U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Relatório Final

LCOM - Laboratório de Computadores

To Infinity and Beyond

Turma 4, Grupo 11

António Cunha Seco Fernandes de Almeida - up201505836@fe.up.pt
João Paulo Madureira Damas - up201504088@fe.up.pt

Índice

1. Introdução

a. Instruções de utilização

Para executar o projeto, deve-se primeiramente o copiar o ficheiro conf para etc/system.conf.d. Depois, este deve ser compilado através do comando make. Finalmente, deve correr o projeto, através de service run `pwd`/proj sem argumentos para obter as instruções de correta inicialização.

b. Descrição do Jogo

Infinity and Beyond é um jogo em que um personagem controlado horizontalmente pelo utilizador atravessa um plano "infinito". Para além da capacidade de movimento, o personagem pode também disparar *lasers*.

Durante o percurso, obstáculos surgem no seu caminho, os quais ultrapassa desviando-se (se possível), ou eliminando-os do seu caminho disparando-lhes com sucesso um determinado número de vezes. Os obstáculos destruídos libertam balas em função da sua dificuldade de destruição.

O jogo termina quando o utilizador é "empurrado" por um obstáculo para fora do percurso (na vertical).

O personagem tem ainda acesso a bónus periódicos, nomeadamente “invencibilidade” temporária - a colisão com obstáculos é ignorada -, balas infinitas e balas duplas - provocam o dobro do dano relativamente às balas normais -.

É de notar que o número médio de disparos necessários para destruir um bloco aumenta com o decorrer do jogo e existe possibilidade de a qualquer altura do jogo pausar o mesmo.

No final, o utilizador regista o seu nome associado à sua pontuação e tem a possibilidade de ver os *highscores* até ao momento.

c. Menu inicial

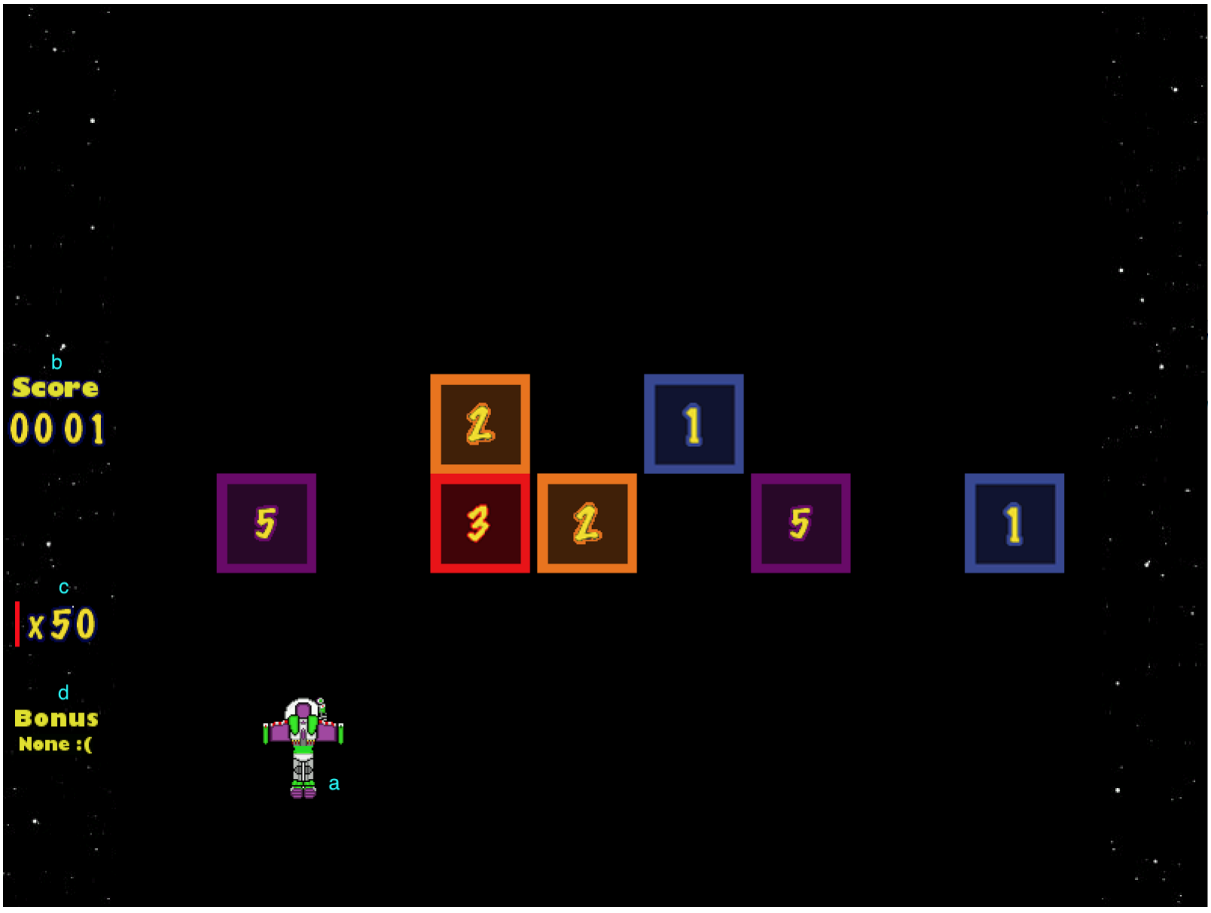
Ao correr o programa, o utilizador depara-se com o seguinte menu:

Neste ecrã inicial, através do rato é possível escolher uma das duas opções apresentadas:

1. **Play** - inicia diretamente o jogo
2. **Exit** - sai do programa. Como se pode observar na imagem, carregar na tecla Escape também executa esta função.

d. Ecrã de jogo

Ao escolher a opção **Play** descrita acima, o utilizador entra em modo de jogo, deparando-se com um ecrã semelhante ao seguinte, tendo em conta que as posições e números dos blocos são calculados aleatoriamente e variam de jogo para jogo.



