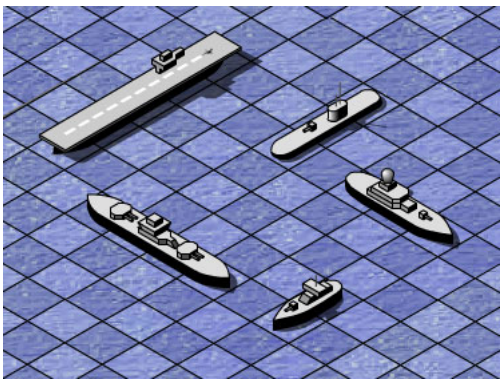
Batalha Naval

Evan Sá
up201707418

Margarida Costa
up201706103

May 2, 2020

Laboratório de Programação
Mestrado Integrado em Engenharia de Redes e Sistemas Informáticos e
Licenciatura em Ciência de Computadores



1 Introdução

Este trabalho tem como objetivo a implementação do jogo Batalha Naval na linguagem C. Para isso foi feito um estudo sobre as regras do jogo no qual nos baseamos para a realização do mesmo, sendo estas as seguintes:

- Cada jogador tem o mesmo número e tipo de navios, variando consoante o tamanho do tabuleiro de jogo (definido no início);
- Cada jogador define o seu modo de definição e inserção do navio no tabuleiro(manual, random ou ambos);
- Para o início da partida, é necessário que ambos os jogadores escolham um número, o que escolher o maior, será o primeiro a jogar;
- Depois de iniciada a partida, cada jogador à vez, deve escolher uma posição para acertar num navio do outro. Caso, acerte, tem a possibilidade de jogar outra vez;
- O jogo termina, quando um dos jogadores tiver os seus navios todos afundados.

2 Implementação

2.1 Bibliotecas usadas da linguagem C

As bibliotecas utilizadas foram as seguintes:

- **stdio.h**: define alguns tipos de variáveis, várias macros e várias funções para executar entrada e saída;
- **stdlib.h**: define alguns tipos de variáveis, várias macros e várias funções para executar funções gerais;
- **unistd.h**: define várias constantes e tipos simbólicos e declara funções diversas; esta foi utilizada essencialmente para implementação da função `sleep`. Esta função serve para suspender a execução do código, permitindo assim uma leitura mais perceptível.

2.2 MACROS

As macros, podem ser constantes ou funções:

2.2.1 Constantes

- **define sizeBitMap** `5`: define tamanho que uma bitmap tem, neste caso 5;
- **define PXREF** `2`: define o ponto de referência em x para a rotação do barco na bitmap;
- **define PYREF** `2`: define o ponto de referência em y para a rotação do barco na bimap;
- **define MAX_BUFFER** `1024`: define o tamanho máximo do buffer utilizado na manipulação do ficheiro.

2.2.2 Funções

- **define sin_int(int degrees)**: define uma função para retornar o valor do seno em relação aos ângulos múltiplos de 90;
- **define cos_int(int degrees)**: define uma função para retornar o valor do cosseno em relação aos ângulos múltiplos de 90;
- **define rotationNumber(number)**: define uma função que recebe um valor de 1 a 4, originário da função `random` da linguagem C, que permite assim saber qual a rotação que o navio deve sofrer.

2.3 Structs

As structs servem para para definir, para uma ou mais variáveis, um novo tipo de dados. Para cada struct utilizamos o `typedef struct` para dar um nome à mesma.

- **struct bitmap_**: define uma matriz de *unsigned char*, com o número de linhas e colunas e também guarda o ponto de referência (refx, refy) de inserção no tabuleiro do jogo;
- **struct ship_**: define um navio, sendo que este tem um tipo associado. O número de linhas e o número de colunas é definido dependendo do tipo do mesmo. A bitmap inicial(`bp`) é alterada, também pelo tipo de navio, ou seja, o desenho criado pelas linhas e colunas, é retratado com a modificação do valor '0' para '1'. A rotação e a translação escolhida pelo jogador, também influenciará a definição da bitmap. Estes valores são guardados nas variáveis (*`int` `rotation`, `int` `translationx`, `int` `translationy`*). O navio também tem guardado uma variável (*`int` `shotCount`* que guarda o número de células que o navio ocupa na bitmap. Cada vez que um jogador acerta numa célula do navio, a variável é decrementada - quando chega a 0, então o barco foi afundado;
- **struct listNode_**: define um nó do tipo SHIP e guarda o próximo nó;

