

DEMON SLAYER

“O Caçador de Demónios” - Inspirado na série de jogos Castlevania.



Trabalho Realizado por:

José Guerra. up20170621@fe.up.pt

Rúben Almeida. up201704618@fe.up.pt

Unidade Curricular:

LCOM

Ano Letivo:

2018/2019

Regente:

Prof. Dr. Pedro Souto (PFS)

Data de Entrega:

2019/06/01

Índice:

Pag:

User Instructions	3
Interface e Menus:	4
Multiplayer	10
Comandos	12
Project Status	13
Timer	13
Void update_variables	14
Keyboard	16
Mouse	18
Mouse – GUI interface	20
RTC	24
Graphics Card	26
Formato BMP e Transparências	26
Double Buffering e Flip Display	27
Colisões	28
Colisões pixel-por-pixel	28
UART/SERIAL PORT	32
Codificação das mensagens	32
Interação com os FIFOs	34
Quadro – Resumo	36
Code Organization/Structure:	37
Memória dinâmica	40
Function call graph	42
Implementation Details	43
Conclusions	46

User Instructions:

Objetivo do jogo

Demon slayer é um jogo de arcade em que o objetivo principal do jogo é matar diversos de demónios e fantasmas que vão aparecendo com o passar do tempo ao jogador. A personagem principal encontra-se a correr por predefinição podendo também voar e saltar.

O jogo é inspirado na série de jogos “Castlevania”. Desta forma, os recursos (imagens) que utilizamos são provenientes destes jogos. Os programadores realizaram apenas adaptações gráficas ao contexto em questão. **Não somos detentores da propriedade intelectual das mesmas.**

Personagens:



Personagem principal, “*Demon slayer*”

Inimigos:



Axe taurus



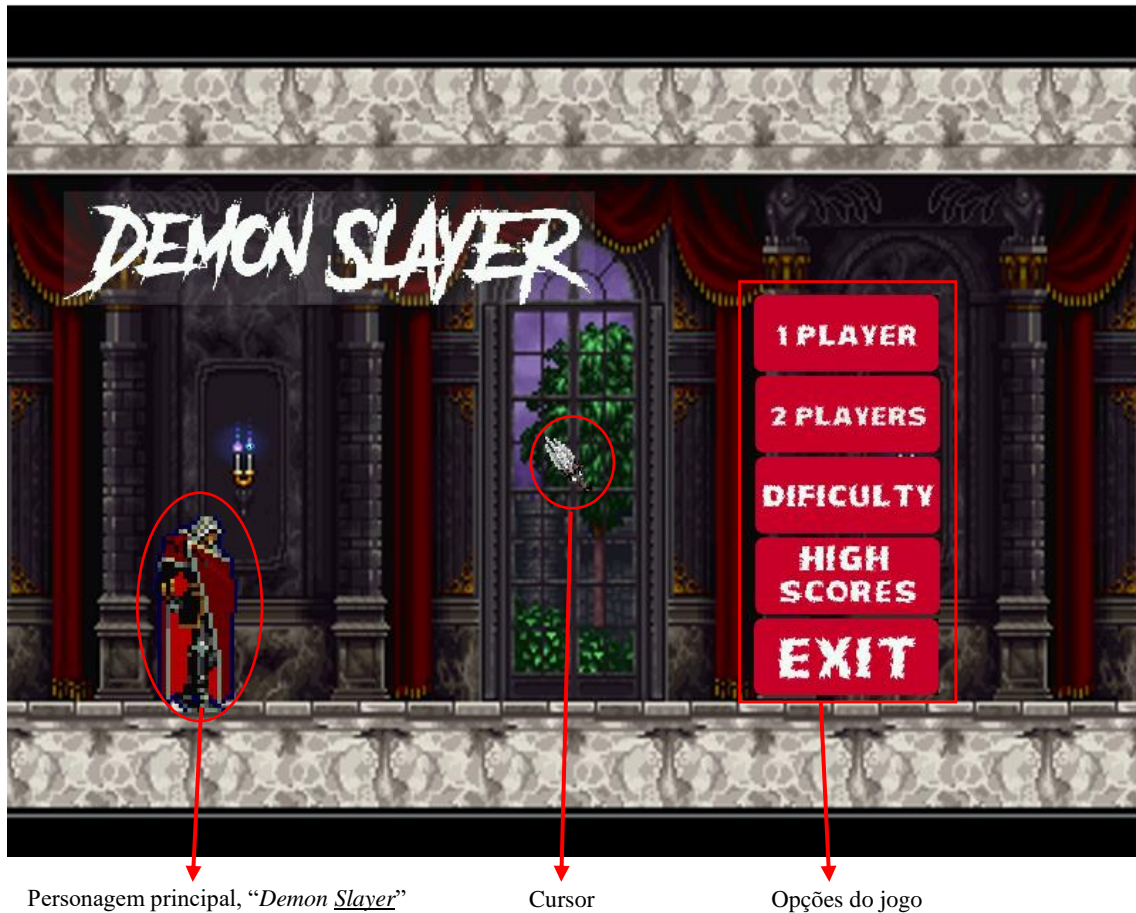
Fantasma



Helldog

Interface gráfica (GUI)

- Main menu:



Quando o jogador inicia o jogo, visualiza, em primeiro lugar, um main menu que contém 5 opções distintas:

- **1 player (1 jogador)**
- **2 players (2 jogadores)**
- **Difficulty (dificuldade)**
- **Highscores (Scores altos)**
- **Exit (Sair)**

Qualquer opção é acessível com o rato, nomeadamente, pressionando o botão esquerdo do rato quando este se encontra por cima da opção. Esta faceta do jogo é válida também para qualquer menu que haja no jogo. Para além disso, é de realçar que mal o jogador entra no jogo é logo introduzido à personagem principal, o “*Demon Slayer*”, que se encontra parado, pronto para entrar em ação.

O jogo suporta duas modalidades:

- **Singleplayer**, que se inicia pela escolha da opção “Um player”. Que se trata, tal como o nome indica, de um modo local de um jogador.
- **Multiplayer**, inicializado, somente pela conexão de duas máquinas autónomas à mesma rede local, clicando na opção “Dois player” em ambas.

Nota: A descrição dos referidos modos é feita mais à frente, quando se descreve cada um deles com detalhe (PAG 7).

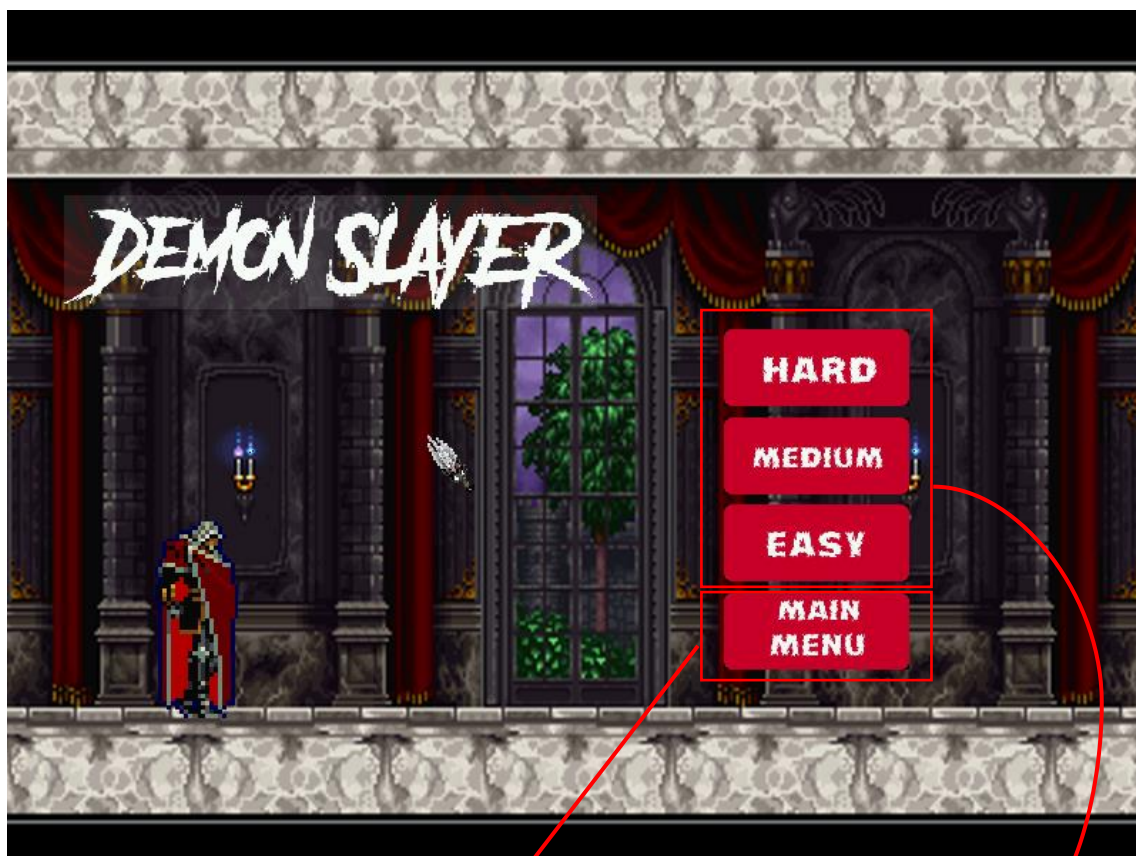
A opção *Difficulty* (dificuldade) leva o jogador para o menu de dificuldades, enquanto que opção *Highscores* leva o jogador para o menu de *Highscores*. A opção Exit faz com que o jogador saia do jogo.

Menu de dificuldades

O menu de dificuldades é acessível apenas através do *Main Menu* e permite seleccionar 3 dificuldades diferentes:

- **Hard (difícil),**
- **Medium (médio)**
- **Easy (fácil).**

Existe também um botão para regressar ao *Main Menu* caso o jogador não queira alterar a dificuldade do jogo. Alterada a dificuldade, o jogador é redirecionado para o *Main Menu*.



