    }

};

// Initializes the Game. Also resets the game if previously initialized

Game.prototype.init = function() {

    this.humanGrid = new Grid(Game.size);

    this.computerGrid = new Grid(Game.size);

    this.humanFleet = new Fleet(this.humanGrid, CONST.HUMAN\_PLAYER);

    this.computerFleet = new Fleet(this.computerGrid, CONST.COMPUTER\_PLAYER);

    this.robot = new AI(this);

    Game.stats = new Stats();

    Game.stats.updateStatsSidebar();

    // Reset game variables

    this.shotsTaken = 0;

    this.readyToPlay = false;

    this.placingOnGrid = false;

    Game.placeShipDirection = 0;

    Game.placeShipType = '';

    Game.placeShipCoords = [];

    this.resetRosterSidebar();

    // Add a click listener for the Grid.shoot() method for all cells

    // Only add this listener to the computer's grid

    var computerCells = document.querySelector('.computer-player').childNodes;

    for (var j = 0; j < computerCells.length; j++) {

        computerCells[j].self = this;

        computerCells[j].addEventListener('click', this.shootListener, false);

    }

    // Add a click listener to the roster

    var playerRoster = document.querySelector('.fleet-roster').querySelectorAll('li');

    for (var i = 0; i < playerRoster.length; i++) {

        playerRoster[i].self = this;

        playerRoster[i].addEventListener('click', this.rosterListener, false);

    }

    // Add a click listener to the human player's grid while placing

    var humanCells = document.querySelector('.human-player').childNodes;

    for (var k = 0; k < humanCells.length; k++) {

        humanCells[k].self = this;

        humanCells[k].addEventListener('click', this.placementListener, false);

        humanCells[k].addEventListener('mouseover', this.placementMouseover, false);

        humanCells[k].addEventListener('mouseout', this.placementMouseout, false);

    }

    var rotateButton = document.getElementById('rotate-button');

    rotateButton.addEventListener('click', this.toggleRotation, false);

    var startButton = document.getElementById('start-game');

    startButton.self = this;

    startButton.addEventListener('click', this.startGame, false);

    var resetButton = document.getElementById('reset-stats');

    resetButton.addEventListener('click', Game.stats.resetStats, false);

    var randomButton = document.getElementById('place-randomly');

    randomButton.self = this;

    randomButton.addEventListener('click', this.placeRandomly, false);

    this.computerFleet.placeShipsRandomly();

};

1. **Grid object**

// Grid object

// Constructor

function Grid(size) {

    this.size = size;

    this.cells = [];

    this.init();

}

// Initialize and populate the grid

Grid.prototype.init = function() {

    for (var x = 0; x < this.size; x++) {

        var row = [];

        this.cells[x] = row;

        for (var y = 0; y < this.size; y++) {

            row.push(CONST.TYPE\_EMPTY);

        }

    }

};

// Updates the cell's CSS class based on the type passed in

Grid.prototype.updateCell = function(x, y, type, targetPlayer) {

    var player;

    if (targetPlayer === CONST.HUMAN\_PLAYER) {

        player = 'human-player';

    } else if (targetPlayer === CONST.COMPUTER\_PLAYER) {

        player = 'computer-player';

    } else {

        // Should never be called

        console.log("There was an error trying to find the correct player's grid");

    }

    switch (type) {

        case CONST.CSS\_TYPE\_EMPTY:

            this.cells[x][y] = CONST.TYPE\_EMPTY;

            break;

        case CONST.CSS\_TYPE\_SHIP:

            this.cells[x][y] = CONST.TYPE\_SHIP;

            break;

        case CONST.CSS\_TYPE\_MISS:

            this.cells[x][y] = CONST.TYPE\_MISS;

            break;

        case CONST.CSS\_TYPE\_HIT:

            this.cells[x][y] = CONST.TYPE\_HIT;

            break;

        case CONST.CSS\_TYPE\_SUNK:

            this.cells[x][y] = CONST.TYPE\_SUNK;

            break;

        default:

            this.cells[x][y] = CONST.TYPE\_EMPTY;

            break;

    }

    var classes = ['grid-cell', 'grid-cell-' + x + '-' + y, 'grid-' + type];

    document.querySelector('.' + player + ' .grid-cell-' + x + '-' + y).setAttribute('class', classes.join(' '));

};

// Checks to see if a cell contains an undamaged ship

// Returns boolean

Grid.prototype.isUndamagedShip = function(x, y) {

    return this.cells[x][y] === CONST.TYPE\_SHIP;

};

// Checks to see if the shot was missed. This is equivalent

// to checking if a cell contains a cannonball

// Returns boolean

Grid.prototype.isMiss = function(x, y) {

    return this.cells[x][y] === CONST.TYPE\_MISS;

};

// Checks to see if a cell contains a damaged ship,

// either hit or sunk.