- **struct list_**: define uma lista que tem associada o seu tamanho e a referência do nó inicial(*ListNode* `head`);
- **struct cell_**: define uma célula que tem um pointer para o navio, uma variável (*unsigned char* `value`) que é um ícone ('.', '*', '+' e '#') apresentado visualmente:
  - '.' : célula do tabuleiro oculta;
  - '*' : shot acertou numa célula do navio, no entanto o `shotCount` é diferente de 0;
  - '+' : shot acertou numa célula vazia;
  - '#' : navio afundado.

  Também existe uma variável(*unsigned char* `shot`) que guarda um valor, consoante o shot do jogador no adversário, sendo assim alterado durante a partida:
  - '0' : se não houver shot;
  - '1' : se o shot não acertar em nenhum navio;
  - '2' : se o shot acertar em algum navio.
- **struct matrix_**: define o tabuleiro do jogo, guardando o seu tamanho e uma matriz do tipo Cell;
- **struct user_**: define o jogador, guardando a matriz do mesmo, a lista de barcos que ele definiu(*List* `shipList`) e também o seu nome.

2.4 Enumerações

As enumerações servem para estipular um novo tipo de variável, limitando os valores da mesma:

- **ShipKind**: define o tipo de navio que existe na nossa implementação, como SOLO, TRIAL, SMALL_QUAD, BIGGEST_QUAD,L_GUY;
- **bool**: define um tipo booleano, como true ou false;

2.5 Ficheiros *source*

- **bitmap.c** : funções relacionados com a struct `bitmap`, como a sua inicialização e alteração do valor das céulas.
- **ship.c**: funções relacionadas com a definição do navio, como a sua criação, o cálculo das transformações geométricas(rotações e translações) que este pode sofrer e a eliminação de um navio.
- **list.c**: funções de implementação da lista, como a sua criação e a do nó e a adição de nós à cabeça da lista;

- **matrix.c**: funções relacionadas com o tabuleiro do jogo, como a sua inicialização, a inserção de navios na matriz, a eliminação de navios na matriz e a demonstração gráfica dos tabuleiros de cada jogador no terminal.

- **initGame.c**: funções de início do jogo, como a criação dos utilizadores, o cálculo do número de barcos que irá ser usado por cada jogador durante o jogo, sendo este feito pela seguinte fórmula:

  $$tamanhodotabuleiro/(sizeBitMap * sizeBitMap),$$

  também encontramos a função que permite definir quem inicia o jogo;

- **game.c**: funções relacionadas com o jogo, como o disparo efetuado pelos jogadores (que vai atualizar as variáveis `value` e `shot` da célula da matriz e o valor guardado na célula da bitmap). A cada disparo verifica, se a célula é um pointer para um navio ou não, em caso afirmativo é apurado se esse disparo afunda o navio e o jogador pode voltar a jogar, caso contrário passa a vez. Foi implementado uma função em que os jogadores podem escolher se desejam definir e posicionar os seus navios de forma manual, aleatória ou ambas.

- **randomShips.c**: funções para criação de pontos para a translação, a rotação e para a inserção no tabulereiro do jogo, utilizando a função já existente `random()` da linguagem C. Nestas funções, depois da criação, os navios sofrem as alterações geométricas com o valor retornado pelo random e também os navios são inseridos.

- **menu.c**: implementada a iteração entre o terminal e o utilizador, demonstrando menus iterativos através da escolha de números, sendo assim possível iniciar um jogo, visualizar as regras e sair do jogo.

- **main.c**: função principal do jogo.

