Group: Badger
Members: Zachary Chan, Curtis Huang, Kyle Mollard, Mark He, Angela Yung

UI Prototyping

Upon analyzing the strengths and weaknesses of the UI prototypes made by each group member, we aimed our attention at those UI features that cultivate as many of the ten heuristics of the Nielsen Norman Group as possible. Please see page 7+ for the UI prototypes.

First, we decided to incorporate the following usability heuristics from our collective UI prototypes into the main gameplay interface, menu interface, and settings interface:

1) User control and freedom
2) Consistency
3) Recognition over recall
4) Minimalist design
5) Help and documentation
6) Flexibility

We incorporated heuristic 1) by using a “reset” button design inspired by Prototype K2. The “reset” button is visible and accessible in our final prototype. The motivation behind implementing such a feature is that users should be able to correct any misplaced ships with ease. The “reset” button is a clear indication to the user that they are in control of the placement of their ships such that in the case of a placement error, the grid is wiped clean for the user to start over. Similarly, during gameplay, the user has an option to forfeit by clicking the “forfeit” button located at the bottom of the screen. The “forfeit” button acts as a clearly labeled exit for the user. This feature was inspired by the “reset” button from Prototype K2. In general, these features minimize the user’s frustration during the game set-up process and during gameplay.

As for heuristic 2), it was a common theme that we observed in all of our prototypes. We agreed that at the minimum, button placement, button size, icon size, grid size, and colours should be consistent. As such, our final prototype implements consistency in all of these areas. For example, during gameplay, all buttons are accessible at the bottom of the screen as opposed to being scattered around the screen. We tried to use colours consistently to communicate to the user, for example, that a specific tile is occupied by a ship, or that a button performs an irreversible action. Lastly, we followed the convention of labeling the x-axis with letters and y-axis with numbers based on our interpretation of the original Battleship game. This re-creates a similar impression of the original game board. Our goal is that with more consistency, the easier it is for recognition to occur.

We implemented heuristic 3) in two ways. First, our prototype follows a colour scheme: a blue tile represents an unexplored tile, a red tile (with the letter H) represents a tile that is occupied by a ship and has been hit, a white tile (with the letter M) represents a missed shot. Any other coloured tiles with letters represent the ships. The tile colour scheme is inspired by prototype Z2, and the ship tiles (with letter labels) are inspired by Prototype A2. The colour scheme for buttons is as follows: a red button indicates that an action cannot be reversed or progress will be lost, a blue button indicates an action can be reversed, and a green button indicates that this action will help the user start the game. We applied this colour scheme consistently for the gameplay UI, menu UI, and settings UI. The motivation behind these button colours is to mimic the traffic light colours - red means it is dangerous to proceed and green means it is safe to proceed. This promotes intuitive recognition. By using a consistent colour scheme, we can minimize the user's memory load by making important actions and options visible.

Next, we considered what kind of aesthetic we wanted for each of the interfaces. We took ideas from Prototype C3, K4, and Z2, and ultimately went for a minimalist design for all interfaces. At the end of a day, displaying two grids on the screen can be overwhelming if not done correctly. Moreover, we did not want to overwhelm the user even further by adding unnecessary imagery or buttons. Therefore, we went with a flat design where the visuals are focused on the essentials. Our gameplay interface includes the name of the game, two side-by-side grids, and three basic buttons - forfeit, menu, and help. The menu interface has the name of the game, a title, and four buttons - play, settings, achievements, and credits. The minimalist design ensures that the user gets the information they actually need.

We liked the idea of including some kind of "help" button based on Prototype C2 and Z2. The motivation behind this feature is to support help and documentation for improving the usability of our UI. The help button that we implemented in the final prototype leads the user to a page that explains the objective, set-up, and how-to's of the game. The documentation listed in the "help" page explains concrete steps for the user to progress in the game.

Finally, we wanted to allow user flexibility when it comes to in-game music. The user has an option to adjust the music. So, if the user wants to immerse themselves in the game, they can turn on the music, and if the user finds the music to be distracting then they can turn it off (by dragging the in-game music volume all the way to the left). These features are accessed in the settings menu. This design was inspired by Prototype K2 and Z1.

Design Patterns

Composite

With React being a component-based library, naturally we were able to use the composite design pattern throughout our code. In general, we created our UIs by composing simple and isolated pieces of code. While interfaces are a useful way to implement the composite design pattern, React itself has objects (components) that can be treated as composite objects (containers). As such, we can have components composed of other components, thus forming a tree-like structure. In our implementation of the composite design pattern, we created a hierarchy of components where each component encapsulates a specific functionality or piece of the UI.

For example, observe lines 17 to 80 in game.jsx (found in /src/pages). The code exhibits composition by combining multiple smaller components together to make the component, 'GameContent'. The return statement lists all of the smaller components that make up 'GameContent' such as 'ShipGrid', 'BattleGrid', 'GameLogDisplay', and 'Button'. Note that each smaller component is well-named to describe its purpose. For example, 'ShipGrid' is the component that manages the grid with ships, whereas 'BattleGrid' is the component that manages the grid for tracking hits and misses.

Similarly, consider lines 144-174 in the same file. The 'Game' component conditionally renders either the 'GameContent' component or the 'PlaceShips' component based on 'gameState'. The 'Game' component acts as a composite object that can contain different components based on certain conditions. Therefore, this pattern allows more flexible rendering of components in the UI based on the state of the game.

