U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

LABORATÓRIO DE COMPUTADORES - MIEIC

RELATÓRIO FINAL - SPACE BATTLESHIPS

Diogo Moura, up201304068

Pedro Arnaldo, up201303890

Pedro Costa, up201303973

ÍNDICE

Secção I

User Instructions

Menu Inicial

Ao iniciar, o programa apresenta imediatamente o menu principal, com as opções mais importantes do jogo.

Fig. 1 - Menu Inicial

- Jogar: ao selecionar esta opção, o programa leva-nos ao inicio do jogo, ou seja, ao posicionamento das naves. Após o posicionamento das naves estar completo, o jogo começa, tendo o jogador de tentar destruir todas as naves do jogador adversário;

- Highscore: esta opção mostra-nos os cinco melhores jogadores, com os respetivos *highscores* e a data na qual os obtiveram;

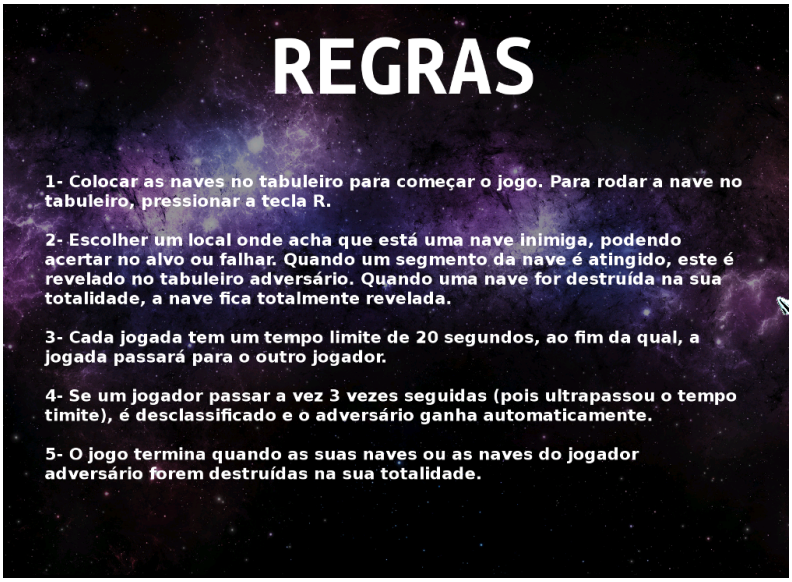


Fig. 2 - imagem instruções

- Regras: tal como o nome indica, mostra as regras do jogo;
- Sair: Sai do jogo, libertando toda a memória alocada durante o programa.

Como jogar

O objetivo do jogo é destruir as naves inimigas antes que o inimigo destrua as nossas.

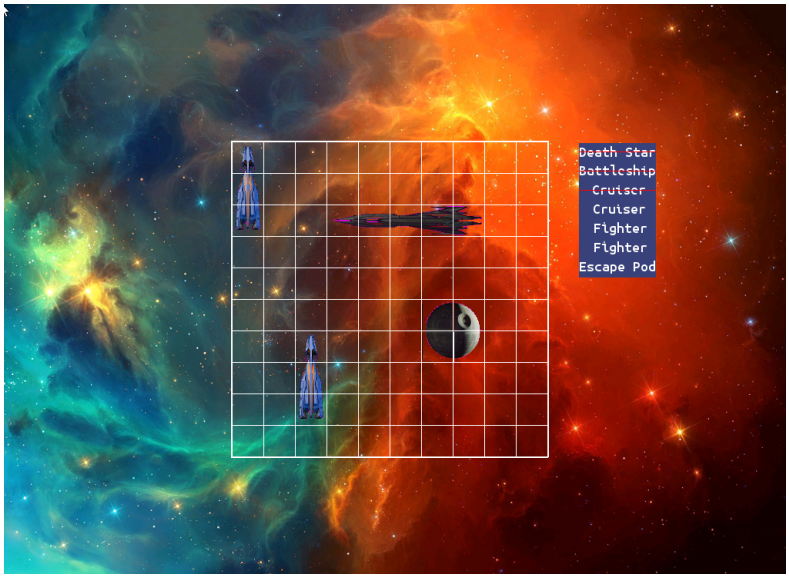


Fig. 3 - Posicionamento das naves

Existem 5 tipos de naves, todas com tamanhos diferentes. O jogo começa com o posicionamento das naves no tabuleiro de jogo. Para as posicionar, usam-se as setas para mexer as naves no tabuleiro. Pode-se pressionar a tecla *R* para mudar a orientação da nave e *Enter* para colocar a nave definitivamente no tabuleiro de jogo. Após todas as naves estarem colocadas, o jogo começa.

