Faculdade de Engenharia da Universidade do Porto
Laboratório de Computadores
2018/19

Typing Invaders

Relatório do Projeto Final

T3G07
David Luís Dias da Silva
up201705373
Luís Pedro Pereira Lopes Mascarenhas Cunha
up201706736

1. Instruções de utilização

Ao iniciar o jogo, o jogador é levado ao menu principal do programa. Lá são-lhe apresentadas quatro opções que podem ser selecionadas clicando no botão do lado esquerdo do rato enquanto a ponta do cursor se encontra em cima do botão pretendido.

- ***Play***: leva o jogador ao jogo single-player;
- ***Help a friend***: inicia o jogo *CO-OP*;
- ***Highscores***: mostra uma tablela com as melhores pontuações no jogo;
- ***Exit***: abandona o jogo.

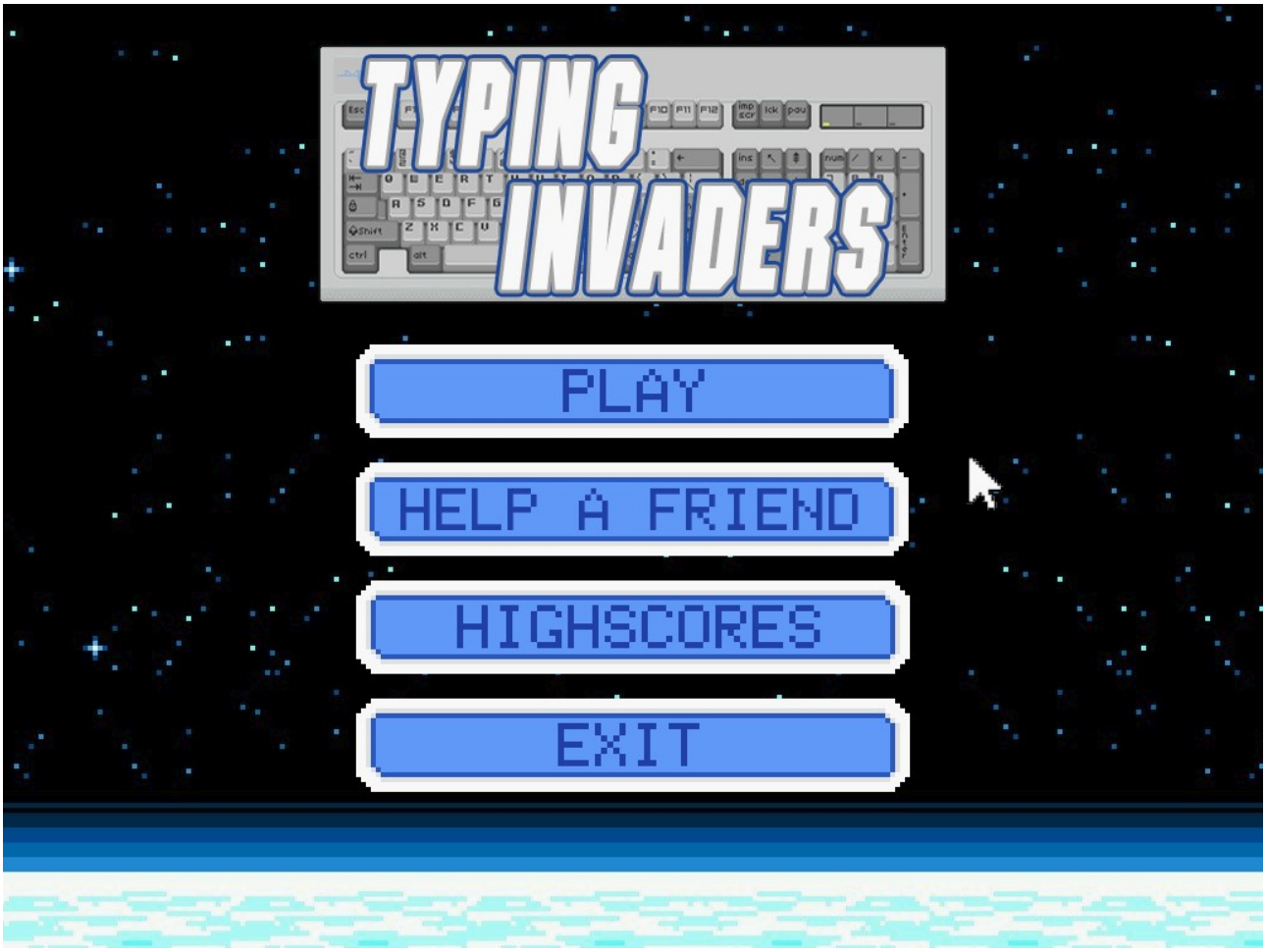


Figura 1: Menu principal

1.1 Jogo *Singleplayer*

No jogo *singleplayer* o jogador controla a Base LCOM(2) que tem que proteger a Terra da ameaça das naves alienígenas. Começando o jogo, naves(3) que contêm carateres começam a surgir no topo do ecrã e a descer em direção ao jogador. Para impedir o seu avanço, o jogador deve clicar nas teclas correspondentes aos carateres, o que faz com que este dispare projeteis "teleguiados"(3) em direção aos inimigos. Um inimigo morre quando é atingido por um projétil que foi criado premindo na tecla que lhe corresponde. Se um projétil atinge um inimigo com uma letra diferente da letra que o originou, o projetil é destruido e nenhum inimigo morre. À medida que o tempo decorre, a dificuldade do jogo aumentará, fazendo com que mais inimigos apareçam, com carateres mais diversificados e mais velozes.

Figura 2: 1. Nave inimiga, 2. Base do jogador, 3. Projétil, 4. Nave de powerup, 5. Vida do jogador, 6. Indicador de powerup, 7. Indicador de Caps Lock, 8. Estatísticas de jogo

Para auxiliar o jogador, existem naves especiais(4) que precisam de varios projeteis para morrerem, mas que dão ao jogador *powerups*(6), isto é, habilidades especiais que podem ser ativadas através do uso do rato:

- **Freeze:** diminui a velocidade das naves visiveis (é ativado desenhando uma linha vertical com o rato enquanto se prime o botão esquerdo do rato);

- **Splitshot:** quando um inimigo é destruido enquanto o splitshot está ativo, vários projeteis são disparados em todas as posições a partir dessa posição (estes projeteis podem eliminar qualquer nave, independentemente da sua letra e são ativados desenhando uma linha vertical com o rato enquanto se prime o botão direito do rato).

Ao longo do jogo o utilizador pode ver as suas estatísticas no canto superior direito do ecrã(8):

- **Score:** pontuação do jogador. A pontuação é obtida passivamente ao longo do jogo, destruindo inimigos e obtendo powerups.
- **CPM:** *characters per minute,* mede a rapidez com que o utilizador escrever;
- **Acc:** *accuracy,* mede a precisão do utilizador a escrever.

A qualquer momento durante o jogo, o utilizador pode premir a tecla '*escape*' que lhe permitirá pausar o jogo, onde, por trás de um *pop up* poderá ver o estado atual do jogo. No ecrã de pausa o jogador pode escolher abandonar o jogo ou retomar, clicando no botão respetivo ou pressionando a tecla escape novamente (no caso de retomar o jogo).