In addition, the 'BattleGrid' component in battle_grid.jsx (found in src/scenes/game) and 'ShipGrid' component in ship_grid.jsx (found in the same directory) are two components that render the 'SelectionGrid' component, thus utilizing the composition.

There are many more examples of composition in the entire project which can be found in most nested component structures. The reason we chose this design pattern is because it was natural and convenient to use in React with JavaScript. Noticeable improvements include having an easier time refactoring our codebase and easily reusing components.

Factory Method

We implemented the factory method design pattern by delegating instantiation logic to specialized functions. In our case, the ‘createGrid’ function in game.jsx allows for flexible object creation. It accepts a grid size parameter as input and can create grids of different sizes. So, ‘createGrid’ doubles as a creator and a factory for grids (the product). Grids are not implemented as interfaces, as they are just arrays.While we did not have to implement this design pattern with classes or interfaces, the application of the pattern is nonetheless equivalent. We have a product, grids, where its concrete implementation (i.e., the specific grid types based on size) is decided by the argument passed into the creator. We achieve the same effect from the factory method design pattern without using subclassing.

We chose this design pattern because it would be easier to create new levels (e.g., based on different grid sizes). Using the factory method to dynamically create grid objects is a lot more efficient than creating unique classes or components for each grid type. There would be a lot of repeated code if the factory method was not used, thus increasing the chances for bugs. In terms of improvements, we were able to encapsulate the logic for grid object creation and abstract away the details from the client code for better maintainability.

Adapter

We decided to implement the adapter design pattern while respecting the architecture of React, which means there is no emphasis on interfaces and more emphasis on composition (i.e., components). Therefore, while the traditional implementation of the adapter design pattern adapts interfaces, we decided to adapt a data structure and representation of an object. We implemented this pattern in the file selection_grid.jsx (found in /src/components/game).

In the code, the ‘SelectionGrid’ component receives a ‘legend’ prop, which can be understood as a form of adaptable data. The ‘legend’ prop contains mappings between grid items and their corresponding display properties (e.g., color, image, and icon). These properties can be considered as the "adaptee" interface in the analogy. The ‘SelectionGrid’ component adapts the ‘legend’ data to render the grid display. It maps the grid items in the grid prop to their corresponding display properties from the ‘legend’ data. It uses this adapted representation to render the grid tiles using the ‘GridTile’ component. This can be seen as the "adapter" in the analogy, as it converts the incompatible representation of the grid into the format expected by the ‘GridTile’ component.

While the code does not directly adapt one interface to another, it indirectly adapts the data structure and representation of the grid using the legend data. It maps the grid items to their corresponding display properties, allowing the ‘GridTile’ component to work with the adapted data representation.

We decided to use this interpretation of the adapter pattern because it would allow us to use the ‘GridTile’ component with different data sources or representations. For example, the legend data may change if we introduce different levels in the future. It helps with extending our code base without introducing more complexity.

We identified two main benefits of using this design pattern in our code. First, there was more code reuse specifically of the ‘GridTile’ component. Because we were able to adapt the legend data to multiple instances of the ‘GridTile’ component, there was no need to create different types of ‘GridTile’ components for different ‘legend’ data inputs. It encapsulates the adaptation logic within the ‘SelectionGrid’ component, allowing for the reuse of the ‘GridTile’ component with different data sources or representations. Second, there was more modularity. By separating the adaptation logic within the ‘SelectionGrid’ component, the code promotes modularity. The logic for adapting the grid data is encapsulated within the component, making it easier to understand, modify, and extend the adaptation process if necessary.

Observer

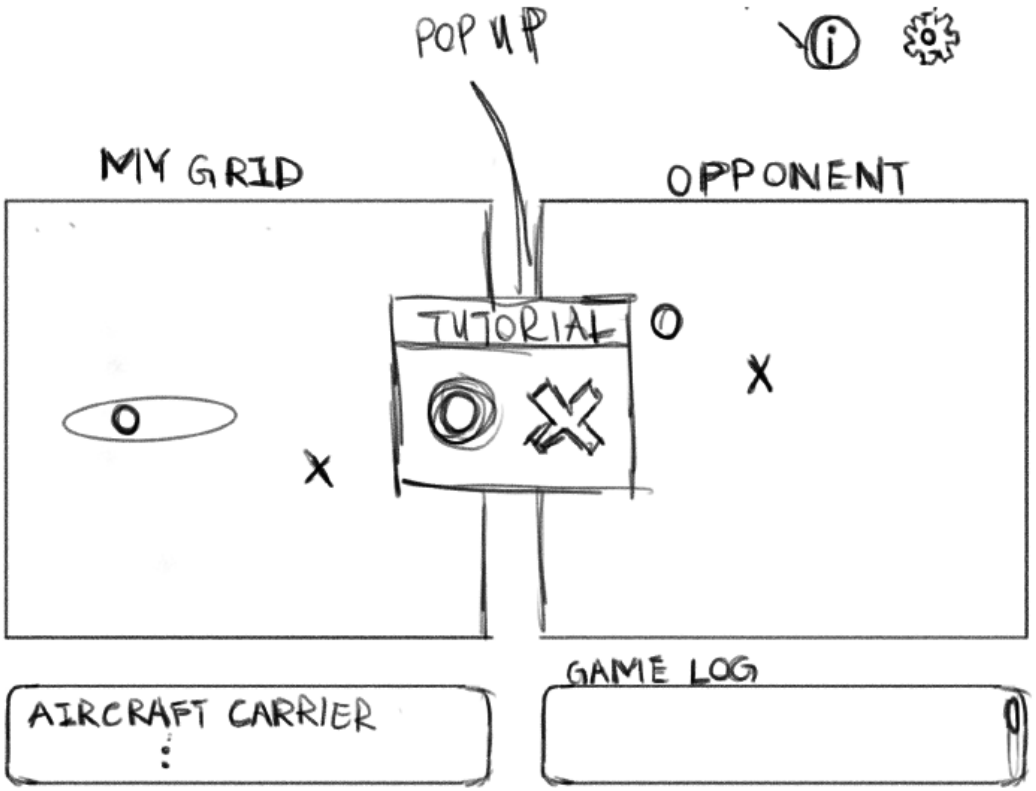
Throughout various parts of the project, various components use the observer pattern indirectly through React’s ‘useEffect’ hook which observes changes in the local storage and updates the corresponding states accordingly. For example, the App component updates ‘settings’, ‘stats’, and ‘obtainedAchievements’ with useEffect in App.js (found in /src). Similarly, in game.jsx, the ‘Game’ component uses React’s ‘useEffect’ hook to watch for changes to ‘gameState’ in order to determine when to place enemy ships randomly. Consider also the ‘PlaceShips’ component in place_ships.jsx (found in /src/scenes/game). It utilizes the useState hook to rerender the component and any child components depending on the state of variables such as ‘selectedShip’, ‘shipOrientation’, and ‘selectedSquare’.

