Project Watt

LCOM Final Report

Turma 6 - Grupo 7

Eduardo da Costa Correia
up201806433

Tiago Duarte Silva
up201806516

January 6, 2020

Contents

Chapter 1

Instruções de Utilização

1.1 Menu Principal

Ao iniciar o programa, é apresentada o menu principal, onde o jogador pode selecionar um dos dois modos de jogos possíveis a iniciar, cada um com a opção de se jogar ou não com outro jogador (se for para jogar em modo **Co-Op**, um computador deve selecionar Co-op P1 e o outro Co-op P2).

Para sair do jogo, o utilizador deve pressionar a tecla ***Esc*** duas vezes, o mesmo para voltar ao menu principal a partir de um dos modos de jogo.

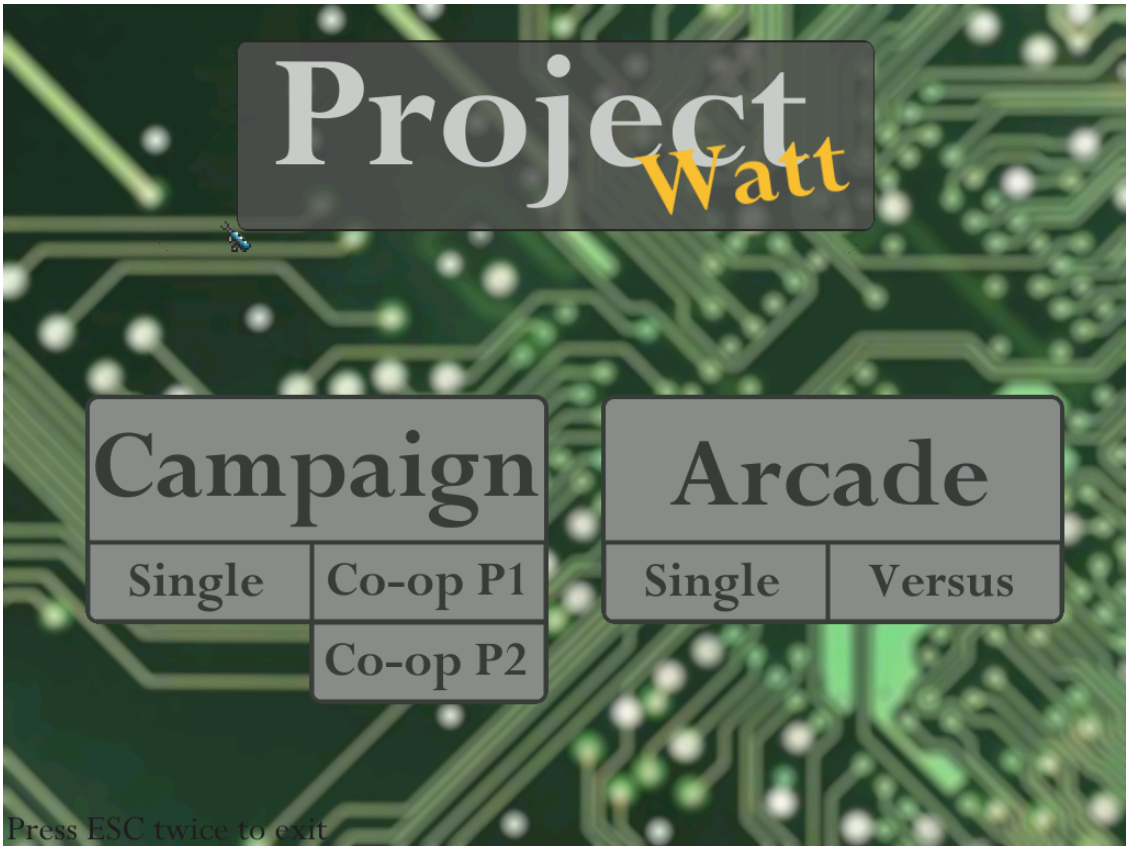

Figure 1.1: Menu principal

1.2 Campaign

Este é o modo de jogo "principal", que se trata de um *platformer* em que um jogador controla a personagem *Watt* (Figura 1.2), uma faísca que ficou "presa" num curto-circuito e que para sair terá de fazer "ligação à terra" (Figura 1.3), ou seja, neste caso, a personagem tem de alcançar o **canto superior direito do mapa** (Figura 1.5[1]). Para isso tem de ultrapassar diversos obstáculos tais como os **espinhos** e **lasers** (se tocar num destes a faísca dissipa-se e volta ao início).

Para além disso, é possível controlar certos aspetos da jogabilidade, como a **altura do salto** da personagem e a sua **velocidade** (através dos *sliders* azul e laranja respetivamente).

Existem ainda dois **powerups** que após obtidos, desbloqueiam o controlo de funcionalidades adicionais, requeridas para acabar o nível. Estes são o powerup dos **lasers** (**canto superior esquerdo** - Figura 1.3) que permitem controlar qual dos lasers está **inativo** (vermelho, azul ou roxo - através dos botões com a cor correspondente) e alterar o sentido da gravidade da personagem (ou seja, a personagem passa a ser puxada para cima em vez de para baixo como seria normalmente).

Todos estes aspetos são controlados pelo **segundo jogador**, se estiver a jogar em co-op através de uma ***switchboard*** (Figura 1.6). Esta ***switchboard*** possui ainda um ***mini-jogo*** que consiste em destruir ***balões*** que aparecem (clicando neles) antes que estes atinjam o **topo do ecrã**, caso contrário, tal gerará ***interferência*** (o ecrã fica com *ruído* para tornar a experiência mais desafiante e fazer com que quem controle a personagem 2 tenha de estar mais ativo).

Figure 1.2: Personagem 1 - Watt

Figure 1.3: Saída

(a) Laser

(b) Anti-gravidade

Figure 1.4: Powerups

[1]As figuras apresentadas para ilustrar o nível da personagem 1 do modo campaign são do modo singleplayer, portanto incluem botões e *sliders* no canto superior esquerdo que não estão presentes no modo *co-op*

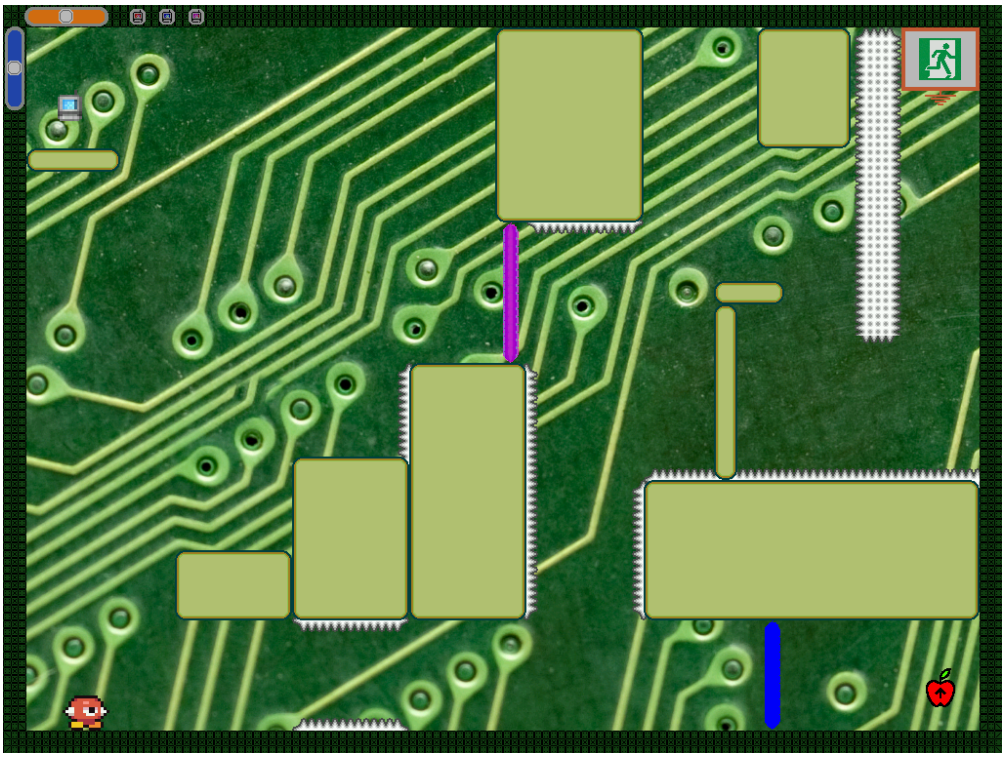

Figure 1.5: Campaign - Nível (Personagem 1)

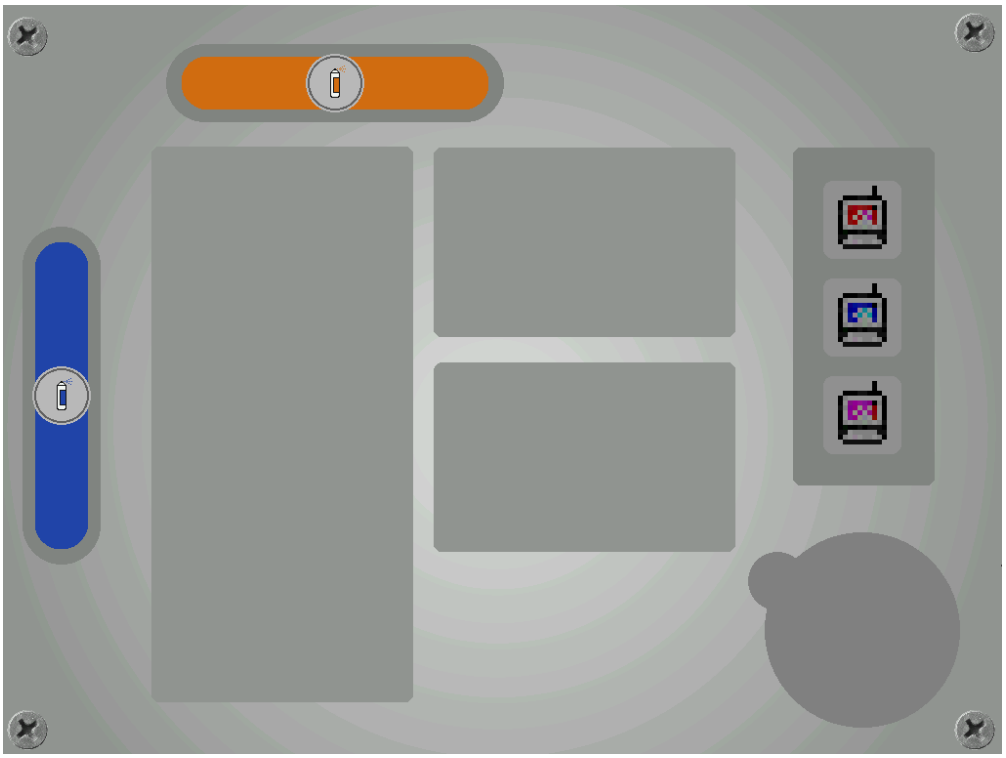

Figure 1.6: Campaign - Switchboard (Personagem 2)

1.2.1 Victory Screen

Quando o jogador conclui o modo ***Campaign***, é exibido um ecrã a parabenizá-lo e a informá-lo do **tempo que demorou**, em segundos, a fazê-lo. Para sair deste ecrã e regressar ao menu principal o utilizador deve pressionar ***Esc*** duas vezes.