Quando um inimigo embate no jogador ou quando uma nave inimiga começa a penetrar a atmosfera terrestre, o jogador perde vida. Se esta chegar ao fim, o jogador perde o jogo. No ecrã de *Game Over* são mostradas ao jogador as suas estatísticas finais e caso se trate te um novo *Highscore* é pedido ao utilizador que insira o seu nome e o highscore é guardado.

Figura 3: Novo highscore

1.2 *Help a friend*

No ecrã “Help a friend” um jogador conecta-se a outro jogador que esteja a jogar e com quem tem de estar ligado pela porta série. Aí, o jogador que clicou no botão deve cooperar com o jogador “host” para derrotar as naves, usando o seu teclado para as destruir.

O segundo jogador não tem acesso a *powerups* e apenas vê no seu ecrã as letras das naves inimigas que se encontram vivas, deve por isso orientar-se pelo ecrã do seu parceiro.

Figura 4: Interface do jogador secundário

2.2 *Highscores*

Neste ecrã podem ser observadas as cinco melhores pontuações obtidas no jogo. Sendo que estão ordenadas por score obtido. Os valores dos highscores são mantidos em múltiplas sessões de jogo, sendo para isso também guardadas a hora e a data da jogada.

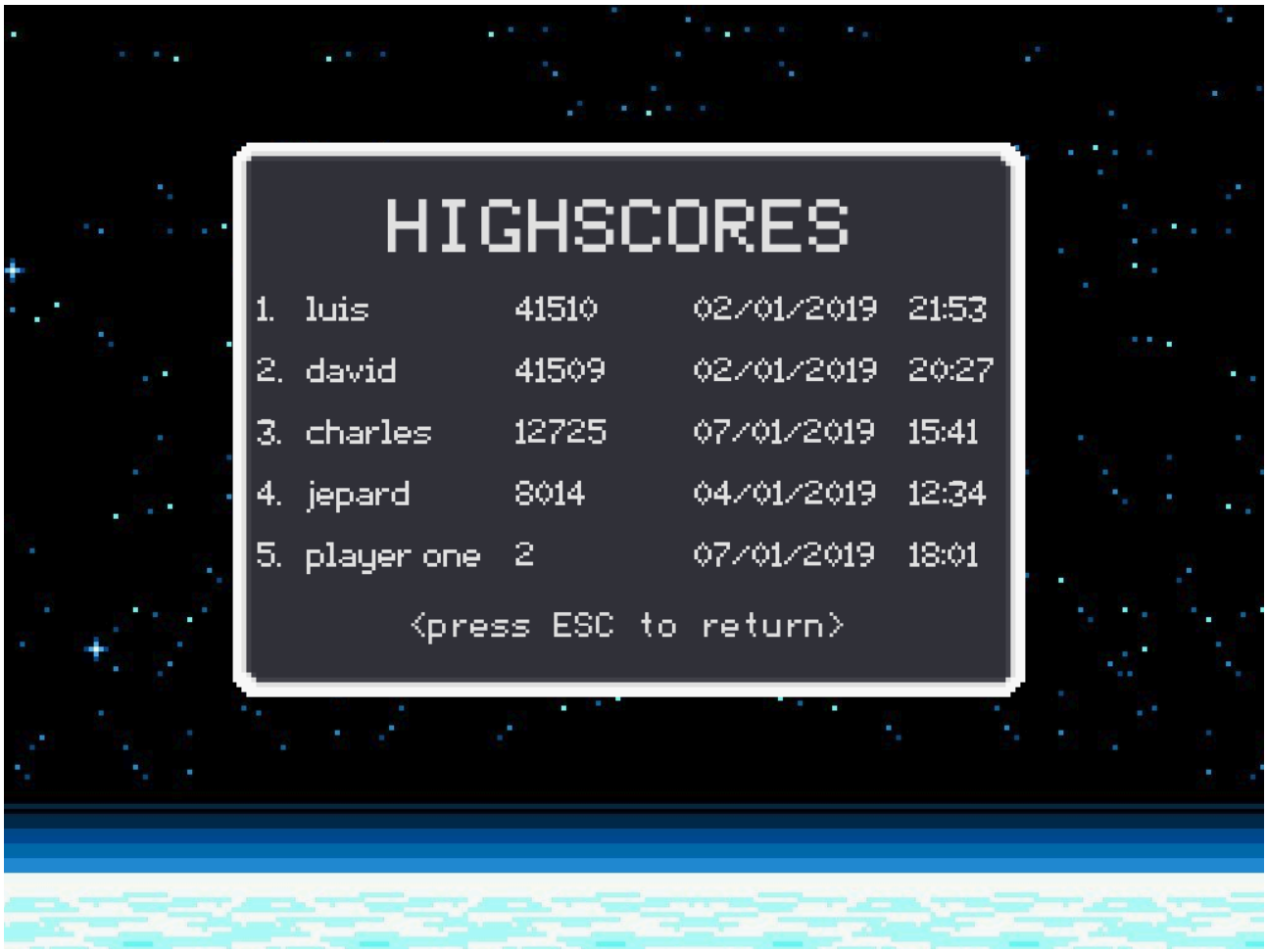


Figura 5: Consulta dos highscores

2. Estado do projeto e uso de dispositivos

Todas as funcionalidades que estão visíveis na interface do utilizador foram implementadas e estão acessíveis.
O uso dos periféricos foi o ilustrado na seguinte tabela:

Dispositivo	Uso no projeto	Interrupções
Timer	Controlo do framerate e atualização do jogo	S
Teclado	*Input* do jogador (projéteis do jogo, texto e menu)	S
Rato	Navegação de menus , ativação de powerups em jogo	S
Video card	*Display* de gráficos do jogo e menus	N
RTC	Alarmes para eventos de jogo, controlo de tempo	S
UART	Jogo multijogador	S

Timer

O timer é usado para controlo de framerate e para ditar a cadência de updates do jogo. O principal uso do periférico dá-se na função proj_main_loop, mais concretamente no "ciclo das interrupções" para controlar quando é que é executado o update do frame que é mostrado no ecrã. Indiretamente este periférico afeta também a state-machine que dita o fluxo do programa e também o motor do jogo, pois é pelo ritmo ditado pelo periférico que estes sub-sistemas evoluem. Quando há uma interrupção de timer, tendo em conta o *frame rate*, é executada a função "update_game_state()" que por sua vez invoca funções de desenho que efetuam os cálculos necessários para o *update* do jogo e desenho do *frame* apresentado. A implementação da interação com o timer está nos ficheiros timer.h e timer.c.

Teclado

O teclado é uma peça fulcral do programa. Através do teclado é feito o input que é usado para a destruição das naves, para o *input* de texto e para alguma navegação nos menus (uso da tecla escape). Interrupções sinalizam o pressionar de uma tecla, estas são detetadas no "*interrupt*" loop e o scancode obtido pelo *interrupt handler* do teclado é enviado para a state machine do programa, que o processa através da função "makecode_to_char()". Uma vez obtido o caracter correspondente à tecla premida, este *input* é usado para procurar alterações tanto nos menus (no caso de input de texto) como no jogo em si, para criar projéteis.