Legenda da foto

a. Personagem *Buzz*
b. Pontuação atual - medida em minutos e segundos
c. Número de *lasers* atuais
d. Bónus atual - varia entre “None”, “Infinite Ammo”, “Invincible” e “Double Bullets”

Os obstáculos da imagem encontram-se em movimento descendente - ou seja, em direção ao Buzz. Quando uma linha atinge o final do ecrã e o jogo ainda não terminou, uma nova volta a aparecer no topo, com obstáculos em posições diferentes - calculados aleatoriamente. A dificuldade de um obstáculo é apresentada diretamente pelo número que tem escrito e as possíveis dificuldades são apresentadas na figura seguinte.

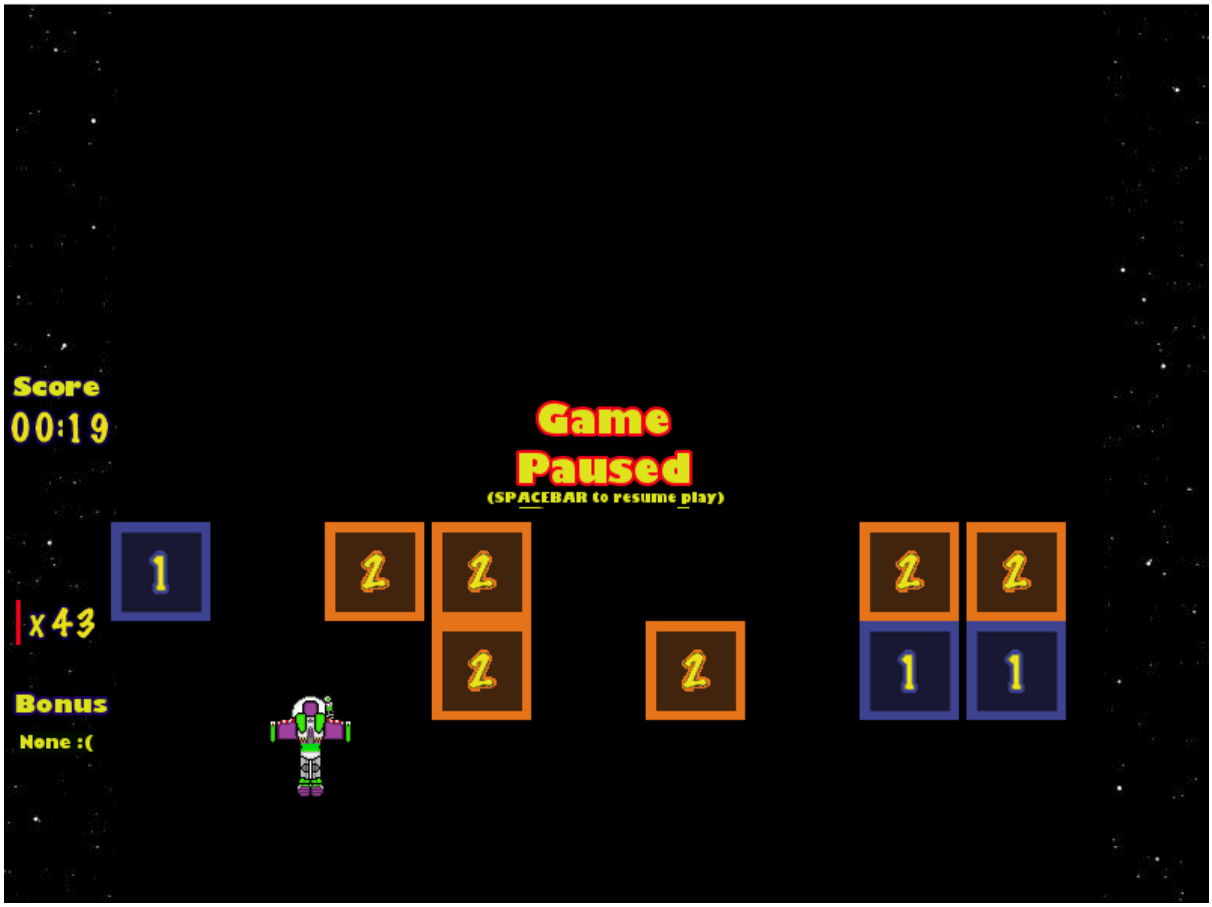
Legenda: os vários obstáculos por ordem crescente de dificuldade.