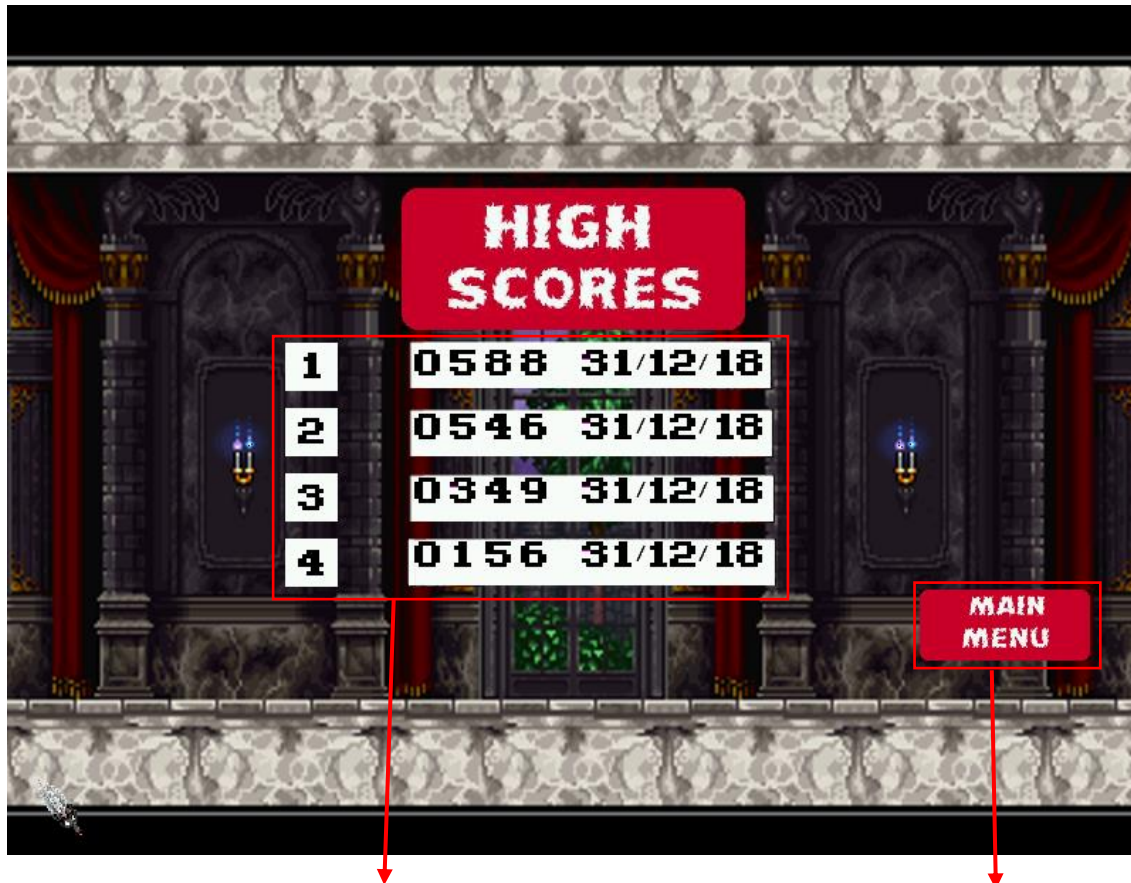
Botão para regressar ao main menu

Dificuldades que o jogo oferece

A dificuldade do jogo afeta a velocidade com que as sprites se movem no ecrã, por exemplo: no Hard os inimigos são gerados e deslocam-se mais rapidamente.

Menu de *Highscores*:

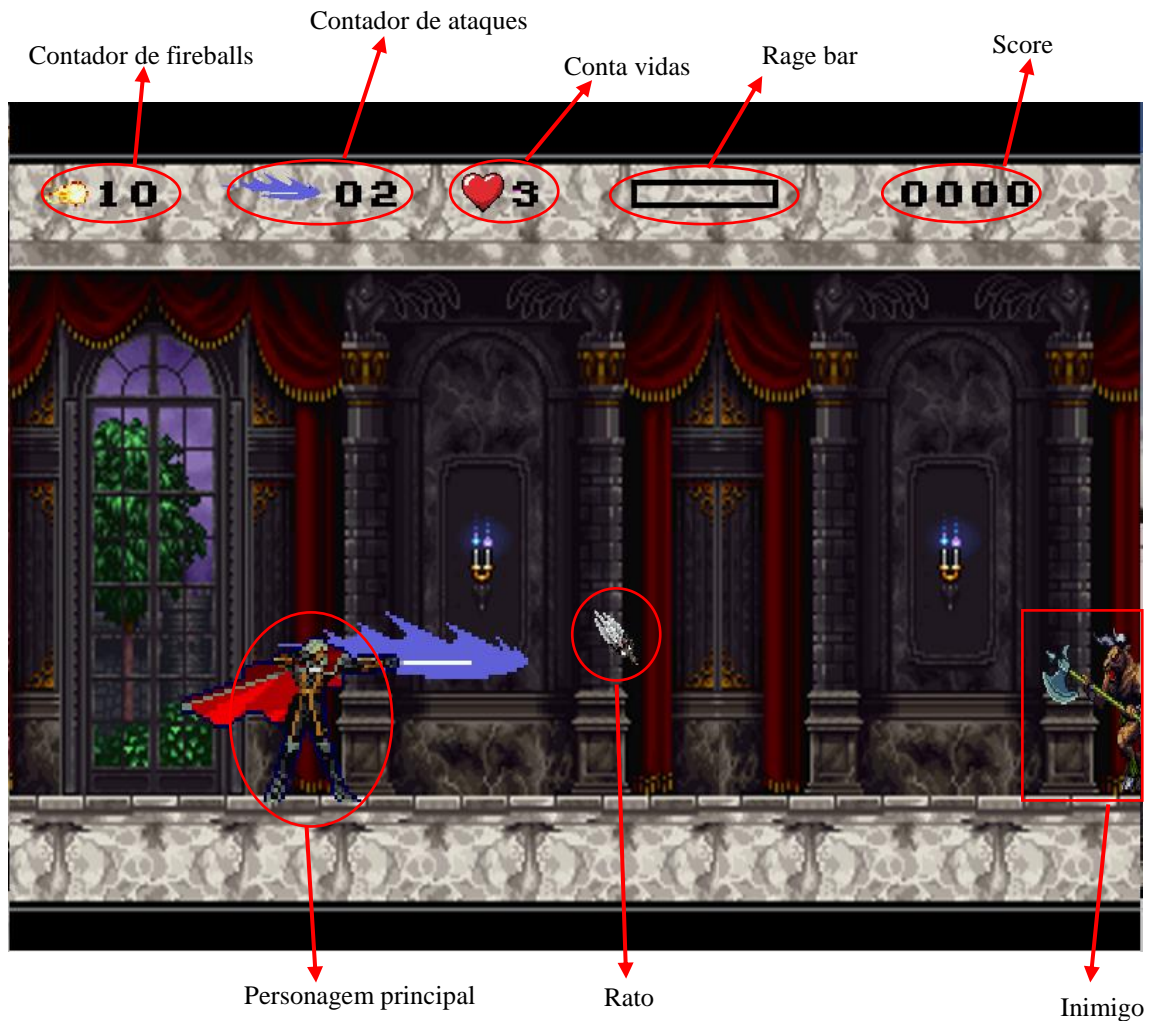
O menu de *Highscores* é apenas acessível através do *Main Menu*. Neste menu estarão presentes os quatros melhores scores que o jogador alcançou no jogo, bem como o dia, o mês e o ano em que o jogador obteve esses resultados. Caso o jogador pretenda regressar ao *Main Menu*, existe um botão para regressar ao *Main Menu*.



4 melhores scores e data dos mesmos

Botão para regressar ao main menu

1 Player (single player):



O movimento da personagem é controlado pelas teclas WASD e a barra de espaços do keyboard, sendo que:

- a **tecla W** faz com que a personagem levante voo, aumentando a altitude,
- a **tecla S** faz com que a personagem reduza a altitude caso esteja no ar,
- a **tecla A** faz com que a personagem se movimente para a esquerda,
- a **tecla D** faz com que a personagem se movimente para a direita,
- a **tecla R** ativa o Rage-Mode,
- a **tecla Barra de Espaços** faz com que a personagem salte.

O jogador começa com 3 vidas. Sempre que haja uma colisão com um inimigo, perde uma vida. Se o jogador tiver 0 vidas, morre e o jogo passa para o menu game over. Existe um contador de vidas que indica quantas vidas o jogador ainda tem.

O jogador para matar os inimigos tem que utilizar uma fireball (bola de fogo) ou um sword attack (ataque de espada). Para mandar uma fireball tem que pressionar o botão esquerdo do rato e para efetuar um sword attack tem que pressionar o botão direito. Para além disto, no caso da fireball o jogador ainda tem a possibilidade de apontar com o rato para onde quer mandar a fireball. Caso um destes ataques atinja um inimigo, este morre, desaparecendo e o score aumenta em uma unidade.

Existe um contador de sword ataques, que começa em 3, e um contador de fireballs, que começa em 10, que indicam quantos ataques, daquele tipo, estão disponíveis naquele momento. Quando o jogador gasta um ataque, o número de ataques disponíveis daquele tipo diminui por uma unidade. Tanto o número de fireballs e o número de sword attacks vai recarregando com o tempo, sendo que o máximo número de fireballs que o jogador tem disponível em qualquer altura é 10 e sword attacks é 3.

Outro detalhe que se implementou foi a *Rage bar* (barra da fúria). A Rage bar vai aumentando à medida que o jogador vai matando inimigos, quando o jogador tiver morto suficientes para encher a rage bar, a opção para entrar no *Rage Mode* vai ficar disponível. O *Rage Mode* é um modo especial, de duração limitada, durante o qual o jogador tem fireballs e sword attacks infinitos. Para além disso, o ritmo ao qual se geram os inimigos aumenta significativamente.

Menu Pause (apenas no single player):

O menu pause está apenas disponível quando o jogador está no modo 1 Player (single player) e é ativado quando o jogador pressiona a tecla ESC do keyboard. O objetivo deste menu é o jogador ter a possibilidade de pausar o jogo, preservando o estado atual do mesmo.



Através deste menu, o jogador tem 4 opções:

- Continue (continuar),
- Restart (recomeçar),
- Main menu,
- Exit.

O botão **Continue** permite ao jogador continuar a jogar o jogo a partir do estado que o deixou quando pressionou o botão ESC para entrar no menu pause. O botão **Restart** permite ao jogador recomeçar o jogo, restaurando as configurações iniciais. O botão **Main Menu** e o botão **Exit** têm funcionalidades idênticas aquelas de outros menus com os mesmos botões, nomeadamente o menu **Highscores** e o **Main Menu**.

Menu Game Over (single player):

O menu *Game Over* é apresentado apenas quando o jogador morre no single player.



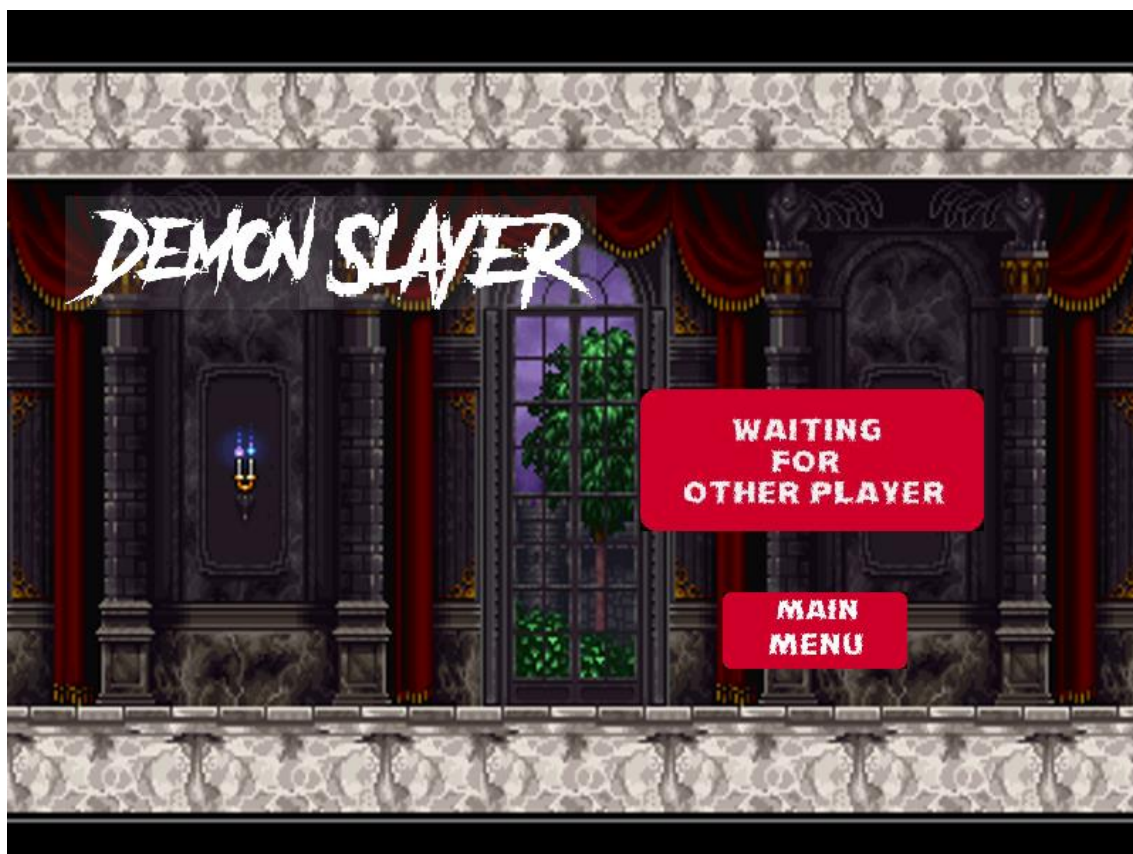
Este menu oferece 3 opções para o jogador:

- **Restart (Recomeçar)**
- **Main Menu (Menu Principal)**
- **Exit (Sair)**

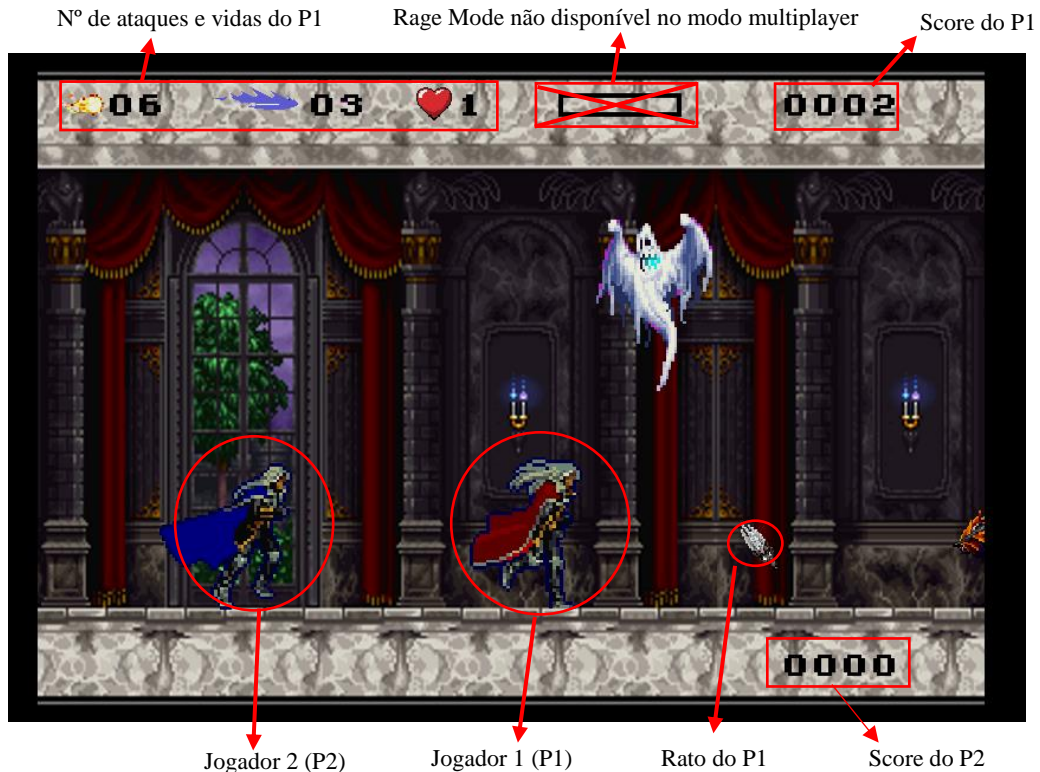
O botão *Restart* permite ao jogador recomeçar o jogo, restaurando as configurações iniciais. O botão *Main Menu* e o botão *Exit* têm funcionalidades idênticas aquelas de outros menus com os mesmos botões, nomeadamente o menu *Highscores* e o *Main Menu*.

Menu Multiplayer (multijogador)

O menu *Multiplayer* é acessível apenas através do *Main Menu* com a opção 2 Players. Neste menu o jogador tem que esperar que um outro jogador passe para este o mesmo estado noutro pc, só assim é que o jogo começa. O menu multijogador oferece apenas uma opção que é a de retornar ao main menu, não é possível configurar a dificuldade do jogo, pelo que esta é predefinida em *Medium*.



2 Players (Multiplayer):



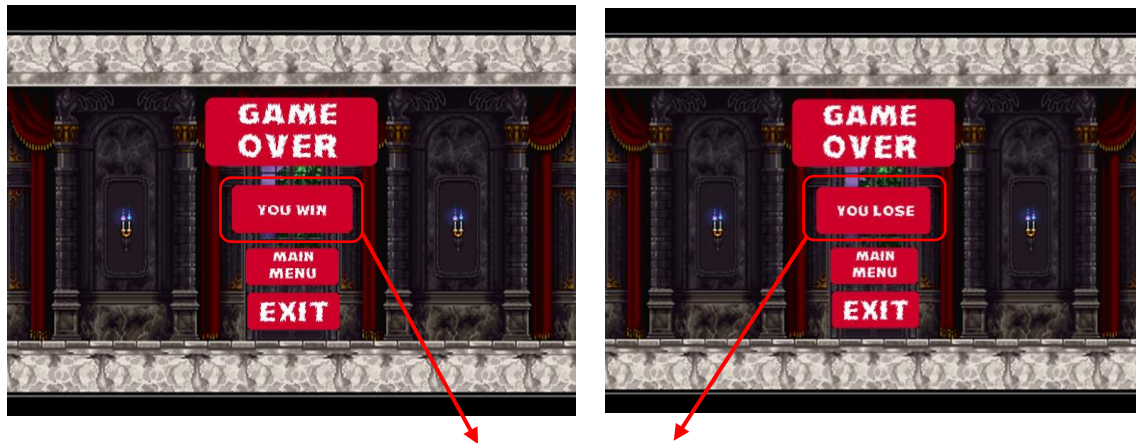
No modo Multiplayer o jogador passa a concorrer com um segundo jogador num outro pc. Existe no canto inferior direito um contador para o score do jogador 2, caso o jogo termine esse contador é comparado com o contador de score do jogador 1 e pode acontecer 3 casos:

- Jogador 1 Wins (ganha), jogador 2 Losses (perde), score jogador 1 < score jogador 2,
- Jogador 2 Wins, jogador 1 Losses, Score jogador 1 > Score jogador 2,
- Draw, Score jogador 1 = Score jogador 2.

Cada jogador tem o seu próprio conjunto de ataques e vida disponíveis, cujo display é dado no pc local. Caso a vida de um jogador chegue a 0, este desaparece do ecrã, tornando-se espetador do jogador remoto. Quando os dois jogadores estiverem mortos o jogo passa para um menu Game Over (Multiplayer).

É de salientar que no multiplayer o rage mode não está implementado, pelo que a rage bar não aparece no ecrã.

Menu Game Over (Multiplayer):



Informação se o jogador perdeu, ganhou, ou empatou o jogo

O menu Game Over em *Multiplayer* é apenas acessível quando ambos os jogadores estiverem mortos. Este menu apresenta a informação se o jogador perdeu, ganhou ou empatou em relação ao outro jogador, tendo em conta a pontuação de cada jogador. Ao contrário do Game Over do Single Player, não oferece uma opção Restart, pelo que caso os jogadores que pretendam começar um novo jogo, necessitam de ir primeiro para o Main Menu e repetir o procedimento inicial. Este menu oferece também a opção para voltar para o Main Menu e para sair do jogo com o botão Exit.

Resumo comandos do jogo:

Nos menus:

Teclado: Não Utilizado

Rato:

- Mover o dispositivo – Deslocamento do cursor,
- Botão lado esquerdo – Efetivar a passagem para o menu pretendido.

Nota: Para que o click no botão esquerdo faça o efeito desejado, o cursor deve, ainda, estar sobreposto ao bloco que efetua a transição pretendida.

Em Jogo (*Singleplayer* ou *Multiplayer*):

Teclado:

- Tecla W – Iniciar o voo da personagem,
- Tecla S (durante o voo) – Descida mais rápida,
- Tecla A – Movimento para a esquerda,
- Tecla D – Movimento para a direita,
- Tecla Barra de Espaço (quando no chão) - Salto.
- Tecla Esc (em modo Singleplayer) – Menu Pausa (apenas singleplayer)
- Tecla R – Ativa o *Rage Mode*.

Rato:

- Mover o dispositivo – Deslocamento do cursor/mira de disparo,
- Botão lado esquerdo – Disparo de fireball,
- Botão lado direito – Golpe de espada.

Project Status:

O estado final do projeto permite inferir que implementamos todos os I/O devices lecionados na disciplina de LCOM. Volvendo à especificação do projeto realizada em novembro, podemos concluir que foram cumpridas as tarefas prometidas, com a implementação da UART excedeu-se as expectativas iniciais.

Em termos temporais, ocorreu um ligeiro derrape na reta final do projeto com a implementação de forma autónoma da UART, contudo, na generalidade os prazos definidos foram cumpridos.

- Na lista abaixo, listam-se as funcionalidades fulcrais do nosso trabalho.

Timer:

O uso do timer é fundamental num trabalho com uma GUI (*Graphical User Interface* - Interface gráfica do utilizador), pois é o componente central da atualização da memória gráfica, naquilo que são designados FPS (*Frames Per Second*, a quantidade de vezes que a informação na memória gráfica é atualizada / *per second*). Para além disso, é a chave para controlar o comportamento do jogo.

Utiliza-se o I/O I8254, em modo Interrupts, mais concretamente, o Timer 0, predefinido a uma frequência de interrupts de **60 Hz**. Como suporte da utilização do controlador, recorreremos exclusivamente à biblioteca definida no Lab2.

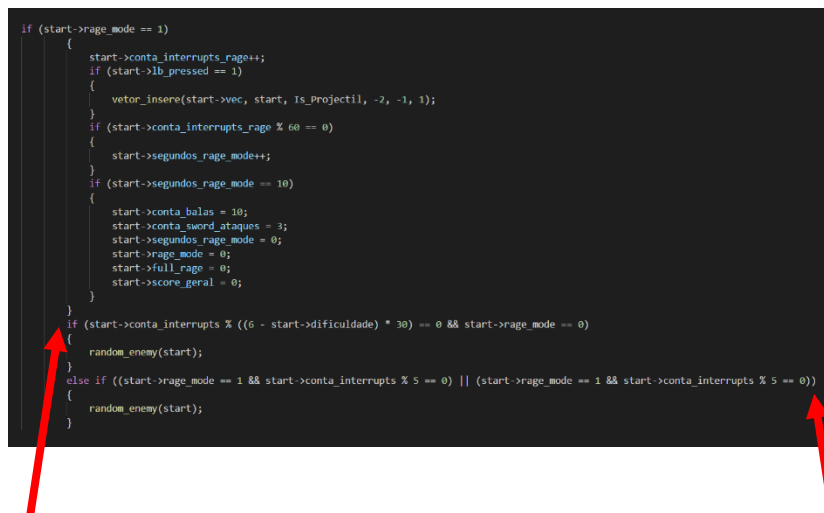
Desta forma, de modo a contornar o recurso a variáveis globais, o nosso handle dos interrupts é feita pela contabilização do nº de interrupts do timer e pela chamada da rotina “**void update_variables(boot_resources*start)**” (descrição da função na PAG 15).

```
0
1
2   if (msg_m_notify.interrupts & start->timer_irq_set){
3
4       start->conta_interrupts++;
5
6       if (start->menu == GAME_2PLAYERS){
7           update_variables_p2(start);
8       }
9
10      update_variables(start);
11
12      }
13
14
```

Tal como referido anteriormente, em *Demon Slayer*, o Timer 0 é utilizado como suporte da GUI, e, ainda, no controlo das *physics* (físicas) das personagens, naquilo que doravante definiremos como controlo do *game flow* (fluxo do jogo). Como tal, são exploradas diferentes cadências, definidas empiricamente, de acordo com o que os programadores consideraram o mais adequado a cada situação.