Mouse

O rato é essencialmente usado para a navegação de menus. Os pacotes provenientes do KBC (obtidos pelo mouse_ih(), interrupt handler do rato) que são referentes ao rato são processados no interrupt loop pela função ms_process_packet(), sendo transformados em eventos de rato que são depois processados pela state machine, que leva à atualização da posição do cursor e/ou ao *push* de botões, bem como a deteção de gestos (função check_vert_line_gest()) usados no uso de *powerups* no jogo *singleplayer*. Em ambas as situações indicadas são usadas tanto a funcionalidade de posição do rato como a dos seus botões.

Video card

O *videocard* é usado para efetuar o *display* do jogo. No programa, é usado o modo 0x116 do VBE, modo de cores direto, que usa 16 bits (um para transparência) e que suporta uma resolução de ecrã de 1024x768. A *vídeo memory* é inicializada através da função "vg_init()", e a sua manipulação é feita através de funções declaradas no ficheiro videocard.h.

Dado a limitações da implementação do VBE oferecida pela Virtual Box, é implementado um "pseudo" *double buffering*, na medida em que é toda a informação referente ao frame a ser desenhado é desenhada num buffer auxiliar (denominado de scene) que depois é copiado para a video memory em si. isto é feito para impedir o aparecimento de *flickering* do ecrã. Esta técnica pode ser vista nas funções "draw_xpm" e "draw_scene", na medida em que a função "draw_xpm"(principal função usada para escrever no ecrã) escreve no buffer "scene" que é depois copiado inteiramente para a *video memory*, podendo aí, ser visualizado no ecrã. A função "draw_scene" é sempre a última a ser executada quando se pretende mostrar algo no ecrã.

Para as imagens que efetivamente são desenhadas, foi dado recurso ao formato XPM, dado à sua simplicidade e à eficiência a ela inerente. Os ficheiros xpm usados no programa estão localizados no ficheiro xpm_db.h e são convertidos para pixmap's (arrays de pixéi cujos valores são escritos na memória vídeo).

O movimento de sprites é conseguido pela repetida sequência de desenhar todo o frame de novo cada ciclo, desenhando todos os sprites na sua nova posição. Dado ao elevado tráfego de sprites inerente ao jogo em si, esta opção é , geralmente, mais eficiente do que alterar apenas as partes do novo frame onde ocorreu uma mudança. Um exemplo disto encontra-se na função "draw_game_play()" do ficheiro game.h, é possível ver que para efetuar o desenho , a cada frame, é desenhado o novo *frame* de "raiz".

O programa conta com o uso de vários "tipos" de *sprites*. Existem sprites estáticos, que podem ser observados nos objetos "Button", e sprites dinâmicos. Os *sprites* dinâmicos por suas vezes podem ser animados (caso dos projéteis e das explosões) e em sprites não-animados (caso dos inimigos e cursores.

A colisão de sprites está implementada no ficheiro "Sprite.h" (função sprite_checkColision). A estratégia usada passa por primeiro detetar colisões entre os retângulos representam os xpm's dos *sprites*. Caso seja detetada uma

interseção entre estes dois retângulos, será feita um estudo mais exaustivo da colisão, verificando o sub-retângulo da interseção dos dois xpm's pixel a pixel, de modo a verificar se existe algum pixel que ambos os sprites pretendem ocupar. Isto é feito para aumentar a eficiência, na medida em que não é necessário verificar uma colisão pixel a pixel para sprites que não estão próximos.

A escrita de texto no ecrã é feita com recurso à função drawWord() (ficheiro wordWriter). A escrita é feita aplicando a função draw_xpm() para cada letra de uma palavra. Os xpm's dos carateres utilizados estão presentes no ficheiro font_xpm.h, sendo guardados num array que posteriormente é convertido em pixmaps. Esta operação é executada no início do programa para evitar que sejam feitas conversões durante o decorrer do programa, operações que são pouco eficientes.

RTC

O *real time clock* é usado ao longo do programa para o controlo de tempo, uso de alarmes e obtenção de tempo e data atuais. Mais concretamente, as funcionalidades de tempo(através de update interrups), são usadas para o cálculo do tempo de jogo o que posteriormente é usado no cálculo de estatísticas como os chars per minute (funções sec_passed_handler do ficheiro "GameStateMachine" e game_calculate_cpm do ficheiro "game"). As funcionalidades de alarme são utilizadas nos *powerups* do jogo, sendo que a sua duração é ditada por um alarme do RTC. A instituição do alarme é feita através da função rtc_set_alarm (usada, por exemplo, na função "right_p_up_handler" do ficheiro GameStateMachine) e os seus efeitos são propagados para o jogo através da função "alarm_ring_handler" do ficheiro GameStateMachine. Finalmente, a leitura do tempo e data é usada para marcar os highscores obtidos pelos jogadores (função "rtc_get_currentTime", declarada no ficheiro "rtc.h" e usada no ficheiro "highscores.c" na função "insert_new_highscore).

UART

A porta série é utilizada no projeto para comunicar entre computadores de modo a fazer o modo multijogador. O modo como utilizamos a porta série consiste em usar a técnica de *polling* na transmissão e interrupções, com a porta série configurada para usar 8 *bits* por carater, 2 *stop bits*, 9600 de *bit-rate* e paridade *odd*. Utilizamos a porta série para transmitir entre jogadores quando um entra ou sai do jogo, bem como os novos caracteres a adicionar como projéteis ao jogo prinicipal ou como inimigos a adicionar ou remover ao jogo secundário. São enviadas mensagens sempre que o jogador secundário introduzir um novo caracter, ou sempre que uma nave aparece ou desaparece no jogo principal.

Apesar da funcionalidade implementada com a porta série funcionar nos casos em que temos testado, reconhecemos que a implementação dos módulos referentes à porta série é débil e pode ser causa de falhas no nosso progama, pois algumas situações de erro ficaram por verificar.

Estrutura e organização do código

A estrutura do código encontra-se dividida em três secções essenciais. A primeira é interação com os periféricos, a qual tem como função estabelecer o contacto do programa com os periféricos utilizados e extrair deles a informação relevante para a execução do programa. Na primeira secção estão incluídas todas as funções dos ficheiros referentes aos periféricos (keyboard.h, mouse.h, rtc.h e timer.c) bem como a secção "main" do programa onde é executado o loop que contém o "driver_receive" e onde são executadas as funções de inicialização da gestão dos periféricos.

A segunda secção é constituída por uma máquina de estados que efetua a "ponte" entre a informação que é obtida dos periféricos e o motor de jogo. A máquina de estados é implementada nos ficheiros GameStateMachine.

A terceira grande secção de código compreende o motor de jogo e todas as funções e estruturas a ele auxiliares. Nesta secção são efetuados os cálculos necessários para o funcionamento de jogo: movimento e colisão de sprites, desenho no ecrã, etc..

Segue-se uma descrição mais detalhada dos módulos usados no projeto:

"proj"

O módulo "proj" contém a função *main* (gráfico de chamada de função na página seguinte) e é onde se localiza a grande maioria da gestão dos periféricos. Nele, são efetuadas as operações de carregamento dos assets do jogo, de "*setup*" dos periféricos, da subscrição de interrupções e posteriormente do inverso das mesmas operações (libertação de memória, dessubscrição, e reposição do estado dos periféricos. É também neste módulo que se encontra o ciclo das interrupções onde são executados os *handlers* que são usados para obter a informação dos periféricos necessárias à execução do jogo.