Figure 1.7: Campaign - Victory Screen

1.3 Arcade

Arcade é um modo de jogo alternativo, baseado na mecânica de **anti-gravidade** do outro modo de jogo. Neste o jogador terá de se desviar de **lasers** que vêm na sua direção (passando por entre estes) tentanto aguentar o máximo de tempo possível sem tocar neles e perder. Por cada par de **lasers** que o jogador passa o seu ***score*** (**canto superior direito** - Figura 1.8) aumenta por um ponto. O ***score*** superior, a **branco**, representa a sua pontuação atual (que é reposta de cada vez que este toca num **laser**), já o ***score*** inferior, a **cinzento**, representa a maior pontuação (***highscore***) do jogador que ele conseguiu obter durante a partida.

Possui também um modo ***versus*** (Figura 1.9), em que dois utilizadores jogam simultaneamente para ver quem obtém a maior pontuação durante um tempo limite (de 45 segundos, após o qual aparece um ecrã a informar de quem foi a vitória e a pontuação respetiva de cada jogador - Figura 1.10). Para distinguir entre os dois jogadores, um deles é apresentado com uma cor **azulada**, tal como o seu ***score*** e o outro a vermelho (Figura 1.9).

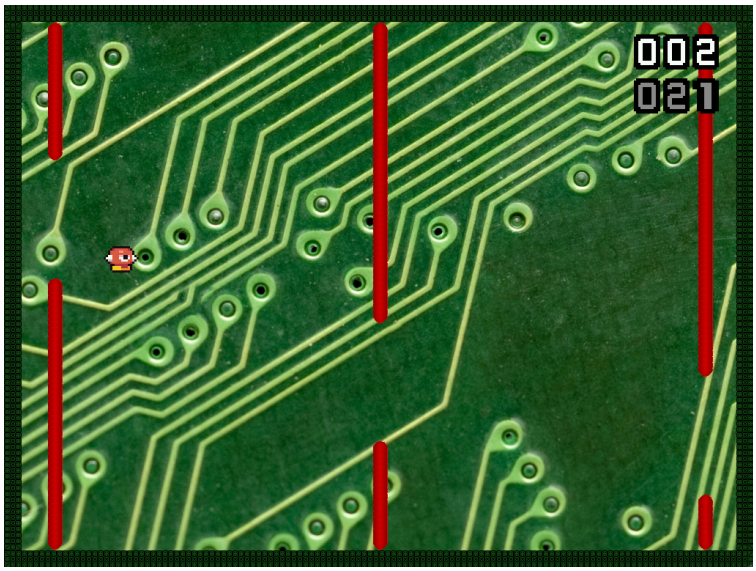

Figure 1.8: Arcade - Singleplayer

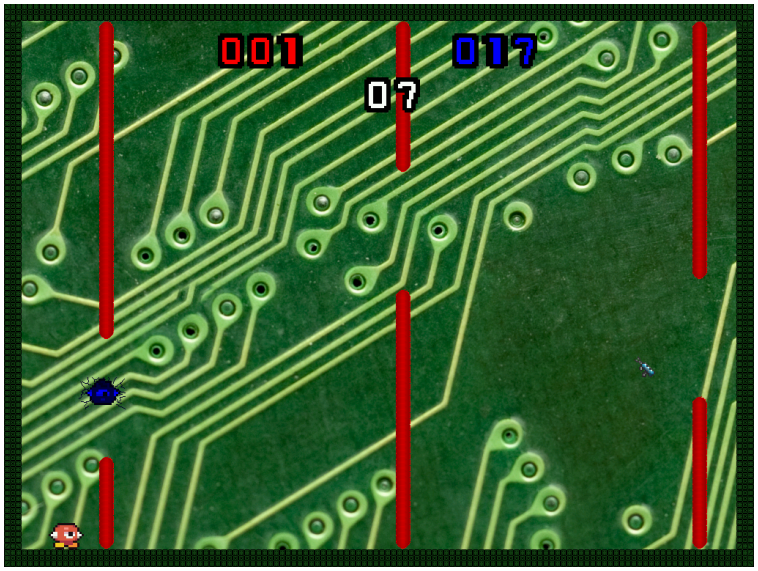

Figure 1.9: Arcade - Versus

Figure 1.10: Arcade - You Win

1.4 Relatório

Se for necessário re-compilar este relatório, em distribuições baseadas em *Arch*, basta instalar a package *texlive-core*, e depois correr o comando *latexmk -pdf final-report.tex* a partir do diretório onde o ficheiro *final-repot.tex* se encontrar.

Chapter 2

Estado do Projeto

2.1 Dispositivos Usados

Dispositivo	Utilização	Interrupção
Timer	Framerate handling e in-game timer	Sim
Teclado	Controlo da personagem 1	Sim
Rato	Menus e personagem 2 (switchboard)	Sim
Placa Gráfica	Desenho dos menus e do jogo	Não
Real Time Clock	Mini-game (switchboard, tempo demorado (campaign) e alarme para terminar o arcade mode versus	Sim
Serial Port	Modos Co-Op e Versus	Sim

2.1.1 Timer

O timer possui como principal função atualizar o estado do jogo, assim, a cada $\frac{1}{60}$ segundos (o Timer 0 possui uma frequência de 60 Hz), todo a informação do nível (lasers ativos, coordenadas/estado do jogador, posição do rato...) é atualizada conforme os inputs do(s) utilizador(es) e o jogo é no ecrã renderizado de acordo com essa informação. Também tem um papel importante no modo arcade ao ser utilizado para avaliar o tempo restante até o alarme do RTC terminar o jogo (foi usado o timer e o número de frames decorridos pois o RTC dava resultados diferentes nos dois PCs). Este número de frames decorrido também é responsável por ajustar a dificuldade do modo de jogo arcade.

2.1.2 Teclado

O teclado é usado para controlar a personagem 1 e o seu movimento, atraveś das teclas **W**, **A**, **S**, **D** ou pelas setas ↑, ←, ↓ e → e o seu salto pela tecla **Z** ou pelo **Espaço** (estas são duas das configurações de inputs mais habituais neste tipo de jogos).

Para além disso, se estiver a jogar em *singleplayer*, o sentido da gravidade da personagem é trocado através da tecla **X**.

2.1.3 Rato

O rato é usado sobretudo no menu principal para selecionar o modo de jogo, clicando no botão correspondente, e no modo *campaign* para controlar os *sliders*, clicando e arrastando o botão esquerdo (o mesmo para a *knob* no modo *Co-Op*) e para selecionar o botão do *laser* inativo. No modo de jogo *co-op*, no jogador dois também é usado para impedir que uns 'balões' cheguem ao topo do ecrã e distorçam a imagem renderizada.

2.1.4 Placa Gráfica

A placa gráfica é usada para renderizar todo o jogo, por **camadas**(conforme a ordem das chamadas de render de cada elemento). É usada no modo **0x117**, que possui resolução **1024x768** e profundidade de cor **RGB 5:6:5** (2^{16} cores possíveis). Para além disso, usámos a técnica de ***double buferring*** e a nossa taxa de atualização do ecrã é de **60 *frames* por segundo**.

2.1.5 Real Time Clock

O ***RTC*** é usado essencialmente em três vertentes diferentes: No modo ***Campaign***, com a **personagem 1**, para ler a hora de quando a personagem inicia o jogo e de quando acaba para ver quanto tempo demorou a concluir o nível; com a **personagem 2** (*switchboard*) para determinar de quanto em quanto tempo surgem os ***balões*** do mini-jogo, através de ***alarmes*** (*interrupts*); e no modo *Arcade Versus* é usado um alarme para nos notificar de quando o tempo do jogo terminou.

2.1.6 Serial Port

A ***serial port*** é usada para coordenadar os dois jogadores conetados nos modos **multiplayer** (Campaign - Co-Op e Arcade - Versus). No **primeiro**, as personagens 1 e 2 são controladas pelos utilizadores em máquinas separadas, sendo que estas interegam entre si do seguinte modo: A personagem 2 envia a informação no que toca aos aspetos que esta pode alterar (altura do salto, velocidade...) à personagem 1 e esta envia informação acerca do estado do jogador (powerups obtidos, morto ou vivo...). Esta informação é a mínima necessária e garante a sincronização entre os estados de ambos os ecrãs.

Já no modo **Arcade - Versus**, ambas as máquinas controlam a personagem 1 (Watt), aparecendo assim dois deles, um com a cor normal e um azul para distinguir entre os jogadores. Uma das máquinas age como ***master***, enquanto que a outra possui o papel de ***slave***. A ***master*** fica encarregue de gerar a altura dos lasers aleatoriamente, com intervalos progressivamente mais curtos, alterar a velocidade dos lasers e controlar (através do alarme do RTC) quando o jogo termina para os dois jogadores. Essa informação é enviada para o ***slave***.

Indiferente ao papel de cada máquina, ambas enviam de uma para a outra a sua posição, animação, estado (se o jogador está morto, a sua direção, se tem antigravidade ativa) e se a sua pontuação aumentou ou não.

Implementada com *FIFO*s e uma receiver *Queue* a nível de *software*.

Chapter 3

Organização e Estrutura do Código

3.1 Geometry

Peso relativo: 5%
Módulo desenvolvido por: Tiago Silva

O módulo ***Geometry*** é dividido em duas partes, ***Vec2d***, que representa um vector **bidemensional** e ***Rect***, que representa um retângulo, e serve como base do nosso sistema de coordenadas, físicas, hitboxes e colisões. É possível fazer diversas operações sobre estes dois elementos, como fazer produto escalar, soma e subtração... entre outras, no caso do Vec2d e verificar interseções, por exemplo, no caso do Rect.

3.2 Bitmap

Peso relativo: 3%
Módulo desenvolvido por: Eduardo Correia (40%) e Tiago Silva (60%)

O módulo ***Bitmap*** consiste em código de nível mais baixo que carrega um ***bitmap*** individual para a memória, lendo o seu header e a imagem em si — *new_bitmap()*. Tem a capacidade de renderizar imagens **normalmente** — draw_bitmap(), ou **invertidas** segundo os eixos x — *draw_bitmap_reversed_x_axis()*, y — *draw_bitmap_reversed_y_axis()* ou ambos *draw_bitmap_reversed_both_axis()* (garantindo a eficência).