2.6 Manipulação de ficheiros

As regras do jogo estão escritas num ficheiro `.txt`. Como a linguagem C permite ler texto atráves de ficheiros, utilizamos o `fopen()`, para imprimir as regras no terminal, facilitando assim a leitura das mesmas. O código usado é o seguinte:

```
char buffer[MAX_BUFFER];
    int c;

    FILE *file;
    file = fopen("rules.txt", "r");

    if(file) {
        while((c = getc(file)) != EOF) putchar(c);
            fclose(file);
    }
```

3 Interface

3.1 Navios

Os navios são desenhados da seguinte forma:

```
SHIPS'S TYPE:

1) x    2) x x x    3) x x    4) x x x x    5) x          6) x x x    7)     x x x
                       x x       x x x x       x             x            x
                                 x x x x       x             x            x
                                 x x x x       x                          x
                                               x x x x x                x x x
```

3.1.1 Bitmap default de cada navio

```
BITMAP SHIP (default):

SOLO: 1 0 0 0 0  TRIAL: 1 1 1 0 0  SMALL_QUAD: 1 1 0 0 0  BIGGEST_QUAD: 1 1 1 1 0
      0 0 0 0 0        0 0 0 0 0              1 1 0 0 0               1 1 1 1 0
      0 0 0 0 0        0 0 0 0 0              0 0 0 0 0               1 1 1 1 0
      0 0 0 0 0        0 0 0 0 0              0 0 0 0 0               1 1 1 1 0
      0 0 0 0 0        0 0 0 0 0              0 0 0 0 0               0 0 0 0 0

           L_GUY: 1 0 0 0 0    T_GUY: 1 1 1 0 0    S_GUY: 0 0 1 1 1
                  1 0 0 0 0           0 1 0 0 0           0 0 1 0 0
                  1 0 0 0 0           0 1 0 0 0           0 0 1 0 0
                  1 0 0 0 0           0 0 0 0 0           0 0 1 0 0
                  1 1 1 1 1           0 0 0 0 0           1 1 1 0 0
```

3.2 Início do jogo

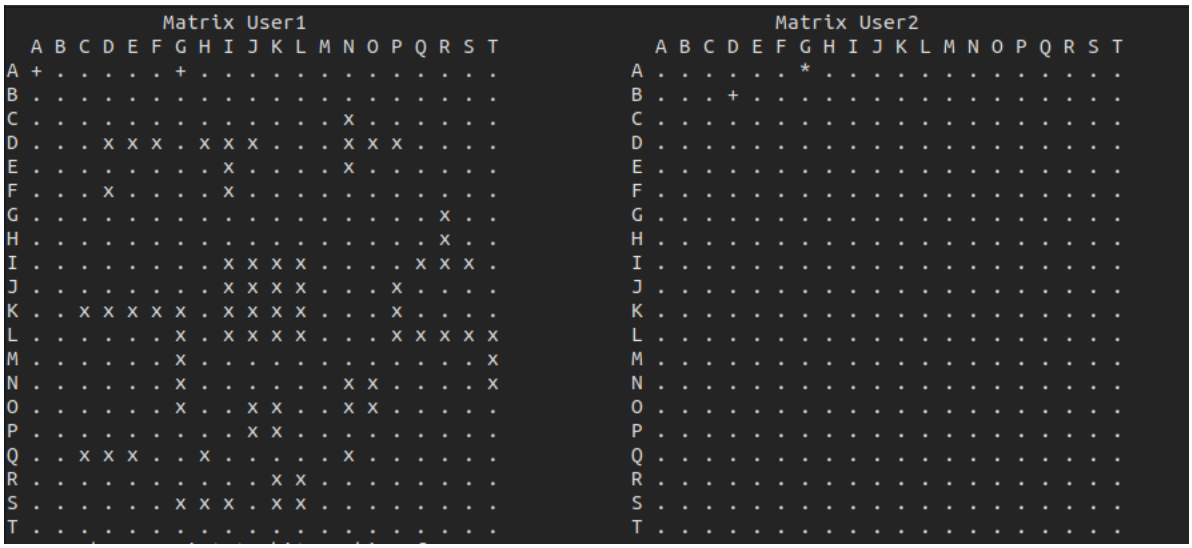
```
--------------------------------------------------
            WELCOME TO BATTLESHIP
--------------------------------------------------

(1) Start
(2) Rules
(3) Exit

Choose one option:
```