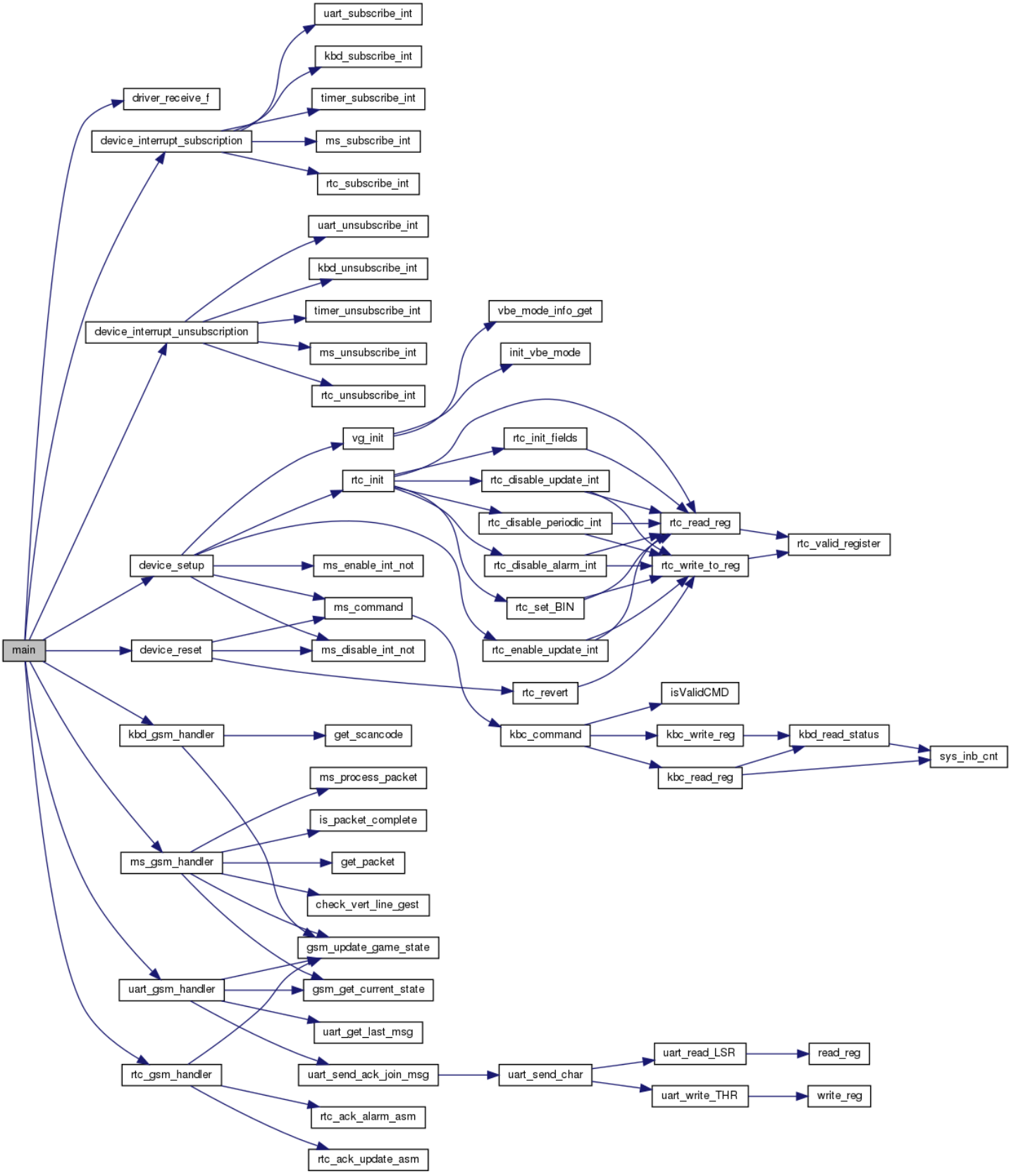
"GameStateMachine"

O módulo "GameStateMachine" representa uma máquina de estados que permite não só manter o estado atual do jogo, como expeditamente tratar de novos eventos consoante o estado atual do jogo. Neste módulo estão implementadas *enums* que contêm os eventos da máquina de estados, bem como os estados possíveis.

Este módulo revelou-se de eleva importância no desenvolvimento do projeto, pois devido à sua modularidade permitia adicionar com facilidade novos eventos, bem como *handlers* para os novos eventos, o que levou a um desenvolvimento incremental de novas funcionalidades do jogo.

Módulo desenvolvido pelo membro Luís Cunha.

Gráfico de chamada de funções pela função *main*:



“game”

O modulo Game, é um dos módulos mais importantes presentes no projeto. Este módulo dá uso de quase todos os outros para executar os cálculos e operações necessárias para executar o jogo. Existem várias estruturas declaradas neste ficheiro. A principal delas, estrutura Game, contém os “*asset”s*, isto é, as informações que são necessárias ao longo do jogo. Mais especificamente, estas informações incluem flags que sinalizam o estado de alguns mecanismos do jogo (*powerups*, aparecimento de inimigos etc.), *pixmaps* usados em objetos do jogo, contadores que guardam estatísticas e outras variáveis de jogo etc. Neste módulo também são declaradas estruturas auxiliares como a estrutura Bounds que representa um intervalo numérico, a estrutura SpawnSplotManager que guarda informações que são usadas no jogo para controlar o aparecimento de inimigos de modo a evitar sobreposições, e enumeráveis que representam conceitos existentes, como por exemplo, os powerups ativos. Possui também vários arrays que guardam informação acerca dos inimigos, projeteis, botões, barra de estatísticas, caixas de texto etc.

As funções presentes neste módulo controlam vários aspetos do jogo como o desenho no ecrã (funções “draw”), adição(funções “game_add”) e remoção (funções “game_remove”) de objetos, funções de verificação (funções “game_check”), carregamento (funções “game_load”) entre outras de função variada.

Módulo desenvolvido com contribuição dos dois membros, com maior preponderância do membro David Luís.

“game-macro”

Neste modulo estão contidas as constantes numéricas simbólicas que são usadas no módulo “Game”. Estas constantes incluem várias categorias que permitem a parametrização do código deste módulo. Permitem o controlo das variáveis que definem o ritmo do jogo (tempo por nível) e de controlo da dificuldade (velocidade e cadência dos inimigos). Permite também controlar outros aspetos como a posição de botões e outros sprites, o número máximo de inimigos, o framerate, o tempo entre frames no caso da animação do sprite de projeteis, etc.

“Sprite”

O modulo Sprite contem a implementação da estrutura Sprite, estrutura que serve de base para outras usadas no programa (Enemy, AnimatedSprite, Projectile). Esta estrutura representa um objeto que é desenhado no ecrã e guarda informações sobre a sua posição(canto superior esquerdo do pixmap) e velocidade usando para isso recurso a estruturas Vector (ver DLMath). Para além disso a estrutura Sprite guarda a informação sobre os gráficos que são mostrados no ecrã quanto este é desenhado (xpm_image_t). Este módulo contém as funções necessárias para usar o Sprite: criar, apagar, desenhar, alterar velocidade, verificar colisões, alterar gráficos etc.

Módulo desenvolvido pelo membro David Luís.