We chose this design pattern because there are lots of events that occur as the user plays the game, and the observer design pattern allows for better maintainability of code. Changes to the subject or the addition of new observers can be done without changing

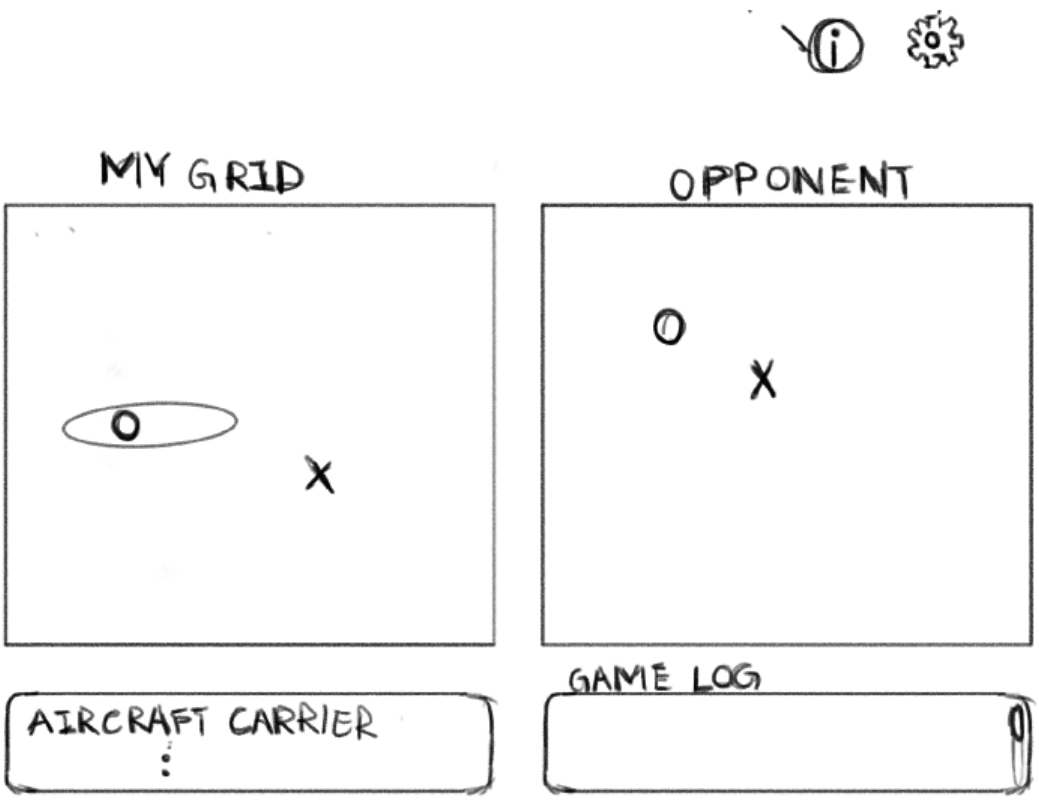
existing observers or the subject. It improved the modularity of our code by separating the subject and observer so that subjects can be reused with different observers, and vice versa.