Permite ainda renderizar o que apelidamos de um ***bitmap* dinâmico**, que corresponde a representar uma imagem de tamanho indicado em *runtime* (abordado no Capítulo 4). Permite **transparência** e a aplicação de uma cor sobre o que é renderizado. O *path* absoluto para os *assets* pode ser alterado como um *command line argument*.

3.3 Sprite

Peso relativo: 5%
Módulo desenvolvido por: Tiago Silva

O módulo ***Sprite*** é, na sua essência,um *wrapper* para o bitmap (tanto o normal como o dinâmico). Como acréscimo, tem uma "camada" extra que permite a animação de qualquer *Sprite*, alternando o *bitmap* a ser desenhada no ecrã na posição da *Sprite* em questão — *set_animation_state()*.

3.4 Game Manager

Peso relativo: 9%

Módulo desenvolvido por: Eduardo Correia (25%) e Tiago Silva (75%)

Módulo essencial que gere todos os aspetos do jogo, desde as mensagens recebidas da UART ao **driver_receive()** e aos ciclos de ***update*** e ***rendering***. É instanciado e usado como um *singleton*.

3.5 Hardware Manager

Peso relativo: 1%
Módulo desenvolvido por: Eduardo Correia (50%) e Tiago Silva (50%)

Este módulo foi uma tentativa de tentar encapsular ao máximo todo o conhecimento e funções responáveis por interagir com o ***hardware*** num único sítio o e transparecer o mínimo possível sobre as nossas implementações internas do *timer*, *keyboard*, *mouse*, etc..

Infelizmente existem outros módulos com um grande conhecimento destes protocolos, como o *InputEvents*, o *GameManager* ou tudo o que envolvesse o *serial port*.

3.6 Level

Peso relativo: 8%
Módulo desenvolvido por: Eduardo Correia (40%) e Tiago Silva (60%)

Vão haver essencialmente dois níveis, um para a **Campaign** — new_prototype_level() e outro para **Arcade** — new_arcade_level(), sendo possível iniciar cada um destes como singleplayer ou não atraveś do parâmetro (bool is_singleplayer).

3.7 Platforms

Peso relativo: 3%
Módulo desenvolvido por: Tiago Silva

As plataformas agem acima de tudo como blocos construtores do nível, para definir tanto as áreas sobre as quais o jogador pode caminhar como delimitar barreiras (nomeadamente as paredes laterais para o jogador não sair fora do nível). Possuem um SpriteDynamic para ser possível desenhá-las com uma tamanho variável e possuírmos assim maior flexibilidade no design do nível.

3.8 Lasers

Peso relativo: 5%
Módulo desenvolvido por: Eduardo Correia (35%) e Tiago Silva (65%)

Os *lasers* são o obstáculo mais comum que o jogador irá encontrar, tendo de evitá-los para não morrer. Cada um possui uma cor que serve também como ID para sabe que lasers estão ativos (estarão sempre todos ativos exceto o de cor correspondente ao ID inativo, controlado pelo utilizador no modo Campaign) e podem ter qualquer a dimensão desejada.

3.9 Spikes

Peso relativo: 1%
Módulo desenvolvido por: Eduardo Correia

São o segundo obstáculo principal do jogo. Mais simples que os lasers e ao contrário destes, são imutáveis e estáticos, não é possível "desativá-los", de modo que o jogador será de encontrar maneiras de se desviar destes para os ultrapassar. São de algum modo Platforms que o jogador não pode tocar e estão apenas presente no modo Campaign.

3.10 Switchboard

Peso relativo: 3%
Módulo desenvolvido por: Eduardo Correia (10%) e Tiago Silva (90%)

Este módulo é exclusivo do modo Co-Op da Campaign e está disponível apenas para o segundo jogador. Está dotado de vários elementos UI que permitem ao segundo jogador interagir com o jogo que está a ocorrer no ecrã do primeiro jogador alterando as suas propriedades (como já foi referido anteriormente). Possui ainda um mini-jogo em que o jogador tem de clicar em balões que surgem esporadicamente.

3.11 Player

Peso relativo: 7%
Módulo desenvolvido por: Eduardo Correia (35%) e Tiago Silva (65%)

O módulo *Player* trata de gerirdor a representação de um jogador e da sua interface no nosso projeto bem como a sua interação com os restantes módulos.

A classe *PlayerTwo* apenas é utilizado no modo Arcade - Versus para representar um segundo jogador que está a jogar noutra máquina.

3.12 UI Elements

Peso relativo: 6%
Módulo desenvolvido por: Eduardo Correia (25%) e Tiago Silva (75%)

O módulo *UI Elements* consiste em vários elementos da *graphical user interface* que são utilizados pelo programa. Consiste em botões, *sliders*, *knobs* e *scoreboards*.[1]

3.13 Powerup

Peso relativo: 2%
Módulo desenvolvido por: Eduardo Correia (10%) e Tiago Silva (90%)

Classe que representa tanto os poderes que o *Player* obtém como a saída do *Level* no modo de jogo *Campaign Single* e *Campaign Co-op*.

3.14 Timer

Peso relativo: 4%
Módulo desenvolvido por: Eduardo Correia (50%) e Tiago Silva (50%)

Código importado do Lab 2 (com melhorias)[1]

3.15 i8042

Peso relativo: 0.5%
Módulo desenvolvido por: Eduardo Correia (50%) e Tiago Silva (50%)

Código importado do Lab 2[2]

3.16 Keyboard

Peso relativo: 4%
Módulo desenvolvido por: Eduardo Correia (50%) e Tiago Silva (50%)

Código importado do Lab 3[2]

3.17 Mouse

Peso relativo: 4%
Módulo desenvolvido por: Eduardo Correia (50%) e Tiago Silva (50%)

Código importado do Lab 4[2]

3.18 i8254

Peso relativo: 0.5%
Módulo desenvolvido por: Eduardo Correia (50%) e Tiago Silva

Código importado do Lab 3/4[2]

3.19 Video

Peso relativo: 6%
Módulo desenvolvido por: Eduardo Correia (50%) e Tiago Silva (50%)

Código importado do Lab 5[2]

3.20 Video Macros

Peso relativo: 1%
Módulo desenvolvido por: Eduardo Correia (50%) e Tiago Silva (50%)

Código importado do Lab 5[2]

3.21 RTC

Peso relativo: 2%
Módulo desenvolvido por: Eduardo Correia

[2]No código importado dos labs fizemos ligeiras alterações, como remover funções que não utilizamos (por exemplo, naquilo em que optássemos por usar *interrupts* removemos o código que lidava com *pollling*) e especificar uma abordagem geral para uma específica ao projeto, para ser mais eficaz (como usar a placa gráfica num modo apenas, correspondente ao que optámos por utilizar no nosso jogo).

O módulo do RTC é responsável por obter informação acerca da hora atual — *get_date()* possuindo uma verificação antes de o fazer para obter a hora correta, bem como programar alarmes. — *rtc_set_alarm()*. — rtc.c

3.22 UART

Peso relativo: 5%
Módulo desenvolvido por: Eduardo Correia (20%) e Tiago Silva (80%)

Este é o módulo que lida com o serial port e com a troca de informação entre os dois computadores que estarão a executar o jogo. Foi feita com FIFO's e em modo full-duplex.

3.23 Queue

Peso relativo: 4%
Módulo desenvolvido por: Tiago Silva

Implementação relativamente simples da estrutura de dados *queue* para uso com a UART. Foi inspirada pela implementação de um site, mas essa mesma possuía diversos erros e limitações, acabando por ser reescrita completamente na prática.

3.24 InputEvents

Peso relativo: 3%
Módulo desenvolvido por: Eduardo Correia (40%) e Tiago Silva (60%)
Composto pelo *KbdInputEvents* e pelo *MouseInputEvents*.
Para guardar o input do *keyboard* recorremos à classe *KbdInputEvents*, que mantém um registo sobre todos os *scancodes* de 1 *byte* e sobre um subset específico de *scandodes* de 2 *bytes*.
Para guardar o input do rato recorremos à classe *MouseInputEvents*, que expõe para o resto do programa os botões premidos pelo utilizador, se eles estão a ser premidos neste instante e se eles começaram a ser pressionados neste frame. Regista também o *delta* do movimento do rato no eixo do x e y em cada frame.

3.25 MouseCursor

Peso relativo: 2%
Módulo desenvolvido por: Tiago Silva
Representa o cursor do rato, atualizando a sua posição no ecrã e renderizando o cursor em si (quando a sua flag assim está definida).

3.26 Utils

Peso relativo: 1%
Módulo desenvolvido por: Eduardo Correia (50%) e Tiago Silva (50%)

Este módulo consiste num conjunto de funções com utilidade diversa que foram usadas um pouco por todo o projeto. A maioria destas funções foram desenvolvidas ao longo dos *labs* e lidam principalmente com escrita e leitura para endereços de memória, bem como manipulação de bits.

3.27 MathUtils

Peso relativo: 1%
Módulo desenvolvido por: Eduardo Correia (50%) e Tiago Silva (50%)

Semelhante ao *Utils*, porém, por questão de organização, decidimos separar as funcionalidades matemáticas num ficheiro à parte. A maior parte das suas funções consistem em adptações de funções já existentes (no *math.h*) mas com diferentes tipos de dados (no caso caso, float, uint16_t e uint8_t).

3.28 Main Menu

Peso relativo: 3%
Módulo desenvolvido por: Tiago Silva

Quando o utilizador inicia o jogo pela primeira vez ou sai de um modo jogo (Campaign ou Arcade) ele depara-se com o Main Menu. Este trata-se de um conjunto de botões que após clicadas iniciam o modo de jogo corresponde. Sempre que isso acontece, toda a informação do modo de jogo anterior (se houver algum) é libertada para prevenir memory leaks e é tratado de iniciar o próximo modo de jogo devidamente.

3.29 Proj

Peso relativo: 1%
Módulo desenvolvido por: Tiago Silva

O módulo *Proj*, por mais que possa parecer contra-intuitivo, é dos que menos tem peso no projeto, uma vez que só trata de chamar a função de iniciar o jogo — *start_game* com o *path* absoluto para as imagens passado como argumento na linha de comandos. O *GameManager* trata de toda a lógica do jogo.

3.30 Call Graph

O nosso ***main_loop()***, o *start_game()* tem várias funções de ***update*** (static), mas visto elas estarem dentro de um objeto (*GameManager*) e são atribuídas aquando da sua instanciação, elas não são detetadas pelo Doxygen. Algo peculiar foi que ao dar set da flag *EXTRACT_ALL* no Doxyfile, ele deixou de gerar sequer alguns *call graphs* (nomeadamente start_level, start_switchboard, start_arcade e start_main_menu).

Em suma, tanto quanto entendemos, pelo facto de utilizarmos *function pointers*, não é possível gerar devidamente o gráfico do start_game() (tendo em conta tudo que ele realmente chama), pelo que decidimos incluir o gráfico para o início de cada modo de jogo que temos e o menu principal em adição ao do start_game().