“Enemy“

O modulo Enemy implementa estruturas Enemy que podem ser consideradas como estruturas “derivadas” da classe Sprite. De facto, a classe Enemy constitui uma especialização de um Sprite guardando para isso não só um objeto Sprite, mas também outros dados importantes para a realização do jogo. Um Enemy possui também um tipo, um atributo que guarda o número de embates que precisa para “morrer” e uma caixa de texto que guarda o carater que está associado a esse inimigo. Assim, este módulo implementa as funções necessárias para a manipulação de inimigos, muitas das quais, inerente à “derivação” de Sprite constituem apenas chamadas ao Sprite associado. O módulo implementa também a estrutura “Textbox” que guarda informações acerca do caracter (mas possibilita strings) associados ao inimigo.

Módulo desenvolvido pelo membro David Luís.

“AnimatedSprite”

Este módulo apresenta a extensão da estrutura Sprite denominada de Animated Sprite. Um AnimatedSprite constitui um Sprite que possui uma animação da sua representação (a imagem desenhada no ecrã varia ao longo do tempo). Para isso é necessário guardar informação acerca de todas as imagens representadas bem como o tempo que é necessário gastar entre cada mudança de frame da animação. A implementação dos AnimatedSprite pode ser vista nos projeteis (ver objeto Projectile) e nas explosões que surgem quando um inimigo é destruído. Mais uma vez, dada à derivação que obtém por conter um objeto Sprite, a implementação das funções de manipulação de Animated Sprites presentes neste módulo é bastante simples.

Módulo desenvolvido pelo membro David Luís.

“Projectile”

O módulo Projectile contém a implementação de uma estrutura denominada Sprite, que é uma extensão de um AnimatedSprite e, por consequente, de um Sprite. A estrutura Projectile é usada para implementar os projeteis que são criados pelo jogador e que destroem os inimigos. Estes projeteis são “teleguiados” e, para o efeito, guardam informação sobre o inimigo que estão a perseguir, a letra que os criou entre outros. Uma outra particularidade inerente a esta “perseguição” é a de que, ao contrário dos outros derivados de Sprite, a velocidade do Projectile é, geralmente, calculada a cada update, sendo que é desenhado um vector desde o projétil até ao inimigo que persegue. Várias funções do módulo Projétil dão uso às funções do objeto AnimatedSprite que contém.

Módulo desenvolvido pelo membro David Luís.

“Button”

Neste módulo encontra-se implementada a *struct* “Button” e funções que permitem manipular “objetos” deste tipo, de uma forma aproximada à maneira como seria feita num paradigma de orientação a objetos. A *struct* contém a posição, x e y, onde deverá ser desenhado o canto superior esquerdo, bem como a imagem que deverá ser apresentada no ecrã. Este módulo permite a fácil criação e manipulação de botões. Uma função importante presente neste módulo é a função button_pressed_in_range, que indica se uma determinada posição está contida no espaço ocupado pela imagem do botão, e é usada para determinar, após um clique esquerdo do rato, se a posição do rato pertence ao botão.

Módulo desenvolvido pelo membro Luís Cunha.

“Cursor”

Neste módulo, de forma idêntica ao módulo Button, encontra-se implementada a *struct* “Cursor”. Esta *struct* tem como objetivo representar o cursor do rato, guardando para isso a posição atual onde se encontra o seu canto superior esquerdo bem como a imagem que o deve representar. Uma função importante deste módulo é a função cursor_update_pos, que é utilizada após a leitura de um *packet* do rato para atualizar a posição do cursor, de modo a que no próximo *frame* ele seja apresentado na posição correta.

Módulo desenvolvido pelo membro Luís Cunha.

“Timer”

O módulo “Timer” implementa as funções necessárias para usar o timer do Minix. Nele estão contidas as funções para subscrever interrupções e para as tratar, incrementando uma variável global ao módulo. Com as funções é possível alterar a frequência do timer e obter as suas configurações. No programa, o timer é usado, sobretudo, para ditar a cadência de updates do jogo e, por cosequente, do ecrã, sendo para isso usadas as interrupções provenientes do “Timer 0”.

Módulo desenvolvido pelos dois membros durante as aulas do lab 2.

“i8254”

Este módulo contém as constantes numéricas simbólcias usadas na programação do timer. É geralmente usada no módulo “Timer” e contém constantes que representam endereços de registod só timer bem como “bit masks” que são usadas para manipulação de valores dos mesmos registos.

Módulo desenvolvido pelos dois membros durante as aulas do lab 2.

“keyboard”

Este módulo tem como objetivo efetuar a conecção entre o teclado e o programa. Neste estão contidas as operações necessárias para obter scancodes provenientes do teclado e os processar de uma forma legível para o utilizador. O módulo contém também as funções necessárias para manipular o KBC e para efetuar subscrições a interrupções provenientes do teclado. Devido a possuir funções que manipulam o KBC, o módulo do teclado também é usado pelo módulo “Mouse” que partilha o KBC com este. As principais estruturas presentes neste módulo são a estrutura “kb_key” que guarda os carateres que são correspondentes aos makecodes de uma tecla do teclado e a estrutura “keys” que guarda um array “kb_keys” representando as teclas do teclado. É possível detetar o uso da tecla “Shift” bem como o da tecla Caps Lock.

Módulo desenvolvido pelos dois membros durante as aulas do lab 3. Funções não pertencentes ao lab 3, desenvolvidas pelo membro Luís Cunha.

“mouse”

Este módulo tem como objetivo gerir o rato, fazendo a correta leitura da informação por ele enviada, interpretando os *bytes* lidos e preenchendo a *struct Ppacket,* de modo a integrar no programa com facilidade o movimento do rato e o estado dos seus botões.

Neste módulo, faz-se ainda uso de uma máquina de estados para detetar eventos significativos do rato, bem como os gestos necessários à ativação dos *power ups* do jogo.

Módulo desenvolvido pelos dois membros durante as aulas do lab 3. Funções não pertencentes ao lab 4, desenvolvidas pelo membro Luís Cunha.

“i8042”

Este módulo contém as constantes numéricas simbólcias usadas na programação do KBC. O KBC é usado para o controlo da informação proveniente do rato e do teclado. Assim, este módulo contém macros que permitem idenfitcar certos eventos relacionados com o teclado e com o rato (scancode da tecla “escapa” por exemplo), bem como constantes que são usadas para representar comandos do KBC, endereços de registos e “bit masks” que são usadas para a manipulação de valores dos registos.

“vbe-macro”

Neste módulo surgem as constantes numéricas simbólicas que são usadas no módulo “Videocard”. Entre elas estão macros que representam funções de VBE, módulos gráficos, entre outros.

“videocard-util”

Este módulo é simplesmente usado para implementar funções auxiliares ao módulo “Videocard”. As funções são , geralmente, usadas nas funções que implementam funções VBE. Entre elas estão funções que convertem “far pointers” to “near pointers”, aplicam mascaras de bits e extraem as palavras mais ou menos significantes de um numero de 32 bits.

Módulo desenvolvido pelos dois membros durante as aulas do lab 5.