3.3 Tabuleiro do jogo de um utilizador

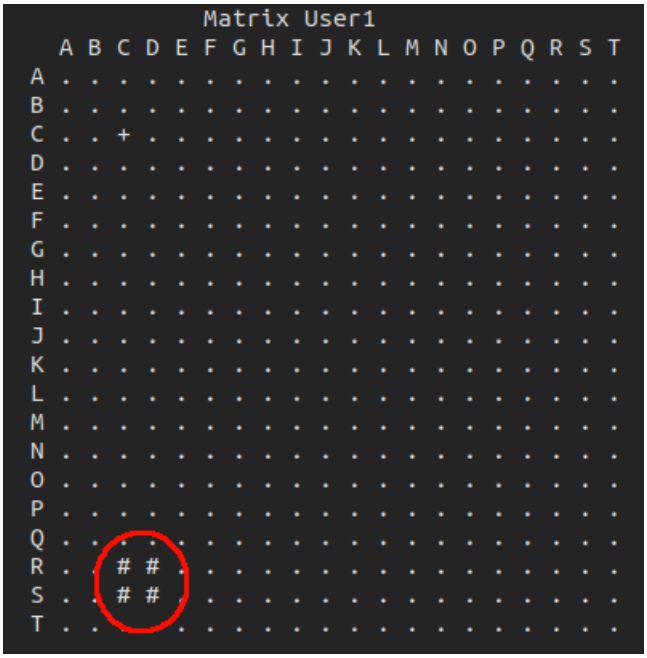
3.3.1 Tabuleiro depois de inserir os navios todos de um utilizador

3.3.2 Tabuleiro do utilizador1 depois de 2 jogadas

A jogada efetuada foi no ponto **G,A** e acertou num barco.
A jogada efetuada foi no ponto **D,B** e acertou numa célula vazia.



3.3.3 Tabuleiro do utilizador1 depois de ter acertado num barco completo do utilizador2



4 Memória

Ao longo do trabalho, foi necessário alocar memória dinamicamente, permitindo assim guardar de acordo com a necessidade da *struct*, reservando até espaço. No entanto, as variáveis criadas com alocação dinâmica não são libertadas automaticamente, sendo assim necessário libertá-las manualmente.
Um exemplo de alocação dinâmica utilizado nesta implementação é o seguinte:

```
SHIP *newShip = malloc(sizeof(SHIP));
```

Este caso, a utilização de `malloc` faz com que seja alocado um espaço para um bloco de bytes consecutivos na RAM, sendo o número de bytes usados igual ao número de bytes da *struct* SHIP.
Sendo assim, o newShip é um apontador que aponta para o endereço de memória que contém um espaço do tipo e tamanho SHIP.
Para libertar a memória neste caso, é necessário:

```
free(newShip -> bp);
free(newShip);
```

Este caso, como nós vimos no ponto 2.3, a *struct* SHIP aloca memória para uma BitMap, então será necessário primeiro libertar a BitMap e depois libertar o SHIP. Esta libertação é permitida devido à utilização da *keyword* `free`, que indica ao sistema que o espaço se encontra disponível.
Uma forma de perceber se a memória alocada dinamicamente foi toda libertada após a execução do programa, é se forem encontradas ***Memory Leaks***. A forma de se detetar a presenças das mesmas, é fazendo o seguinte comando:

```
gcc -fsanitize=address main *.c
```

Caso, no fim da execução não sejam detetadas *Memory Leaks*, indica que a memória foi toda libertada; caso contrário, dará um erro, indicando onde estão a ocorrer as mesmas.
Na nossa implementação, não são detetadas *Memory Leaks*, o que assinala que a memória que já não é necessária, foi toda libertada.

5 Execução do Código

Para a execucação do código foi criado um `Makefile`, pois são utilizados vários ficheiros e é mais fácil de utilização. O MakeFile é definido pelos objetos(**OBJS**): `bitmap.o, ship.o, list.o, matrix.o, initGame.o, game.o, randomShips.o, menu.o` e `main.o`, sendo o programa(**PROGRAM**) igual a `./main`. A **CCFLAG** usada é `-g`, uma vez que é importante para a otimização e depuração da compilação do programa. O ficheiro a ser executado (**all**) é o que se encontra em PROGRAM. Para uma nova compilação deve ser feito `make clean`, definido por `rm -f` (*PROGRAM*)`(OBJS)` (**clean**).