Delas destaca-se a função “**void update_variables(boot_resources*start)**”, cujas principais funções se listam a seguir:

- **Animação de boas vindas:** Quando se inicia do jogo, nos menus de espera, o utilizador poderá desfrutar de uma animação da personagem, manifestando a sua intenção de iniciar o movimento com um gesto na sua capa. O objetivo de potenciar a interação homem/máquina é aqui central, este objetivo é alcançado pelo recurso aos interrupts do timer, que nos permite atualizar as animações a uma cadência 2 / *per second*.
- **Spawn de Inimigos:** Através da função **random_enemy(start)**.
- **Controlo do Rage-Mode:** A implementação deste modo especial de jogo para além de propósitos estéticos, **tem uma vertente inteiramente ligada ao propósito da cadeira**, destacando-se a capacidade do nosso jogo lidar sem qualquer problema com overload de inimigos, e com uma interação intensa, altamente concentrada com os I/O devices . É um modo que executa o derradeiro teste na robustez da biblioteca e na capacidade de lidar com informação do mouse, keyboard, timer, Graphics Card, não descurando o RTC, tudo em simultâneo. Recorre-se ao timer para limitar em cálculo modular, a duração deste modo especial, bem como diferenciar o spawn de inimigos que ocorrem.



```
if (start->rage_mode == 1)
{
    start->conta_interrupts_rage++;
    if (start->lb_pressed == 1)
    {
        vetor_insere(start->vec, start, IS_Projectil, -2, -1, 1);
    }
    if (start->conta_interrupts_rage % 60 == 0)
    {
        start->segundos_rage_mode++;
    }
    if (start->segundos_rage_mode == 10)
    {
        start->conta_balas = 10;
        start->conta_sword_ataques = 3;
        start->segundos_rage_mode = 0;
        start->rage_mode = 0;
        start->full_rage = 0;
        start->score_geral = 0;
    }
}
if (start->conta_interrupts % ((6 - start->difichuldade) * 30) == 0 && start->rage_mode == 0)
{
    random_enemy(start);
}
else if ((start->rage_mode == 1 && start->conta_interrupts % 5 == 0) || (start->rage_mode == 1 && start->conta_interrupts % 5 == 0))
{
    random_enemy(start);
}
```

Caso o Rage Mode esteja ativo, a cadência da geração é diferente.

- **Animação da personagem:** É o recurso ao Timer que nos permite introduzir animação à personagem, por outras palavras, permite emular a animação através de sucessivas *Sprites*. Na imagem fica ainda patente outro dado empírico, a frequência com que estas são alteradas.

```
if (start->conta_interrupts % 11 - start->dificuldade == 0 && start->state != JUMPONCE){
    start->conta_frames_character++;

    if (start->conta_frames_character == 5){
        start->conta_frames_character = 0;
    }

    if (start->conta_frames_character == 7){
        start->conta_frames_character = 5;
    }

    if (start->conta_frames_character == 14){
        start->p1.width = start->character_p1_move[0]->bitmapInfoHeader.width;
        start->p1.height = start->character_p1_move[0]->bitmapInfoHeader.height;
        start->conta_frames_character = 0;
    }
}
```

Nota: A velocidade com que as Sprites são atualizadas podem ser encaradas como a simulação da velocidade da personagem, daí a introdução do parâmetro velocidade, na atualização das Sprites.

Maior a dificuldade → Maior a Velocidade.

- **Desenho gráfico:** A função **void draw_frame(boot_resources *start)** (descrita no parágrafo dedicada à placa gráfica) é função do timer (depende deste), chamada a cada interrupt deste I/O , perfazendo um desenho gráfico a 60 FPS.

Keyboard:

No jogo, o keyboard, mais concretamente a programação do microcircuito I8042 é fundamental no game flow, pois permite a execução de diferentes comandos em jogo que envolvem o movimento da personagem, bem como o acesso aos menus de pausa. Torna-se, por essa razão, pertinente uma exaustiva descrição da implementação e do uso que lhe é destinado em *Demon Slayer*.

O I8042 é utilizado em modo interrupts, subscrito em modo exclusive, em conformidade com as recomendações providenciadas no Lab 3. De referir ainda que, o handling dos interrupts gerados, bem como a interação com este microcircuito, à semelhança do Timer, ou dos I/O s descritos a seguir, recorrem em exclusivo às implementações criadas para as aulas práticas de cada dispositivo.

```
if ((msg.m_notify.interrupts & start->kbd_irq_set)){  
    kbc_ih();  
    start->scancode = return_scancode();  
  
    pointer = (unsigned char *)malloc(sizeof(unsigned char));  
    *pointer = 'k';  
    makecode = (unsigned char *)malloc(sizeof(unsigned char));  
    *makecode = (unsigned char)start->scancode;  
    fila_push(start->fila_transmitter, pointer);  
    fila_push(start->fila_transmitter, makecode);  
    transmit_queue(start);  
  
    kbd_processing(start);  
}
```

Tem como função ler o output
enviado pelo teclado

Processamento da informação
lida no handler

Um aspeto importante a salientar é a robustez que esta implementação permite a nível do jogo, pois somos capazes de processar diferentes tipos de informação, mesmo que dados do rato ou de outro I/O sejam enviados em simultâneo. Esta estratégia quando concluída com sucesso, permite um nível de liberdade considerável e uma maior facilidade no desenvolvimento de eficientes máquinas de estado que permitam controlar o flow do jogo com alguma facilidade, como veremos no capítulo dedicado às mesmas.

void kbd_processing(boot_resources *start): Um desenvolvimento low-level como LCOM requiere, exige em primeiro lugar que os dados obtidos dos I/O sejam processados. Para que possam ser considerados informação. Em *Demon Slayer*, esse processamento é realizado em **void kbd_processing(boot_resources *start)**.

```
if (start->scancode == W_KEY_MAKECODE)
{
    update_character(UP, start, 1);
}
if (start->scancode == A_KEY_MAKECODE)
{
    update_character(LEFT, start, 1);
}
if (start->scancode == D_KEY_MAKECODE)
{
    update_character(RIGHT, start, 1);
}
if (start->scancode == S_KEY_MAKECODE)
{
    update_character(DOWN, start, 1);
}
if (start->scancode == SPACE_KEY_MAKECODE)
{
    update_character(JUMP, start, 1);
}
if (start->scancode == ESC_BREAK)
{
    update_menu(ESC_PRESSED, start);
}
if (start->scancode == R_KEY_MAKECODE && start->full_rage == 1)
{
    start->full_rage = 0;
    start->rage_mode = 1;
}
check_borders_p1(start);
}
```

Nota: As funções `update_character` e `update_menu` são essenciais na compreensão desta rotina, a sua **descrição é realizada mais à frente, na secção dedicada às máquinas de estado** (PAG. 39).

Mouse:

A programação do mouse, é deveras semelhante à do Keyboard (KB) (descrito acima, PAG. 17), a semelhança advém da partilha do microcontrolador I8042 para a comunicação I/O – OS, esta partilha poderia em tese, caso os graus de qualidade na implementação não fossem os mais assertivos, levantar questões no que concerne a potenciais interferências de Scancodes do KB e consequente perda de informação. Tendo esta questão em mente, recorreu-se às bibliotecas desenvolvidas em Lab4 para este periférico, garantindo-se a robustez pretendida.

O Mouse no trabalho é usado em **modo de Interrupts**, subscrito em modo exclusivo à semelhança do KB. O Handling deste periférico condensa as funcionalidades necessárias ao processamento dos packets do mouse numa solução a três tempos. **De modo a potenciar o desempenho do jogo**, optou-se por processar a informação imediatamente após a receção de cada byte que compõe os packets do mouse.

Em *Demon Slayer*, o papel do rato é central, nomeadamente na GUI, desempenhando diferentes funções, todas elas resultado de um processamento em **void mouse_processing(boot_resources *start)**.

```
void mouse_processing(boot_resources *start)
{
    start->mouse_byte = return_mousebyte();
    if (start->contador_bytes == 0)
    {
        if ((start->mouse_byte & BIT(3)) != 0)
        {
            start->mouse_packet.bytes[0] = start->mouse_byte;
        }
        else
            return;
    }
    else if (start->contador_bytes == 1)
    {
        start->mouse_packet.bytes[1] = start->mouse_byte;
    }
    else if (start->contador_bytes == 2)
    {
        start->mouse_packet.bytes[2] = start->mouse_byte;

        data_processing(&(start->mouse_packet));

        start->mouse_stats.x += start->mouse_packet.delta_x;
        start->mouse_stats.y -= start->mouse_packet.delta_y;
    }
}
```

Processamento a 3 tempos


```

uint8_t(data_processing)(struct packet *leitura)
{
    int sinalx, sinaly;
    uint16_t mousebyte16bits;
    if ((leitura->bytes[0] & L_B) != 0)
        leitura->lb = TRUE;
    else
        leitura->lb = FALSE;
    if ((leitura->bytes[0] & M_B) != 0)
        leitura->mb = TRUE;
    else
        leitura->mb = FALSE;
    if ((leitura->bytes[0] & R_B) != 0)
        leitura->rb = TRUE;
    else
        leitura->rb = FALSE;
    if ((leitura->bytes[0] & Y_OVFL) != 0)
        leitura->y_ov = TRUE;
    else
        leitura->y_ov = FALSE;
    if ((leitura->bytes[0] & X_OVFL) != 0)
        leitura->x_ov = TRUE;
    else
        leitura->x_ov = FALSE;
    if ((leitura->bytes[0] & X_SIGN) != 0)
    {
        sinalx = 1;
    }
    else
        sinalx = 0;
    if ((leitura->bytes[0] & Y_SIGN) != 0)
    {
        sinaly = 1;
    }
    else
        sinaly = 0;
    if (sinalx == 1)

```

```

    {
        mousebyte16bits = 0;
        mousebyte16bits = (mousebyte16bits | leitura->bytes[1]);
        mousebyte16bits = mousebyte16bits | complemento_2;
        leitura->delta_x = mousebyte16bits;
    }
    else
    {
        mousebyte16bits = 0;
        mousebyte16bits = (mousebyte16bits | leitura->bytes[1]);
        leitura->delta_x = mousebyte16bits;
    }
    if (sinaly == 1)
    {
        mousebyte16bits = 0;
        mousebyte16bits = (mousebyte16bits | leitura->bytes[2]);
        mousebyte16bits = mousebyte16bits | complemento_2;
        leitura->delta_y = mousebyte16bits;
    }
    else
    {
        mousebyte16bits = 0;
        mousebyte16bits = (mousebyte16bits | leitura->bytes[2]);
        leitura->delta_y = mousebyte16bits;
    }
    return 0;
}

```

A função **data_processing()** (excerto de código acima) é uma função utilizada pelo **mouse_processing()** para retirar a informação dos mouse bytes e transpô-la para uma estrutura do tipo packet que irá possuir informação acerca do mouse, por exemplo: se o botão esquerdo do mouse está a ser pressionado ou não.