“videocard”

O módulo videocard contém as funções usadas para a manipulação do video buffer. Nele estão também as implementações de funções VBE (00 – "vbe_controller_info_get”, 01- “vbe_mode_info_get”, 02- “init_vbe_mode”), que são usadas para inciar a video memory e para obter informações acerca da mesma. No entanto, as principais funções deste módulo são as funções “draw” (“draw_xpm”, ”draw_background” e “draw_scene”) que são usadas para o desenho de xpm’s ou de “fundos” no buffer da memória de video. È também através do draw_scene que é obtido o efeito de double buffering, como explicado na secção “Graphics Card”. A função “draw_xpm” é usada por muitos dos módulos para desenhar o xpm dado no buffer “scene”, sendo que esta função funciona para todos os modos de video do VBE e permite o desenho de imagens de forma parcial (que ultrapassam os limites do ecrã).

Módulo desenvolvido pelos dois membros durante as aulas do lab 5.

“rtc”

Tal como o nome indica, este módulo destina-se a controlar o dispositivo RTC. Aqui estão todas as funções necessárias para aceder à hora e data local do computador. O módulo permite também establecer um menacismo de interrupções entre o programa principal e o perifério, estando implementados protocolos que lidam com interrupções de “update” e interrupções de alarme. As interrupções de “update” são usadas pelo programa para atualizar o relógio e jogo. Já os alarmes são usados para marcar o fim do tempo de uso do powerup “splitshot”. Este módulo centra-se à volta da estrutura “RTCFields” que guarda o valor dos registos de data e tempo do RTC.

Módulo desenvolvido pelo membro David Luís.

“rtc_ih_asm”

Este modulo implementa, em assembly, o interrupt handler do Real Time Clock (documentado no ficheiro RTC.h). Para efeito foi usada a sintaxe da arquitetura Intelx86 dado ter-nos sido lecionado na UC de MPCP. Apesar disso, foi notada alguma dificuldade na implementação, dado que existem, ainda assim, diferenças entre essas sintaxes (por exemplo na invocação de sub-rotinas).

Módulo desenvolvido pelo membro David Luís.

“rtc-macro”

Este módulo contém as constantes numéricas usadas na implementação do uso do Real Time Clock. Estão documentadas constantes que representam os endereços, absolutos ou relativos, de registos do RTC bem como mascaras binárias para manipulação de alguns desses registos.

“serialport”

Este módulo tem como objetivo manipular a porta série e contém a implementação de funções que permitem ler e alterar a configuração da porta série e também de enviar caracteres (utilizando *polling*) e receber caracteres (utilizando interrupções), permitindo assim a comunicação entre dois computadores.

Este módulo contém ainda o protocolo por nós usado para permitir a comunicação entre dois utilizadores do nosso jogo. Para facilitar a identificação de mensagens entre os dois computadores, dividimos as mensagens em tipos (indicados na secção detalhes de implementação), sendo que cada uma tem um *header* identificativo diferente, e todas são terminadas por um *trailer*. Há ainda mensagens que contêm um corpo, constituído apenas por um caracter.

Módulo projetado pelos dois membros e desenvolvido pelo membro Luís Cunha.

“uart”

Neste módulo encontram-se constantes simbólicas utilizadas nas funções que lidam com a porta série.

“debug”

Este módulo implementa apenas duas funções que foram usadas ao longo do desenvolvimento para efetuar o “debug” do código. O ficheiro inclui uma instrução de pre-processador “#define DEBUG” que quando comentada “invalida” todas as invocações presentes no código das funções definidas no ficheiro.

“dlmath”

O modulo “dlmath” contem as funções e estruturas “matemáticas” do projeto. Contem funções que permitem obter números aleatórios, operação importante no jogo *singleplayer* (usado no “spawn” de naves, nos *powerups* etc.), carregar tabelas trigonométricas, obter o máximo divisor comum, etc. A componente mais importante neste módulo encontra-se na estrutura “Vector” que representa um vetor 2D . O módulo contem funções que permitem cálculos vetoriais básicos como a soma, a norma e a multiplicação por fatores. Este cálculo permite uma maior facilidade na representação de conceitos como posição e velocidade, que por sua vez ajudam na implementação de mecanismos como a deteção de colisões entre sprites. Devido a problemas

com a função de geração de números aleatórios do C, implementamos a nossa própria versão da mesma com recurso a mecanismo “Xorshift”.

Módulo desenvolvido pelo membro David Luís.

“linked_list”

O módulo “Linked List” implementa uma lista simplesmente ligada de caracteres. A lista é usada no jogo singleplayer para guardar os carateres aleatórios que são usados nas naves que são colocadas em jogo. Foi usada uma pois possui uma fácil implementação e pela sua baixa complexidade na inserção, especialmente no início da lista. As estruturas criadas para o propósito são a “CharLinkedList” e a “CharLinkedListNode”. Esta última é constituída por um elemento do tipo “char” e por um apontador para outro CharLinkedListNode. Assim sendo, uma lista ligada de caracteres é conseguida através da ligação de sucessivos nós, tendo o último um apontador nulo. A estrutura “CharLinkedList” contem um apontador para um Node que é o cabeçalho, elemento que é abstraído do utilizador dado não conter qualquer informação relevante e apenas servir que a lista fique completamente vazia. Esta estrutura contem também informação acerca do tamanho da lista. Foram implementados métodos para inserir, apagar, fazer print (para debug) , esvaziar e fazer “pop”. Esta última é usada para dar à lista uma característica de “pseudo-fila” sendo que o seu prepósito é retirar o primeiro caracter de uma lista. As filas usadas no jogo, operam da seguinte forma: é criada uma lista ligada de forma aleatória com os caracteres pretendidos (isto é conseguido por sucessivas inserções em posições aleatórias), ao longo do jogo é feito o pop da cabeça da lista para obter o caracter de um novo inimigo sendo que quando este inimigo morre o caracter que este representa é reinserido na lista numa posição aleatória.

Módulo desenvolvido pelo membro David Luís.

“highscores”

O módulo “Highscores” tem como objetivo guardar uma lista das melhores pontuações obtidas pelo jogador num ficheiro de texto, para consulta em utilizações posteriores do programa. Neste módulo está implementada a *struct* “Highscore”, que guarda a pontuação, data e hora do fim de jogo e nome de utilizador relativas ao *highscore*. As pontuações são guardadas num ficheiro de texto, uma por linha, no formato “username@pontuação ano mês dia hora minutos segundos”. O programa permite apenas que se guarde um número *highscores* menor que um determinado valor fixo, que está indicado na constante simbólica NUMBER_HIGHSCORES, e o nome do utilizador não pode ser maior que o valor indicado na constante simbólica USERNAME_MAX_SIZE.

Este módulo é utilizado quando os dados de jogo são carregados, ao iniciar o programa, colocando os *highscores* num *array* pertencente à *struct* “game”. Sempre que um novo *highscore* é alcançado, este é introduzido na posição certa do *array* e o ficheiro de texto é atualizado, o que permite que os *highscores* sejam guardados mesmo que o programa termine de forma não convencional.

Módulo desenvolvido pelo membro Luís Cunha.

“wordWriter”

O módulo “WordWriter” é utilizado para escrever palavras no ecrã enquanto este se encontra em modo gráfico. Para utilizar o módulo, é necessário executar primeiro a função “loadAlphabet” que carrega os xpm’s dos carateres (guardados individualmente e agrupados num array no ficheiro “font_xpm”). A principal função deste módulo é a função “drawWord” que, como o nome indica, desenha a palavra dada como argumento no ecrã. Para desenhar a função aplica a função “draw_xpm” (ver módulo “videocard”) sucessivamente para cada letra. Para tal, é importante que os xpm’s guardados para os carateres sejam guardados com um formato uniforme (com a mesma altura, em pixeis, para todos os carateres). A principal estrutura deste módulo não está acessível ao utilizado e guarda os pixmaps dos carateres.