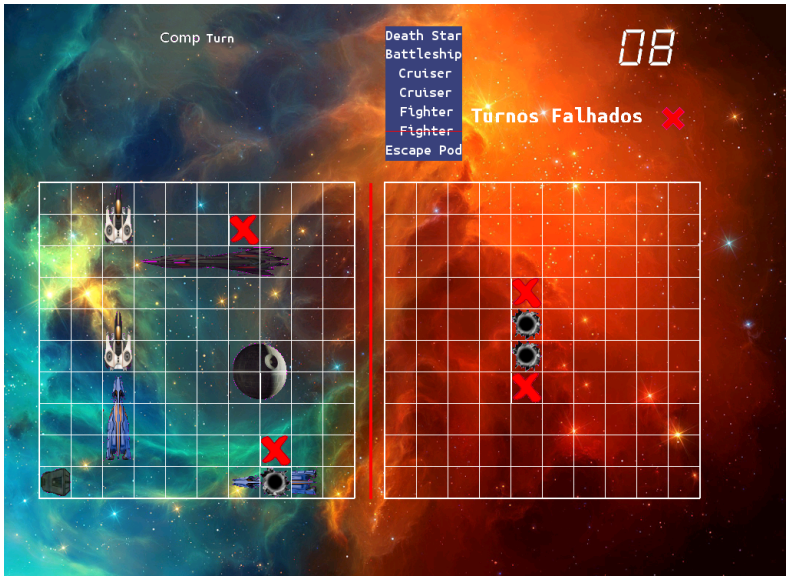


Fig. 4 - Jogo

Mais uma vez, utilizam-se as setas para a navegação no tabuleiro e *Enter* para selecionar para onde quer disparar. O jogador pode acertar numa nave (*hit shot*) ou no espaço (*missed shot*). Cada jogada têm um tempo limite de 10 segundos e caso o jogador acerte numa nave com um tiro, pode jogar novamente.

Se o jogador quiser, pode premir P para ir para pausa e uma vez lá, se premir *Escape*, o jogo acaba e vai para o menu inicial.

No final do jogo é atribuído um *highscore* ao vencedor. Caso seja superior aos cinco melhores *highscores*, é colocado no array de *highscores* e mostrado no menu de *highscores*.

Secção II

Devices

Device	Utilização	Interrupções
Timer	Controlar *frame rate*, contar o tempo de cada jogada e do jogo.	SIM
KBD	Navegar pelo tabuleiro para jogar. Introdução de nomes no highscore.	SIM
Mouse	Navegação dos menus, jogo em si após a colocação das naves	SIM
Video Card	Desenho das imagens de fundo, tabuleiro de jogo e as naves	NÃO
RTC	Data associada ao *highscore* de um jogador	NÃO

Função de cada *device*

- Teclado: colocação das naves no tabuleiro e jogar (utilizando interrupções). Usamos as setas para a navegação no tabuleiro, *Enter* para selecionar a quadrícula, P para ir para pausa e *Esc* para sair do jogo.

```
case KEY_ARR_UP_BRK:
    if (game_state->com.tab.selected_y > 0) {
        game_state->com.tab.selected_y--;
    }
    battle->kb_code = KEY_NONE;
    break;

case KEY_ARR_DOWN_BRK:
    if (game_state->com.tab.selected_y != 9) {
        game_state->com.tab.selected_y++;
    }
    battle->kb_code = KEY_NONE;
    break;
```

Fig. 5 - Reconhecimento de inputs

- Rato: Usado na navegação do menu, utilizando o botão direito para selecionar o modo que o jogador pretende.

Fig. 6 - Rato sobre a opção *Player vs AI*

- *Real Time Clock (RTC)*: atribuição de uma data ao highscore de um jogador.

- Placa Gráfica: responsável pela interface gráfica (em modo de vídeo).