// Returns boolean

Grid.prototype.isDamagedShip = function(x, y) {

    return this.cells[x][y] === CONST.TYPE\_HIT || this.cells[x][y] === CONST.TYPE\_SUNK;

};

1. **Ship object**

// Ship object

// Constructor

function Ship(type, playerGrid, player) {

    this.damage = 0;

    this.type = type;

    this.playerGrid = playerGrid;

    this.player = player;

    switch (this.type) {

        case CONST.AVAILABLE\_SHIPS[0]:

            this.shipLength = 5;

            break;

        case CONST.AVAILABLE\_SHIPS[1]:

            this.shipLength = 4;

            break;

        case CONST.AVAILABLE\_SHIPS[2]:

            this.shipLength = 3;

            break;

        case CONST.AVAILABLE\_SHIPS[3]:

            this.shipLength = 3;

            break;

        case CONST.AVAILABLE\_SHIPS[4]:

            this.shipLength = 2;

            break;

        default:

            this.shipLength = 3;

            break;

    }

    this.maxDamage = this.shipLength;

    this.sunk = false;

}

// Checks to see if the placement of a ship is legal

// Returns boolean

Ship.prototype.isLegal = function(x, y, direction) {

    // first, check if the ship is within the grid...

    if (this.withinBounds(x, y, direction)) {

        // ...then check to make sure it doesn't collide with another ship

        for (var i = 0; i < this.shipLength; i++) {

            if (direction === Ship.DIRECTION\_VERTICAL) {

                if (this.playerGrid.cells[x + i][y] === CONST.TYPE\_SHIP ||

                    this.playerGrid.cells[x + i][y] === CONST.TYPE\_MISS ||

                    this.playerGrid.cells[x + i][y] === CONST.TYPE\_SUNK) {

                    return false;

                }

            } else {

                if (this.playerGrid.cells[x][y + i] === CONST.TYPE\_SHIP ||

                    this.playerGrid.cells[x][y + i] === CONST.TYPE\_MISS ||

                    this.playerGrid.cells[x][y + i] === CONST.TYPE\_SUNK) {

                    return false;

                }

            }

        }

        return true;

    } else {

        return false;

    }

};

// Checks to see if the ship is within bounds of the grid

// Returns boolean

Ship.prototype.withinBounds = function(x, y, direction) {

    if (direction === Ship.DIRECTION\_VERTICAL) {

        return x + this.shipLength <= Game.size;

    } else {

        return y + this.shipLength <= Game.size;

    }

};

// Increments the damage counter of a ship

// Returns Ship

Ship.prototype.incrementDamage = function() {

    this.damage++;

    if (this.isSunk()) {

        this.sinkShip(false); // Sinks the ship

    }

};

// Checks to see if the ship is sunk

// Returns boolean

Ship.prototype.isSunk = function() {

    return this.damage >= this.maxDamage;

};

// Sinks the ship

Ship.prototype.sinkShip = function(virtual) {

    this.damage = this.maxDamage; // Force the damage to exceed max damage

    this.sunk = true;

    // Make the CSS class sunk, but only if the ship is not virtual

    if (!virtual) {

        var allCells = this.getAllShipCells();

        for (var i = 0; i < this.shipLength; i++) {

            this.playerGrid.updateCell(allCells[i].x, allCells[i].y, 'sunk', this.player);

        }

    }

};

/\*\*

 \* Gets all the ship cells

 \*

 \* Returns an array with all (x, y) coordinates of the ship:

 \* e.g.

 \* [

 \*  {'x':2, 'y':2},

 \*  {'x':3, 'y':2},

 \*  {'x':4, 'y':2}

 \* ]

 \*/

Ship.prototype.getAllShipCells = function() {

    var resultObject = [];

    for (var i = 0; i < this.shipLength; i++) {

        if (this.direction === Ship.DIRECTION\_VERTICAL) {

            resultObject[i] = {'x': this.xPosition + i, 'y': this.yPosition};

        } else {

            resultObject[i] = {'x': this.xPosition, 'y': this.yPosition + i};

        }

    }

    return resultObject;

};

// Initializes a ship with the given coordinates and direction (bearing).

// If the ship is declared "virtual", then the ship gets initialized with

// its coordinates but DOESN'T get placed on the grid.

Ship.prototype.create = function(x, y, direction, virtual) {

    // This function assumes that you've already checked that the placement is legal

    this.xPosition = x;

    this.yPosition = y;

    this.direction = direction;

    // If the ship is virtual, don't add it to the grid.

    if (!virtual) {

        for (var i = 0; i < this.shipLength; i++) {

            if (this.direction === Ship.DIRECTION\_VERTICAL) {

                this.playerGrid.cells[x + i][y] = CONST.TYPE\_SHIP;

            } else {

                this.playerGrid.cells[x][y + i] = CONST.TYPE\_SHIP;

            }

        }

    }

};

// direction === 0 when the ship is facing north/south

// direction === 1 when the ship is facing east/west

Ship.DIRECTION\_VERTICAL = 0;

Ship.DIRECTION\_HORIZONTAL = 1;

1. **Stats object**

// Game Statistics

function Stats(){

    this.shotsTaken = 0;

    this.shotsHit = 0;

    this.totalShots = parseInt(localStorage.getItem('totalShots'), 10) || 0;

    this.totalHits = parseInt(localStorage.getItem('totalHits'), 10) || 0;

    this.gamesPlayed = parseInt(localStorage.getItem('gamesPlayed'), 10) || 0;

    this.gamesWon = parseInt(localStorage.getItem('gamesWon'), 10) || 0;

    this.uuid = localStorage.getItem('uuid') || this.createUUID();

    if (DEBUG\_MODE) {

        this.skipCurrentGame = true;

    }

}

Stats.prototype.incrementShots = function() {

    this.shotsTaken++;

};

Stats.prototype.hitShot = function() {

    this.shotsHit++;

};

Stats.prototype.wonGame = function() {

    this.gamesPlayed++;

    this.gamesWon++;

    if (!DEBUG\_MODE) {

        ga('send', 'event', 'gameOver', 'win', this.uuid);

    }

};

Stats.prototype.lostGame = function() {

    this.gamesPlayed++;

    if (!DEBUG\_MODE) {

        ga('send', 'event', 'gameOver', 'lose', this.uuid);

    }

};

// Saves the game statistics to localstorage, also uploads where the user placed

// their ships to Google Analytics so that in the future I'll be able to see

// which cells humans are disproportionately biased to place ships on.