Prototype C1 - Curtis

Prototype C2 - Curtis

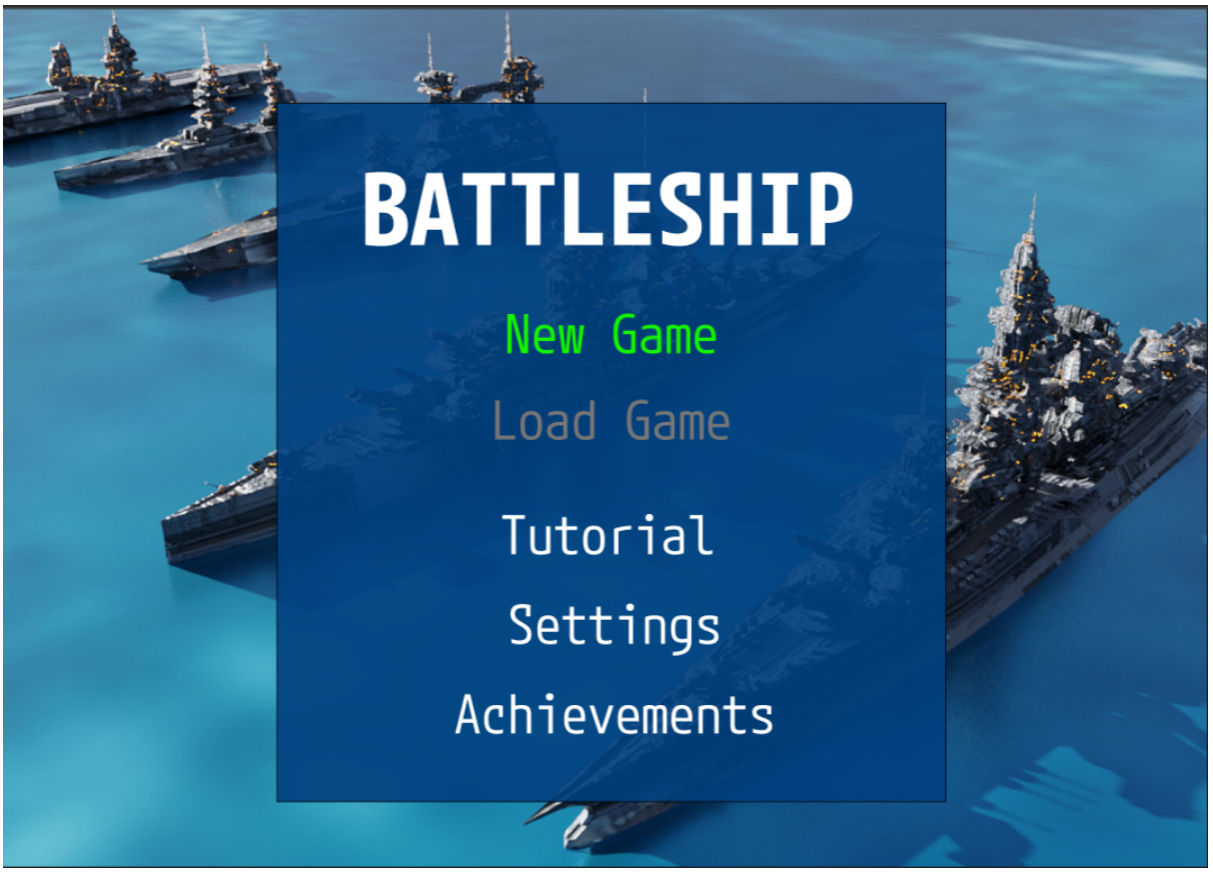


Prototype C3 - Curtis