A implementação garante que no fim do jogo a **linha de comandos do MINIX é retornada com correção**, através da leitura do output buffer de potenciais bytes residuais que nele existam. Para tal feito é utilizada na função **game_end()** a função **keyboard_command_receive()** (excerto de ambas em baixo).

```

5
6
7 void game_end(boot_resources *start)
8 {
9     uint8_t lixo;
10     .....
11     keyboard_command_receive(&lixo);
12
13     return;
14 }
15

```

```

int(keyboard_command_receive)(uint8_t *leitura)
{
    int tries = 5, i = 0;
    uint32_t stat32bit, commandbyte32bits = INITIALIZE0;
    uint8_t stat8bit, commandbyte8bits;
    while (i < tries)
    {
        sys_inb(STAT_REG, &stat32bit);
        stat8bit = (uint8_t)(stat32bit & CAST_PARA_8BITS);
        if (stat8bit & 0BF)
        {
            sys_inb(OUT_BUF, &commandbyte32bits); /* assuming it returns OK */
            commandbyte8bits = (uint8_t)(commandbyte32bits & CAST_PARA_8BITS);
            if ((stat8bit & (PAR_ERR | TO_ERR)) == 0)
            {
                *leitura = commandbyte8bits;
                return 0;
            }
            else
            {
                printf("Par error\n");
                return -1;
            }
        }
        i++;
        tickdelay(micros_to_ticks(DELAY_US));
    }
    return -1;
}

```

Mouse – GUI interface:

De forma a garantir a que a posição do cursor do rato não excede os limites do ecrã, incluiu-se esta funcionalidade em **void mouse_processing(boot_resources *start)**, esta função está implementada de forma a funcionar com qualquer modo gráfico. Esta característica é proporcionada pela chamada de funções como **get_h_res()** (indica qual a resolução horizontal do modo) e **get_v_res()** (indica qual a resolução vertical do modo).

Verificação se o cursor
excedeu os limites
horizontais do ecrã

Verificação se o cursor
excedeu os limites verticais
do ecrã

```
if (start->mouse_stats.x >= get_h_res())
{
    start->mouse_stats.x = get_h_res() - 10;
}
else if (start->mouse_stats.x <= 0)
{
    start->mouse_stats.x = 0;
}

if (start->mouse_stats.y >= get_v_res())
{
    start->mouse_stats.y = get_v_res() - 10;
}
else if (start->mouse_stats.y <= 0)
{
    start->mouse_stats.y = 0;
}
```

Como referido anteriormente, o mouse é o elemento de navegação dos menus, a biblioteca apresenta por isso, a programação necessária para essa integração:

```
if (start->menu == MAIN_MENU)
{
    if (start->mouse_stats.x > 524 && start->mouse_stats.x < 654 && start->mouse_stats.y < 258 && start->mouse_stats.y > 201 && start->mouse_packet.lb == TRUE)
    {
        update_menu(PLAY_PRESSED, start);
    }
    if (start->mouse_stats.x > 524 && start->mouse_stats.x < 654 && start->mouse_stats.y < 315 && start->mouse_stats.y > 260 && start->mouse_packet.lb == TRUE)
    {
        update_menu(MULTIPLAYER_PRESSED, start);
    }
    if (start->mouse_stats.x > 524 && start->mouse_stats.x < 654 && start->mouse_stats.y < 371 && start->mouse_stats.y > 317 && start->mouse_packet.lb == TRUE)
    {
        update_menu(DIFICULTY_PRESSED, start);
    }
    if (start->mouse_stats.x > 524 && start->mouse_stats.x < 654 && start->mouse_stats.y < 428 && start->mouse_stats.y > 374 && start->mouse_packet.lb == TRUE)
    {
        update_menu(HIGHSCORE_PRESSED, start);
    }
    if (start->mouse_stats.x > 524 && start->mouse_stats.x < 654 && start->mouse_stats.y < 486 && start->mouse_stats.y > 432 && start->mouse_packet.lb == TRUE)
    {
        update_menu(EXIT_PRESSED, start);
    }
}
else if (start->menu == DIFICULTY_MENU)
{
    if (start->mouse_stats.x > 504 && start->mouse_stats.x < 639 && start->mouse_stats.y < 278 && start->mouse_stats.y > 224 && start->mouse_packet.lb == TRUE)
    {
        update_menu(HARD_PRESSED, start);
    }
    if (start->mouse_stats.x > 504 && start->mouse_stats.x < 639 && start->mouse_stats.y < 339 && start->mouse_stats.y > 287 && start->mouse_packet.lb == TRUE)
    {
        update_menu(MEDIUM_PRESSED, start);
    }
    if (start->mouse_stats.x > 504 && start->mouse_stats.x < 639 && start->mouse_stats.y < 402 && start->mouse_stats.y > 347 && start->mouse_packet.lb == TRUE)
    {
        update_menu(EASY_PRESSED, start);
    }
    if (start->mouse_stats.x > 504 && start->mouse_stats.x < 639 && start->mouse_stats.y < 465 && start->mouse_stats.y > 410 && start->mouse_packet.lb == TRUE)
    {
        update_menu(MAIN_MENU_PRESSED, start);
    }
}
```

Nota: As funções `update_menu`, implementações de máquinas de estado, num sistema gerido por eventos são descritos no capítulo (Máquinas de Estado PAG.39)

Demon Slayer possui dois tipos de medidas ofensivas de modo a destruir inimigos. O disparo de uma *fireball* ou um golpe de espada. Ambas necessitam de processamento da informação proveniente dos mouse bytes, funcionalidades que são executadas em **void mouse_processing (boot_resources *start)**.



```
if (start->rage_mode == 0)
{
    if (start->mouse_packet.lb == TRUE && start->mouse_packet.rb == FALSE)
    {
        if (start->conta_balas > 0 && start->lb_pressed == 0)
        {
            vetor_inserir(start->vec, start, Is_Projectil, -2, -1, 1);
            start->lb_pressed = 1;
            start->conta_balas--;
        }
    }
    else if (start->mouse_packet.lb == FALSE)
    {
        start->lb_pressed = 0;
    }
}
```

Mouse Processing – Fireballs

Nota: A referência a Vetor_Inserir pode ser encontrada na secção Memória Dinâmica (PAG .40)

O Processamento do botão direito do rato passa pela atualização da Sprite a ser representada para uma que representa essa animação durante um frame (consultar PAG 8 - Desenho gráfico). De notar que esta animação está restrita à sua utilização quando a personagem se encontra naquilo que é considerado por FLOOR_POSITION (“Posição do chão”).



```
if (start->mouse_packet.lb == FALSE && start->mouse_packet.rb == TRUE && start->p1.y == FLOOR_POSITION)
{
    if (start->conta_sword_ataques > 0 && start->rb_pressed == 0)
    {
        start->rb_pressed = 1;
        start->p1.width = start->sword[0]->bitmapInfoHeader.width;
        start->p1.height = start->sword[0]->bitmapInfoHeader.height;
        start->conta_frames_character = 9;
        start->conta_sword_ataques--;
    }
}
else if (start->mouse_packet.rb == FALSE)
{
    start->rb_pressed = 0;
}
```

Mouse Processing – Golpe de Espada

Cálculo da trajetória do Projétil: De referir ainda, a forma como num sistema baixo nível como o proposto em LCOM, é calculada a trajetória do projétil. Uma extensão em código do cálculo físico da trajetória de um projétil.

```
double hipotenusa, seno, cosseno;  
hipotenusa = sqrt(delta_x * delta_x + delta_y * delta_y);  
seno = delta_y / hipotenusa;  
cosseno = delta_x / hipotenusa;  
  
vec->element[pos].union_type_elementos.proj->obj->speedx = 15 * cosseno;  
vec->element[pos].union_type_elementos.proj->obj->speedy = 15 * seno;
```

Esta solução, permite uma forma fiável e portátil de cálculo da trajetória. Recorrendo a valores angulares, determinados pela decomposição do triângulo efetuado entre a posição onde é disparado o projétil e a posição do cursor do rato / mira é calculada uma trajetória independente. Desta forma, conseguimos um sistema baixo nível, independente do modo gráfico selecionado de elevada eficácia.

RTC:

A inserção do Real Time Clock no jogo tem como objetivo estabelecer um registo horário e da data onde determinada classificação é estabelecida. O RTC isolado não possui um papel que se possa considerar preponderante, a tradução gráfica da informação que este contém é que é uma tarefa árdua.

O RTC em *Demon Slayer* é utilizado em modo de interrupts, dada a simplicidade da implementação e o papel que este microcircuito possui no projeto, essa subscrição demonstrou-se desnecessária, no entanto, sem prejuízo do jogo, optou-se por manter essa funcionalidade com objetivos de portabilidade e tendo em visto um futuro aditamento ao projeto, sendo a obtenção dos dados nele representado obtido por via de Polling. O desfasamento temporal que as leituras dos diferentes parâmetros poderiam apresentar, um problema abordado nas aulas, é resolvido nesta biblioteca pela primeira das três soluções apresentadas, **a leitura do UIP_BIT do Registro A do RTC**.

O diagrama mostra um trecho de código C para a leitura do RTC. O código está em um fundo escuro com uma fonte de código clara. Há duas anotações em português com setas vermelhas apontando para partes do código:

- Uma seta aponta para o laço `for (i = 0; i < 3; i++)` com o texto: "Executa um número de 3 tentativas de forma a dar delay a um possível update em curso".
- Outra seta aponta para a linha `sys_inb(RTC_DATA_REG, &leitura);` dentro do primeiro bloco de código, com o texto: "Leitura do UIP_BIT".