- *Timer*: É utilizado para contar o tempo de cada jogada, para ter a certeza que cada jogada não demora mais de 10 segundos. Se demorar mais, o jogador perde essa jogada, ou seja, a jogada passa para o jogador adversario. Também serve para controlar a *frame rate* do jogo.

```
battle->IRQ_SET_TIMER = timer_subscribe_int();

getMouse();

// Por frequencia do timer a 60
timer_set_square(0, 60);
```

Fig. 7 - Definição da frequência do *timer*

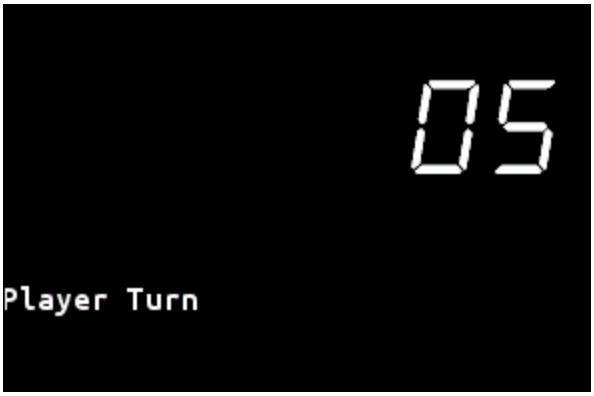


Fig. 8 - Tempo restante para a jogada

Secção III

Organização/Estrutura do código

Módulos e respetiva descrição

- ***Battleship:***

    - Importância para o projeto: 16 %
    - Membros responsáveis: Diogo Moura (6%) e Pedro Costa(10%)
    - O módulo principal do programa, onde estão localizadas as funções do jogo que permitem que este seja inicializado, desenhado, actualizado e terminado. Também é onde se encontram as funções que levam ao funcionamento da máquina de estados, permitindo a mudança, atualização e eliminação dos estados em que o programa possa estar.
    - Utiliza os seguintes módulos: *Keyboard_mouse*, *Timer*, *Game*, *Button*, *Bitmap*, *Graphics* e *MainMenu*

- ***Bitmap:***
    - Importância para o projeto: 5 %
    - Membros responsáveis: Diogo Moura (5%)
    - Módulo responsavel pelo desenho de todas as imagens .bmp no programa.
    - Utiliza os seguintes módulos: *Graphics*.

- ***Button:***
    - Importância para o projeto: 2 %
    - Membros responsáveis: Pedro Costa (2%)
    - Responsável pela criação, atualização e eliminaçao dos botões do menu inicial.
    - Utiliza os seguintes módulos: *Graphics* e *Keyboard_mouse*.

- ***Game:***
    - Importância para o projeto: 25 %
    - Membros responsáveis: Diogo Moura (12.5%) e Pedro Costa (12.5%)
    - Módulo encarregue modo de jogo. Está encarregue da inicialização/atualização dos jogadores, desenho do jogo, inteligência artificial, colocação das naves, reconhecimento de colisões, etc. Encarregue também da eliminação do jogo.
    - Utiliza os seguintes módulos: *Battleship*, *Keyboard_mouse* e *Graphics*.

- ***Graphics:***
    - Importância para o projeto: 12 %
    - Membros responsáveis: Diogo Moura (12%)
    - Usado para desenhar objetos de menor tamanho (quadrados, linhas ...).
    - Utiliza os seguintes módulos: *VBE* e *Keyboard_mouse*.

- ***Handler:***
    - Importância para o projeto: 4 %
    - Membros responsáveis: Diogo Moura (2%) e Pedro Costa (2%)
    - Código em *assembly* para verificar em que teclas o jogador esta a carregar.

- ***Highscore:***
    - Importância para o projeto: 8 %
    - Membros responsáveis: Diogo Moura (8%)
    - Membros responsáveis:
    - Módulo responsável pela atribução de um *highscore* a um jogador e *display* do mesmo (seja esta superior as 5 melhores).
    - Utiliza o seguinte módulo: RTC.

- ***Keyboard_mouse:***
    - Importância para o projeto: 10 %
    - Membros responsáveis: Diogo Moura (totalidade do rato e metade do teclado) e Pedro Costa (metade do teclado)

- Módulo que trata de todas as interações do jogador com os dispositivos de *input* (teclado e rato).
- Utiliza o seguinte módulo: *Bitmap.*

- ***Main Menu:***
  - Importância para o projeto: 10 %
  - Membros responsáveis: Pedro Costa (10%)
  - Membros responsáveis:
  - Responsável pela criação do menu inicial.
  - Utiliza os seguintes módulos: *Graphics*, *Bitmap* e *Keyboard_mouse.*

- ***Timer:***
  - Importância para o projeto: 6%
  - Membros responsáveis: Diogo Moura (3%) e Pedro Costa (3%)
  - Contém as funções de tratamento do *timer* (interrupções e definição da frequência).