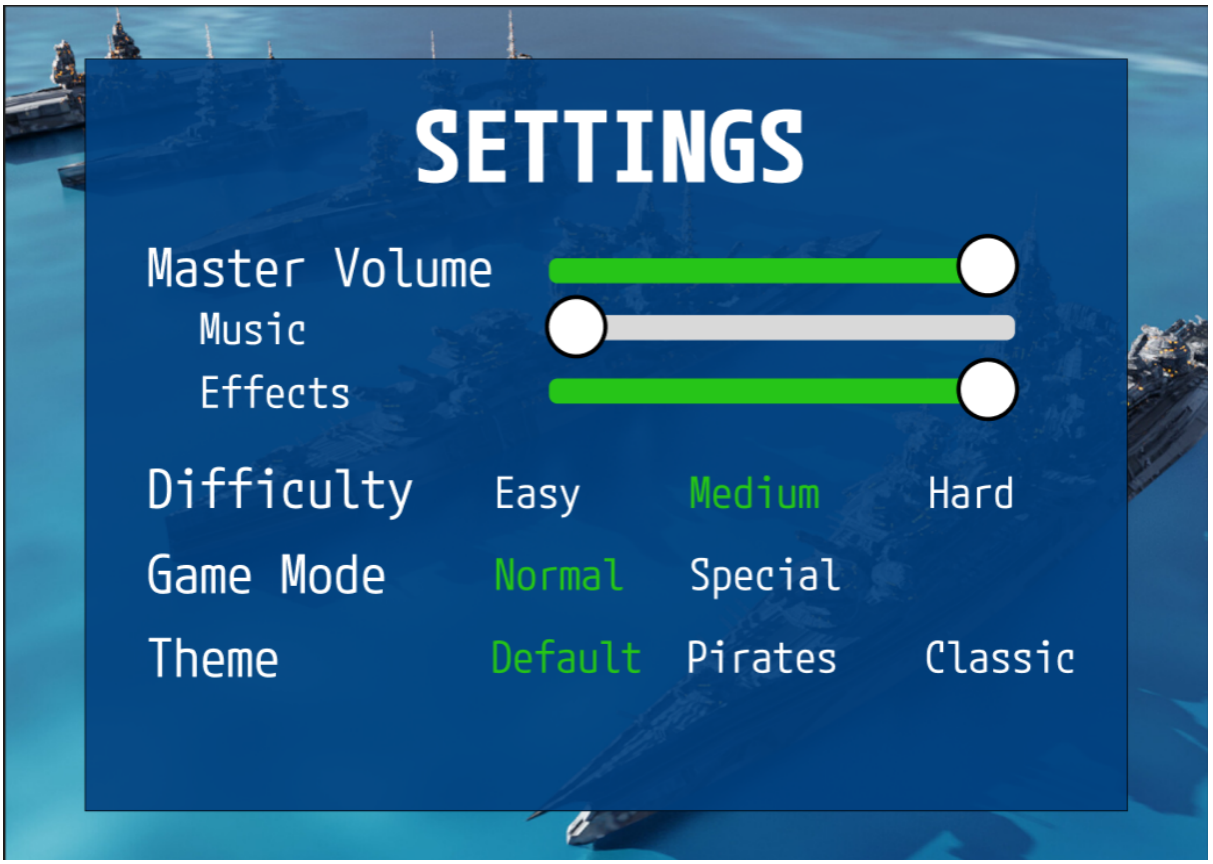


Prototype C4 - Curtis

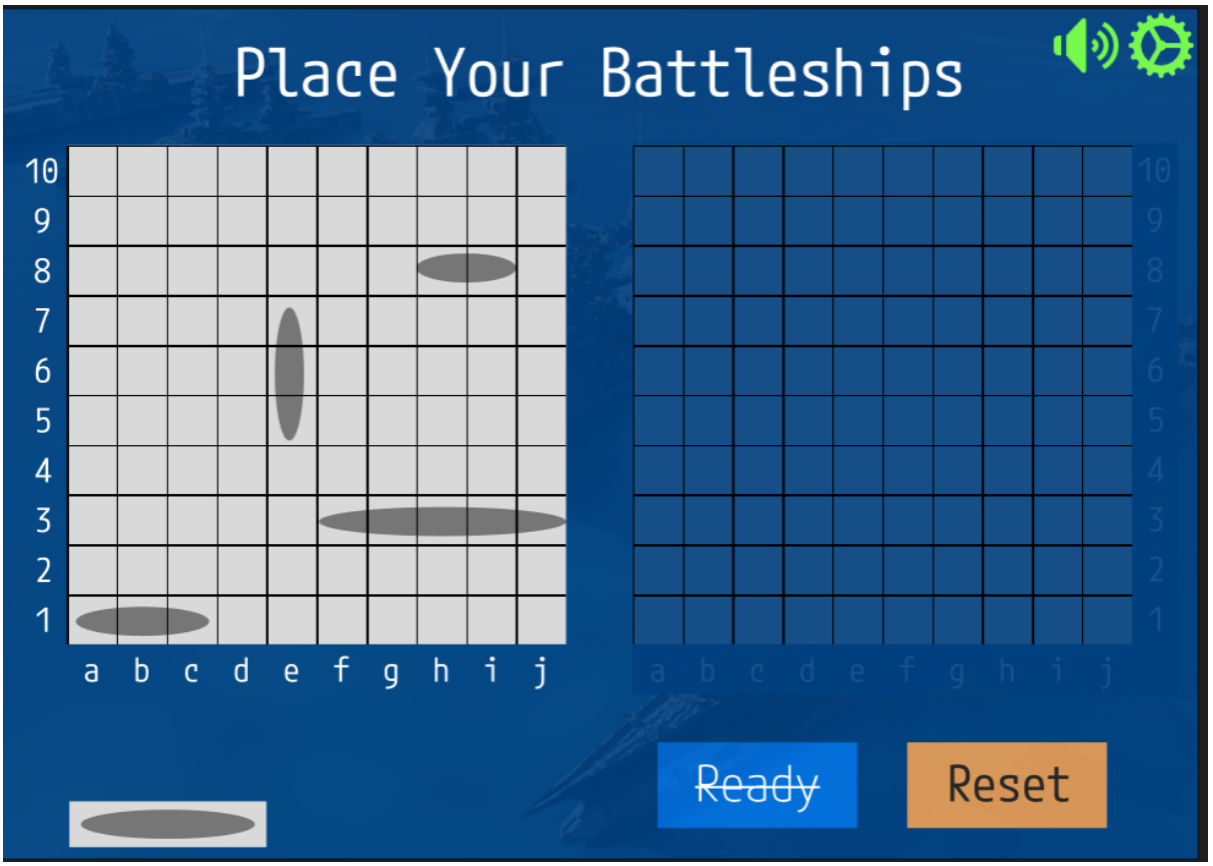
Prototype K1 - Kyle