Membro desenvolvido pelo membro David Luís.

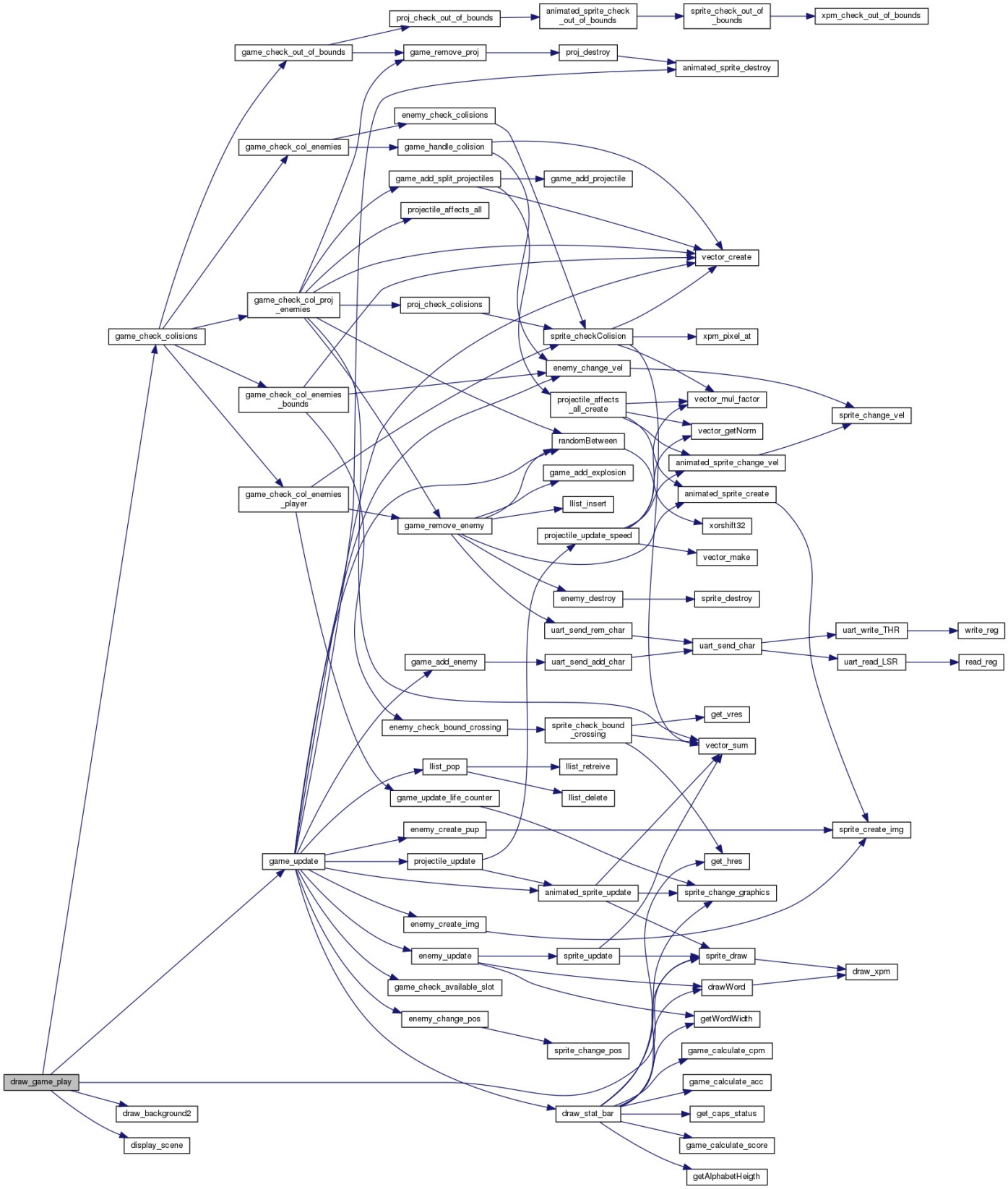
“font_xpm”

Este modulo contem os xpm’s usados para as letras usadas no projeto. Neste caso, foi usada a font “M7SingleH”, sendo que cada carater valido se encontra num array (formato xpm) separado. No final do ficheiro, existe um array em que são guardados todos os xpm’s, na ordem correta (ordem de aparição na tabela ASCII) para posteriormente serem lidos.

“xpm_db”

Este módulo serve apenas para guardar as imagens que são usadas ao longo do projeto para fundos, botões, menus, projeteis, inimigos etc. As imagens são guardadas em formato xpm. O módulo contém também os array’s de xpm’s que são usados para agrupar os frames que são usados nos AnimatedSprite’s.

Gráfico de chamadas de função da função draw_game_play:



4. Detalhes de implementação

Como foi referido anteriormente, foi feita uma divisão do projeto em etapas, para facilitar não só a implementação progressiva, bem como a modularidade do mesmo.

Em geral, o mecanismo é o seguinte: o segmento do projeto que gere os periféricos e que contém o ciclo das interrupções, obtém destes a informação que é necessária para o update do jogo. Aí, trata de enviar estas informações (carateres correspondentes a makecodes, packets de rato, interrupções do timer, alarmes do RTC, etc.) para a statemachine que controla o jogo. A statemachine, segunda etapa do projeto, processa as informações recebidas e executa os handlers corretos para que seja tomada a ação correta no jogo. Finalmente, é procedida a etapa de cálculos, que usa as informações que recebe da state machine e efetua os cálculos necessários para a execução do jogo(update do frame, cálculo de colisões, criação de inimigos e/ou projeteis, etc.).

A primeira etapa, que gere os periféricos, inclui uma sub-etapa que gere a inicialização do programa, onde são efetuadas as subscrições às interrupções, inicialização dos periférios e funções como a “loadAphabet” a “load_game_data” que carregam os *assets* necessários à execução do jogo.

Grande parte da implementação da gestão dos periféricos usa as funções que foram desenvolvidas ao longo dos “labs”, dado a terem sido pensadas com primazia à modularidade. No entanto, foram feitos alguns ajustes e correções para melhor adaptar as funções às nossas necessidades, bem como para a correção de erros.

Desta primeira etapa, faz ainda parte a função *driver receive* e o ciclo de interrupções, bem como os *interrupt handlers*, desenvolvidos com o objetivo de serem independentes da aplicação, e as funções que depois desses mesmos *interrupt handlers* geram os eventos relevantes para o jogo, usados para atualizar a máquina de estados.

A parte que faz a ligação entre os periféricos e a deteção dos eventos por eles gerados e o jogo que é apresentado ao utilizador, consiste na máquina de estados implementada no módulo “GameStateMachine”, que, fazendo uso de uma máquina de estados e de programação orientada por eventos, serve como *dispatcher,* utilizando apontadores para funções, tendo como principal tarefa interpretar os eventos levantados na primeira etapa e executar as funções corretas, tanto nos estados em que o jogo está a decorrer, como nos estados correspondentes aos menus.