Stats.prototype.syncStats = function() {

    if(!this.skipCurrentGame) {

        var totalShots = parseInt(localStorage.getItem('totalShots'), 10) || 0;

        totalShots += this.shotsTaken;

        var totalHits = parseInt(localStorage.getItem('totalHits'), 10) || 0;

        totalHits += this.shotsHit;

        localStorage.setItem('totalShots', totalShots);

        localStorage.setItem('totalHits', totalHits);

        localStorage.setItem('gamesPlayed', this.gamesPlayed);

        localStorage.setItem('gamesWon', this.gamesWon);

        localStorage.setItem('uuid', this.uuid);

    } else {

        this.skipCurrentGame = false;

    }

    var stringifiedGrid = '';

    for (var x = 0; x < Game.size; x++) {

        for (var y = 0; y < Game.size; y++) {

            stringifiedGrid += '(' + x + ',' + y + '):' + mainGame.humanGrid.cells[x][y] + ';\n';

        }

    }

    if (!DEBUG\_MODE) {

        ga('send', 'event', 'humanGrid', stringifiedGrid, this.uuid);

    }

};

// Updates the sidebar display with the current statistics

Stats.prototype.updateStatsSidebar = function() {

    var elWinPercent = document.getElementById('stats-wins');

    var elAccuracy = document.getElementById('stats-accuracy');

    elWinPercent.innerHTML = this.gamesWon + " of " + this.gamesPlayed;

    elAccuracy.innerHTML = Math.round((100 \* this.totalHits / this.totalShots) || 0) + "%";

};

// Reset all game vanity statistics to zero. Doesn't reset your uuid.

Stats.prototype.resetStats = function(e) {

    // Skip tracking stats until the end of the current game or else

    // the accuracy percentage will be wrong (since you are tracking

    // hits that didn't start from the beginning of the game)

    Game.stats.skipCurrentGame = true;

    localStorage.setItem('totalShots', 0);

    localStorage.setItem('totalHits', 0);

    localStorage.setItem('gamesPlayed', 0);

    localStorage.setItem('gamesWon', 0);

    localStorage.setItem('showTutorial', true);

    Game.stats.shotsTaken = 0;

    Game.stats.shotsHit = 0;

    Game.stats.totalShots = 0;

    Game.stats.totalHits = 0;

    Game.stats.gamesPlayed = 0;

    Game.stats.gamesWon = 0;

    Game.stats.updateStatsSidebar();

};

Stats.prototype.createUUID = function(len, radix) {

    var chars = '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'.split(''),

    uuid = [], i;

    radix = radix || chars.length;

    if (len) {

        // Compact form

        for (i = 0; i < len; i++) uuid[i] = chars[0 | Math.random()\*radix];

    } else {

        // rfc4122, version 4 form

        var r;

        // rfc4122 requires these characters

        uuid[8] = uuid[13] = uuid[18] = uuid[23] = '-';

        uuid[14] = '4';

        // Fill in random data.  At i==19 set the high bits of clock sequence as

        // per rfc4122, sec. 4.1.5

        for (i = 0; i < 36; i++) {

            if (!uuid[i]) {

                r = 0 | Math.random()\*16;

                uuid[i] = chars[(i === 19) ? (r & 0x3) | 0x8 : r];

            }

        }

    }

    return uuid.join('');

};

1. **Tutorial object**

/ Tutorial Object

// Constructor

function Tutorial() {

    this.currentStep = 0;

    // Check if 'showTutorial' is initialized, if it's uninitialized, set it to true.

    this.showTutorial = localStorage.getItem('showTutorial') !== 'false';

}

// Advances the tutorial to the next step

Tutorial.prototype.nextStep = function() {

    var humanGrid = document.querySelector('.human-player');

    var computerGrid = document.querySelector('.computer-player');

    switch (this.currentStep) {

        case 0:

            document.getElementById('roster-sidebar').setAttribute('class', 'highlight');

            document.getElementById('step1').setAttribute('class', 'current-step');

            this.currentStep++;

            break;

        case 1:

            document.getElementById('roster-sidebar').removeAttribute('class');

            document.getElementById('step1').removeAttribute('class');

            humanGrid.setAttribute('class', humanGrid.getAttribute('class') + ' highlight');

            document.getElementById('step2').setAttribute('class', 'current-step');

            this.currentStep++;

            break;

        case 2:

            document.getElementById('step2').removeAttribute('class');

            var humanClasses = humanGrid.getAttribute('class');

            humanClasses = humanClasses.replace(' highlight', '');

            humanGrid.setAttribute('class', humanClasses);

            this.currentStep++;

            break;

        case 3:

            computerGrid.setAttribute('class', computerGrid.getAttribute('class') + ' highlight');

            document.getElementById('step3').setAttribute('class', 'current-step');

            this.currentStep++;

            break;

        case 4:

            var computerClasses = computerGrid.getAttribute('class');

            document.getElementById('step3').removeAttribute('class');

            computerClasses = computerClasses.replace(' highlight', '');

            computerGrid.setAttribute('class', computerClasses);

            document.getElementById('step4').setAttribute('class', 'current-step');

            this.currentStep++;

            break;

        case 5:

            document.getElementById('step4').removeAttribute('class');

            this.currentStep = 6;

            this.showTutorial = false;

            localStorage.setItem('showTutorial', false);

            break;

        default:

            break;

    }

};