Quando um obstáculo é atingido por um laser, é substituído por outro de dificuldade imediatamente inferior - por exemplo, um obstáculo de dificuldade 4 é substituído por um de dificuldade 3. No caso do bónus atual ser “Double Bullets”, a redução é de 2 níveis de dificuldade. Caso o nível de dificuldade atinja 0, o obstáculo desaparece e abre caminho para o *Buzz.*

Ao longo do jogo, a dificuldade média dos obstáculos aumenta em função do tempo, sendo que, a uma certa altura, se verificará apenas a criação de blocos de dificuldade máxima.

i. Estado de pausa

A qualquer altura durante o decorrer do jogo, o utilizador pode pressionar na tecla Spacebar, de forma a parar indefinidamente a sessão, que recomeça após a tecla ser pressionada novamente.



ii. Ecrã final - gravar pontuação

Quando o utilizador perde o jogo, ou seja, o seu personagem é empurrado por um obstáculo para fora do ecrã, o programa entra na fase de registo de pontuação.

Neste ecrã, o utilizador tem acesso à sua pontuação da sessão, bem como aos *highscores* dos jogos decorridos até ao momento, em ordem decrescente. É-lhe então pedido que registe o seu nome - com quatro letras - de forma a associar o mesmo à sua pontuação. Essa informação, juntamente com a data e hora do sistema são armazenadas e poderão surgir como *highscore* numa sessão futura.

Terminado o registo, o utilizador é redirecionado para o menu inicial, onde poderá começar um novo jogo ou sair do programa.

Legenda: exemplo de um ecrã final de registo de pontuação.

2. Estado do Projeto

a. Funcionalidades (não) desenvolvidas

As funcionalidades descritas na secção anterior estão desenvolvidas na totalidade. Em relação à especificação, não foi implementado o modo multijogador, visto que não chegou a ser implementado o uso da porta série.

b. Utilização de periféricos

As funcionalidades descritas abaixo (na tabela e depois) representam um *overview*, mencionando por alto as mesmas e as funções gerais onde são implementadas. Para uma descrição e local de implementação mais detalhados (menção de linhas específicas em *source files*) ver a secção 4, detalhes da implementação.

Periférico	**Função**	**Interrupts**
Timer	Controlo da frequência de desenho para a placa gráfica e de fatores de jogo (pontuação, geração de bónus e aumento gradual da dificuldade)	Sim
Keyboard	Disparo de lasers e registo de nome do utilizador.	Sim
Mouse	Navegação no menu e controlo do movimento do personagem do jogo.	Sim
RTC	Leitura da data e hora do sistema para registo de *highscores*.	Não
Video Card	Desenho e apresentação dos grafismos do jogo.	Não
Serial Port	Não utilizado.	------

i. Placa gráfica

Modo Utilizado: 0x114
Resolução: 800x600 (2 bytes/pixel, RGB565)
Implementações a notar:

- Double Buffering: quer o menu, quer o jogo têm um *buffer* secundário na respetiva 'classe', alocado dinamicamente aquando da sua criação, no qual são registadas as alterações e sucessivas frames, sendo depois utilizada a instrução memcpy para transferir a informação para o buffer principal (funções *update_game_running* e *update_game_score*, em game.c, e *update_menu*, em menu.c. As funções para copiar para o buffer

principal, *draw_game* e *draw_menu*, encontram-se nos mesmos header files respetivos).

- Objetos em movimento: Não são utilizados sprites animados (o sprite para cada objeto é sempre o mesmo), no entanto verifica-se a existência de vários objetos em movimento simultaneamente: os obstáculos deslocam-se verticalmente para baixo no ecrã (atualização realizada na parte inicial de *update_game_running*, em game.c). Para além disso, o personagem, controlado pelo utilizador, move-se horizontal de acordo com as movimentações do rato (atualização feita na função anterior através da chamada de *update_player_mouse* quando um packet é recebido, em player.c). De notar ainda as balas (lasers) disparados pelo jogador que se deslocam com direção vertical e sentido contrário ao dos obstáculos (atualização realizada logo a seguir à atualização dos obstáculos, na mesma função).
- Deteção de colisões: O personagem pode ainda mover-se na vertical, no caso de estar a ser empurrado por um obstáculo. Para além disso, quando uma bala é disparada, ao longo do seu movimento no ecrã esta pode colidir com um obstáculo, resultando na destruição da bala e diminuição do nível de dificuldade do obstáculo. Os mecanismos de colisão são utilizados também na função *update_game_running*, em game.c, estando implementados em *update_player_collision* (em player.c, para colisões jogador-obstáculo) e em *bullet_obstacle_collision* (em bullet.c, para colisões bala-obstáculo).

ii. Teclado

Implementações a notar:

- Controlo do disparo do personagem durante o jogo: Durante o estado de jogo, o personagem pode disparar lasers para ajudar a abrir caminho para a sua passagem (desde que tenha um número não nulo dos mesmos) a qualquer altura premindo a tecla A. Este controlo é realizado na função *interrupt_handler*, quando é recebida uma interrupção do rato, no ficheiro dispatcher.c
- *Input* de texto na fase de guardar a pontuação do utilizador: Assim que o jogo termina, o utilizador é levado para o ecrã final onde pode ver a sua pontuação final e os *highscores* até ao momento. Depois é convidado a inserir um nome para a sua sessão. Quando o nome estiver preenchido, ao premir a tecla Enter o utilizador é levado de volta para o menu inicial. O tratamento dos carateres lidos é realizado na função *update_game_score*, chamada sempre que for detetada uma nova tecla e o jogo estiver neste estado final, em game.c.
- Outra forma de sair do programa: Quando o utilizador se encontra no menu inicial, é possível sair não só escolhendo a opção Exit, mas também premindo a tecla Escape, verificação realizada aquando da atualização geral do estado de menu, *update_menu*, em menu.c