- ***RTC:***
  - Importância para o projeto: 2%
  - Membros responsáveis: Pedro Arnaldo (2%)

Classes e respetiva descrição

- ***MainMenuState***
  - Classe responsável pelo Menu Inicial
    - **done:** variável que nos diz se podemos sair do *Main Menu;*
    - ***exit_button, play_ai_button, instructions_button, highscores_ button***: Butões de cada umas das opções do menu;
    - **background:** fundo do menu inicial

```
typedef struct {
    int done;
    currently_selected selected_button;
    Button* exit_button;
    Button* play_ai_button;
    Button* instructions_button;
    Button* highscores_button;
    Bitmap* background;
} MainMenuState;
```

Fig. 9 - *MainMenuState Class*

- ***SetShipState***
    - Classe responsável pelo modo de colocação das naves no tabuleiro.
        - **tab, tab_com:** tabuleiro do jogador e do computador (respetivamente);
        - **ship_temp:** nave que esta a ser posicionada no tabuleiro;
        - **ship_list:** imagem com o nome das naves;
        - **ship_map:** imagem com todas as naves;
        - **done:** variável que indica que já acabamos de colocar as naves

```
typedef struct {
    tabuleiro tab;
    tabuleiro tab_com;
    ship* ship_temp;
    Bitmap* ship_list;
    Bitmap* ship_map;
    int done;
} SetShipState;
```

Fig. 10 *SetShipState Class*

- ***GameState***
    - Classe responsável pelo jogo em si.
        - **turn_time_counter:** contador para contar o tempo de cada jogada;
        - **turn:** indica que é a jogar;
        - **winner:** indica quem ganhou;
        - **hum, com:** jogadores;
        - **ai_comp:** inteligência artificial do computador;
        - **alarm_clock:** contador decrescente do tempo da jogada (de 10 a 0);
        - **turn:** mostra um texto que indica que é que está a jogar (hum ou com);
        - **pause_screen:** imagem do menu de pausa durante o jogo;
        - **done:** indica que o jogo acabou;
        - **pause:** indica se estamos no menu de pausa.

```
typedef struct {
    unsigned short turn_time_counter;
    unsigned short turn;
    unsigned short winner;
    player hum;
    player com;
    bot_ai ai_comp;
    Bitmap* ship_map;
    Bitmap* alarm_clock;
    Bitmap* ship_list;
    Bitmap* turns;
    Bitmap* pause_screen;
    int done;
    unsigned int pause;

} GameState;
```

Fig. 11 - *GameState Class*

- ***Highscore_State***
    - Classe responsavel pelo menu dos highscores, *display* dos mesmos.
        - **year, month, day, hour, min:** elementos da data atribuida ao *highscore;*
        - **jogador_array:** array que contém as estruturas dos jogadores com os 5 melhores *highscores*;

      As restantes variáveis são auxiliares ao estado do *Highscore,* sendo necessárias para o seu funcionamento e visualisação.

```
typedef struct {
    unsigned int year;
    unsigned int month;
    unsigned int day;
    unsigned int hour;
    unsigned int min;

    Jogador jogador_array[5];

    unsigned short show;

    char nome_player[10];

    unsigned short score_player;

    unsigned short done;

    Bitmap* fonts;

} Highscore_State;
```

Fig. 12 - *Highscore_State Class*

- ***Button***
    - Classe responsável por todos os botões do programa.

```
typedef struct {
    int x_ini, y_ini, x_final, y_final;
    int width, height;
    int mouse_hover;
    short color_border;
    unsigned short available;
} Button;
```

Fig. 13 - *Button*

- ***Mouse***
    - Classe responsável pelo rato.

```
typedef struct {
    int x, y;
    double speedMultiplier;

    int bytesRead;
    long packets[3];

    int leftButtonDown;
    int rightButtonDown;
    Bitmap* mouse_up;
    Bitmap* mouse_down;
    int hasBeenUpdated;
    int draw;
} Mouse;
```