## **View**

**@import url(https://fonts.googleapis**.com/css?family=Open+Sans:400italic,700italic,400,700);

html, body {

    height: 100%;

    width: 100%;

    padding: 0;

    margin: 0;

    background-color: rgb(218,227,241);

    min-width: 860px;

}

h1, h2, h3, h4, h5, h6, p, li, span {

    font-family: 'Open Sans', Helvetica, Arial, sans-serif;

    color: #222222;

}

a {

    color: #222222;

    font-weight: bold;

}

    a:hover {

        text-decoration: none;

    }

div {

    padding: 0;

    margin: 0;

}

button {

    font-family: 'Open Sans', Helvetica, Arial, sans-serif;

    font-size: 1.2em;

    color: #EEEEEE;

    background-color: #25567B;

    border: none;

    margin: 1em auto 0 auto;

    padding: 10px;

    border-radius: 5px;

}

    button:hover {

        cursor: pointer;

        background-color: #ffffff;

    }

    button:active {

        background-color: #99C2E1;

    }

    #start-game,

    #place-randomly,

    #restart-game {

        position: relative;

        color: #FFFFFF;

        background-color: #FF9200;

    }

        #start-game:hover,

        #place-randomly:hover,

        #restart-game:hover {

            background-color: #FFB655;

        }

        #start-game:active,

        #place-randomly:active,

        #restart-game:active {

            background-color: #FFCE8E;

        }

#prefetch1 {

    background: url('../img/cross-icon.svg');

}

#prefetch2 {

    background: url('../img/cross-icon.png');

}

#prefetch3 {

    background: url('../img/crosshair.png');

}

.prefetch {

    background-repeat: no-repeat;

    background-position: -9999px -9999px;

}

.tagline {

    margin-bottom: 2em;

}

    .instructions {

        width: auto;

        display: inline-block;

        text-align: left;

        margin: 0 auto 3em auto;

    }

.container {

    width: 870px;

    text-align: center;

    margin: 20px auto 100px auto;

}

.game-container {

    position: relative;

}

    #roster-sidebar, #stats-sidebar, #restart-sidebar {

        width: 150px;

        padding: 20px;

        margin: 0;

        background-color: #EEEEEE;

        position: absolute;

        top: 4.5em;

        -webkit-border-radius: 5px;

        -moz-border-radius: 5px;

        border-radius: 5px;

    }

    #roster-sidebar, #restart-sidebar {

        left: -200px;

    }

    #stats-sidebar {

        right: -200px;

    }

        #roster-sidebar h2, #stats-sidebar h2, #restart-sidebar h2 {

            margin: 0 0 1em 0;

            text-align: center;

        }

.fleet-roster {

    position: relative;

    margin: 0;

    padding: 0;

    list-style: none;

}

.fleet-roster, button {

    opacity: 1;

    -webkit-transition: opacity 0.5s ease-out;

    -o-transition: opacity 0.5s ease-out;

    transition: opacity 0.5s ease-out;

}

    .fleet-roster li {

        margin: 0.5em 0;

    }

        .fleet-roster li:hover {

            color: #aaa;

            cursor: pointer;

        }

    .fleet-roster .placing {

        font-weight: bold;

    }

    .fleet-roster .placed {

        visibility: hidden;

        font-weight: bold;

        opacity: 0;

        -webkit-transition: all 0.2s ease-in;

        -o-transition: all 0.2s ease-in;

        transition: all 0.2s ease-in;

    }

    .invisible {

        opacity: 0;

        -webkit-transition: opacity 0.5s ease-in;

        -o-transition: opacity 0.5s ease-in;

        transition: opacity 0.5s ease-in;

        z-index: -20;

    }

    .hidden {

        display: none;

    }

.grid-container {

    width: 430px;

    display: inline-block;

}

.grid-container h2 {

    width: 430px;

    display: inline-block;

}

.grid {

    position: relative;

    vertical-align: top;

    padding: 5px;

    height: 420px;

    width: 420px;

    background-color: #25567B;

    -webkit-border-radius: 5px;

    -moz-border-radius: 5px;

    border-radius: 5px;

}

    .grid-container:last-child {

        margin-left: 10px;

    }

    .no-js {

        width: 300px;

        height: 300px;

        margin: 100px auto;

        display: inline-block;

        color: #EEEEEE;

        font-size: 2em;

    }

    .grid-cell {

        vertical-align: top; /\*clears the vertical space between rows\*/

        height: 40px;

        width: 40px;

        display: inline-block;

        background-color: #99C2E1;

        margin: 1px;

        -webkit-border-radius: 2px;

        -moz-border-radius: 2px;

        border-radius: 2px;

    }

        .grid-cell:hover {

            cursor: pointer; /\* Fallback for IE \*/

            background-color: #66A3D2;

        }

        .human-player .grid-cell:hover {

            background-color: #99C2E1;

        }

        .computer-player .grid-cell:hover, .computer-player:hover {

            cursor: url('../img/crosshair.png') 16 16, crosshair;

        }

    .grid-ship, .human-player .grid-ship:hover {

        background-color: #808080;

    }

    .grid-miss, .grid-miss:hover, .human-player .grid-miss:hover {

        background-color: #FFFFFF;

        background-image: url('../img/cross-icon.png'); /\* Fallback \*/

        background-image: url('../img/cross-icon.svg');

        background-position: center;

        background-repeat: no-repeat;

    }

    .grid-hit, .grid-hit:hover, .human-player .grid-hit:hover {

        background-color: #F60018;

        background-image: url('../img/cross-icon.png'); /\* Fallback \*/

        background-image: url('../img/cross-icon.svg');

        background-position: center;

        background-repeat: no-repeat;