Além disso, há uma bracejada azul vertical à direita do código, abrangendo os blocos de leitura dos registros (SECONDS_REG, MIN_REG, HOUR_REG, DAY_WEEK, DAY_MONTH, MONTH, YEAR), com o texto: "Leitura dos registos do RTC".

```
for (i = 0; i < 3; i++)
{
    sys_outb(RTC_ADDR_REG, REGISTER_A);
    sys_inb(RTC_DATA_REG, &leitura);

    if ((leitura & UIP_MASK) == 0)
    {
        sys_outb(RTC_ADDR_REG, SECONDS_REG);
        sys_inb(RTC_DATA_REG, &leitura);
        *seconds = leitura;

        sys_outb(RTC_ADDR_REG, MIN_REG);
        sys_inb(RTC_DATA_REG, &leitura);
        *minutes = leitura;

        sys_outb(RTC_ADDR_REG, HOUR_REG);
        sys_inb(RTC_DATA_REG, &leitura);
        *hour = leitura;

        sys_outb(RTC_ADDR_REG, DAY_WEEK);
        sys_inb(RTC_DATA_REG, &leitura);
        day_week = leitura;

        sys_outb(RTC_ADDR_REG, DAY_MONTH);
        sys_inb(RTC_DATA_REG, &leitura);
        *day_month = leitura;

        sys_outb(RTC_ADDR_REG, MONTH);
        sys_inb(RTC_DATA_REG, &leitura);
        *month = leitura;

        sys_outb(RTC_ADDR_REG, YEAR);
        sys_inb(RTC_DATA_REG, &leitura);
        *year = leitura;

        break;
    }
}
```

```

void scores_update(boot_resources *start)
{
    if (start->contador_highscore > start->high_data.lugar_1)
    {
        start->high_data.lugar_4 = start->high_data.lugar_3;
        start->high_data.lugar_3 = start->high_data.lugar_2;
        start->high_data.lugar_2 = start->high_data.lugar_1;
        start->high_data.read[3] = start->high_data.read[2];
        start->high_data.read[2] = start->high_data.read[1];
        start->high_data.read[1] = start->high_data.read[0];
        start->high_data.lugar_1 = start->contador_highscore;
        rtc_read_time(&start->high_data.read[0].seconds, &start->high_data.read[0].minutes, &start->high_data.read[0].hour, &start->high_data.read[0].day_month, &start->high_data.read[0].year_month, &start->high_data.read[0].year);
    }
    else if (start->contador_highscore > start->high_data.lugar_2)
    {
        start->high_data.lugar_4 = start->high_data.lugar_3;
        start->high_data.lugar_3 = start->high_data.lugar_2;
        start->high_data.read[3] = start->high_data.read[2];
        start->high_data.read[2] = start->high_data.read[1];
        rtc_read_time(&start->high_data.read[1].seconds, &start->high_data.read[1].minutes, &start->high_data.read[1].hour, &start->high_data.read[1].day_month, &start->high_data.read[1].year_month, &start->high_data.read[1].year);
        start->high_data.lugar_2 = start->contador_highscore;
    }
    else if (start->contador_highscore > start->high_data.lugar_3)
    {
        start->high_data.lugar_4 = start->high_data.lugar_3;
        start->high_data.read[3] = start->high_data.read[2];
        rtc_read_time(&start->high_data.read[2].seconds, &start->high_data.read[2].minutes, &start->high_data.read[2].hour, &start->high_data.read[2].day_month, &start->high_data.read[2].year_month, &start->high_data.read[2].year);
        start->high_data.lugar_3 = start->contador_highscore;
    }
    else if (start->contador_highscore > start->high_data.lugar_4)
    {
        rtc_read_time(&start->high_data.read[3].seconds, &start->high_data.read[3].minutes, &start->high_data.read[3].hour, &start->high_data.read[3].day_month, &start->high_data.read[3].year_month, &start->high_data.read[3].year);
        start->high_data.lugar_4 = start->contador_highscore;
    }
    return;
}

```

RTC_READ_TIME é utilizada para atualizar os scores quando ocorre um GAME OVER

Ver HIGHScores MENU (PAG.6)

Graphics Card:

Em *Demon Slayer* a GUI tem um papel da maior relevância. Desta forma, a programação deste I/O é imperativa ser descrita minuciosamente, bem como as funcionalidades que essas funções implementam no jogo. De reforçar ainda o já referido papel do Timer na execução do desenho gráfico (PAG. 7)

No projeto, **o modo gráfico escolhido foi o VBE 2.0 0x114**. Este modo possui uma resolução de 800x600, a 16 bits por cor (5:6:5) em modo **Direct Color**.

Quanto às Sprites, a importação de imagens foca-se no recurso ao formato **.bmp**, servindo-se de uma biblioteca (**Bitmap.c, cortesia de Henrique Ferrolho PAG. 40**) para assegurar a compatibilidade com o modo usado pela VBE.

As funcionalidades gráficas presentes no jogo concentram-se no módulo **graphics.c e vbe1.c**, esta última, conjunto de funções desenvolvidas em Lab5, à semelhança da maioria dos periféricos utilizados.

Listam-se a seguir os tópicos essenciais à compreensão da implementação gráfica do jogo:

- **Formato BMP e Transparências:** Uma das limitações prende-se com o estilo do formato das Sprites, o bmp. Na realidade, as Sprites são representadas sob a forma de retângulos, o que obriga a estabelecer um pigmento de transparência para poder programar a biblioteca de modo a ocultar o desenho de pixéis indesejados. Em *Demon Slayer*, selecionou-se o cor-de-rosa para efetuar esse truque gráfico, já que esta cor não está presente, originalmente, em nenhum bitmap.



Exemplo de um BMP usado no jogo. Usa-se a cor rosa para não representar os pixéis que compõe o retângulo, mas nada tem que ver com a personagem.

- **Double Buffering e Flip Display:** De forma a contornar a ocorrência de possíveis resíduos gráficos existentes pelo desfasamento temporal existente entre a CPU/GPU, foi implementado um buffering de memória local, alocado dinamicamente com as dimensões exatas à memória disponível na VBE. Esta abordagem garante que potenciais erros de escrita são erradicados.

```
void draw_frame(boot_resources *start)
{
    draw_background(start);
    draw_enemy(start);

    if (start->collision_time == 0)
    {
        draw_character(start, 1);
    }
    else if (start->collision_time == 1 && start->conta_interrupts_collision % 2 == 0)
    {
        draw_character(start, 1);
    }
    if (start->menu == GAME_2PLAYERS)
    {
        if (start->collision_time2 == 0)
        {
            draw_character(start, 2);
        }
        else if (start->collision_time2 == 1 && start->conta_interrupts_collision2 % 2 == 0)
        {
            draw_character(start, 2);
        }
    }

    draw_projetil(start);
    draw_contador_de_ataques(start);
    draw_rage_bar(start);
    draw_mouse(start);
    flip_display();

    return;
}
```

A escrita no Buffer local é uma constante em graphics.c, tendo por base a função **void drawBitmap(Bitmap *bmp, int x, int y, Alignment alignment, char *video_mem)**. A invocação a esta rotina ocorre com a passagem por referência no parâmetro **char *video_mem**, do apontador para o buffer dinamicamente alocado.

```
{
    type = Is_Axe_taurus;
    drawBitmap(aux_taurus->bmp, aux_taurus->x, aux_taurus->y, ALIGN_LEFT, get_double_buffer());
    aux_taurus->x -= start->background_sprite.speed; //atualiza a posicao dos taurus
}
```

Exemplo de invocação

Em *Demon Slayer* todos os elementos gráficos são representados primeiro no referido buffer local de memória dinâmico, após a escrita de todos os elementos ter sido concluída, a função **void flip_display()** entra em cena, efetuando a escrita desse buffer local na VRAM.

```
void flip_display()
{
    memcpy(get_video_mem(), get_double_buffer(), (get_vram_size()));
    memset(get_double_buffer(), 0, (get_vram_size()));
    return;
}
```

Recurso à função **memcpy** para facilitar o uso do sistema D.Buffering.

- **Recurso a funções VBE:** A implementação apresentada recorre às informações retornadas pelas funções VBE. Estas são utilizadas em *Demon Slayer* de forma a garantir um nível superior de portabilidade e uma execução independentemente do modo gráfico utilizado. Destaca-se `int(vbe_get_mode_info_ours)(uint16_t mode, vbe_mode_info_t *vmi_p)`:

```
int(vbe_get_mode_info_ours)(uint16_t mode, vbe_mode_info_t *vmi_p)
{
    mmap_t map;
    struct reg86u reg;
    phys_bytes buf;
    if (lm_alloc(sizeof(vbe_mode_info_t), &map) == NULL) //alloc memory
    {
        printf("ERRO: lm_alloc falhou\n"); //reboot minix if there is prob
        return -1;
    }
    memset(&reg, 0, sizeof(reg));
    buf = map.phys;
    reg.u.w.ax = VBE_GET_MODE_INFO; // VBE get mode info (0x4F01)//funcç
    reg.u.w.es = PB2BASE(buf); // PB2BASE Is a macro for computing
    reg.u.w.di = PB2OFF(buf); // PB2OFF Is a macro for computing t
    reg.u.w.cx = mode;
    reg.u.b.intno = BIOS_CALL; // (0x10)//mete serviços bios
    if (sys_int86(&reg) != OK)
    {
        //all BIOS
        printf("ERRO: sys_int86() falhou\n");
        lm_free(&map); //free memory
        return -1;
    }
}
```

Preenchimento de estrutura do tipo
reg86u

- **Funções Draw:** A biblioteca goza de um certo **encapsulamento no que toca à parte gráfica**, tendo em vista facilitar um trabalho que se adivinhava árduo. Desta forma, foi contruído uma função `Draw_Frame`, chamada (consultar PAG.9) a cada interrupt do Timer 0 que abarca as funcionalidades de desenho gráfico. Esta, por sua vez, possui 7 sub-rotinas com vertente gráfica, visando o desenho de cada um dos elementos que compõe o jogo (personagens, cursor do rato, inimigos, projétil, layout, rage bar e background).

Esta solução permite outra funcionalidade muito importante, em sistemas desta natureza, **a informação gráfica tem de ser sobreposta frequentemente. É, por isso, imperativo estabelecer uma prioridade no desenho gráfico.**

Exemplo: Quando o cursor do rato se sobrepõe a um inimigo, o cursor do rato deve ser representado naquela posição, ao invés dos inimigos:



```
draw_background(start);
draw_enemy(start);

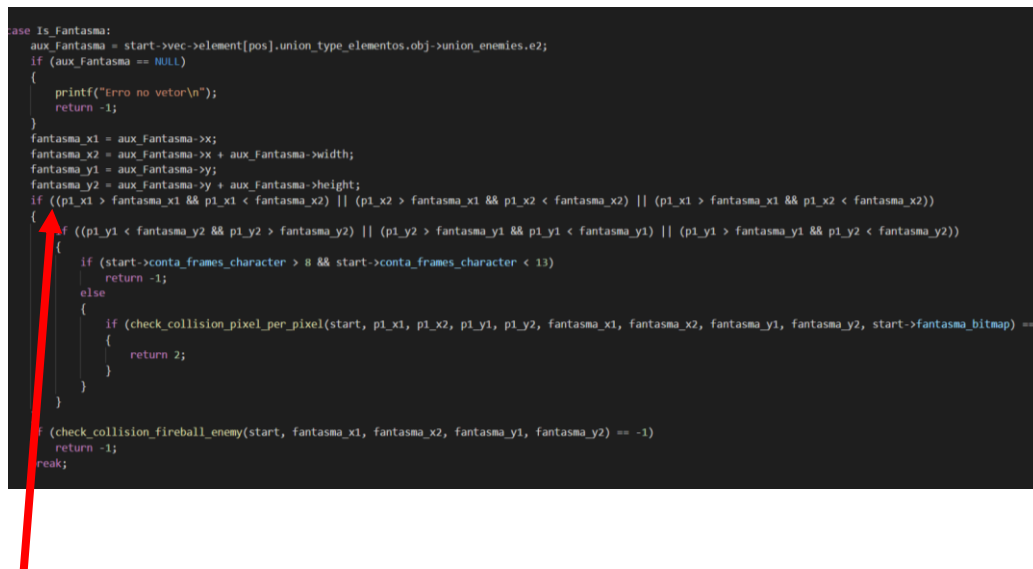
draw_character(start, 2);
draw_projetil(start);
draw_contador_de_ataques(start);
draw_rage_bar(start);
draw_mouse(start);
```

Através da solução implementada, estes tipos de prioridades são reduzidos a uma simples troca na ordem das linhas.

- **Colisões:** Em *Demon Slayer* são implementados **testes de colisão pixel-a-pixel**. Esta funcionalidade apesar de mais complexa, garante um maior nível de fiabilidade e realidade ao jogo. O problema que se impunha neste momento residia nas limitações do formato bmp. (A transparência - PAG. 27), desta forma, urge garantir que a colisão detetada não é de facto sobre o domínio da transparência da imagem, isso só é alcançado através deste tipo de teste.

As **colisões pixel-a-pixel**, é um processo a dois passos, para não exigir um grande poder computacional.