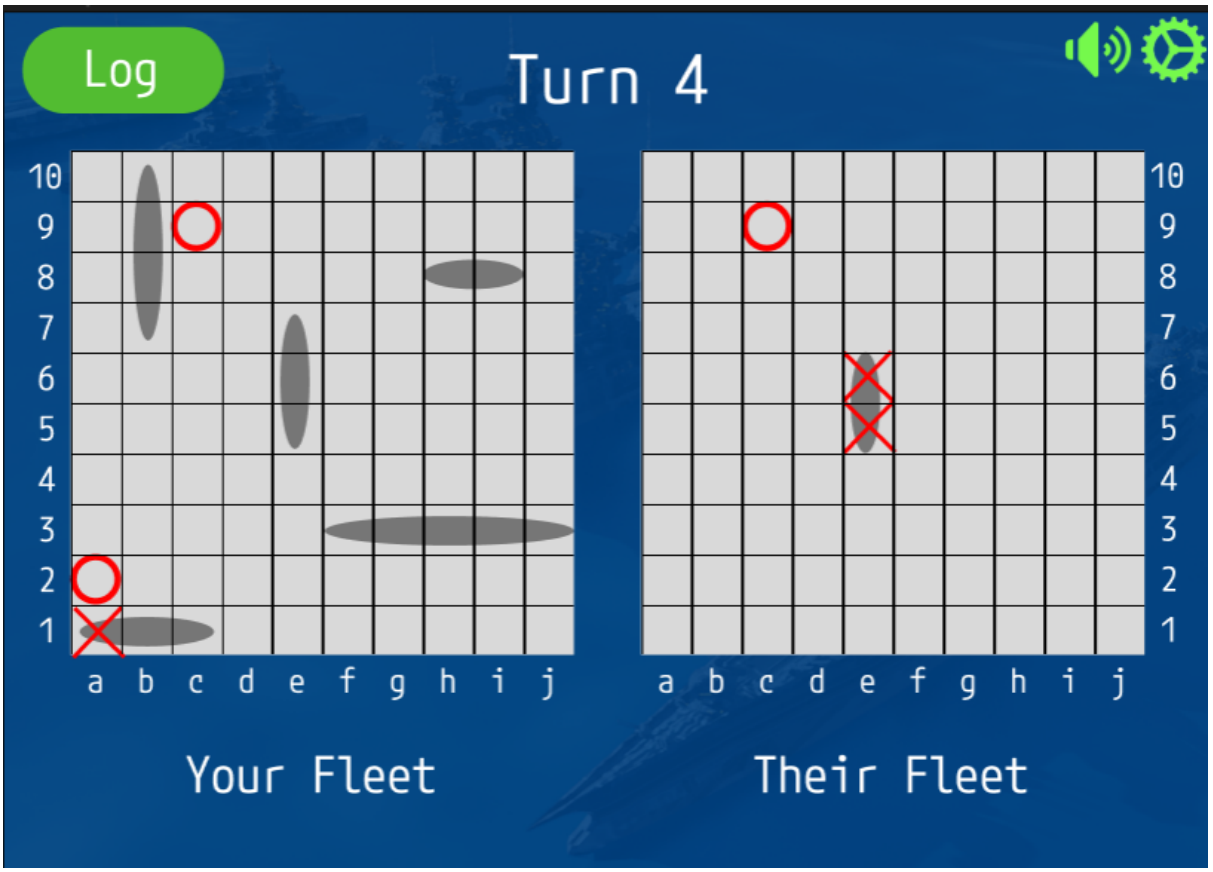
Prototype K2 - Kyle



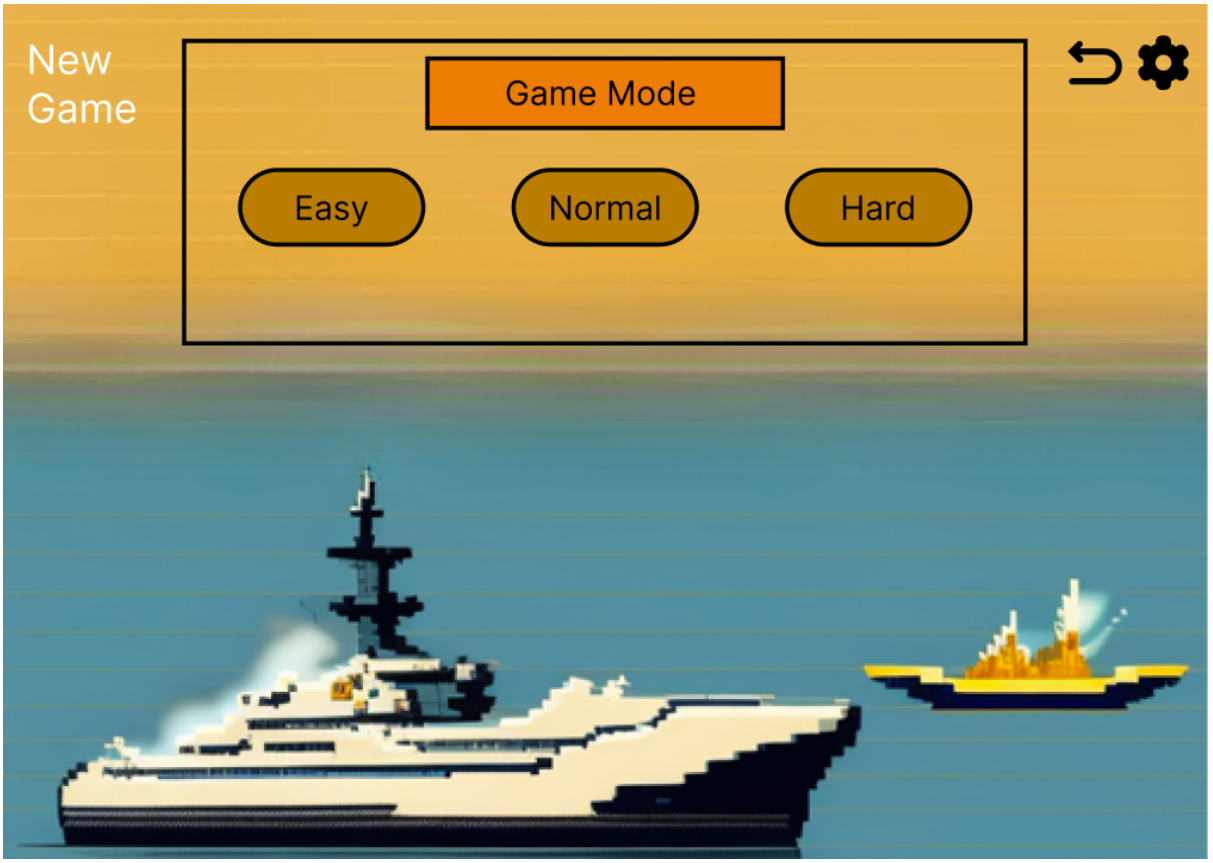
Prototype K3 - Kyle



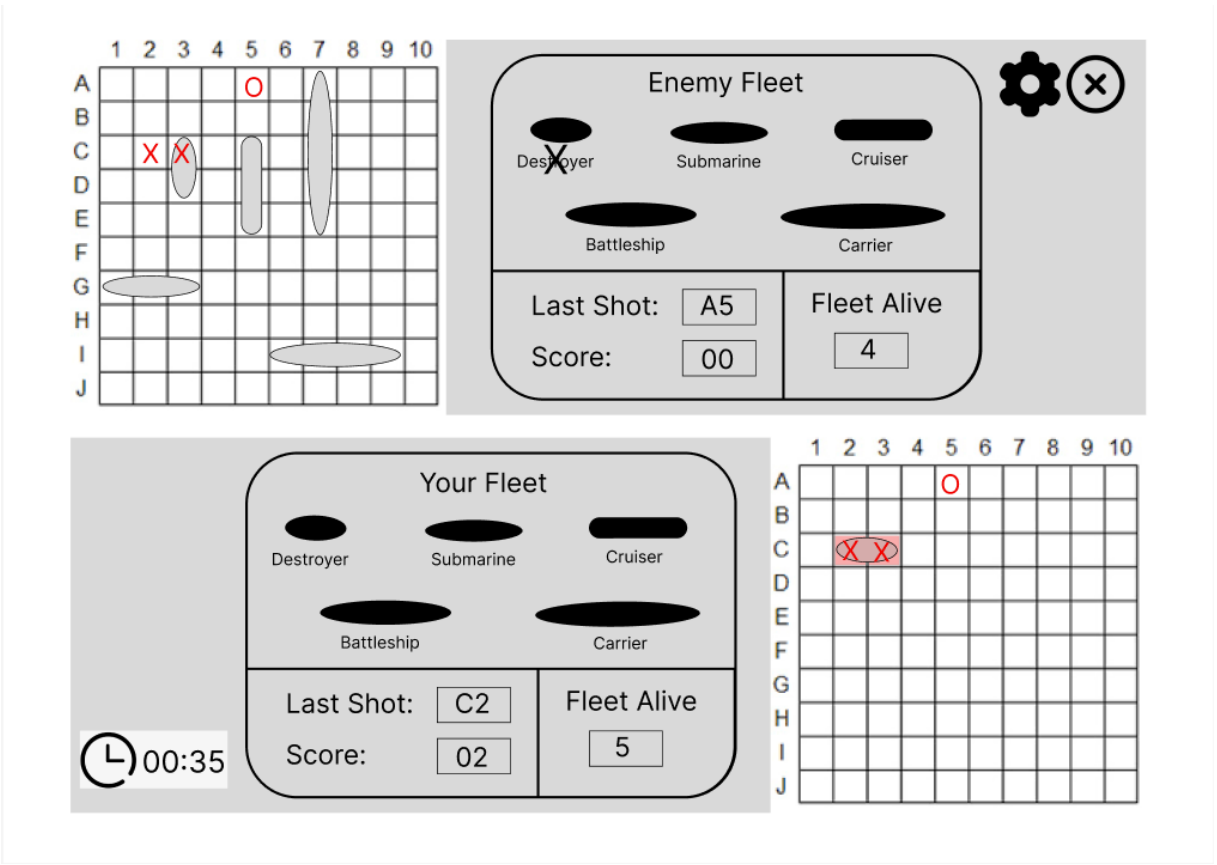