3.30.1 Start Game

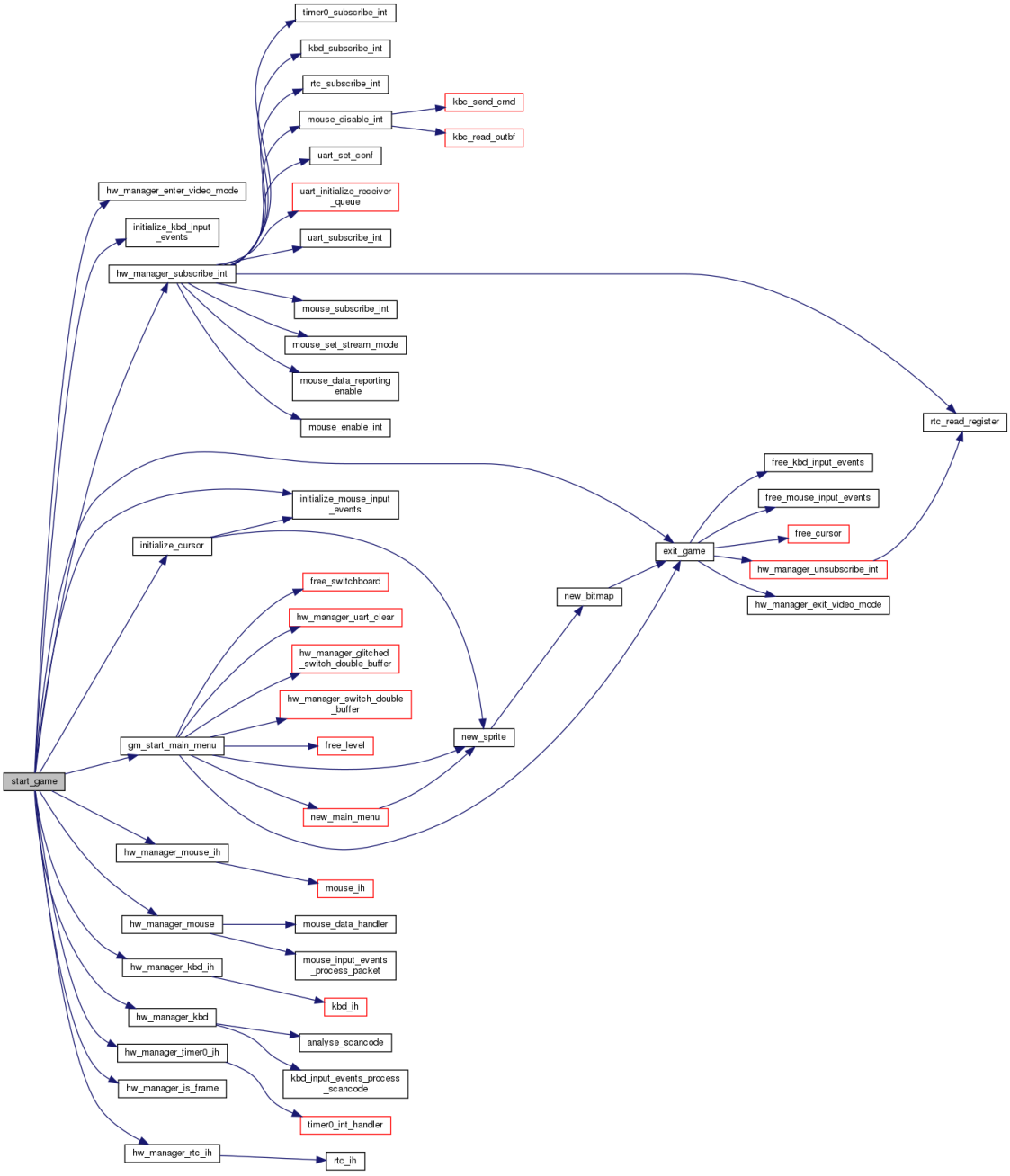

Figure 3.1: Call Graph - gm_start_game()

3.30.2 Menu Principal

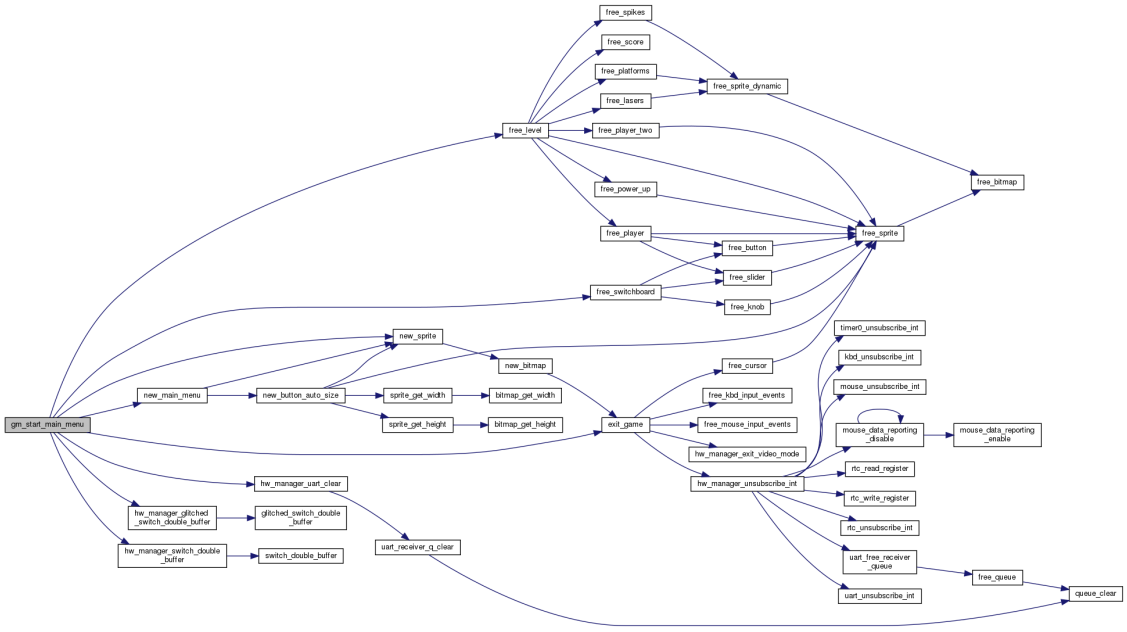

Figure 3.2: Call Graph - gm_start_main_menu()

3.30.3 Start Arcade

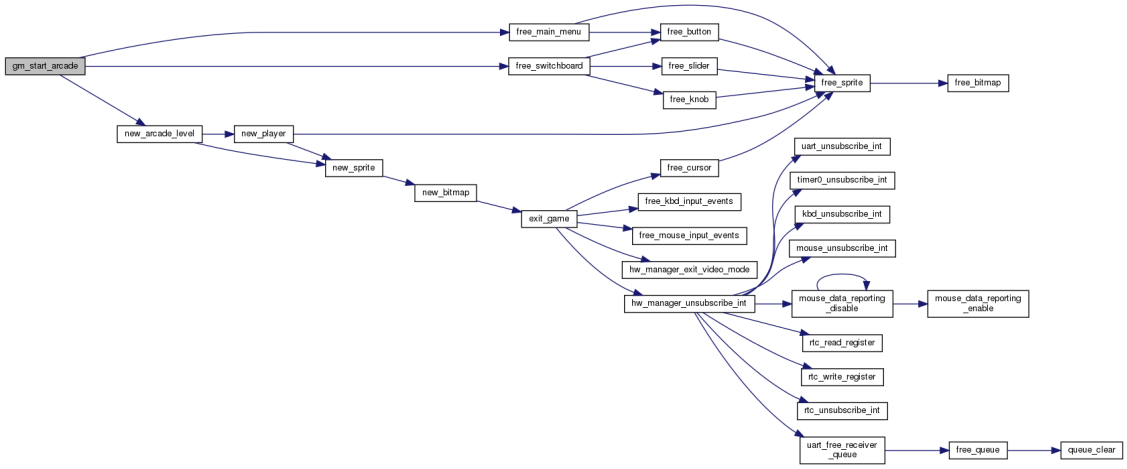

Figure 3.3: Call Graph - gm_start_arcade()

3.30.4 Start Level

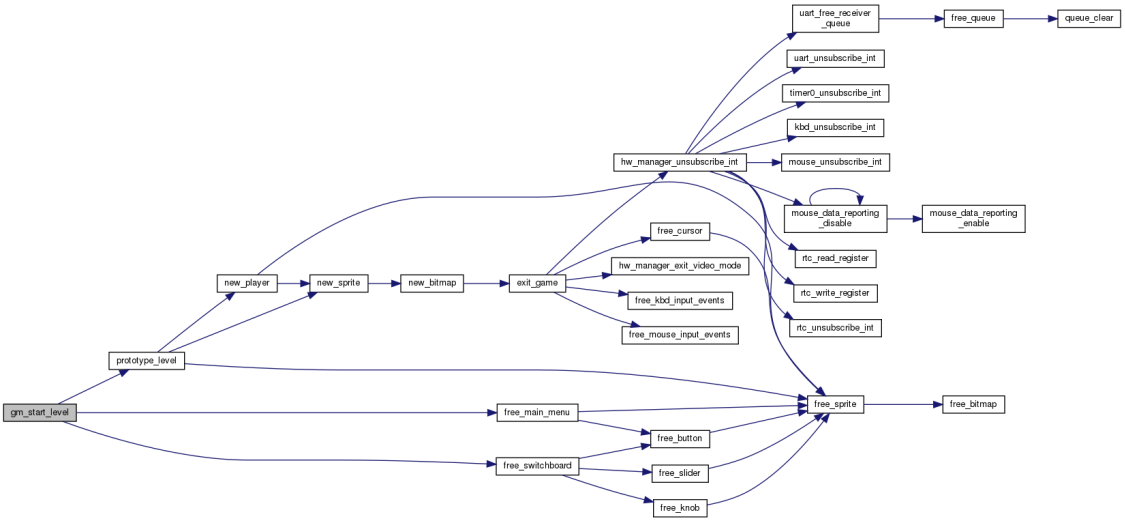

Figure 3.4: Call Graph - gm_start_level()