A última grande camada do jogo é a camada que efetua os cálculos necessários para a execução do jogo como o cálculo de posições, velocidades, colisões, estatísticas, entre outros, e que efetua o desenho dos elementos no buffer scene.

Nesta parte, está patente um “design” de código orientado a objetos, na medida em que foram construidas classes para representar conceitos do jogo e foi usada derivação entre elas para simplificar o codigo. O exemplo deste design são as classes “Sprite”, “Enemy”, “AnimatedSprite” e “Projectile”. A base destas classes é a classe “Sprite” que contém os elementos básicos para o desejo de uma imagem em movimento no ecrã e cujo módulo implementa as funções necessárias a manipular os mesmos. A classe “Sprite” especializa-se

nas restantes, sendo que a especialização mais “básica” é a da classe “Enemy”, que adiciona membros que representam a “Textbox” que contém, o número de embates que precisa para ser destruida e o seu tipo. A classe AnimatedSprite é também contruida tendo a classe Sprite por base, conferindo-lhe apenas as operações e informação necessárias para que o frame que é apresentado no ecrã, aquando do desenho do mesmo, mude ao longo do tempo, resultando no efeito de um Sprite animado. A classe Projetile constitui a derivação mais complexa de um “Sprite”, pois é uma classe que é construida com uso a um objeto AnimatedSprite.

Na camada que faz os cálculos, estão também incluidas toda a criação e desenho dos frames. Isto é realizado em três passos. O primeiro passo efetua os cálculos das colisões entre os vários objetos do jogo de modo a identificar, entre outros, as suas novas velocidades, se devem ou não ser destruidos ou se o jogador deve perder vida. De seguida, é feito o update do jogo, onde é feito o cálculo da criação de novos inimigos e das novas posições dos objetos, sendo que no final da etapa, todos os objetos são desenhados no scene buffer. O terceiro e último passo aplica a cópia do scene buffer para a memória vídeo (através de memcpy), método que aplica o “double buffering”. È de notar que no projeto é utilizado um frame-rate de 60 frames por segundo, isto é, é executado um update ao jogo 60 vezes por segundo.

Como referido em cima, uma parte importante dos cálculos a efetuar para a execução do jogo é o cálculo de colisões. Este é efetuado com recurso à implementação da estrutura Vector (representativa de um vetor 2D), para simplificar os a compreensão e posterior implementação do conceito. Para calcular a colisão entre dois Sprites é calculada a sua nova posição somando, para cada um, o seu vetor posição e velocidade. Nas novas posições, é verfiicado se existe uma interseção entre os retângulos que representam os gráficos de cada Sprite. Caso não seja o caso, a colisão não acontece, o que permite prevenir o programa de fazer cálculos mais complexos desnecessáriamente. Caso haja interseção, é analizada a interseção entre os dois gráficos para averiguar se existe um pixel da memória onde ambos os sprites pretendem desenhar um pixel sólido (não transparente), em caso afirmativo, ocorreu colisão.

Quando a colisão ocorre é executado um handler que decide que é o caminho a tomar pelos Sprites de modo a que essa colisão não aconteça no update seguinte.

O RTC foi usado para controlar o tempo (dado a possuir uma funcionalidade mais adaptada que o timer para o controlo de segundos), para verificar a hora e data atuais e para criar alarmes, usados para a gestão de powerups. O update da estrutura RTCFields é feita de forma a apenas atualizar a data quando o RTC é inicializado pelo programa ou quando o dia muda (00:00), isto previne que durante as atualizações que são feitas ao longo do dia sejam feitas atualizações desnecessárias à data, no entanto, o utilizador pode escolher se quer forçar a atualização da data. Aquando da obtenção da estrutura RTCFields correspondente ao tempo atual é verificado se alguma atualização está a decorrer, de modo a minimizar erros na leitura derivados dessas mesmas atualizações.

No projeto, programação em assembly é usada para implementar o interrupt handler do *Real Time Clock.* É usada a sintaxe da arquitetura Intelx86 dado a ter-nos sido lecionada na UC de MPCP. Apesar disso, foi notada alguma

dificuldade na implementação, dado que existem, ainda assim, diferenças entre essas sintaxes (por exemplo na invocação de sub-rotinas).

Outra parte do projeto, consiste no modo de cooperação multijogador, para o qual usamos as funções implementadas no módulo “serialport”. Para além da configuração já mencionada acima, no nosso protocolo de comunicação utilizamos os seguintes caracteres para identificar as mensagens e o seu tipo:

- 0xFA – ACK_JOIN_MSG – Mensagem enviada pelo jogador principal a reconhecer e permitir a entrada do segundo jogador no jogo.
- 0xFB – JOIN_MSG – Mensagem enviada pelo jogador secundário ao tentar entrar na partida do jogador principal.
- 0xFC – LEAVE_MSG – Mensagem enviada pelo jogador secundário quando abandona o jogo.
- 0xFD – KBD_CHAR_MSG – Mensagem enviada pelo jogador secundário quando este “envia um projétil” ao utilizar o teclado. Contém um corpo, constituído pelo caracter a adicionar ao jogo principal.
- 0xEF – ADD_CHAR_MSG – Mensagem enviada pelo jogador principal quando uma nova nave apareceu no jogo, de modo a comunicar a existência desta ao jogador secundário. O novo carater vem no corpo da mensagem.
- 0xFE – REM_CHAR_MSG – Mensagem enviada pelo jogador principal quando uma nave foi destruído, de modo a comunicar o desaparecimento de um caracter do jogo. O carater a remover vem no corpo da mensagem.
- 0xEE – P1_LEAVE_MSG – Mensagem enviada pelo jogador principal quando o jogo termina.
- 0xFF – TRAILER_ID – Caracter que sucede a todas as mensagens e indica o fim destas.

Durante a execução do projeto, surgiu-nos o problema de como gerar carateres de forma aleatoria, para a criação de inimigos, e de forma a que não existe nenhuma letra repetida no ecrã. Para isto, e dado às limitações da linguagem usada no que toca a estruturas de dados, efetuamos a implmentação de uma lista ligada de carateres. Para gerar os carateres são então criadas listas através de sucessivas inserções de novos carateres em posições aleatorias das mesmas. Quando um novo inimigo é criado, é feito “pop” do carater que se encontra no topo da lista e quando este morre, esse carater é colocado novamente na lista numa posição aleatória. A lista simplesmente ligada foi a estrutura escolhida dado a possuir uma baixa complexidade na operação de inserção, operação que é mais utilizada no processo. Para além disso, conferiu-se uma caraterística de fila, na medida em que é feito o pop do início da lista.

5. Conclusão

A cadeira de LCOM apresentou-se como um grande desafio, que nos obrigou a pensar, em grande parte, autonomamente na resolução de diversos problemas.

Um aspeto positivo é que o tema e conceção e desenvolvimento do projeto final é deixada ao nosso critério, o que nos permite praticar uma gestão e planeamento autónomos, tendo em conta as nossas capacidades. Um outro aspeto positivo é a prontidão dos professores a responder ao fórum.

Um aspeto que consideramos que LCOM poderia melhorar seria em lecionar nas aulas temas relacionados com a manipulação de *sprites* como as colisões e a rotação por exemplo. Para além disso, consideramos que faz falta uma aula laboratorial que tratasse da *UART* devido à sua elevada complexidade e diferentes nuances.