1. Processo de deteção de sobreposição das Sprites (retângulos):



```

case Is_Fantasma:
    aux_Fantasma = start->vec->element[pos].union_type_elementos.obj->union_enemies.e2;
    if (aux_Fantasma == NULL)
    {
        printf("Erro no vetor\n");
        return -1;
    }
    fantasma_x1 = aux_Fantasma->x;
    fantasma_x2 = aux_Fantasma->x + aux_Fantasma->width;
    fantasma_y1 = aux_Fantasma->y;
    fantasma_y2 = aux_Fantasma->y + aux_Fantasma->height;
    if ((p1_x1 > fantasma_x1 && p1_x1 < fantasma_x2) || (p1_x2 > fantasma_x1 && p1_x2 < fantasma_x2) || (p1_x1 > fantasma_x1 && p1_x2 < fantasma_x2))
    {
        if ((p1_y1 < fantasma_y2 && p1_y2 > fantasma_y2) || (p1_y2 > fantasma_y1 && p1_y1 < fantasma_y1) || (p1_y1 > fantasma_y1 && p1_y2 < fantasma_y2))
        {
            if (start->conta_frames_character > 8 && start->conta_frames_character < 13)
                return -1;
            else
            {
                if (check_collision_pixel_per_pixel(start, p1_x1, p1_x2, p1_y1, p1_y2, fantasma_x1, fantasma_x2, fantasma_y1, fantasma_y2, start->fantasma_bitmap) ==
                {
                    return 2;
                }
            }
        }
    }
    if (check_collision_fireball_enemy(start, fantasma_x1, fantasma_x2, fantasma_y1, fantasma_y2) == -1)
        return -1;
    break;

```

Deteção de sobreposição dos retângulos

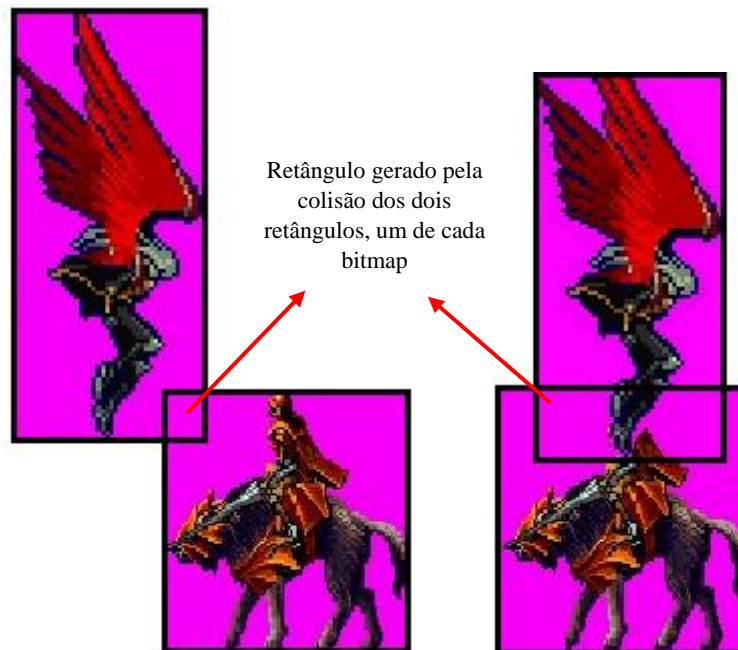
2. Análise dos pixels da imagem, em busca da ausência de transparências no espaço colidido:

```

character_left_rectangle_point_x = left - start->p1.x;
character_left_rectangle_point_y = p1.y2 - bottom;
enemy_left_rectangle_point_x = left - enemy.x1;
enemy_left_rectangle_point_y = enemy.y2 - bottom;
for (int i = 0; i < (bottom - top); i++)
{
    character_start_pos = start->p1.bitmap->bitmapData + (character_left_rectangle_point_x + character_left_rectangle_point_y) * 2 + (start->p1.width * i * 2);
    enemy_start_pos = enemy_bitmap->bitmapData + (enemy_left_rectangle_point_x + enemy_left_rectangle_point_y) * 2 + (enemy_bitmap->bitmapInfoHeader.width * i * 2);
    for (int j = 0; j < (right - left) * 2; j = j + 2)
    {
        if (character_start_pos[j] != PINK1 && character_start_pos[j + 1] != PINK2 && enemy_start_pos[j] != PINK1 && enemy_start_pos[j + 1] != PINK2)
        {
            if (start->collision_time == 0)
            {
                start->collision_time = 1;
                start->contador_de_vida--;
            }
            if (start->contador_de_vida == 0)
            {
                return -1;
            }
        }
    }
}
return 0;

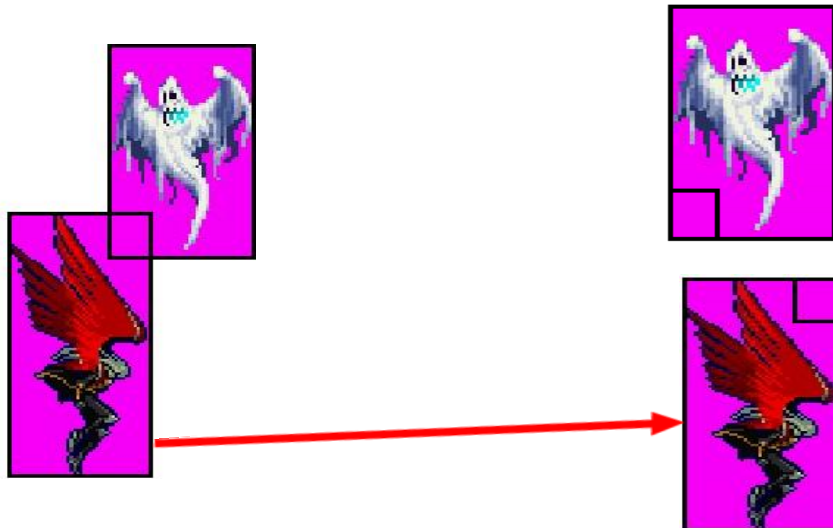
```

Exemplo de uma colisão entre dois bitmaps:



Na imagem da esquerda, embora tenha havido uma colisão dos retângulos dos bitmaps pode observar-se que nenhum dos pixels que compõe a personagem se sobrepõem em ambas as sprites. A detecção de colisões passa a resumir-se a um problema que envolve verificar se o espaço em colisão não passa de meras transparências em ambos, o que se pode testemunhar no caso da esquerda. Na imagem da direita, pelo contrário, podemos ver que há sobreposição de pixels entre a personagem e o inimigo gerando assim uma colisão de pixels.

Em suma, a verificação da colisão pixel-a-pixel é a demonstração que uma simples sobreposição não é suficiente para garantir colisão. Quando se deteta sobreposição, temos de proceder ao cálculo das coordenadas do retângulo de colisão dos bitmaps e com essa informação calcular a posição desse mesmo retângulo só que agora em cada um dos bitmaps.



Para calcular a posição do retângulo e em seguida o retângulo em cada bitmap usa-se o seguinte trecho de código:

Cálculo da posição do lado esquerdo do retângulo no eixo dos x	→	<code>if (p1_x1 > enemy_x1)</code>
		<code>{</code>
		<code>left = p1_x1;</code>
		<code>}</code>
		<code>else</code>
		<code>{</code>
		<code>left = enemy_x1;</code>
		<code>}</code>
Cálculo da posição do lado superior do retângulo no eixo dos y	→	<code>if (p1_y1 > enemy_y1)</code>
		<code>{</code>
		<code>top = p1_y1;</code>
		<code>}</code>
		<code>else</code>
		<code>{</code>
		<code>top = enemy_y1;</code>
		<code>}</code>
Cálculo da posição do lado direito do retângulo no eixo dos x	→	<code>if (p1_x2 > enemy_x2)</code>
		<code>{</code>
		<code>right = enemy_x2;</code>
		<code>}</code>
		<code>else</code>
		<code>{</code>
		<code>right = p1_x2;</code>
		<code>}</code>
Cálculo da posição do lado inferior do retângulo no eixo dos y	→	<code>if (p1_y2 > enemy_y2)</code>
		<code>{</code>
		<code>bottom = enemy_y2;</code>
		<code>}</code>
		<code>else</code>
		<code>{</code>
		<code>bottom = p1_y2;</code>
		<code>}</code>
Cálculo do ponto no bitmap da personagem onde vamos começar a percorrer o retângulo	→	<code>character_left_rectangle_point_x = left - start->p1.x;</code>
		<code>character_left_rectangle_point_y = p1.y2 - bottom;</code>
Cálculo do ponto no bitmap do inimigo onde vamos começar a percorrer o retângulo	→	<code>enemy_left_rectangle_point_x = left - enemy_x1;</code>
		<code>enemy_left_rectangle_point_y = enemy_y2 - bottom;</code>

Em seguida depois de saber todas as coordenadas necessárias, percorremos os retângulos do bitmap do inimigo e da personagem ao mesmo tempo para verificar se existe algum pixel, que não tenha a cor rosa, que esteja ativo nos dois e que esteja na mesma posição do retângulo de colisão dos dois bitmaps. Se houver algum pixel ativo nos dois ao mesmo tempo então sucedeu-se uma colisão entre o inimigo e a personagem.

Para percorrer os dois bitmaps utiliza-se o seguinte trecho de código, que contém dois ciclos for (), um para percorrer a altura e outro para percorrer a largura do retângulo.

```
character_left_rectangle_point_x = left - start->p1.x;
character_left_rectangle_point_y = p1.y2 - bottom;
enemy_left_rectangle_point_x = left - enemy_x1;
enemy_left_rectangle_point_y = enemy_y2 - bottom;
for (int i = 0; i < (bottom - top); i++)
{
    character_start_pos = start->p1.bitmap->bitmapdata + (character_left_rectangle_point_x + character_left_rectangle_point_y) * 2 + (start->p1.width * i * 2);
    enemy_start_pos = enemy_bitmap->bitmapdata + (enemy_left_rectangle_point_x + enemy_left_rectangle_point_y) * 2 + (enemy_bitmap->bitmapinfoheader.width * i * 2);
    for (int j = 0; j < (right - left) * 2; j = j + 2)
    {
        if (character_start_pos[j] != PINK1 && character_start_pos[j + 1] != PINK2 && enemy_start_pos[j] != PINK1 && enemy_start_pos[j + 1] != PINK2)
        {
            if (start->collision_time == 0)
            {
                start->collision_time = 1;
                start->contador_de_vida--;
            }
            if (start->contador_de_vida == 0)
                return -1;
        }
    }
}
return 0;
```

UART/SERIAL PORT:

A implementação da UART é o passo final da conclusão do projeto. **Apesar da tentativa inicial de operar em modo de Interrupts, não fomos capazes de concluir** essa metodologia por um problema ainda hoje indeterminado, desta forma, de modo a concluirmos a tarefa que nos propúnhamos, optamos por um sistema de **polling, recorrendo a FIFOS**. Em *Demon Slayer*, o Papel da UART reside na transmissão da informação proveniente do mouse, do teclado e alguns dados de controlo, em duplex, para que ambas as máquinas as possam processar.

Usa-se a configuração com o recurso ao COM1, utilizando FIFOS, quer para a transmissão, quer para a receção, 8 bits/ per char, a um baud-rate de 115200.