    }

    .grid-sunk, .grid-sunk:hover, .human-player .grid-sunk:hover {

        background-color: #222222;

    }

.highlight {

    overflow: visible; /\* Bugfix for IE \*/

}

    .highlight:before {

        content: "\2193";

        font-weight: bold;

        font-size: 75px;

        color: #FF9200;

        text-shadow: 0 0 5px #FF9200;

        position: absolute;

        width: 100px;

        height: 100px;

        top: -100px;

        left: 50%;

        margin-left: -50px;

        -webkit-animation: highlight 1.5s infinite;

        -o-animation: highlight 1.5s infinite;

        animation: highlight 1.5s infinite;

    }

.current-step {

    font-weight: bold;

}

@media(max-width: 1300px) {

    .container {

        width: 440px;

    }

    .grid-container:last-child {

        margin-left: 0;

    }

}

@-webkit-keyframes highlight {

    0% {

        -webkit-transform: translateY(0);

        transform: translateY(0);

    }

    50% {

        -webkit-transform: translateY(-20px);

        transform: translateY(-20px);

    }

    100% {

        -webkit-transform: translateY(0);

        transform: translateY(0);

    }

}

@-moz-keyframes highlight {

    0% {

        transform: translateY(0);

    }

    50% {

        transform: translateY(-20px);

    }

    100% {

        transform: translateY(0);

    }

}

@-o-keyframes highlight {

    0% {

        -o-transform: translateY(0);

        transform: translateY(0);

    }

    50% {

        -o-transform: translateY(-20px);

        transform: translateY(-20px);

    }

    100% {

        -o-transform: translateY(0);

        transform: translateY(0);

    }

}

@keyframes highlight {

    0% {

        -webkit-transform: translateY(0);

        -ms-transform: translateY(0);

        -o-transform: translateY(0);

        transform: translateY(0);

    }

    50% {

        -webkit-transform: translateY(-20px);

        -ms-transform: translateY(-20px);

        -o-transform: translateY(-20px);

        transform: translateY(-20px);

    }

    100% {

        -webkit-transform: translateY(0);

        -ms-transform: translateY(0);

        -o-transform: translateY(0);

        transform: translateY(0);