- Pausar o jogo: A qualquer altura no decorrer do jogo premir a tecla Spacebar provoca uma pausa no jogo, sendo a sessão retomada pressionando a mesma tecla de novo. Esta verificação também é feita na função *interrupt_handler*, em dispatcher.c, com o auxílio de uma pequena máquina de estados relativa ao estado do jogo.

iii. Rato

Implementações a notar:

- Movimento
    a. No menu inicial é apresentado um cursor que tem uma posição inicial pré-definida mas que a partir daí se move consoante os movimentos que o utilizador faz com o rato, sendo o cursor atualizado com a função *update_mouse*, em mouse.c, chamada quando um packet inteiro for detetado. É depois atualizado o estado do menu, em menu.c, na função *update_menu*. De notar ainda a limitação do cursor aos limites do ecrã para evitar erros de acesso indevido a memória.
    b. No jogo a leitura dos deslocamentos do rato é parcialmente utilizada: apenas os deslocamentos horizontais são utilizados na implementação do movimento do personagem de jogo. Novamente, a função *update_mouse* é chamada, seguida de *update_player_mouse*, em player.c (e esta depois em game.c).
- Botões
    a. O menu inicial contém dois botões que podem ser selecionados utilizando o cursor implementado. A verificação da seleção é feita aquando da atualização do estado geral do menu, na função *update_menu*, em menu.c.

iv. RTC

Implementações a notar:

- Leitura de data/tempo: Os scores obtidos em cada sessão de jogo são guardados num ficheiro de texto onde a data e hora em que foram obtidos são registadas por questões de identificação. A chamada para obtenção da data e hora atuais é feita em *detect_game_end*, em game.c. A respetiva implementação da sub-rotina é feita no ficheiro rtc_get.S.

3. Estrutura do código

A seguir apresenta-se a descrição breve dos módulos implementados para o projeto, quer os da autoria do grupo, quer os de terceiros.

a. Código próprio

i. Bullet.c

Funções relacionadas com o processamento de balas (lasers) no jogo - criação, atualização de posição e remoção.

Peso relativo: 3%

Responsável: António Almeida

ii. Dispatcher.c

Funções relacionadas com o estado geral do programa, gestão das interrupções recebidas do hardware utilizado e transições gerais de estado. É neste módulo que é realizada a chamada a driver_receive().

Estruturas a notar:

- Disptacher: 'classe' que contém informação sobre os IRQs dos periféricos (onde é feita, portanto, a chamada de subscribe e unsubscribe aos mesmos)

Peso relativo: 15%

Responsável: João Damas (implementação inicial) e António Almeida (adaptação do código para conter apenas uma chamada a driver_receive())

iii. Game.c

Funções relacionadas com o processamento do jogo e a lógica a si associada.

Estruturas a notar:

- Game: 'classe' que contém toda a informação sobre a sessão atual de jogo. Responsável por atualizar quase toda a informação relativa ao mesmo.

Peso relativo: 20%

Responsável: João Damas

iv. Keyboard.c

Funções relacionadas com o processamento de input vindo do teclado.
Estruturas a notar:

- Keyboard: ‘classe’ que guarda a informação sobre as teclas premidas

Peso relativo: 5%
Responsável: João Damas

v. Menu.c

Funções relacionadas com o processamento do menu inicial do programa.
Estruturas a notar:

- Menu: ‘classe’ que guarda a informação sobre os componentes do menu (rato, teclado, fundo…)

Peso relativo: 10%
Responsável: António Almeida

vi. Mouse.c

Funções relacionadas com o processamento de input vindo do rato.
Estruturas a notar:

- Mouse: ‘classe’ que guarda a informação sobre o input recebido (movimento e cliques de botões)

Peso relativo: 5%
Responsável: António Almeida

vii. Obstacle.c

Funções relacionadas com o processamento de obstáculos no jogo - criação, atualização e remoção.
Estruturas a notar:

- Obstacle: ‘classe’ que guarda informação sobre um obstáculo (número de vidas, imagem correspondente, posição no ecrã)

Peso relativo: 5%
Responsável: João Damas

viii. Player.c

Funções relacionadas com o processamento do personagem do jogo (o seu estado, posição, número de balas e bónus) e da sua pontuação atual.
Estruturas a notar:

- Player: ‘classe’ que guarda toda a informação sobre o personagem. É responsável por atualizar a posição do mesmo

quer seja devido a movimento do rato ou a uma colisão, gerar os bónus atribuídos e a gestão do número de balas.

Peso relativo: 10%

Responsável: João Damas (implementação geral) e António Almeida (implementação do mecanismo de colisões)

ix. Read_scancode_asm.S

Ficheiro onde é implementada a sub-rotina que lê um byte (podendo ser um scancode ou parte de um) do OUT_BUF do teclado. Esta função corresponde à implementada aquando da realização do lab3, com uma nova adição: contém, agora, a implementação da espera de 20ms no caso do buffer não estar pronto a ser lido, através da chamada de tickdelay(), que não chegou a ser implementada no lab.