3.30.5 Start Switchboard

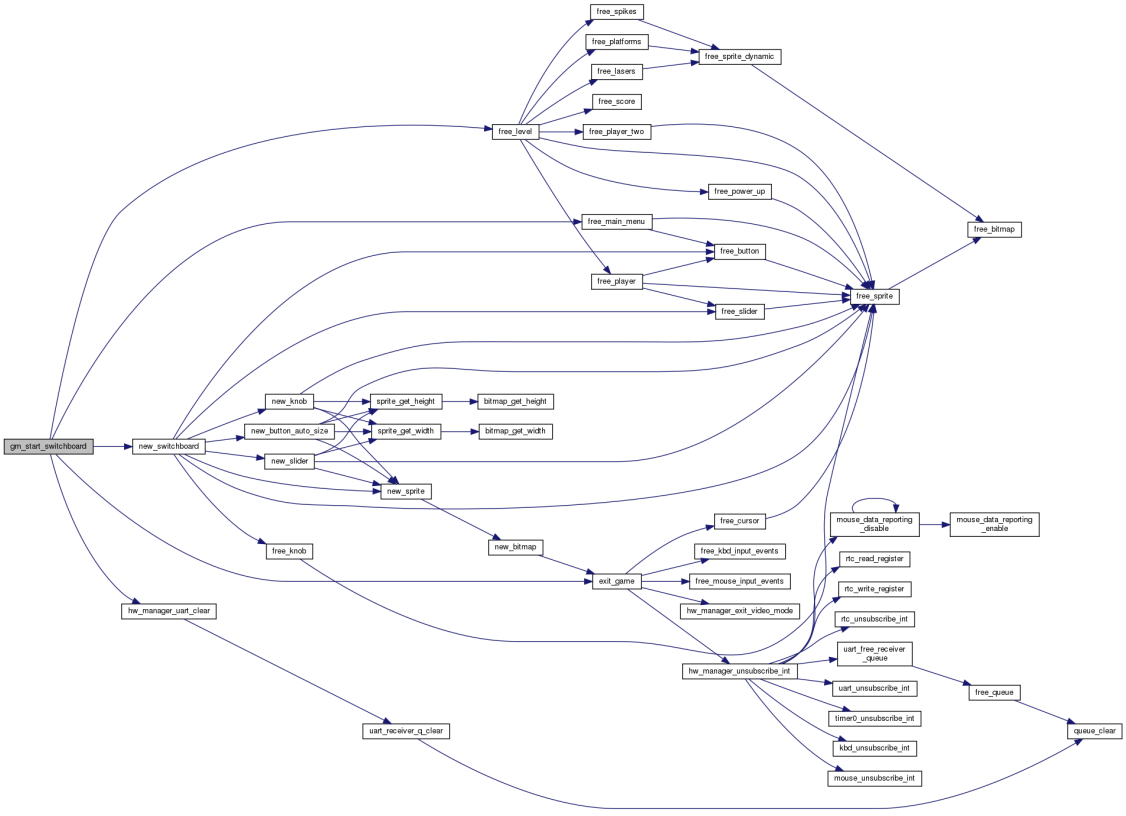

Figure 3.5: Call Graph - gm_start_switchboard()

3.31 Peso Relativo dos Módulos no Projeto

Módulo	Peso relativo (%)
Geometry	5
Bitmap	3
Sprite	5
Game Manager	9
Hardware Manager	1
Level	8
Platforms	3
Lasers	5
Spikes	1
Switchboard	3
Player	7
UI Elements	6
Powerup	2
Timer	4
i8042	0.5
Keyboard	4
Mouse	4
i8254	0.5
Video	6
Video Macros	1
RTC	2
UART	5
Queue	4
InputEvents	3
MouseCursor	2
Utils	1
Math Utils	1
Main Menu	3
Proj	1
Total	100

Chapter 4

Detalhes de Implementação

4.1 Geometry

O módulo *Geometry* é dividido em duas partes, ***Vec2d*** e ***Rect***, e serve como base do nosso sistema de coordenadas, físicas, hitboxes e colisões.

4.1.1 Vec2d

Este módulo é usado para representar **vetores** de duas dimensões, e consequentemente, **pontos** no espaço (não se trata de nada mais que o vetor aplicado à origem do referencial). Cada ***Vec2d*** é composto pelas suas componentes **horizontal** (x) e **vertical** (y), seguindo o referencial do *frame buffer* (x no sentido da esquerda para a direita e y no sentido de cima para baixo).

Existe um conjunto de **operações** que podemos efetuar com esta classe, nomeadamente somar — *sum_vec2d()*, subtrair — *subtract_vec2d*, multiplicar por um escalar —multiply_by_scalar_vec2d()), obter a norma do vetor — *norm_vec2d()*, o ângulo entre dois vetores (usado para a *Knob*) — *angle_vec2d()*, produto escalar — *internal_prod_vec2d()*, a distância entre dois pontos do plano — *distance_vec2d()*, calcular a posição de um ponto em coordenadas polares — *circumference_vec2d()*, entre outros (todas estas funções encontram-se expostas no *Doxygen* deste módulo).

4.1.2 Rect

A classe *Rect* representa, ao seu nível mais simples, *retângulos*. Eles são usados para representar a ***hitbox*** tanto dos diversos elementos do jogo (*Player*, *Lasers*, *Platforms*, ...), como dos *UI elements* (que área do ecrã é dedicada a ativar um botão, por exemplo).

Cada *Rect* é composto pelas **coordenadas** do seu ponto superior esquerdo, tanto como pela sua **largura** e **altura**. Existem vários construtores disponíveis com propósitos diferentes: *rect()*, *rect_from_vec2d()*, *rect_from_points()* (gera o retângulo definido por dois pontos).

4.1.3 Sistema de Colisões

O nosso sistema de colisões foi construído pensando em várias **camadas**, isto para permitir que um determinado objecto só colida com os necessários, como, por exemplo, podemos querer verificar se um jogador colide com uma plataforma, ficando parado no sítio onde colidir, ou se morre por colidir com *lasers* — *lasers_collide_player()* ou espinhos — *player_touches_spike()*. Isto seria dificultado se estivessem todos juntos 'na mesma camada'.

4.2 Rendering

O nosso sistema de ***rendering***, tal como o nosso sistema de colisões, foi pensado como tendo várias **camadas**, isto é, sabemos que temos que desenhar o ***Player*** por cima do ***background***, e que temos que

desenhar os ***Lasers*** por cima deles.

A solução encontrada para esta questão foi dedicar diversas funções de ***rendering*** ao *GameManager*. Isto permitiu-nos renderizar um nível em *Campaign* — *render_level()*, a *Switchboard* — *render_switchboard()* num dos computadores em *Campaign - Co-op*, os níveis e os diversos contadores (*Score*) — *render_score()* em modo *Arcade* — *render_arcade_single()* — *render_arcade_versus()*, etc.. O método de trocar entre estas funções de *render* será abordado quando analisarmos o ***GameManager***.

Outro aspeto peculiar é podermos aplicar **filtros** sobre o ecrã inteiro sem diminuir drasticamente a **performance**. É possível definir várias funções extra para assumirem o papel de copiar do ***buffer* temporário** para o ***frame buffer***, que usam efeitos especiais, como foi realizado na *Switchboard* para o efeito de o ecrã aparenter ter 'estática' na imagem (só quando o utilizador não cumpre uma tarefa corretamente) — *glitched_switch_double_buffer()*.

4.2.1 Bitmap

O código que usamos ler **bitmaps** foi parcialmente retirado da internet (do seguinte site: http://difusal.blogspot.com/2014/09/minixtutorial-8-loading-bmp-images.html), mas com várias alterações (descritas em baixo) para melhor se ajustar ao tipo de ***bitmaps*** que usamos (criados no GIMP).

Todos os ficheiros externos necessários (*.bmp*) encontram-se organizados dentro de uma pasta ***assets***, cuja localização pode ser especificada ao iniciar o jogo (até um limite de 255 carateres, incluíndo os ficheiros no interior dessa pasta).

Com o intuito de reduzir a quantidade de *bitmaps* criados, ao desenhar no ecrã cada *bitmap* **"normal"**, existe a opção de o desenhar alinhado à **esquerda**, ao **centro** ou à **direita** do ponto do ecrã indicado num argumento. Para além desse argumento existe ainda a possibilidade de desenhar o **simétrico** de cada *bitmap* segundo quer o seu eixo (centrado no *bitmap*) horizontal, vertical ou ambos. Esta segunda ***feature*** foi especialmente útil e importante para as **animações** do jogador.

Ao desenvolver o projeto considerámos necessário uma maneira de poder renderizar todas as nossas plataformas, paredes, espinhos e lasers de uma maneira mais **dinâmica**, para evitar criar uma nova sprite manualmente por cada objeto novo dessas classes. Assim implementamos uma função *bitmap_draw_dynamic()* que nos permite criar *bitmaps* bastante reduzidos e que ocupam extremamente pouca **memória RAM**. Cada um destes *bitmaps* pode ser subdividido em **9 quadrados**, todos de tamanho igual, específico de cada imagem. A função irá depois reproduzir cada secção apropriadamente para produzir no ecrã uma imagem do tamanho pretendido.

Existe ainda a possibilidade de cada *pixel* ser "multiplicado" por uma cor. Devido ao custo pesado de computação desta operação, decidimos que usar uma operação ***bitwise*** entre a cor do *pixel* e da cor a multiplicar, visto que o efeito que ela produz seja satisfatório o suficiente para as nossas necessidades, mantendo um baixo **custo computacional**. Esta operação irá sempre escurecer a cor do *bitmap* original, aplicando-lhe a cor multiplicada, e quando aplicado a um pixel branco irá ficar apenas a cor a multiplicar. Esta propriedade foi usada a renderizar todas as plataformas e paredes.

Todos os *bitmaps* podem ser desenhados com **"transparência"**, isto é, definimos a cor rosa, **0xFF00FF** (em RBG 888), como a cor que será sempre ignorada ao desenhar.

O *path* para o diretório *Assets* (onde se encontram todas as imagens), pode ser alterado como um *command line argument* ao correr o projeto.

4.2.2 Sprite

No nosso programa existem duas classes de *sprite*: a **normal** e a **dinâmica**. A **dinâmica** é usada sempre que quisermos usar a função *bitmap_draw_dynamic()* referida acima, a **normal** é usada em todos os outros casos.

Foi tomada a decisão de implementar as animações seguindo o popular ***Unity***. Este *game engine* tem uma propriedade *Animator* que pode ser acoplado a qualquer objeto com um *Renderer*. O *Animator* é apenas uma máquina de estados que indica, no momento de renderizar o objeto no ecrã, qual a *Sprite*, textura ou *3D Model* a usar.

O sistema que decidimos implementar está imbutido diretamente na nossa classe *Sprite*, que permite criar cada objeto com até 255 *bitmaps*. Para permitir a instanciação de tantos *bitmaps*, recorremos a um construtor

que aceita um número **variável** de argumentos — *new_sprite()*. A nossa classe tem uma propriedade "estado da animação", que indica qual dos *n bitmaps* a desenhar. Este **estado** pode ser obtido através de um *getter* — *get_animation_state()* e alterado através de um *setter* — *set_animation_state()*.

A **máquina de estados** responsável por controlar qual o valor deste estado (por *default* este valor é 0, ou seja, o primeiro *bitmap*). Este tema será abordado novamente ao discutir o *player*.