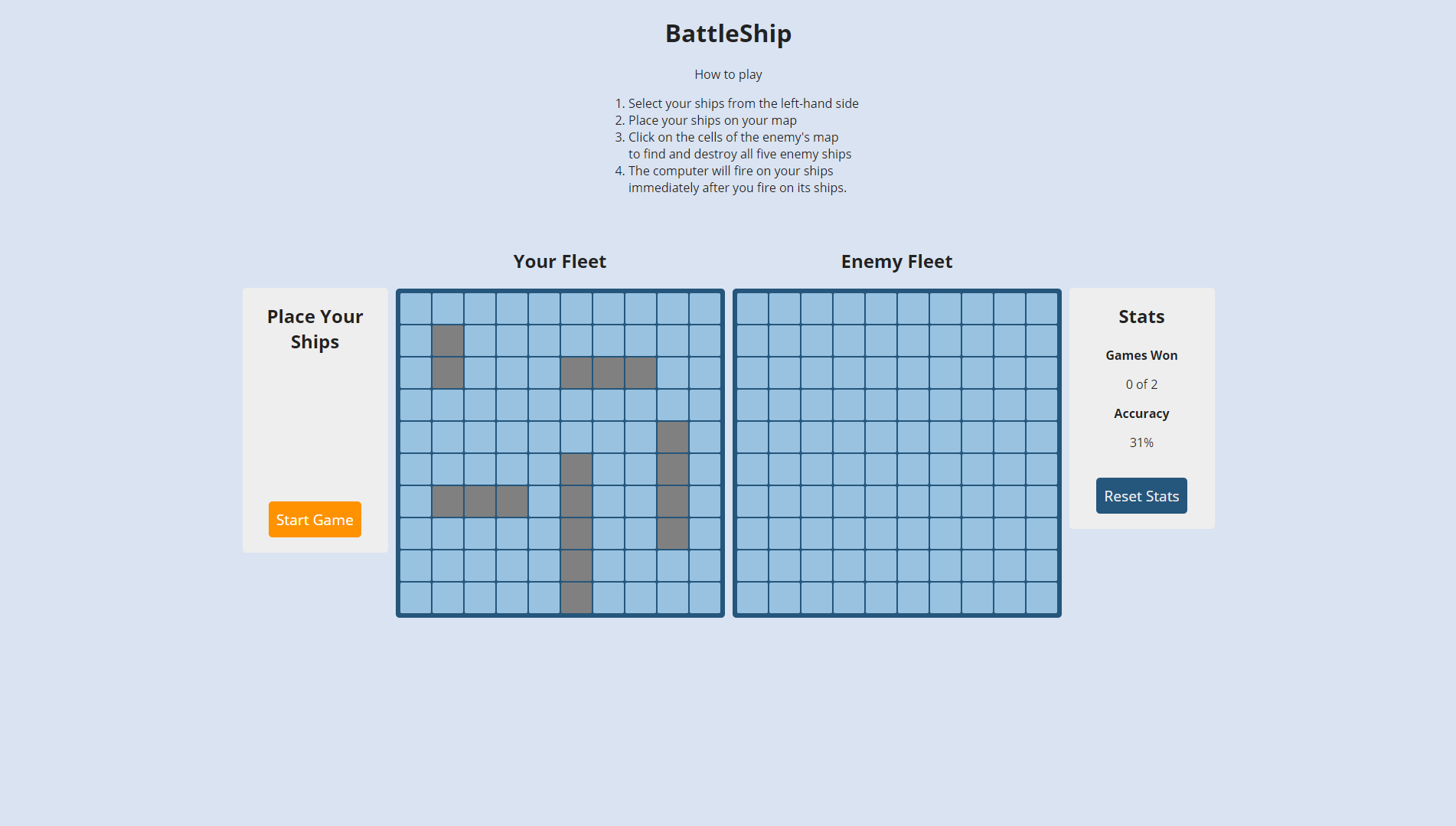
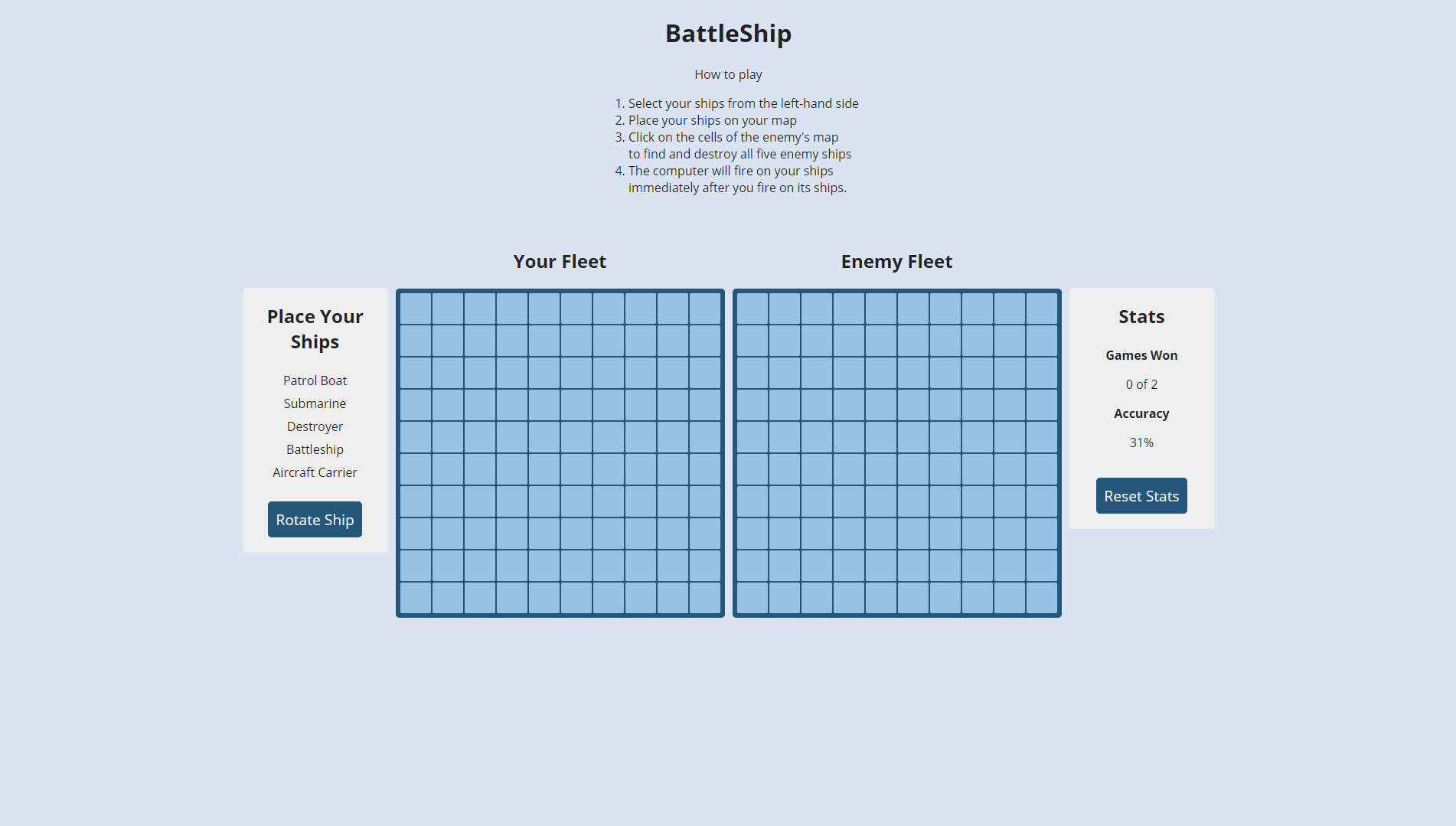
    }