Peso relativo: Percentagem já incluída no módulo Keyboard

Responsável: João Damas

x. Rtc.c + rtc_get.S

Antes de mais, a descrição destes módulos encontra-se agrupada pois não se justifica um módulo isolado para rtc.c havendo o módulo rtc_get.S que acaba por lhe ser complementar e torna o módulo mais relevante.

O módulo rtc.c apenas contém as chamadas de subscribe e unsubscribe ao RTC.

No ficheiro rtc_get.S é implementa a sub-rotina que lê a configuração correspondente ou à hora atual, ou à data atual. É utilizado um argumento para distinguir que leitura o utilizador pretende e, por isso, apenas uma sub-rotina consegue executar o necessário.

Peso relativo: 5%

Responsável: João Damas

xi. Score.c

Funções relacionadas com a leitura, processamento e escrita de scores (pontuação e informações relativas a uma sessão de jogo). É onde se regista toda a interação do programa com ficheiros de texto.

Peso relativo: 5%

Responsável: João Damas (leitura e escrita em ficheiros de texto) e António Almeida (apresentação dos valores no ecrã)

xii. Timer.c

Chamadas subscribe e unsubscribe para utilização das interrupções do timer 0.

Peso relativo: 1%

Responsável: João Damas

xiii. Utils.c

Funções úteis para a execução do programa mas que não se enquadram exatamente noutros módulos (desenho de letras/números, obtenção de caminhos absolutos para ficheiros, …)

Peso relativo: 5%

Responsável: João Damas e António Almeida

xiv. Vbe.c

Chamada relativa à função VBE 0x01 para obter as informações sobre o modo gráfico utilizado.

Peso relativo: 2%

Responsável: António Almeida

xv. Video_gr.c

Funções relativas à inicialização do modo gráfico e obtenção de informações sobre o mesmo (relativamente a resolução e bytes/pixel).