Fig. 14 - *Mouse Class*

Sub-”classes” e respetiva descrição

- ***ship_part*:** contem a informação de um dos segmentos das naves: qual o segmento, qual é o tipo de nave (death star, cruiser …), direção e se foi atingido ou não.
- ***ship*:** como o nome indica, possui a informação de uma nave: tipo de nave, arrays dos segmentos, número de vezes que foi atingida, se está destruída ou não, posição, direção e o tamanho.
- ***tabuleiro*:** classe responsável pelo tabuleiro. Contém o array do tabuleiro, posição selecionada, array de naves, e qual a nave que estamos a colocar no tabuleiro.
- ***player*:** classe com a informação do jogadar: tipo de jogador (humano ou computador), tabuleiro desse jogador, número de tiros falhados, tempo de jogo, número de vezes que passou a jogada e número de naves destruidas.
- ***bot_ai*:** classe com a informação do bot: última posição em que acertou numa nave, direção, orientação (0 anda para trás e 1 para a frente) e informação sobre a mudança de orientação e sobre a jogada anterior (se atingiu uma nave ou não).
- ***Bitmap*:** contém a informação de um *bitmap* e os dados da mesma.

Funcionalidades Implementadas

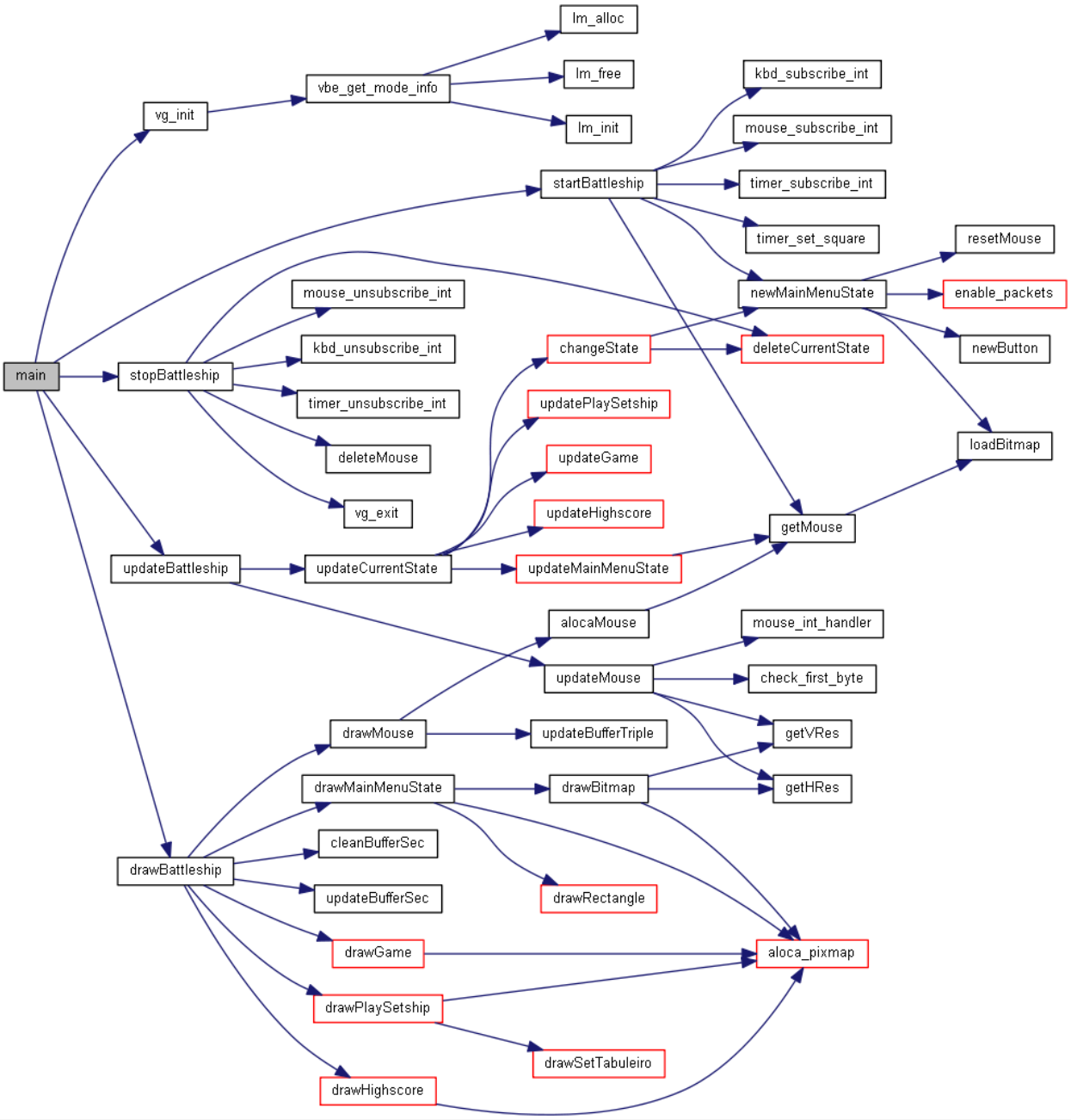
Em relação à proposta do projeto, foram implementadas as seguintes funcionalidades:

- Implementação do *timer*, teclado, rato, placa gráfica (usando .bmp’s) e a lógica de jogo.
- Menus (através de máquinas de estados) e implementação do RTC para mostrar a data atual (ano, mês, dia e minutos).

Funcionalidades não Implementadas

A única funcionalidade especificada na proposta do projeto que não foi implementada foi o modo *multi-player*, através do uso da *UART*.

Secção IV



A descrição das principais funções chamadas neste gráfico encontram-se na secção seguinte.

Secção V

Buffering:

No nosso programa foi implementado "*triple buffering*" para o desenho do rato funcionar corretamente. Em primeiro são colocadas todas as imagens e o fundo no "second" *buffer*. Posteriormente copia-se o "second" *buffer* para o "triple"(entenda-se: terceiro) *buffer* e adiciona-se o rato. Por último, este é movido para a *video memory.* Esta operacão apenas é realizada no estado do Main Menu, onde se usa o rato, sendo nos outros estados apenas implementado o *double buffering* (não se usa o terceiro). A *video memory* está a ser atualizada a uma taxa fixa (30 FPS), controlada pelo timer, sendo que a cada 2 ticks do timer o "second" *buffer* é limpo, rescrito e novamente copiado.

Máquina de Estados:

No nosso programa decidimos utilizar uma máquina de estados para progredirmos no programa. A máquina funciona da seguinte maneira: existem quatro estados principais, simulados pelos vários módulos do programa e por um *enum* e por um apontador que guardam o estado atual em que o programa se encontra.

O programa corre em 3 partes principais:

1. Inicialização do programa - *startBattleship()*
2. Atualização e desenho do programa, baseado no estado atual - ciclo while que chama *updateBattleship(battleship)* e *drawBattleship(battleship)*
3. Finalização do programa - *stopBattleship(battleship)*

1 - Inicialização do programa

Aqui é inicializado o programa. São efetuadas as subscrições do rato, timer e teclado e é inicializado o modo de vídeo (0x117). É também alocado espaço para as variáveis da battleship, estrutura principal do programa que é passada como argumento para as funções que necessitem dados desta. O estado inicial é definido como sendo o *Main Menu*, inicializando-se este estado para a variável *void* state* (apontador genérico). Houve necessidade de usar um apontador genérico, pois tínhamos os vários estados em estruturas dentro dos respetivos módulos e queríamos guardar os seus apontadores em apenas uma variável da *Battleship*. É a estrutura da *Battleship*

que guarda o estado do programa, sendo esta a *Battleship* estrutura principal e a mais importante.

2 - Atualização e desenho do programa

A atualização do programa é o passo mais importante da máquina de estados, pois **analisa o estado atual**, **progride com o estado, consoante as interrupções detetadas (alterando as variáveis do estado)** e **permite a mudança de estados**.

A seguir à inicialização do programa existe um ciclo while que, enquanto o programa não tiver a flag de terminação ativa (variável *done*), vai correr duas funções: a função de atualização do programa e a função para o desenhar.

- A função de atualização do programa verifica o estado atual da máquina.
- Consoante o estado, a máquina vai chamar a função de atualização específica desse estado. Isto significa que cada estado tem as suas funções de atualização, que vão ser o *core* do programa, detetando alterações e progredindo com o programa, alterando variáveis do estado.
- Após a atualização, a máquina de estados verifica a *flag* "*done"* do estado e analisa a necessidade de mudar de estado com a função *changeState(battleship, statetochange)*. A variável *statetochange* é o retorno das funções update dos estados e indica o estado para o qual se vai mudar. Esta função é chamada se a *flag "done"* do estado atual se encontrar a 1.

