Project report

Introduction:

For my project I followed one of the ideas proposed to us and made a website consisting of 3 minigames: Snake, Tic-Tac-Toe and Battleship. Instead of focusing on HTML and CSS, which in my opinion get boring fairly quickly, I decided to focus on making interactive experiences with JavaScript.

Snake:

This game was the easiest to implement out of the three. My first attempt at coding Snake was using divs and frame animations. Using this approach, I managed to get a snake which was only 1-block long that could move around, and spawn an apple which the snake could eat (Basically just a div that could move based on user input and that could 'eat' another div which was spawned randomly), but the problem that I encountered was making the snake grow after having eaten the food(I still included the file for it Snake.html and Snake.js but please do not take Snake.js into consideration as it is a real mess, I just kept it there to illustrate my point) . When I looked it up online I found that the more conventional and modern approach at making Java-script video games is to use HTML canvases. More specifically, I was inspired by (**http://www.competa.com/blog/how-to-build-a-snake-game-using-javascript-and-html5-canvas/**), notably I got the idea of handling a snake that is bigger than 1-block long by making the snake into an array of positions. Using this approach, I was able to make a functioning snake game. I did not encounter any real difficulties while coding this game. The only problem that was hard to identify was that if the snake was moving in some direction, let's say left, I had put an if statement (line 59-64 in Snake-Canvas.js) to prevent the change from changing directions and going directly right, as that would cause collision. But for a reason that I ignored at first, pressing the up-key then the right-key really fast caused the snake to ignore my if statement and go backwards. Upon further inspection, I understood that this was because the value of the variable direction was changing twice in between two animation frames. Hence while the snake was indeed moving left on my screen, the value of the variable direction was being set to up, and hence setting it to right while the snake was moving left was allowed, which caused the bug. I fixed this by implementing two direction variables, one 'actual_direction' and one 'direction_to_set', and I set the value of actual_direction to be direction_to_set only in the main game loop. Hence, all I need is to check if the direction to set is not opposite to the actual direction.

Tic-tac-toe:

This is the first minigame that I coded for this website. For the display, I decided to hardcode some divs and style them to look like a tic-tac-toe grid rather than make a grid using JavaScript because in my opinion since I was just dealing with a 3 by 3 grid, this was fairly simple short and very straightforward. Adding user-input to append an image of either a cross or a circle to each box onclick was also very simple. The difficult part of implementing this game was to code the 'not so easy mode'. For this part, I had first decided to use full game-tree exploration and a minmax algorithm where x would be the maximizing player and o would be the minimizing player. But for a reason that I still do not understand, my recursion was stopping before hitting the base case. Using this algorithm would have allowed for an unbeatable opponent, since it is literally impossible to lose at tic-tac-toe if you know what you're doing. However, I still wanted a fairly challenging computer opponent and not

just one that picks random moves like in 'easy mode'. So, I decided to use a heuristic evaluation, where I assign a score for each board in the game-tree in the following manner:

- $+\infty$ if x won
- $+10^i$ for each line of i x's and everything else empty
- -10^i for each line of i o's and everything else empty
- $-\infty$ if x won

So the script fetches the current board, computes it's children boards and their heuristic score, then picks the minimum one if it's playing o, or the maximum one if it's playing x (by default the computer is playing o, but if you first set twoplayermode on, play a turn with x, turn it off then play a turn with o, the computer will take over x and it will still try to win). It turns out that this works fairly well, because except for strategies where you 'trick' the computer to play the middle cell in order to capitalize on the corner cells and lock the victory.

Battleship:

This was by far the most complex of the three games to implement because the game is fundamentally more advanced than the other two. Firstly, it would have been virtually impossible to hardcode the grids like I did in tic-tac-toe, so I used JavaScript to divide a board into cells. Each grid is associated to a 2x2 matrix, where the values of the matrix are pointers to the cells of the grid. Then the boats are Js objects where the attributes are explained exhaustively in the comments of the codes. One difficulty in this program was figuring out a way to wait for user input: my code had the following basic structure:

```
user_did_something = false

while (!user_did_something){

        function dosomething(event){

                user_did_something = true

                do stuff

        }

}
```

But this made the code impossible to run and crashed Firefox many, many times. So I decided to get rid of the while loop and do everything inside of the gameloop, while separating the gameloop itself into different 'gamephases'. This coupled with the fact that JavaScript syntax is still new and confusing to me, resulted in a fairly messy to read code, with a lot more global variables than I probably needed, but this was the only way I could see to make it work, which it does perfectly. The other main difficulty in programming Battleship was again making a challenging computer opponent. I had a choice, I could have made the program 'cheat' since it knows where the boats are, I could have just made it choose a cell with a boat in it with some probability. But this seemed very uninteresting to me. So instead I designed the following algorithm, which almost corresponds to what a human would do:

1. Choose a non-hit cell at random
2. If it's a miss return to 1, else:
3. Choose a direction you have not explored yet randomly
4. Keep exploring that direction
5. If you hit water or a border, go back to 3

6. If you sunk a boat go back to 1

Again, this makes for a fairly engaging and entertaining opponent. However, the main problem with my algorithm is at step 1, where it chooses a cell randomly. If a human sees that a non-hit cell is surrounded by misses, i.e it's empty but there is no way a boat can be there, it will not pick it, whereas in my algorithm this cell has the same value as any other cell. Another way I could have improved my algorithm would have been by keeping track of which boats have been sunk, in order to choose the random cell more intelligently.

General notes on the three games:

There are a couple of things that I needed done in JavaScript, namely shuffling an array and getting a random int uniformly in an interval, but that I did not manage to implement a way to do them in time, so I used some code on Stack Exchange and linked the sources.

Lastly, I have decided to implement some 'cheat' options in Battleship and Snake to allow for easy testing of the code. In snake 'god mode' means that the game will ignore collisions, I have also added a speed slider, and a reveal enemy boats button in Battleship.