Peso relativo: 5%

Responsável: António Almeida

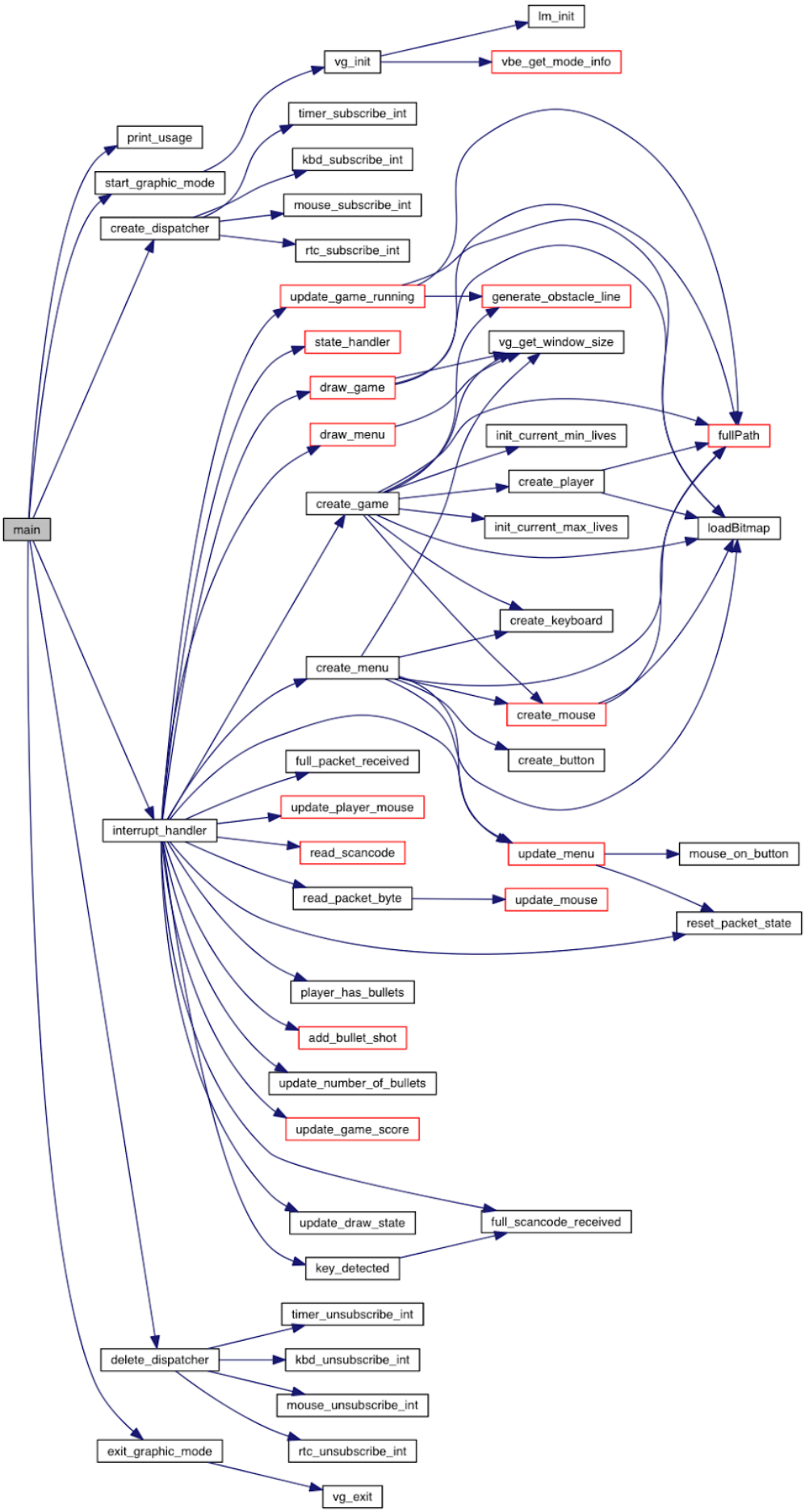
b. Código de terceiros

i. Bitmap.c

Funções relativas à criação, desenho e remoção de imagens no formato Windows Bitmap (.bmp). O código fonte foi realizado pelo aluno do MIEIC Henrique Ferrolho a quem o grupo agradece pela enorme utilidade deste pequeno conjunto de funções na realização do trabalho. De notar algumas alterações realizadas ao código original: as coordenadas do bitmap são guardadas na estrutura Bitmap, em vez de serem fornecidas independentemente na função drawBitmap. Para além disso, na função drawBitmap, a imagem não é automaticamente desenhada para o buffer principal, mas sim para um à escolha do utilizador (fornecido como argumento da função). Código original retirado de: http://difusal.blogspot.pt/2014/09/minixtutorial-8-loading-bmp-images.html

Peso relativo: 4%

c. Diagrama de chamada de funções



4. Detalhes da implementação

Umas das preocupações mais prioritárias ao longo do projeto foi a estrutura em do código em geral, de modo ao mesmo ser de fácil legibilidade, manutenção, e organizado de forma a que todas as estruturas mais "profundas" fossem de simples inclusão. Assim, é de destacar no módulo Dispatcher a forma como se efetua o tratamento das várias interrupções dos periféricos, nomeadamente através de uma máquina de estados geral ao programa cujos estados variam entre MAIN_MENU, GAME e EXIT_PROGRAM.

Por outro lado, os vários módulos do projeto são caracterizados pela presença de estruturas típicas de programação orientada a objetos, nomeadamente 'classes', com o propósito de ter acesso estruturado aos vários componentes do jogo ao longo de todo o projeto. Da mesma forma, foram uma boa solução para ter código reutilizável e mais facilmente manuseável em termos de manutenção.

A nível de frame generation, tendo a necessidade de implementar *double buffering*, alguma pesquisa foi fulcral para o desenvolvimento da mesma, dado que nos slides e labs foi apenas mencionada como uma possível solução. No código, a implementação é visível nas funções de update (que podem ser assíncronas; exemplo - um jogo pode ser atualizado devido a uma interrupção do timer ou então a um movimento do rato) onde, após atualização do estado do programa, é preparada a próxima frame no buffer secundário: game.c ll.160-196 e ll.257-277, menu.c ll.47-68; quando chegar a altura de desenhar no ecrã, é então utilizada a função draw, correspondente à invocação de memcpy: game.c ll.303-305, menu.c l.43.

Quanto a objetos em movimento, foi assinalado o facto de os obstáculos se deslocarem para baixo no ecrã de forma linear em função do tempo. Este movimento acontece aquando da atualização do estado de jogo (na função de update), chamada a cada interrupção do timer (game.c ll.83-101). Nesta função é também tratado o movimento das balas, semelhante mas de sentido inverso ao dos obstáculos, presentes no ecrã através da chamada da respetiva função de update (chamada em game.c l.117, função em bullet.c ll.14-18). Por fim, o jogador move-se horizontalmente de acordo com as informações vindas do rato num update que, geralmente, é assíncrono em relação ao resto dos elementos do jogo (player.c ll.19-50).

Outra particularidade de implementação é a presença de mecanismos de colisões entre várias componentes do jogo. Um exemplo é a colisão entre balas e obstáculos, realizada na fase de atualização geral do estado de jogo. Para cada bala, é determinado o index do obstáculo com que pode entrar em colisão e depois é testada a colisão em si (o índice é determinado pela função *determine_index*, chamada em game.c l.113 e a possível colisão é tratada logo a seguir, ll.114-149; de notar que este cálculo ajuda na fluidez do jogo visto que não é preciso testar uma possível colisão para todos os obstáculos, para cada bala: cada bala é testada, no máximo, com 2 objetos (um por cada linha do índice determinado). É relevante acrescentar que a colisão é testada através da posição dos pontos chave da bala, nomeadamente o topo esquerdo e direito, relativamente ao obstáculo: se pelo menos um destes estiver na zona do obstáculo, é detetada a colisão) e entre o personagem e obstáculos, que pode ser de

dois tipos: vertical, testada na mesma altura da situação supra descrita (game.c l.150, função definida em player.c ll.54-63), e horizontal, verificada aquando da atualização da posição consoante o movimento do rato. No entanto, a colisão aqui (em ambos os casos) é testada de outra forma: em vez de ser determinado o possível obstáculo de colisão, são verificadas as cores em redor dos limites do bitmap do personagem. Se não forem preto (correspondente ao fundo do jogo), é porque se detetou uma colisão e por isso o movimento fica impedido. De notar que esta alternativa é melhor no caso do jogador visto que a mudança de posição é tão frequente que seriam calculados e testados demasiados obstáculos quase simultaneamente (pelo contrário, a bala, devido ao seu movimento estritamente vertical, estará sempre no caminho do(s) mesmo(s) obstáculo(s)).