Após a atualização é chamada a função que desenha o programa. Consoante o estado, é chamada a função de desenho correspondente, tenho cada estado uma função própria. Uma particularidade da função *drawBattleship* é que a sua chamada, apesar de se encontrar dentro do ciclo while, é controlada pelo timer. Assim, estando o timer a 30 FPS, esta função é chamada 30 vezes por segundo, mas sempre após a atualização.

3 - Finalização do programa

No final do programa, ou seja, quando a *flag done* da *battleship* ficar a 1, por alguma razão (como quando o utilizador carrega em "Sair"), o programa é terminado. Assim, é em primeiro apagado o estado em que o programa se encontrar (mais uma vez, cada estado tem uma função de terminação própria), é libertada toda a memória

anteriormente alocada, são efetuados os *unsubscribes* dos vários dispositivos (timer, keyboard e mouse) e sai-se do modo de vídeo do programa.

A partir daqui, a finalização e mudança de estados é controlada pela variável "*done*" de cada estado. A própria estrutura principal battleship possui também esta variável, para quando se quiser sair do programa. A lógica da máquina de estados é a seguinte: há sempre um estado atual ativo, e quando a variável "*done"* estiver a 1, passa-se para o estado seguinte, consoante as ações do utilizador.

Temos cinco estados. Cada um equivalente a uma opção do menus (sendo que o modo de jogo tem dois) e o estado do menu principal.

Lógica de Jogo:

A grande dificuldade do nosso projeto residiu na própria lógica e implementação de uma versão do jogo da "batalha naval". Houve uma certa dificuldade na decisão das estruturas e métodos a usar, mas achamos que encontrámos uma solução mais ou menos engenhosa e eficaz.

Decidimos implementar como estrutura principal uma estrutura de Jogo, que guardasse os tabuleiros de jogo do computador e do jogador. No entanto, como tínhamos um estado para o posicionamento dos navios e outro para o jogo em si, decidimos usar o mesmo ficheiro para as estruturas de ambos os estados. Decidimos assim que estes iriam ser "mini-estados" que transferiam as estruturas tabuleiro de um para o outro. Os tabuleiros são constituídos por arrays multidimensionais *10x10,* estando em cada espaço do tabuleiro uma *ship_part*, ou seja, uma peça de navio, que possuei **um tipo de parte** e **um tipo de navio**.

Outro engenho que usámos para facilitar o trabalho foi definir que nos **tipos de parte** poderiamos ter água, primeira peça, segunda peça e por aí fora, enquanto que nos **tipos de navio** poderiamos ter os nomes do navio ou "Nothing" que simula um quadrado sem navio presente. A presença destas duas variáveis (baseadas nos valores dos *enums*) permitiram-nos criar as várias combinações para os quadrados (com restrições, por exemplo, nunca se vai ter um quadrado com água e um navio lá ao mesmo tempo). Um exemplo de quadrado/elemento array: type_part First e type_ship Fighter.

Para que os quadrados não fossem independentes, criámos então a estrutura Ship, que aglomera alguns dos quadrados num array, permitindo assim identificar quantas peças o navio tem, quantas peças foram destruídas, etc. Uma particularidade desta estrutura é que, além do array de ship_parts, o navio possui uma peça principal,

uma direção para a qual se estende e um tamanho, variáveis que nos foram muito utéis.

No 1º mini-estado, SetShipState, colocam-se as naves e criam-se os tabuleiros. O jogador coloca as naves no seu tabuleiro, sendo que estas nunca podem passar dos limites e é detetada a sua colisão de tal modo que não podem ser colocadas. Criámos uma função que coloca aleatoriamente as naves do computador.

Na verdade é colocada aleatoriamente a peça central de cada nave, à vez, e usando a direção e tamanho da nave analisam-se as colisões. Se existir alguma peça que contenha um tipo diferente de “water” e coincida com uma peça do navio, existe colisão e é feito um novo “random” para o local da peça. No final de serem colocados os tabuleiros, estes poderão ser usados pelo 2º mini-estado, GameState, através de um static pointer que se encontra no ficheiro Game.c (que os estados partilham). Estes tabuleiros vão tambem ser inseridos nas estruturas Player, cujas instâncias são criadas para o jogo.