Cada *sprite* criada também tem um ***offset*** individual que é utilizado ao renderizar o *bitmap* no ecrã. Existe ainda um getter das *sprites* — *sprite_get_size()* que permite obter o tamanho do primeiro *bitmap* carregado, especialmente útil para obter dinâmicamente a *hitbox* do *player* e dos botões.

4.3 User Inputs

4.3.1 KbdInputEvents

Com o propósito de criar um ***dispatcher*** eficiente, foi usado um ***map*** para mapear cada *scancode* ao seu valor respetivo. É registada sempre se uma tecla está a ser pressionada e se uma tecla foi pressionada durante este frame (isto é, se o utilizador empurrou para baixo a tecla).

A função *kbd_input_events_process_scancode()* trata de interpretar qualquer scancode que seja recebido do *keyboard*. Visto recebermos mais de 90% do teclado (incluindo numpad), estamos a um pequeno passo de distância de permitir o utilizador escrever texto no nosso programa.

Para mais é permitido um acesso mais simples a estas informações a partir de qualquer ponto do programa, esta classe foi tornada um *singleton*, com dois *getters* para saber se uma **tecla** se encontra premida — *get_key()*, e se uma **tecla** foi premida neste frame, *get_key_down()*.

O *KeyboardMap* utilizado como argumento é um ***enum*** com todas as teclas suportadas. Isto também permite que se o layout do teclado for diferente do *default*, os scancodes podem ser alterados apenas nesse *enum*.

Para detetar se uma **tecla** foi premida num certo ***frame***, foi necessário limpar esse segundo *map* no final de cada *frame*, isto é, após a chamada da função *update()*, para garantir que os inputs são interpretados corretamente, recorrendo à função *reset_kbd_input_state()*.

4.3.2 MouseInputEvents e MouseCursor

Para permitir saber se um dado **botão do rato** começou a ser premido num determinado *frame* e para garantir que o ***delta*** (deslocação) do rato entre *frames* é o correto, recorremos à mesma ténica do *KbdInputEvents*, utilizando a função *reset_mouse_input_state()*.

4.4 Graphical User Interface

Este módulo é composto pelas classes *Button*, *Slider*, *Knob*, *Number* (usado internamente para o *Score*) e o próprio *Score*.

Todos os **elementos de *UI*** com que o utilizador pode interagir (botão, *sliders* e *knob*) têm as propriedades de *hidden* e *active*. Quando a propriedade ***hidden*** se encontra ativa, não é renderizado e não é possível o utilizador interagir com ele. Quando a propriedade ***active*** se encontra true, ele é renderizado e é possível o utilizador interagir com ele (se não estiver ativo, é renderizado com uma cor mais **escura** e o utilizador não pode interagir com ele, propriedade usada quando um jogador não tem acesso a um *power up*).

Estes possuem ainda um estado de **ativo** ou **inativo** e a qualquer momento é possível mudar o seu estado — *set_activation()*. Caso um elemento esteja **inativo**, a sua cor aparece ligeiramente menos **saturada** e não é possível interagir com o mesmo, ou seja, se for um *Button*, não é **clicável**, se for um *Slider*, não é **arrastá-lo** e, por último, se for uma *Knob*, não é possível **rodá-la**. Se estiver ativo, a sua aparência é **normal** e já é permitida a **interação** com o utilizador.

Sempre que o utilizador passa o **cursor** por **cima** de algum elemento de *UI* com que ele possa interagir, ele será rendered com uma saturação ligeiramente diferente, com o propósito de dar *feedback* visual ao utilizador de que ele pode interagir com o elemento — *render_button()* — *render_slider()* — *render_knob()*.

Um **desafio** inesperado (mas óbvio) que surgiu neste módulo foi que quando o utilizador "agarra" **num elemento de UI**, por exemplo o *Slider* ou a *Knob*, temos que ter em conta o *offset* desde o ponto que indica onde uma *Sprite* é desenhada e o **ponto** onde o utilizador clicou, tendo que fazer sempre um **cálculo** para garantir que interpretamos e renderizamos corretamente as ações do utilizador.

4.4.1 Button

Tal como o nome indica, esta "classe" representa um simples **botão** com uma **aparência** desejada e formato **retangular** que pode ser **clicado** para ativar alguma funcionalidade.

Caso esteja **ativo** e o utilizador clicar nele com o **cursor do rato**, ativa a sua **funcionalidade**.

Para este propósito, cada botão pode ser instanciado com a sua pŕopria *Sprite*, especificadando **manualmente** o **tamanho** do botão — *new_button()* ou auto-definindo-o como o tamanho da própria *Sprite* — *new_button_auto_size()*. Pode ainda ser instanciado usando uma *Sprite* já **definida**noutro ponto do jogo (para evitar carregar os mesmos *bitmaps* em memória e duplicados).

Cada **botão** também tem definida uma ***hitbox***, isto é, a área do ecrã em que o cursor pode ativar o botão. Ao instanciar um botão temos a escolha de ajustar esta *hitbox* automaticamente ao tamanho da *Sprite* (como foi falado em cima).

Para os botões provocarem alterações no código, cada um deles tem um **apontador para função** —*void (*func)()*, que é chamado quando o utilizador **clica** nele.

4.4.2 Slider

Um ***Slider*** consite numa **barra** com um **manípulo** no seu interior que pode ser **deslizado** e colocado numa posição que provocará um efeito de ***magnitude*** proporcional à posição do manípulo no *Slider*, relativamente à sua extremidade. Este *Slider* é composto por uma *sprite* de *background* e outra para o *handle* (parte com que o utilizador interage).

Cada *slider* tem um ponto inicial e final, que podem estar alinhados em qualquer direção (horizontal, vertical, diagonal), desde que o inicial seja o ponto mais **acima** e à **esquerda**. Esta restrição não traz qualquer problema, pois quando implementámos o *slider* **vertical** para o **multiplicador do salto**, em que o início deveria ser em **baixo**, simplesmente interpretamos o input ao contrário (255 passa a 0 e vice-versa).

Tal como o botão, o slider também tem um apontador para função — *void (*func)(uint8_t)*, que é chamada sempre que o utilizador **larga** a *handle*, enviando como argumento uma **aproximação** entre [0, número de estados máximos] (normalmente 255) da **posição** em que o utilizador **largou** o *handle*, desdo o ponto inicial até ao ponto final.

4.4.3 Knob

Uma espécie de **"maçaneta"** que pode ser **rodada**, clicando na sua **extremidade** com o **botão esquerdo** do rato e **arrastando-a**, ativando algum **efeito** durante o tempo que ocorre entre o instante em que é **rodada** até rodar de **volta** à sua posição inicial.

Uma *knob* tem um ***handle***, isto é, a **componente** em que o utilizador **clica** e **arrasta** pelo ecrã, um apontador para função chamado quando o utlizador larga o *handle* — *on_drop()* e outro quando o *handle* volta à sua posição inicial — *on_reset()*.

Para resumir de forma simplificada o seu **modo de funcionamento**, o utilizador clica no *handle* e arrasta-a na **direção** da posição final (mas é sempre analisado o **ângulo** e nunca a posição absoluta). Quando o utilizador larga o *handle*, ele chama o seu apontador para função *on_drop*. O *handle* irá automaticamente voltar para a sua posição original. Durante este **período** o utilizador não pode interagir com a *Knob*. Quando o *handle* volta à sua posição original, chama o seu apontador para função *on_reset* e permite novamente a interação com o utilizador.

Outra característica adicional, é que o utilizador não pode rodar na direção **errada**, isto é: Se a direção do início para o fim for no sentido do ponteiro do relógio, se o utilizador arrastar no sentido contrário até à posição final, o *handle* não se irá **mover** até o utilizador voltar à posição do *handle* no ecrã.

Este elemento foi bastante **desafiante**, visto a **complexidade** do método usado para determinar o **ângulo** e a **posição** do *handle* dado o **ângulo** determinado. Foram usadas extensivamente as **funções**

associadas ao *Vec2d* de *circumference_vec2d()*, *angle_vec2d()* (que por sua vez necessita do *prod_vec_vec2d()* e do *norm_vec2d()*), *sum_vec2d()*, *subtract_vec2d()*, entre outras.

4.4.4 Number

A classe ***Number*** serve para representar um único algarismo entre 0 e 9, sendo ainda possível incrementá-lo dinamicamente por um unidade — *updade_number()*.

4.4.5 Score

A classe ***Score*** serve para representar um número com um determinado número de algarismos (*Numbers*). É utilizada para mostrar a pontuação atual de um jogador no modo de jogo *arcade*, o tempo restante no modo *arcade versus*, e para exibir o tempo (em segundos) que o jogador demorou a concluir o nível no modo *campaign* (tanto *single player* como *co-op*).

É possível iniciar o *Score* com um **valor** inteiro positivo qualquer — new_score(), bem como repô-lo (colocar a 0) — *reset_score()* ou mudar o seu valor — *set_score()*.

Estas duas classes, *Number* e *Score*, poderiam ser facilmente **expandidas** para renderizar **texto**, bastaria alterar o tipo de dados para *char* e renderizá-lo (a pior parte seria os *Sprites* necessários para compor uma *font*). Quanto à disposição do texto, bastava fazer com que o *Score* deixasse de ter tamanho fixo.

4.5 Level

A nossa classe *Level* é extremamente versátil, o que nos permitiu efetuar tanto a *Campaign* como o *Arcade* usando-o. Originalmente tencionávamos criar um *level editor*, daí a natureza tão modular do *Level*, mas como implementámos o modo *Arcade* para satisfazer melhor os critérios do uso da UART, acabámos por não ter tempo suficiente para o terminar.

4.5.1 Platforms e Spikes

A classe Platforms representa o conjunto de **plataformas** que existirão em cada **nível**, tanto em Campaign — *prototype_platforms()*, como em Arcade — *new_arcade_platforms()*. Estas podem ou não serem **paredes** (is_wall).

Os espinhos são em tudo semelhantes às plataformas exceto pela sua aparência e pelo facto do jogador morrer ao tocar nestes. Estão apenas presentes no nível do modo Campaign.

4.5.2 Lasers

A classe Lasers representa o conjunto de **lasers** que existirão em cada nível.

No modo Campaign estes serão **quatro**, dois azuis, um vermelho e um rosa, só os lasers de uma dada **cor** é que estarão **inativos** de cada vez (podendo-se definir qual — *lasers_set_link_id()*) e não mudam de posição.

Já no modo Arcade, o conjunto de lasers é todo da **mesma cor** (vermelho), estão sempre **ativos** e são **gerados** novos pares de lasers — *arcade_add_laser()*, com uma **altura aleatória** — *arcade_generate_laser_height()* **periodicamente** — *arcade_lasers_set_correct_delay()*, que surgem da ponta **direita** do ecrã — *arcade_spawn_next_laser()*, e se **deslocam** até à **esquerda** do mapa — *arcade_move_lasers()*, com uma certa **velocidade** *arcade_lasers_set_speed()*, sendo **destruídos** quando lá chegam.