}

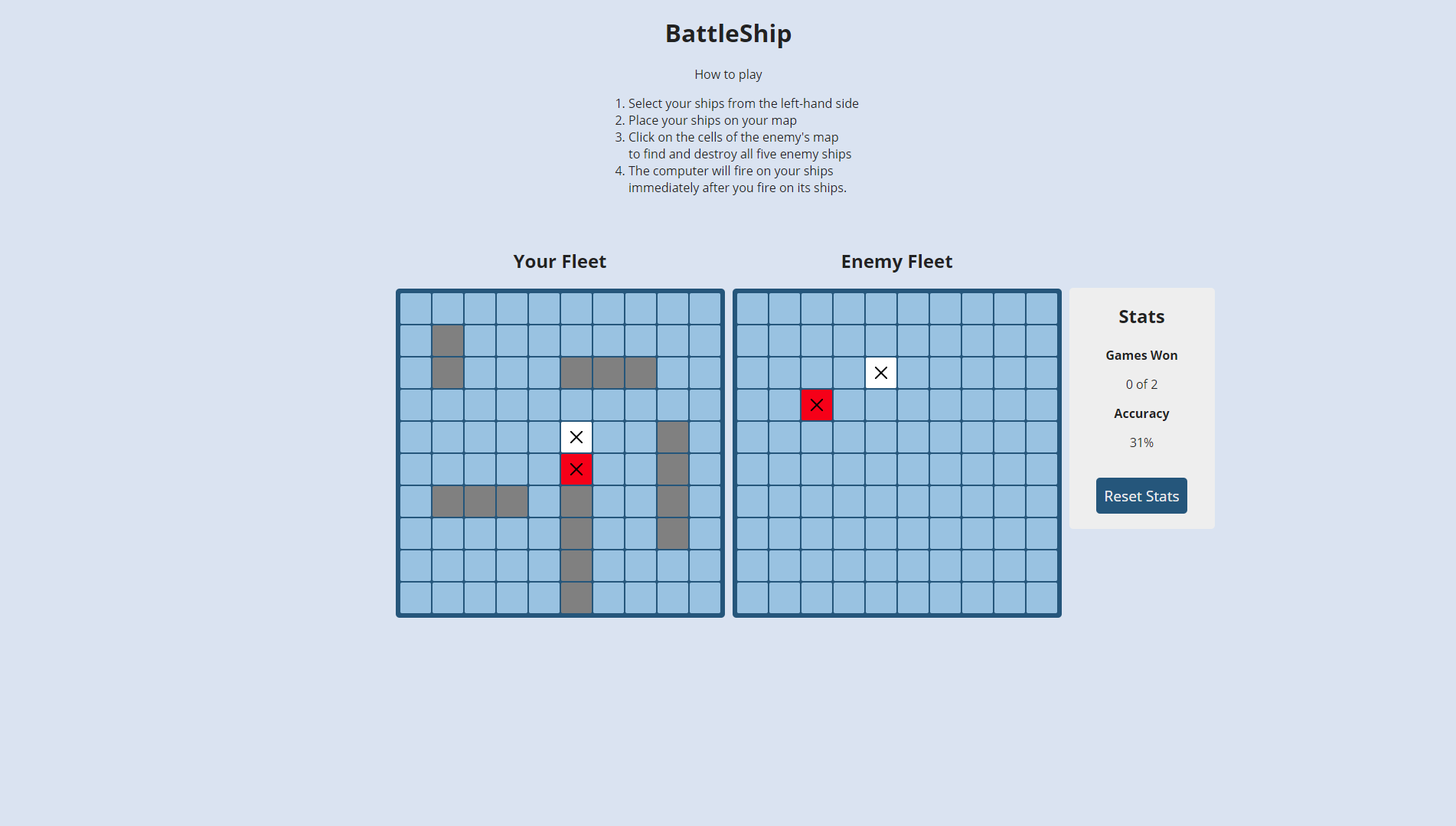
## **Controller**

# **BATTLESHIP GAMEPLAY**

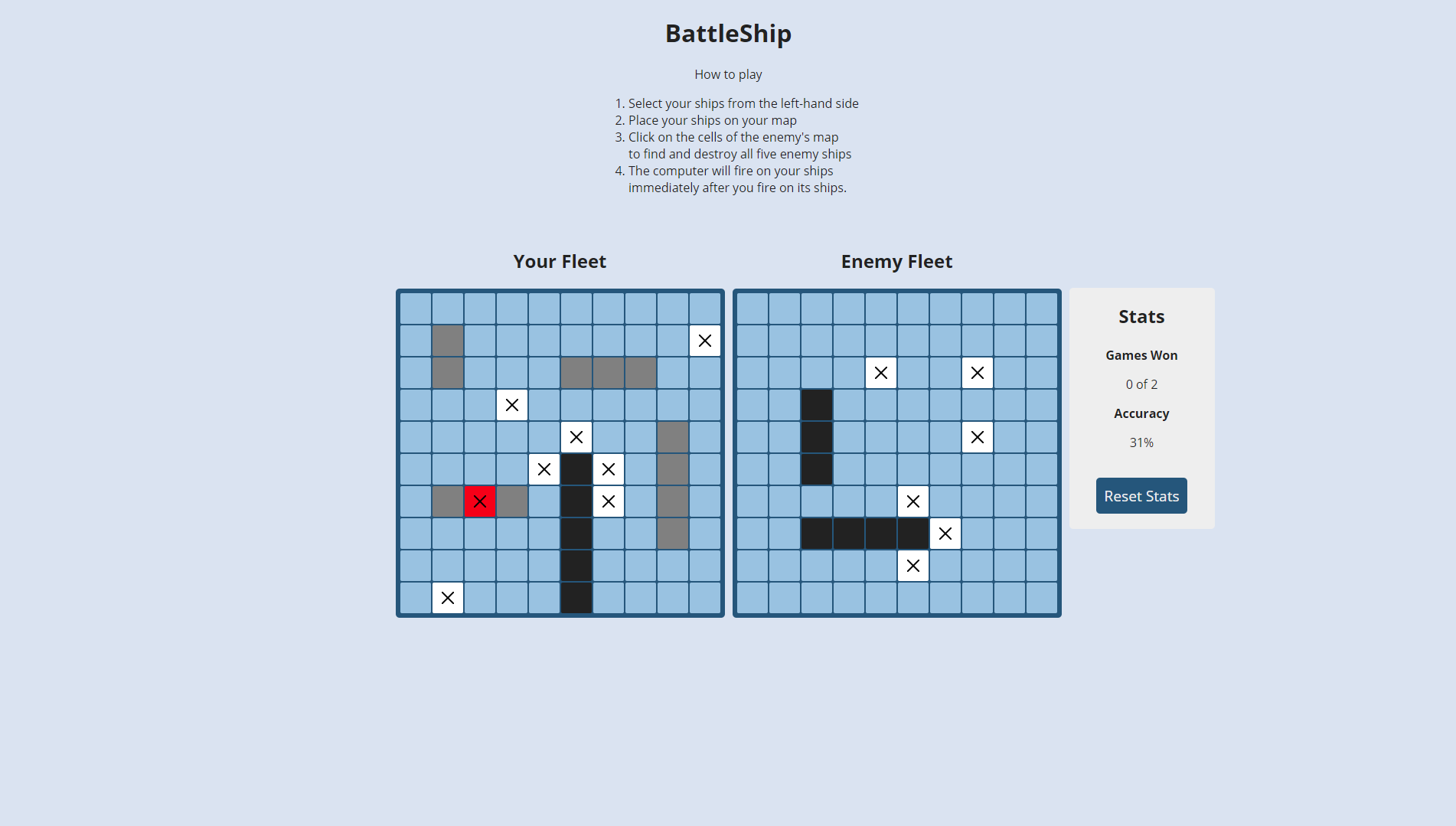
## **Starting a game**



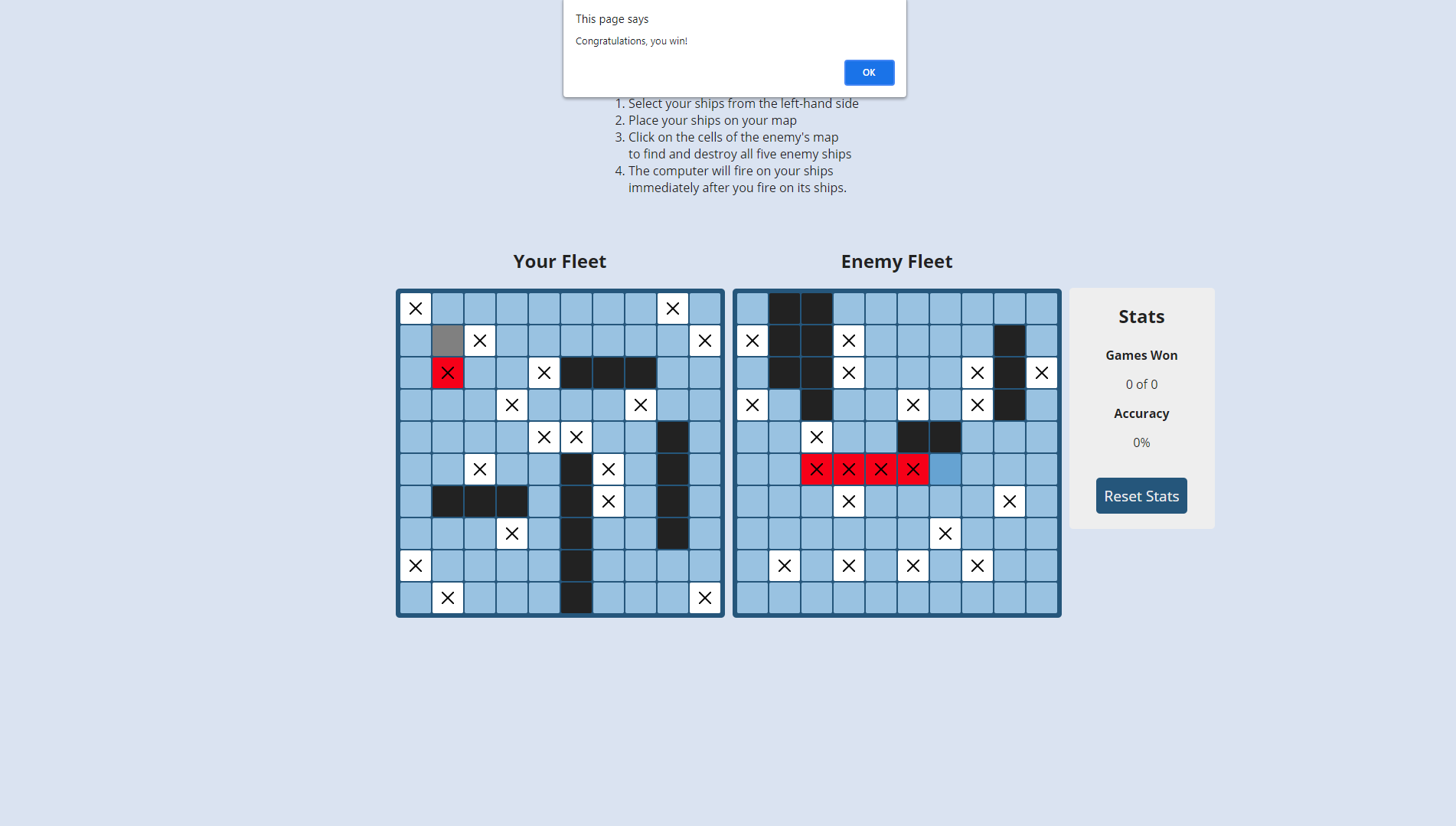
## **A ship was hit**



## **A ship was sunk**



## **All ship sunk**



# **CONCLUSION**

Through examining the field of interest, Battleship games, and formulating the requirements

needed for a basic implementation of such a game, a plausible software design was arrived at,

decided and implemented.

If development of the Battleship game should continue in the future, it would certainly be interesting to look at the possibilities for implementing more AI functionality. It would also make sense, to develop a manual ship placing functionality, so the human player would be able to place ships themself. If the Battleship game featured both AI functionality and manual placing of ships, it would be a complete solution that could be distributed on the app or run as an applet on one’s homepage.