Relativamente a sub-rotinas implementadas em assembly, o objetivo foi aplicar o maior número de conceitos possível abordados nas aulas teóricas. Assim, para completar a sub-rotina desenvolvida no lab3, *read_scancode_asm()*, como já foi dito numa secção anterior, foi implementada a chamada de *micros_to_ticks()* e consequentemente *tickdelay()* de modo a demonstrar o uso de invocações de funções C em assembly. Na outra sub-rotina desenvolvida, a propósito do uso do Real Time Clock, cuja leitura se processou através do auxílio da flag UIP do registo A (ciclo inicial presente em rtc_get.S), para obter a configuração atual relativa à hora ou data do computador através do RTC, foram aplicados outros conceitos, nomeadamente: passagem de resultados através de variáveis entre C e assembly, e ainda passagem de argumentos para a função vindos da invocação em C. Em relação ao primeiro, que requereu alguma pesquisa extra-aula, o modelo implementado contém dois arrays alocados estaticamente no ficheiro assembly para os quais são lidos a data e a hora, respetivamente. A necessidade de ter 2 arrays provém do facto da chamada da função no programa ser feita duas vezes, consecutivamente, para ler data e hora. Assim, se só houvesse 1 array, a informação da primeira chamada seria perdida. Por outro lado, em relação à passagem de argumentos para a função vindos da invocação em C, de modo a tornar a função mais genérica (uma só função é capaz de ler a data e a hora, sendo a escolha feita pelo utilizador), foi criada uma *stack frame* e posteriormente acedeu-se ao argumento pelo deslocamento relativo ao registo ebp. Todas as sub-rotinas foram implementadas usando a convenção C para poderem ser usadas no programa principal.

(Os parágrafos seguintes descrevem mecanismos abordados com mais detalhe nas aulas teóricas, nomeadamente relativamente a teclado e rato, no entanto, para além disso, fornecem uma especificação dos locais de implementação mais detalhada)

Em relação às funcionalidades do teclado, destacam-se controlo da aplicação: Para disparar balas, o utilizador deve premir a tecla A. Depois de ser detetado o *breakcode* desta tecla, uma bala (e uma só) é disparada da posição atual do jogador. Esta verificação e mecanismo são feitos aquando da leitura/tratamento de interrupções (dispatcher.c ll.70-72). Para além disso, é possível pausar o jogo a qualquer altura premindo a tecla Spacebar. Enquanto pausado, as interrupções são lidas e tratadas na mesma mas não causam nenhuma atualização no estado de jogo até o jogo ser retomado, premindo novamente a tecla Spacebar (dispatcher.c ll. 74-80). Por fim, no menu, é possível sair premindo Escape. Ao ser detetado o respetivo *breakcode*, o programa termina (menu.c ll.59-60); é de referir ainda o input de texto: no ecrã final, o

utilizador é convidado a inserir um nome de 4 letras para a sua sessão de jogo, havendo a possibilidade de correção em caso de erro: ao premir a tecla Backspace, o utilizador pode anular a inserção do último carater registado (cada *scancode* é convertido para a respetiva letra: game.c l.254 sendo imediatamente depois chamada a função de *handling* da máquina de estados do nome da sessão onde é processado o descrito: game.c ll.202-249).

Por fim, em relação ao rato, de realçar não só o uso do movimento do cursor (no menu principal e do jogador *in-game*), como também o uso de botões. Quanto ao movimento, nomeadamente no menu, no tratamento da interrupção do rato são lidos os bytes de cada packet numa função específica cuja chamada se encontra em dispatcher.c l.48 e a função em si em mouse.c ll.102-116. Se for detetado um *packet* na sua íntegra (o estado de receção de um packet é controlado através de uma pequena máquina de estados presente na 'classe' Mouse), a função de leitura supramencionada chama ainda a função de update ao rato, definida em mouse.c ll.120-150, e a função de update ao estado do menu sabendo que o update se deveu a um novo packet recebido (função chamada em dispatcher.c ll.49-50 e função de update em si: menu.c ll.47-57). No movimento do jogador, novamente, quando for detetado um packet na íntegra (dispatcher.c l.54), a função de update ao jogador devido a um movimento do rato é chamada (dispatcher.c ll.56-57, estando a função implementada em player.c ll.19-50), sempre tendo em conta os limites do ecrã. Em relação ao uso de botões, na função de update ao rato, não só as coordenadas do cursor são atualizadas, mas também informação sobre o clique do botão esquerdo (mouse.c ll.128-139). Na função de update ao estado de menu é depois verificada a posição do rato e, caso esteja sobre um dos botões e o botão esquerdo tenha sido premido, o programa assume essa como a escolha do utilizador e transita para o respetivo estado (menu.c ll.47-57).