Prototype K4 - Kyle



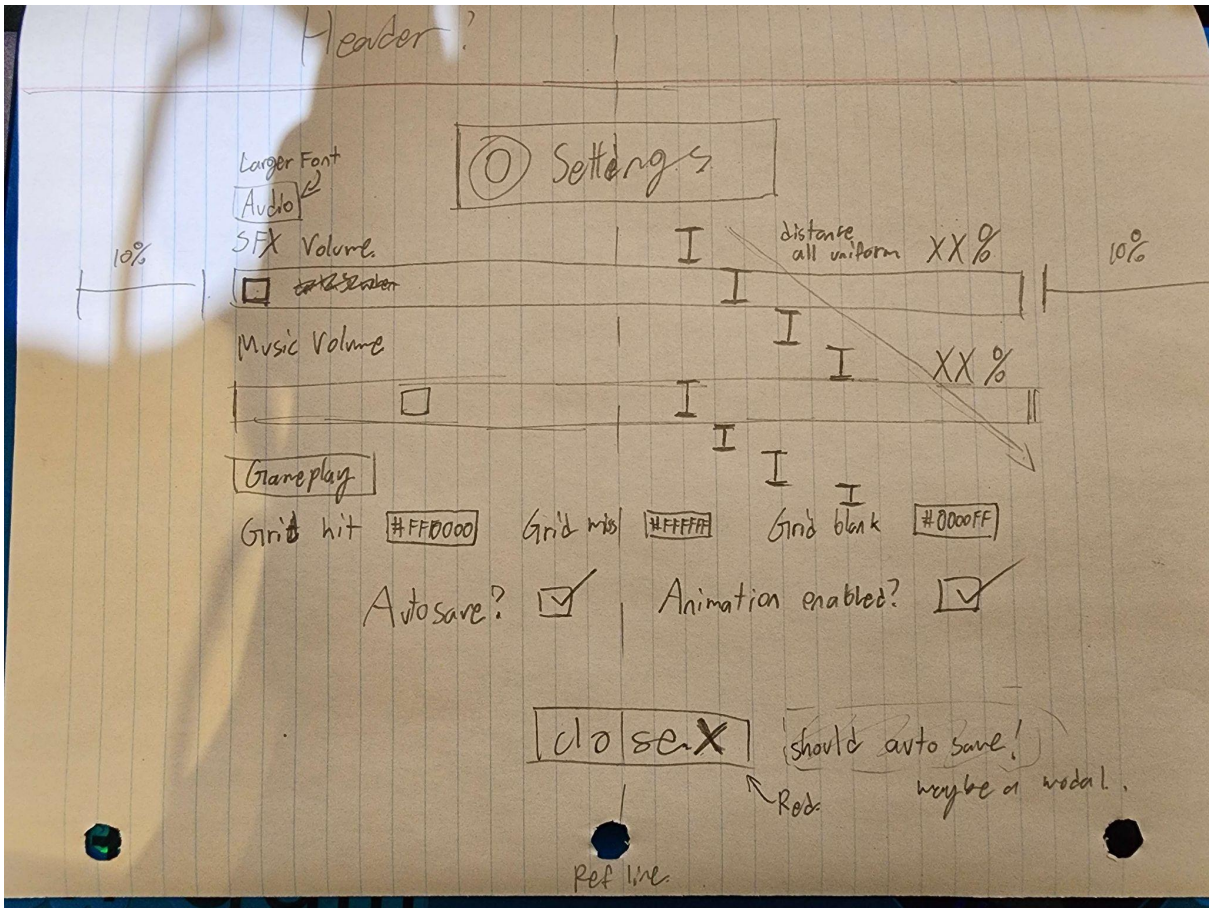
Prototype M1 - Mark



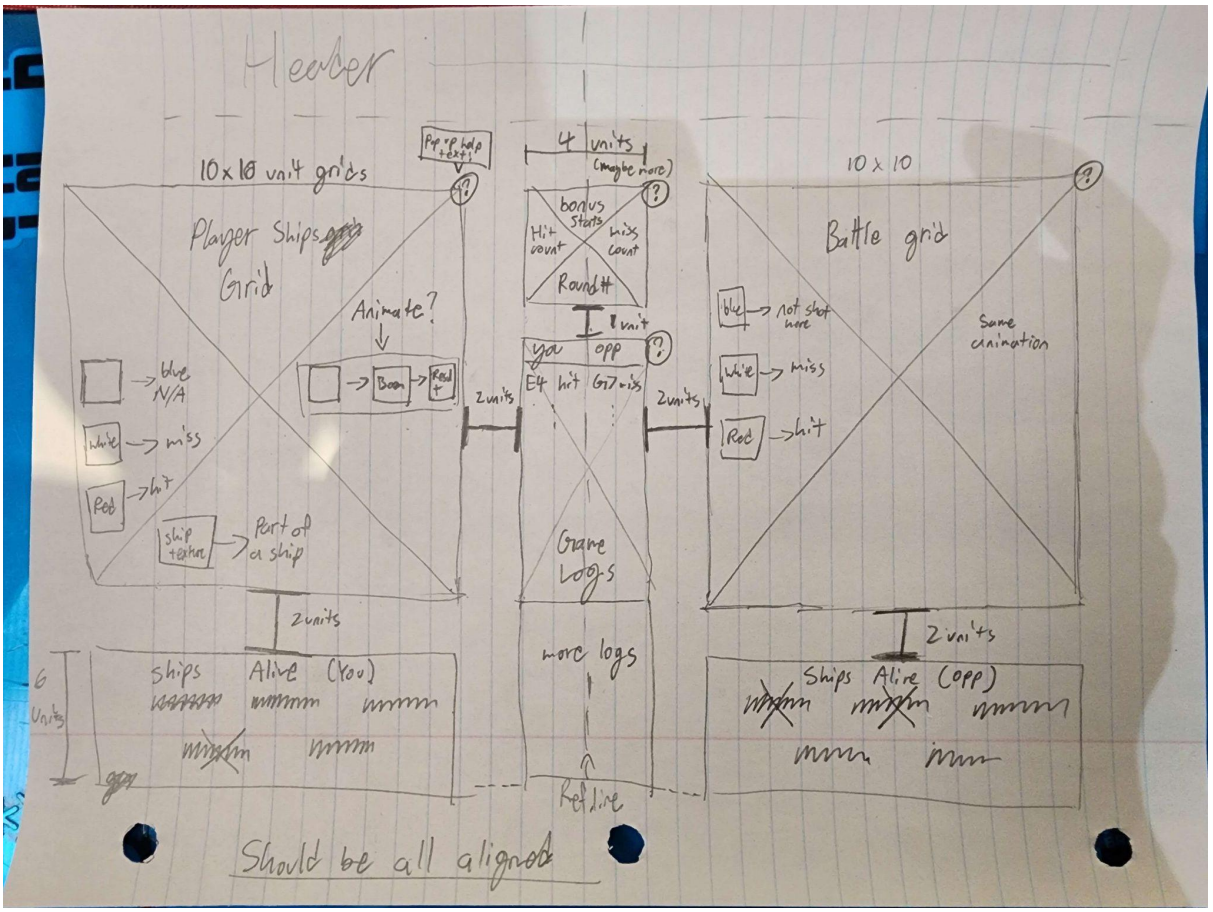
Prototype M2 - Mark



Prototype Z1 - Zach



Prototype Z2 - Zach



Prototype A1 - Angela

Prototype A2 - Angela