**Project Overview**

The primary objective behind our project design was to create a scalable and extensible game framework. This structure allows:

* Adding more players with minimal code adjustments.
* Easily expanding the grid size.
* Introducing new ship types, attacks, or game operations.
* Streamlining the next development phases and reducing repetitive code.

**Key Features**

**Dynamic Grid System**

* **Customizable Grid Size**: The game supports grids of any size that fits on the screen. Changing the GRIDSIZE macro in the defs.h file automatically updates the grid size while ensuring numeral alignment.
* **Flexible Numeral System**:
  + Rows and columns use a dynamic numeral system to accommodate various formats, including alphabetical.
  + For array manipulation, the system can convert between alphabet and integer indices (e.g., AAA = 0+0+0, AAB = 0+0+1, ..., AAZ = 0+0+25, … FBC = (5 \* 26^2)+(1\* 26^1)+ 2 \* 26^0 ...). CMPS 221 was handy.
  + Additionally, the numeral system can adapt to any characters within specified ASCII bounds, and we’re considering updating it to support any array-based character system for even greater flexibility.

**Driver Design**

* **Organized Code Structure**: We maintained a clear, front-end oriented design in the driver to manage the user interface (UI).
  + **Functionality**: The driver handles input, calls backend functions to process it, and displays results or moves to the next game turn as appropriate.
  + **UI Control**: The driver manages around 90% of the UI, using custom functions for:
    - Centered printing and formatting.
    - Color adjustments.
    - String concatenation (using a variadic function for flexible argument counts).
    - Safe input scanning.
* **UI Functions**: All UI-related functions are implemented in the UITools.c file.

**Input System**

* **Custom Input Management**: To improve upon the unsafe and cumbersome built-in C functions (scanf\_s(), getchar()), we developed a safer and more versatile input system inspired by Java’s Scanner class:
  + **alloc\_Input()**: Reads input from the user (using getchar()) and stores it as a single, dynamically allocated string (char \*).
  + **next()**: Moves a pointer across the string word by word until \0. It uses a separate pointer to avoid issues with memory management.
  + **nextInt()**: Utilizes next() to parse the current word into an integer (using ParseInt() from ShortcutFuncs.c).
* **Dynamic Memory**: Dynamic allocation allows the input to be efficiently passed across functions.

**Future Improvements**

While the current input and display systems are functional, we intend to fix, improve, and explore more efficient techniques to further enhance performance and maintainability as the project develops.