A **frequência** a que os lasers **surgem** e a **velocidade** a que eles **movem** são ainda **parâmetros** que vão-se **alterando** conforme o jogador vai progredindo — *arcade_update_laser_values* — *arcade_versus_update_laser_values()*.

Por último, sempre que o *Player* toca num destes *lasers*, estes são **repostos**, tal como a **frequência** a que são gerados e a sua **velocidade** — *arcade_reset_lasers()*, caso contrário, se ele conseguir **passar** por entre os lasers — *arcade_player_passes_lasers()* recebe um ponto.

4.5.3 Player

Físicas

O *Player* é o único objeto ao qual é aplicado um sistema de **físicas** (embora só no eixo **vertical**). O intuito das físicas não foi elas serem **realistas**, mas tal como podemos aprender com os jogos mais bem sucedidos deste *genre* (*platformer*), o mais importante é que elas sejam bastante **satisfatórias** de jogar, algo que acreditamos ter conseguido.

O jogador tem sempre uma **velocidade** vertical (podendo ser nula), e sofre uma **força gravítica** (com **sentido** do eixo positivo ou negativo). Sempre que o jogador está a **"cair"**, isto é, a sua velocidade tem o mesmo sentido que a **gravidade** atual, o efeito da gravidade é **multiplicado**. Se o jogador fosse contra algum **obstáculo** (verticalmente), a sua velocidade é drasticamente **reduzida** de modo a impedir que ele ultrapasse/entre dentro do obstáculo.

Animações

O jogador tem um sistema complexo de **animações**, tendo animações ***idle*** (quando se encontra **parado** e no chão), animações de **andar/correr** e ainda animações de **"faíscas"** que são mostradas exclusivamente quando o jogador ativa a **anti-gravidade**.

Para transitar entre cada animação utilizamos várias **máquinas de estados** (uma para a *idle animation*, uma para a *walk animation* e outra para a das 'faíscas'). As animações vão dando *cycle* entre elas a um ritmo fixo.

Entre a animação de *idle* e *walk* só é renderizada uma de cada vez, portanto temos algumas variáveis responsáveis por essa decisão (inpirado pelo sistema usado pelo *Unity*, tal como mencionado anteriormente, no nosso caso as animações serão geridas pelo *animator_player()* de modo a serem renderizadas de modo correto pelo *render_player_background()* | *render_player_foreground()*). Dependendo se o jogador se encontrar vivo ou morto, e de se ele se encontrar no chão (*grounded*) e sem qualquer input. Para a *Sprite* do player morrer simplesmente reaproveitamos um dos frames da animação de idle, mas recolorida.

Para evitar a duplicação de *Sprites*, utilizamos a vantagem de refletir a imagem sobre si própria para renderizar o jogador em todas as direções, isto é a andar para esquerda, para a direita e para cada uma delas, a andar com a gravidade para baixo (normal) ou para cima (a personagem 'caminha' no teto).

Um detalhe extra que implementámos foi que ao alterar a velocidade horizontal do jogador, alteramos a velocidade da sua animação conforme. Ao reduzir a velocidade ao mínimo a animação é lenta, mas ao colocar a velocidade ao máximo, na animação ele corre extremamente rápido.

4.5.4 PlayerTwo

O PlayerTwo posui a utilidade de representar um **segundo jogador** que está a jogar noutro computador no modo Arcade, completo com a sua animação exata. No modo de jogo *Arcade Versus*, o que num PC é Player será representado como o PlayerTwo no outro.

A cada frame são sempre enviadas via UART infomações do Player para o PlayerTwo, tais como a sua posição, a animação em que ele está, e as informações extra para determinar outros aspetos, como por exemplo a sua direção (horizontal e vertical), se está idle, se está morto, entre outras.

Visto ser o tanto maior 'pacote' que enviamos via UART como o que é enviado com maior frequência, existem casos em que ele não chega a tempo ao outro jogador. Isto implica existirem frames em que o jogador recebe dois pacotes ao mesmo tempo. Para resolver esta situação, só atualizamos a informação do PlayerTwo quando recebemos o seu pacote respetivo. Durante frames em que recebemos múltiplos pacotes, só damos parse ao último recebido.

4.5.5 Campaign

Power Ups

Ao longo do modo Campaign, o **jogador** precisa de obter ***powerups*** para ganhar novas **funcionalidades** de modo a passar o nível. Cada vez que apanha um powerup, desbloqueia o controlo de uma mecânica

adicional e só é possível obtê-los uma única vez (a não ser que o jogador morra, sendo eles repostos na sua posição inicial).

Existem portanto três powerups:

- **Laser** — Após o jogador obter este powerup, torna possível controlar qual das três cores dos lasers está inativa, controlada pelo próprio jogador nos *sliders* do canto superior esquerdo do nível, se estiver a jogar Singleplayer, e pela Switchboard se estiver a jogar Co-Op. Representada por um pequeno computador.
- **Anti-Gravity** — Após o jogador obter este powerup, torna possível alterar o sentido da gravidade durante instantes (passando a ser puxado para o teto) através da **tecla *X*** (em Singleplayer) ou de uma *Knob* (em modo Co-Op) que simultaneamente ilustra o tempo que o efeito de anti-gravidade estará ativo. Representada por uma maçã com um símbolo de uma seta.
- **Exit** — Ao contrário dos outros powerups, a saída não dá qualquer poder ao jogador, mas por uma questão de simplificação do código, ela está implementada como um power_up, visto partilhar todas as características com eles (só pode ser 'apanhada' uma vez por nível, ativa uma função noutro lado para causar um efeito, tem uma Sprite). Representado como uma placa de saída de emergência fundida com o símbolo de ligação à terra.

Singleplayer

A diferença do modo Singleplayer para o modo Co-Op 1 reside no facto de, devido ao facto de já não existir uma *Switchboard*, os elementos de interface gráfica que se encontravam lá situados, terão de ser acoplados no próprio nível. Para isso colocamos sliders mais pequenos no canto, junto com os botões dos lasers, por cima das paredes, de modo a não interferir com a jogabilidade visualmente e subsituímos a Knob pela tecla X.

4.5.6 Co-Op

Para um bom funcionamento do modo Co-Op, é essencial estabelecer um protocolo de comunicação sólida e identificar devidamente a informação trocada, como tal utilizámos *headers* para indicar que informação estava a ser enviada.

Será ainda de destacar que em modo Co-op, na *Switchboard*, temos um minijogo permanente, em que baseado num alarme do RTC, instancia alguns 'ciruitos'/'balões' em que o utilizador tem que clicar para destruir, pois caso cheguem ao final do ecrã, irão ativar temporariamente um filtro de distorção tipo 'interferência' no ecrã. Será de notar que estes 'circuitos' aparecem por baixo da switchboard (truque com a order no renderer e uma *mask*).

4.5.7 Arcade Versus

Neste modo de jogo foi necessário sincronizar bastante informaçao, tal como as suas pontuações, os jogadores e os lasers. A relação e o envio de informação relativos ao Player e ao PlayerTwo já foram descritos anteriormente, mas falta abordar os lasers.

Na 'sincronização' inicial dos dois jogadores é definido um papel a que chamámos *laser master*, que é sempre o primeiro dos dois jogadores a abrir o modo de jogo (e coincidentalmente sempre o jogador vermelho). O computador que ficar com este papel fica responsável por gerir o *spawn* dos lasers tal como descrito no capítulo deles. Este computador é o única que cria os lasers, sabe quando vem o próximo e a velocidade deles. A cada alteração da velocidade dos lasers, o *laser master* envia uma mensagem ao outro a indicar essa alteração (e o seu novo valor). A cada *spawn* de um laser, o *laser master* envia a altura do novo laser ao outro.

O *laser master* é o responsável por terminar o jogo, como tal também envia uma mensagem ao outro jogador a notificá-lo deste evento. Os timers dos dois jogadores no entanto estão implementados a mostrar o tempo usando o número de frames desde o começo do jogo. Inicialmente usávamos leituras ao RTC para obter esse tempo, mas visto que o RTC dos dois PCs mudava o segundo em alturas diferentes (um dos PCs ficava sempre atrasado/adiantado por cerca de 0.5 segundo), o que nos levou à solução descrita acima.

4.6 Game Manager

O *GameManager* é o responsável por coordenar todas as várias componentes do nosso projeto, para não mencionar o facto de o *proj_main_loop()* estar integrado dentro da função do GameManager *start_game()*.

Esta classe é composta por apontadores para os três tipos de classes 'principais', o *Level*, a *Switchboard* e o *MainMenu*. Tem dois *arrays* de apontadores para funções que representam as diversas funções de *update* e *render* que existem. Um membro dado do tipo *GameModeEnum* que representa o modo de jogo atual. Dois apontadores para função que representam as funções de double buffering que desenvolvemos. Diversos *booleans* para sabermos os estado da UART a qualquer altura. Para finalizar, tem uma variável responsável por detetar se o utilizador premiu duas vezes a tecla Esc num curto intervalo de tempo.

4.6.1 Singleton

Como cencetualmente só pode existir um *GameManager*, para permitir que o restante código seja mais simples e tenha fácil acesso ao *GameManager*, decidimos torná-lo num ***Singleton***. Esta técnica muito usada na indústria dos vídeo jogos implica que só pode existir uma única instância do *GameManager* na aplicação inteira. Para isto o GameManager é definido como um *static GameManager *gm*, e para aceder ao GameManager fora do ficheiro game_manager.c é utilizada a função *get_game_manager()*, que retorna um apontador para o *GameManager* e inicializa-o se ele ainda não o tiver sido.

4.6.2 GameModeEnum

Este enum representa sempre o modo de jogo atual, através da atribuição de significado a cada bit. Ou seja, sendo o BIT(0) o bit que indica se a UART é utilizada ou não, o BIT(1) a indicar se é um *Level* normal (*Campaign*). Assim podemos identificar unicamente o modo de jogo atual com um *bitwise and*.

Mode de jogo 2 (b10) indica que nos encontramos em *Campaign Singleplayer*, mas modo de jogo 3 (b11) indica que nos encontramos em *Campaign Co-op Player 1*.

4.6.3 Update e Render

Devido a cada um dos modos de jogo ter um método de atualizar a informação e de a renderizar diferente, adotamos o método de ter um *array* de apontadores de funções cujo índice é o próprio *GameModeEnum*. Isto faz com que a chamada da função estática *update()* e *render()* cada uma chame dentro dela a função correta para o modo de jogo atual.

Assim evitamos a necessidade de um *switch case* todos os frames para determinar qual a função correta. Para diminuir a probabilidade de *segmentation faults*, as funções do array que não têm um valor possível são preenchidas com um apontador para uma função void vazia.