5. Notas finais

a. Relativamente à documentação em Doxygen

O Doxyfile que gera o diagrama de chamada de funções requer que a flag EXTRACT_ALL esteja marcada com YES, fazendo com que seja criada documentação para todos os ficheiros e funções, incluindo as que não tem comentário identificativo apropriado. Devido a isto, decidimos incluir 2 doxyfiles no repositório:

- Doxyfile: cria a documentação "útil" (as funções que realmente estão comentadas de acordo com o propósito), mas que não gera o diagrama de chamada de funções;
- DoxyfileGraph: cria a documentação na íntegra, incluindo o diagrama de chamada de funções (autenticando a figura presente no relatório; requer a aplicação Graphviz)

Ambos os doxyfiles funcionam automaticamente e guardam o resultado da sua execução em /nographs e /withgraphs, respetivamente. No entanto, é possível editar facilmente quer o caminho de entrada, quer o de saída, já que as flags correspondentes se encontram no topo dos respetivos ficheiros.

b. Relativamente aos sprites utilizados

A larga maioria das imagens utilizadas foi feita pelo grupo. No entanto, os sprites do personagem principal, Buzz Lightyear (quer o usado no jogo, quer os demonstrados no menu) foram retirados da Internet e utilizados com a devida permissão do autor, a quem também agradecemos pelo trabalho poupado. Sprites retirados de: http://landsverk96.deviantart.com/art/Custom-Buzz-Lightyear-Spritesheet-Toy-Story-494004136

c. Relativamente à UC

No geral, a avaliação da UC é positiva. Trabalhar com os periféricos mais comuns do computador é interessante e perceber o funcionamento dos mesmos através da programação a relativo alto nível (exceto programação em assembly, que não deixa de ser dos tópicos mais interessantes e sem dúvida uma secção a manter e possivelmente a ser expandida), mas de detalhes de "baixo" nível é de todo o interesse para nós enquanto alunos do MIEIC. No entanto, alguns aspetos da UC não são tão positivos, nomeadamente:

- A dificuldade/carga de trabalho da UC não corresponde ao seu valor em ECTS, sendo bem maior que este
- A falta de familiarização com a linguagem C faz com que a maior parte dos alunos perca muito tempo em certos detalhes de implementação que não se prendem bem às funcionalidades mas à sintaxe da linguagem. Uma possível solução seria, ao longo do leccionamento dos labs, serem

apresentados (por exemplo, em alguns slides intermédios) conteúdos da linguagem importantes para o desenvolvimento do lab (tal como existem as transparências relativas a apontadores em C, mas não tendo uma apresentação inteiramente dedicada ao assunto), transmitindo então a informação de um modo gradual mediante a crescente necessidade dos alunos. Para além disso, a comparação "Implementação em C++ vs Implementação em C" pode ser algo que facilita a compreensão dos conceitos, pelo menos numa fase inicial, já que quem entrou no curso com pouco/nenhum contato prévio com programação tem em C++ a maioria do seu conhecimento (derivado de PROG), conhecendo por isso os conceitos, mas não a sua "versão C" (ex: alocação dinâmica, new vs malloc)

- Relativamente ao uso da linguagem assembly, apesar de não ser o foco da UC, é reconhecido como uma grande vantagem e por isso achamos que algo subvalorizado. Em relação ao modo como é lecionado, a conclusão retirada é que os slides se deveriam focar mais nas diferenças entre a sintaxe AT&T e Intel, visto que os conceitos sobre criação de sub-rotinas, convenções (nomeadamente convenção C), parâmetros na stack, entre outros já foi estudado com relativa exaustão na UC MPCP. Algo relativamente à linguagem que não foi estudado com tanto detalhe em MPCP e que valeria a pena focar é a passagem de valores por variáveis entre C e assembly, possivelmente do interesse para quem usar sub-rotinas em assembly no seu projeto e que as transparências atuais pouco focam.
- Relativamente aos labs, apesar de não ter sido lecionado este ano (devido à semana de aulas a menos), reconhecemos a existência do lab1 que foca a placa gráfica em modo texto. Sendo um periférico muito pouco utilizado em projetos (no modo texto, claro), achamos que era uma mais valia o lab do RTC voltar a ser lecionado, já que é bastante mais utilizado e sendo um periférico novo (a placa gráfica, apesar de noutro modo, já é lecionada no lab5) e relativamente simples era uma troca "justa". Algumas funções relativamente à placa gráfica em modo de texto poderiam na mesma lecionadas, por exemplo, como parte integrante do lab5, se fossem consideradas do interesse geral (enquadrar as funções mais importantes no lab em detrimento de, por exemplo, uma das funções de desenho, test_line ou test_square - este modo de representar imagens não é usado em projetos, sendo normalmente preferido o uso de imagens já existentes e carregadas previamente para o programa - e da função test_controller).
- Finalmente, a própria organização do conteúdo dos labs e a relação dos mesmos com os slides das aulas teóricas deixa um pouco a desejar. A constante necessidade de acesso a várias fontes de informação, nomeadamente páginas de documentação externas, provoca confusão em relação a que conteúdos são efetivamente necessários para efetuar uma determinada função do lab. As próprias secções finais de compilação e execução do código que são, no essencial, iguais em todos os labs,

podiam ser retiradas e agrupadas num módulo à parte de “utilidades” comuns à UC, juntamente com dicas em geral e comandos de terminal úteis - muitos estão perdidos nas várias secções dos primeiros labs, de difícil pesquisa.