No 2º mini-estado, GameState, decorre o jogo. Existem turnos, controlados por um timer e o jogador/computador seleciona peças para tentar acertar nos navios. O jogador navega pelo tabuleiro do computador com o teclado e ao carregar enter, é analisado o tipo de parte selecionado e é revelado se era uma nave ou “água”. Esta parte não foi particularmente difícil de implementar. Sempre que o jogador ou o computador atingem água ou deixam passar os 10 segundos contados a partir do timer (usa-se o mesmo timer que para os fps, apenas se usam fatores multiplicativos), o turno muda. Aqui encontrámos um pequeno desafio: como fazer o computador jogar no seu turno?

Ao início pensámos em apenas usar jogadas aleatórias, mas apercebemo-nos que isso não teria muita piada, pois seria muito difícil para o computador destruir as naves. Desenvolvemos então uma função chamada “*bot_ai*”, que é chamada no turno deste, e que não tornámos mais complexa por falta de tempo.

A 1ª jogada é aleatória. A partir daí, sempre que acerta em água, o computador faz a próxima jogada aleatória, mas quando acertar numa peça de uma nave, o computador decide numa direção e sentido aleatórios, para tentar destruir o resto da nave. Se o computador estiver a destruir uma nave e chegar a uma parede, também começa a jogar em sentido contrário. Finalmente, se o computador acertar numa peça já rebentada, joga novamente, pelas regras anteriores (atenção: se estiver a destruir e bater contra uma parede, volta para trás e começa a jogar do último quadrado não rebentado, pois continua no seu sentido e passa as peças rebentadas à frente).

O jogo vai-se assim sucedendo, até que alguém destrua todas as naves, verificação que está sempre a ser executada.

Secção VI

Avaliação da Disciplina

Aspetos Negativos:

- Elevada quantidade de trabalho em relação aos créditos da disciplina: o número de horas dispendidas na preparação dos laboratórios e do projeto é demasiado elevado, em parte devido a estrutura labiríntica da cadeira, que nos obriga a gastar muito tempo à procura de soluções para problemas que nos deveriam ser melhor explicados logo de início.

- A avaliação não é efetuada de forma justa: a avaliação da disciplina apenas conta com trabalhos de grupo. Isto leva a que haja sempre alguém nos grupos que trabalha mais/menos que o parceiro, apesar do esforço dos professores em contrariar esta tendência. Notou-se que, em quase todas as aulas e para quase todos os grupos, havia um membro que se esforçava menos, apresentando pouco interesse para com os trabalhos. Achamos que algum tipo de avaliação individual poderia corrigir isto e incentivar os alunos a esforçarem-se mais, semana a semana. Poderiam até substituir algum dos laboratórios por uma avaliação individual sobre o dispositivo desse laboratório.

- Algumas turmas têm aula laboratorial antes da teórica: devido aos atrasos que as aulas teóricas sofrem de vez em quando e à diferença de horários entre turmas, acontece que algumas turmas ficam em desvantagem em relação a outras, por não terem tido a aula teórica do laboratório correspondente. Assim, os elementos destas turmas dispendem muito mais tempo para se prepararem, para além de que já aconteceu terem menos tempos que os colegas para realizarem os laboratórios, devido à data de disponibilização dos laboratórios (que, nas primeiras semanas chegou a acontecer a uma sexta feira à noite, havendo aula laboratorial na segunda seguinte).

Aspetos Positivos:

- Aprendemos a utilizar a interface de *hardware* dos periféricos mais habituais de um computador, desenvolver *software* de baixo nível, utilizar a linguagem de programação C de modo estruturado e a utilizar várias ferramentas de desenvolvimento de software, que nos ensinaram a trabalhar e programar em grupo.

Auto-avaliações

	Participação	Contribuição
Diogo Moura	**50**	**55**
Pedro Arnaldo	**10**	**10**
Pedro Costa	**40**	**35**

No geral destacou-se mais o trabalho do Diogo Moura, especialmente no que toca às funções de alguns dos dispositivos e da estruturação geral do projeto.

Em termos de escrita de código, destacou-se um pouco mais pela negativa o Pedro Arnaldo, que compensou, no entanto, noutras áreas, como o design/criação de imagens e desenvolvimento de partes do relatório.

O Pedro Costa ajudou a desenvolver a parte a que se tinha proposto na especificação, tendo dividido o trabalho de escrita de código, estruturação e escrita do relatório com os restantes elementos. Infelizmente, devido à falta de tempo e esforço por parte do grupo, não nos foi possível implementar a Serial Port.