4.6.4 Mensagens recebidas via UART

Todas as nossas mensagens são pacotes de tamanho irregular que começam num *header* predefinido e terminam num terminador *Header_Terminator*. A menor mensagem tem tamanho 2 e a maior 8. Após cada mensagem limpamos toda a *Receiver Queue* até ela estar vazia ou encontrarmos um terminador. Propositadamente, se algum erro ocorrer na *Queue*, receberemos o valor 0xFF (o valor de *HEADER_TERMINATOR*), portanto é um mecanismo de prevenção de erros. Se recebermos um *header* que não estávamos à espera, essa mensagem é simplesmente eliminada.

Quando é iniciado um novo modo de jogo, a *receiver Queue* do RTC é limpa, para garantir que não vamos ler mensagens antigas.

As mensagens recebidas via UART são processadas apenas pela função de *update* correta, isto é, o programa ignora as mensagens de que ele não está à espera (por exemplo receber uma mensagem da *Switchboard* em *Arcade Mode*). Existem duas etapas em cada função de *update*: Uma primeira etapa em que os *serial ports* ainda não estão 'sincronizados' (por isto entenda-se que não estão juntos no mesmo *Gamemode*), depois de eles estarem 'sincronizados' ele espera um conjunto diferente de mensagens da *UART*.

Em cada frame lemos e interpretamos o maior número de mensagens possíveis. Para garantir que não lemos uma mensagem incompleta no entanto, verificamos sempre se a Queue tem pelo menos o tamanho

do 'pacote' que estamos a receber. Nesses casos deixamos o pacote na Queue e continuamos o jogo com a informação atual, para no frame imediatamente a seguir, essa informação seja processada.

4.6.5 Main loop

O *main loop* está dentro da função *start_game()*. Esta última função inicializa todas as classes necessárias (*InputEvents*, *MouseCursor*, *GameManager*), subscreve a todas as interrupções, inicializa uma seed aleatória no *srand()*.

Durante o ciclo do *driver_receive()*, processamos o mais rápido possível os interrupção e aquando de uma interrupção do *timer 0*, chamamos as funções *update()*, *render()* e a função para tratar do *double buffering*, nesta ordem.

Ao terminar o *start_game()* dá *unsubcribe* aos interrupts, ele liberta a memória ainda ocupada (por exemplo pelos *InputEvents* ou pela *Queue*) e termina o jogo.

4.6.6 Outros aspetos

A classe *GameManager* também é responsável por garantir que toda a memória ocupada é libertada assim que possível, de modo a evitar memory leaks (que acreditamos ter corrigido todas) e *segmentation faults*. Caso aconteça algum *segmentation fault*, ao correr o programa novamente ele não irá conseguir subscrever aos interrupts, mas implementámos uma verificação nesse caso em que pede ao utilizador para correr novamente o programa (pois resolvemos o problema nesta 'iteração' do jogo).

A mecânica de premir Esc duas vezes consecutivamente para ou voltar ao *MainMenu* ou sair do jogo é manuseada pelo *GameManager*. Dependedo do *GameModeEnum* atual ele faz a ação correta.

Seguindo a indicação do docente Souto, decidimos permitir que o *path* absoluto para o diretório dos *assets* seja definida como um *command line argument*. Por omissão é utilizado o *path* que usámos durante o desenvolvimento do programa.

4.7 UART

O UART foi implementado usando FIFOs e interrupções, com a ajuda de uma Queue de receção a nível de software (abordada na secção seguinte). Manualmente configuramos ambas os serial port — *uart_set_conf()* a usarem a mesma configuração no início do programa: 8 bits de transmissão, paridade ímpar e dois stop bits. Para verificar as configurações também foi desenvolvida uma função para as imprimir de uma forma *human readable* — *uart_print_conf()*.

Decorreram diversos problemas durante o desenvolvimento deste módulo, por exemplo subscrevíamos a interrupções de receiver e do THR vazio, mas não recebíamos sempre a interrupção do THR vazio, a não ser que tentássemos descobrir se o THR se encontrava vazio durante outros processamos (sem ter recebido a notificação). Nesses casos o IIR indicava existir a interrupção, mas ela não era recebida.

Em grande parte por causa deste facto não conseguimos implementar o serial port com todas as verificações que desejaríamos, mas pelo menos conseguimos pô-lo bastante estável. Ocorreram mais problemas enquanto não descobrimos um erro crítico com a Queue que usámos (abordado no próximo capítulo), mas depois de corrigida não ocorreram mais problemas.

Para qualquer parte do nosso projeto poder enviar um pacote via serial port com facilidade, utilizamos o wrapper do hw_manager() — *hw_manager_uart_send_char()* para enviar um uint8_t de cada vez.

Uma característica que decidimos implementar é detetar quando um dos jogadores se desconectou, seja por fecharem a *virtual machine*, perderem a conexão ou voltarem ao menu principal.

Ambos os PCs têm dois contadores, um que conta o número de frames desde a última mensagem recebida — *uart_receive_char()*, e outro desde a última mensagem enviada. Se a algum momento a última mensagem recebida ultrapassar o tempo predefinido (macro *UART_DC_TIME*, atualmente 3 segundos), volta ao menu principal, visto o outro jogador ter-se desconectado por algum motivo. Para prevenir que, por exemplo, no modo *Campaign Co-op*, em que as mensagens enviadas podem ter um intervalo de tempo superior a 3 segundos entre elas, definimos também que se o tempo desde a última mensagem enviada ultrapassar metade do tempo máximo da de receção (neste caso seria de 1.5 segundos), enviará um *header* especial dedicado só a

assegurar o outro PC de que ainda estamos conectados. Estas interações são descritas pelo valor da variável *synced* do *GameManager*.

4.7.1 Queue

Inicialmente fomos ver a implementação da Queue a um website (https://codeforwin.org/2018/08/queue-implementation-using-linked-list-in-c.html), mas ela estava bastante longe das nossas necessidades. Como tal, tivemos que reescrever grande parte dela, alterando-a para permitir mais do que uma Queue por projeto, criando uma estrutura Queue que funcionasse mesmo como uma Queue, resolvemos diversos problemas de lógica, segmentation fault e de memory leaks. Em geral, pouco se pode aproveitar da Queue que vimos originalmente, mas decidimos mencionar em que foi encontrada a implementação para respeitar os requeridas deste projeto.

Um dos maiores problemas dessa implementação (que foi referido na secção do serial port) era que ao dar pop() da queue quando ela tinha um elemento levava a que ela apagasse o nodo da lista ligada (implementação interna da Queue) mas não alterava o apontador rear para NULL, o que mais tarde (ao reinserir nesse local de memória) causava ou segmentation faults ou reescrevia dados de outro local qualquer do projeto (por exemplo chegou a alterar consecutivamente o dynamic sprite size do SpriteDynamic para um endereço de memória, o que os impedia de serem renderizados). Visto quão comum a operação de dar pop() do único elemento da Queue é, o erro teve proporções muito elevadas.

Após desenvolver a Queue, planeavamos ter uma *Queue* (a nível de software) para receção e outra para transmissão, mas dado os desenvolvimentos da UART, foi decidido usar apenas a queue de receção.

Implementámos as seguintes operações base duma Queue: queue_pop(), queue_push_back(), queue_front(), queue_back() (visto que a nossa implementação de listas ligadas o permite), queue_is_empty(), queue_is_full(), queue_clear(). Por default, o construtor cria uma queue vazia.

No que toca à receiver Queue, destas operações, decidimos expor apenas algumas delas através de wrappers do hw_manager, com o prefixo *hw_manager_uart_q_*: pop(), front(), is_empty(), size() e clear().

4.8 RTC

Neste projeto utilizamos o RTC maioritariamente em três vertentes diferentes. Como "Timer" no modo de jogo Campaign, lendo a hora a que este se inicia e acaba para calcular o tempo decorrido (para facilitar as contas, decidimos fazê-lo em segundos), para realizar uma contagem decrescente no modo de jogo Arcade - Versus, colocando um alarme para o final período de tempo que pretendemos contar e por último, para coordenar os instantes a que são geradas novos "balões" na Switchboard, através de vários alarmes.

Para tal, usamos sobretudo as funções *get_date()* e *rtc_set_alarm()*.

Elaboramos ainda diversas funções auxiliares, nomeadamente para ler e escrever dos registos do RTC (*rtc_read_register()* — *rtc_write_register()*) e para converter BCD em decimal — *bcd_to_dec()* e vice-versa — *dec_to_bcd()* (uma vez que a informação obtida pelo *get_date()* pode estar em BCD e desejamos interpretá-la como decimal ou porque queremos dar set de um alarme que nos é passado em decimal e temos de o passar para BCD para voltar a escrever no registo).

Como na leitura da data podem surgir erros (a data pode alterar-se, enquanto a lemos) fazemos uma verificação se esta se encontra em atualização e só então, se não estiver, é que a lemos.

Chapter 5

Conclusões

Para concluir o nosso projeto, gostaríamos de realçar o que achámos de pior e melhor na cadeira de Laboratório de Computadores, constatar quais foram as nossas maiores dificuldades na elaboração do projeto final e fazer uma avaliação geral do nosso aproveitamento desta.

Em primeiro lugar, quanto aos maus aspetos desta unidade curricular gostaríamos de ressaltar o aspeto que toca à organização da informação que nos era fornecida tanto nos guiões dos *Labs*, como nas transparências das aulas.

Frequentemente tínhamos dificuldade em encontrar o que procurávamos, a documentação nem sempre estava completa e a navegação pela mesma era confusa, sugerimos portanto uma melhoria nesta vertente.

Por outro lado, a cadeira possuiu ainda um lado bastante positivo, na medida em que: Aprendemos novas abordagens e técnicas da informática, aprofundámos o nosso conhecimento de programação a baixo nível e pela primeira vez até agora no curso, tivemos um projeto de maior dimensão e também com uma maior liberdade para explorarmos ao máximo as potencialidades do que aprendemos nas aulas.

Quanto às nossas maiores dificuldades, estas foram principalmente a implementação da UART, uma vez que os Interrutps *THR Empty* nem sempre funcionavam devidamente e devido ao facto do seu debug ser relativamente difícil (o que se aplica na realidade ao projeto inteiro).

Em suma, achamos que tiramos um bom proveito desta cadeira e que nos vai expandir os horizontes no futuro, gostaríamos portanto de agradecer aos docentes da cadeira pela sua prestação a leccionar, pelo esclarecimento das dúvidas que foram surgindo ao longo do semestre e pelo acompanhamento nas aulas práticas. Quanto a este último tópico, fomos constantemente coordenando com o professor os aspetos a melhorar a nível de *hardware*, tendo ele salientado o facto de não enviarmos informação em quantidade e frequência o suficiente pela UART. Cremos ter corrigido essa questão com a implementação do modo Arcade - Versus e de resto, como não foram apontadas mais falhas relevantes, acreditarmos ter cumprido com as nossas expetativas para este projeto.