RELATÓRIO TÉCNICO: DESENVOLVIMENTO DO JOGO BATALHA NAVAL

Disciplina: PIF – Projetos de Sistemas de Informação **Professor:** João Victor Tinoco

Equipe de Desenvolvimento:

- Fabio Henrique Dantas Layme Lopes de Albuquerque
- Felipe Borba De Carvalho
- Francisco Rodrigues de Oliveira Junior

---

1. Introdução

O objetivo deste projeto foi desenvolver uma implementação robusta do jogo Batalha Naval utilizando a linguagem C, operando exclusivamente via interface de linha de comando (CLI). O foco principal do desenvolvimento não foi apenas a funcionalidade do jogo, mas a aplicação prática de conceitos fundamentais de baixo nível, especificamente: **alocação dinâmica de memória**, **manipulação de ponteiros**, **uso de structs** e **modularização de código**.

O sistema permite partidas entre dois jogadores (ou humano vs computador na lógica de posicionamento), com tabuleiro configurável e validação rigorosa de dados.

2. Arquitetura do Projeto

O sistema foi arquitetado para separar responsabilidades, facilitando a manutenção e o isolamento de erros. A estrutura de módulos adotada foi:

- **`main.c`**: Ponto de entrada. Gerencia o ciclo de vida da aplicação (Menu Principal -> Configuração -> Jogo -> Encerramento).
- **`board.c`** / **`.h`**: Módulo de dados. Responsável estritamente pela alocação, acesso e liberação da memória do tabuleiro.
- **`fleet.c`** / **`.h`**: Módulo de dados. Gerencia a coleção dinâmica de navios (`Fleet`) e suas propriedades (`Ship`).
- **`io.c`** / **`.h`**: Camada de interface. Abstrai toda a entrada e saída (`stdin`/`stdout`), garantindo sanitização de dados.
- **`game.c`** / **`.h`**: Core logic. Orquestra as regras do jogo, turnos, lógica de disparo e algoritmos de posicionamento (manual e automático).

Diagrama de Fluxo de Dependências

Plaintext

[Main]

|

+---> [Game] (Controlador)

|

+---> [Board] (Memória do Mapa)

+---> [Fleet] (Lista de Navios)

+---> [IO] (Interação com Usuário)

3. Estruturas de Dados

A escolha das estruturas foi determinante para o desempenho e organização da memória:

1. **`Board` (Tabuleiro Linearizado):** Optamos por não utilizar uma matriz de ponteiros (ponteiro para ponteiro). O tabuleiro é representado por um **vetor unidimensional contíguo** (`malloc(rows * cols)`). Cada elemento é uma struct `Cell` contendo o estado (Água/Navio/Tiro) e o ID do navio.
2. **`Fleet` (Frota Dinâmica):** A frota não possui tamanho fixo no código. Ela é um ponteiro para `Ship` que cresce conforme necessário usando `realloc`. Isso permite flexibilidade caso novas regras exijam mais navios no futuro.
3. **`Player` (Agregador):** Utilizamos uma struct `Player` para agrupar o `Board`, a `Fleet` e estatísticas, facilitando a passagem de dados por referência entre as funções do jogo.

4. Fluxo de Execução

O ciclo de vida do software segue a seguinte lógica:

1. **Inicialização:** O `main` cria uma struct de configuração padrão.
2. **Configuração (Opcional):** O usuário pode alterar o tamanho do tabuleiro e o modo de jogo.
3. **Setup (`setup_game`):**
    - Aloca memória para os tabuleiros (`create_board`).
    - Inicializa e aloca a frota (`init_fleet` / `add_ship`).
    - Executa o algoritmo de posicionamento (Manual ou Automático).
4. **Game Loop (`play_game`):**

    - Exibe o tabuleiro do oponente (ocultando navios intactos).
    - Solicita e valida coordenada de tiro.
    - Processa o impacto e atualiza o estado da célula.
    - Verifica condição de vitória (`all_ships_sunk`).
    - Alterna o jogador.
5. **Encerramento (`cleanup_game`):** Libera toda a memória alocada na ordem inversa à criação.

5. Decisões de Design

Durante o desenvolvimento, tomamos decisões técnicas específicas para garantir robustez:

- **Linearização da Matriz:** Decidimos usar `vetor[linha * colunas + coluna]` ao invés de `matriz[linha][coluna]`. Isso melhora a **localidade de cache** do processador e simplifica drasticamente a liberação de memória (apenas um `free` em vez de um loop de `free`).
- **Substituição do `scanf`:** O uso direto de `scanf` foi abolido em favor de `fgets` com parsers manuais no módulo `io.c`. O motivo foi evitar loops infinitos ou sujeira no buffer de entrada caso o usuário digitasse letras em campos numéricos.
- **Separação de Lógica e IO:** O módulo `game.c` não faz `printf` direto de coordenadas brutas; ele delega a leitura para o `io.c`. Isso permite que a lógica de validação de regras (jogo) fique separada da lógica de validação de string (interface).

6. Gestão de Memória

O projeto segue rigorosamente o gerenciamento manual de memória:

- **Alocação:** Utilizamos `malloc` para criar o tabuleiro com tamanho definido pelo usuário em tempo de execução.
- **Expansão:** Utilizamos `realloc` na função `add_ship` (`fleet.c`), demonstrando capacidade de gerenciar listas dinâmicas.
- **Liberação:** Implementamos a função `destroy_board` e `destroy_fleet`. A liberação ocorre na função `cleanup_game` ao final da partida, garantindo que não haja vazamento de memória (*memory leaks*) entre execuções consecutivas no menu.
- **Segurança:** Todas as alocações verificam se o ponteiro retornado é `NULL` antes de prosseguir.

7. Testes e Validação

Para garantir a estabilidade, realizamos testes focados em casos de borda:

1. **Teste de Limites do Mapa:** Tentativa de posicionar navios grandes (ex: Porta-aviões) nas últimas colunas/linhas. A função `validate_manual_position` calcula matematicamente o índice final antes de escrever na memória, impedindo *Segmentation Faults*.
2. **Teste de Entrada Inválida:** Inserção de caracteres não numéricos ("abc") e coordenadas fora do range ("Z99"). O sistema bloqueia essas entradas e solicita nova digitação.
3. **Teste de Sobreposição:** Tentativa de colocar dois navios na mesma coordenada. A função `can_place_ship` verifica colisão antes da aplicação.

8. Conclusão

O desenvolvimento deste projeto permitiu consolidar o entendimento sobre como o computador gerencia recursos. A maior dificuldade encontrada foi tratar a entrada do usuário no posicionamento manual para evitar corrupção de memória. A solução foi implementar uma camada de pré-validação matemática.

Concluímos que a modularização não apenas organizou o código, mas permitiu que diferentes partes do sistema (como a validação de entrada) fossem testadas isoladamente, resultando em um software estável e confiável.