De forma diferente da idealizada, pelo contratempo da incapacidade de operar com o recurso a Interrupts, foi necessário recorrer a uma solução mais limitada, no entanto funcional. De referir ainda, talvez o maior contratempo com que nos deparamos foi a incompatibilidade que o nosso código apresentava para suportar o modo multiplayer, o que obrigou a múltiplas replicações de funções **para interpretar a informação do computador remoto, doravante designado de Player 2 (P2)**.

A forma utilizada para potenciar a utilização e o processamento da informação entre as diferentes máquinas, enviada pela UART, baseia-se na ideia de State Machine-Replication, ou seja, na replicação do código que já efetuava o processamento no computador local, para passar a processar também, informação proveniente do computador remoto. O recurso a este método, prende-se, sobretudo, pela impraticabilidade de transitar a informação gráfica entre computadores, o que obriga, desde logo a manter as capacidades gráficas e de processamento em ambos os lados da comunicação. Esta restrição origina algumas questões que tornam o processo trabalhoso, das quais se realça a preservação da integridade/codificação da mensagem enviada e a organização de estratégias para processar informações provenientes de dois computadores distintos, que apresentam informações semelhantes.

- **Codificação das mensagens:** Para distinguir a informação, visto que os inputs que processamos estão bem definidos, podemos optar por uma modalidade de envio mais simples. Desta forma, antes de processarmos o envio de informação de determinado I/O, enviamos “à cabeça” um caracter distintivo que nos permita interpretar a mensagem. Seguindo a seguinte codificação:

m - Mensagem do Mouse,

k - Keyboard,

r - Multiplayer ready,

u - Multiplayer unready.

A comunicação entre as máquinas deve, de forma a potenciar o desempenho do jogo, restringir-se ao essencial. Neste caso, aos comandos referidos acima. Como referido anteriormente, a identificação do tipo de mensagem é realizada através de um carater distintivo “à cabeça da mensagem”. Após a detenção desse caracter, desencadeiam-se metodologias fixas, dependendo do tipo de mensagem recebida:

Rato:

Leitura de um caracter “m” a sinalizar que um mouse packet chegou

```
if (aux == 'm')
{
    while (i < 3)
    {
        if (sys_inb(LINE_STATUS_REG, &LSR_read))
        {
            return -1;
        }
        if (LSR_read & BIT(LSR_RECEIVER_DATA_BIT))
        {
            if (i == 0)
            {
                if (sys_inb(RECEIVER_BUFFER_REG, &mouse_byte))
                {
                    return -1;
                }
                i++;
            }
            else if (i == 1)
            {
                if (sys_inb(RECEIVER_BUFFER_REG, &mouse_byte1))
                {
                    return -1;
                }
                i++;
            }
            else if (i == 2)
            {
                if (sys_inb(RECEIVER_BUFFER_REG, &mouse_byte2))
                {
                    return -1;
                }
                i++;
            }
        }
    }
    i = 0;
}
```

Leitura dos 3 bytes do mouse packet do RECEIVER_BUFFER_REG

Keyboard:

```
else if (aux == 'k')
{
    if (sys_inb(RECEIVER_BUFFER_REG, &scancode2))
    {
        return -1;
    }
    start->scancode2 = (uint8_t)scancode2;
    kbd_processing_p2(start);
}
```

Receção de um “k” a simbolizar que se trata de uma mensagem do KB.

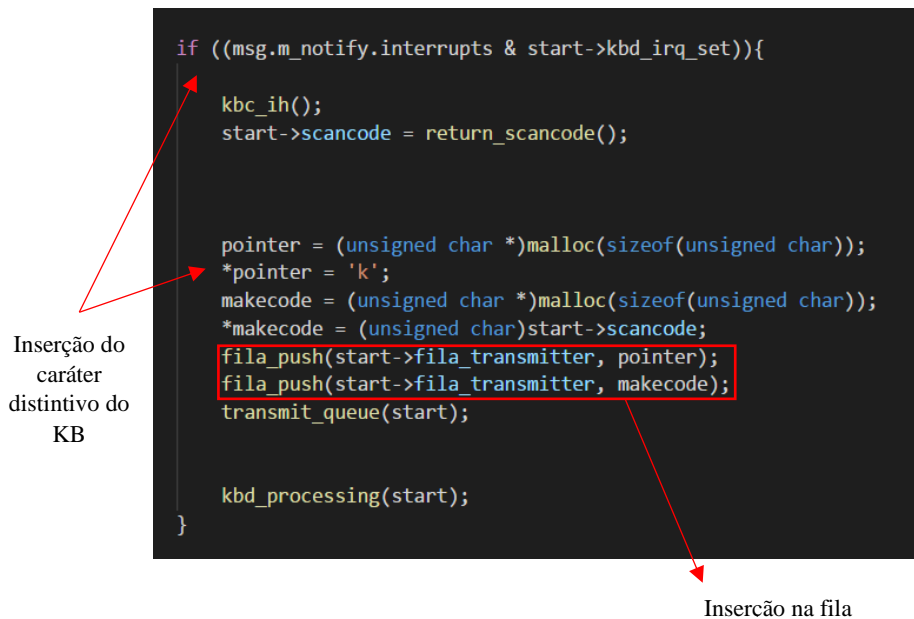
São desencadeados os procedimentos de leitura e interpretação

Controlo do Multiplayer:

Controlo do Multiplayer

```
else if (aux == 'r')
{
    start->maquina2_pronta_para_multiplayer = 1;
}
else if (aux == 'u')
{
    start->maquina2_pronta_para_multiplayer = 0;
}
```

- **Implementação de filas / Interação com os FIFOs:** A interação com o FIFO pressupõe a comunicação a nível do recetor e do transmissor em ambas as máquinas, desta forma introduz-se um novo elemento, o uso de Filas dinâmicas (PAG. 40):
 - **Transmissão de dados dinâmicos:** Quando se gera um interrupt do mouse ou do KB em ambas as máquinas comunicantes, finda a chamada do Handler essa informação é ainda transmitida para o P2 de modo a preservar a integridade do jogo em Multiplayer. Tal como indicado previamente (PAG.33), “à cabeça” de cada informação é introduzido um carater distintivo, só depois os dados correspondentes. Dependendo do caso, os dados a enviar podem ter dimensões variáveis, como tal, é imperioso a utilização de um sistema dinâmico de armazenamento de dados. **De modo a facilitar a compatibilidade e a portabilidade da biblioteca, pelo uso de FIFOs, decidi armazenar-se esses dados numa Fila (de transmissão).**



- **Receção:** A impossibilidade de operar em modo interrupts, e consequente uso de polling para operar a UART, obriga que a receção de informação seja feita de forma periódica. Desta feita, optou-se por efetuar essa leitura durante a invocação à função **void update_variables_p2(boot_resources *start)**, que à semelhança da função com nome idêntico descrita na referência ao timer (PAG. 8), a sua chamada ocorre a cada interrupt do timer. (em modo multiplayer). A receção da informação tem como elemento central a invocação de **unsigned char receive_char(boot_resources *start)**, rotina que integra o módulo **uart.c**.

Quadro Resumo dos IO devices, funcionalidades e o modo utilizado para cada um:

Device	Funcionalidades	Modo
Timer	Auxiliar da memória gráfica, controlo do flow do jogo	Interrupts(I)
Keyboard	Movimentos da Personagem e menus de pausa	I
Mouse	Utilização dos menus, utilização das medidas ofensivas	I
Graphics Card	GUI	Polling(P)
RTC	Contabilização do momento onde são estabelecidos records	P
UART	Multiplayer	P

Code Organization/Structure:

Demon Slayer é o fruto do trabalho de dois estudantes do 2º ano do MIEIC para o projeto final de LCOM (Laboratório de Computadores).

O projeto foi desenvolvido sempre que possível em conjunto para evitar incompatibilidades motivadas pela inexperiência dos programadores. Apesar deste facto, listam-se abaixo, um sumário do papel de cada um no desenvolvimento dos módulos usados, bem como um resumo final das funcionalidades de cada módulo.

Módulos desenvolvidos nas aulas práticas: Previamente descritos nas Google Forms:

- **Timer:**
 - I8254: Peso Relativo: 1%
 - Timer.c: Peso Relativo: 3%
- **KBC.C:**
 - Peso Relativo: 4%
- **KBD:**
 - Peso Relativo: 4%
- **I8042:**
 - Peso Relativo: 1%
- **Mouse:**
 - Mousedriver: Peso Relativo: 7%
- **Graphics Card.**
 - VBE: Peso Relativo: 4%

Desenvolvimento autónomo:

- **RTC,**
- **UART,**
- **Game.c,**
- **Proj.c,**
- **Statemachine.c,**
- **Graphics.c.**

Importação da WEB (com alterações):

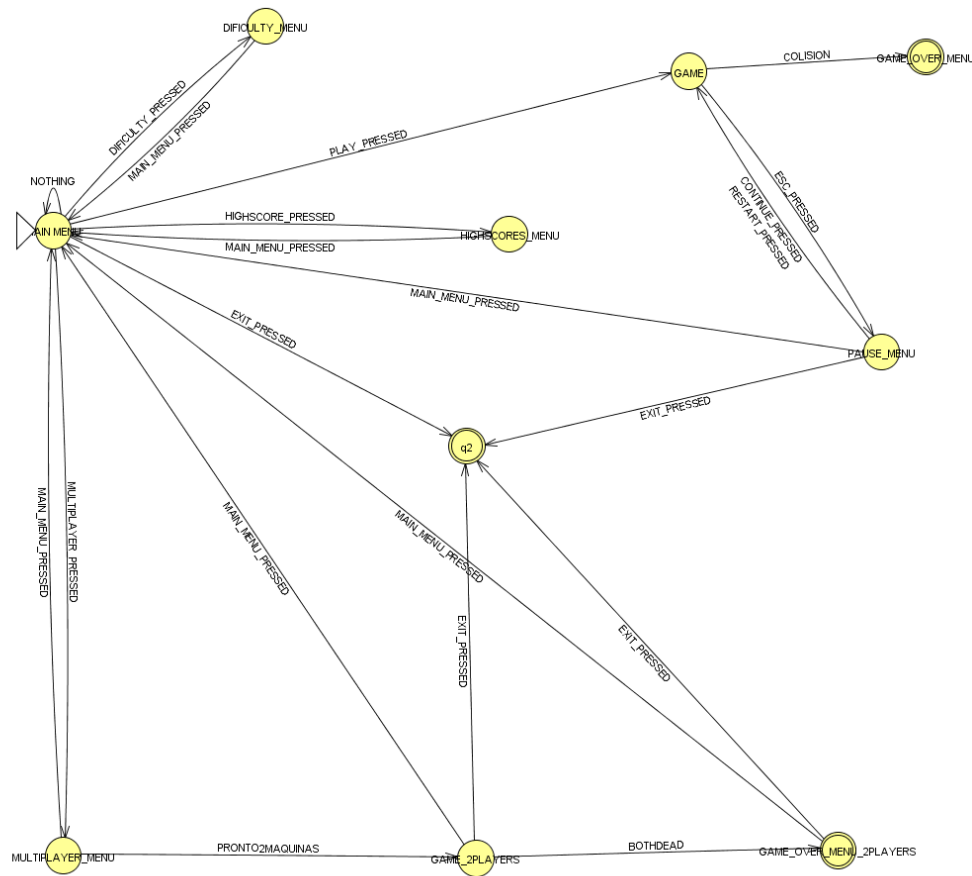
- **Bitmap.c,**
- **Vetor.c,**
- **Fila.c.**

Nota: A omissão da referência ao responsável da implementação deve ser interpretada como um desenvolvimento em conjunto (50/50).

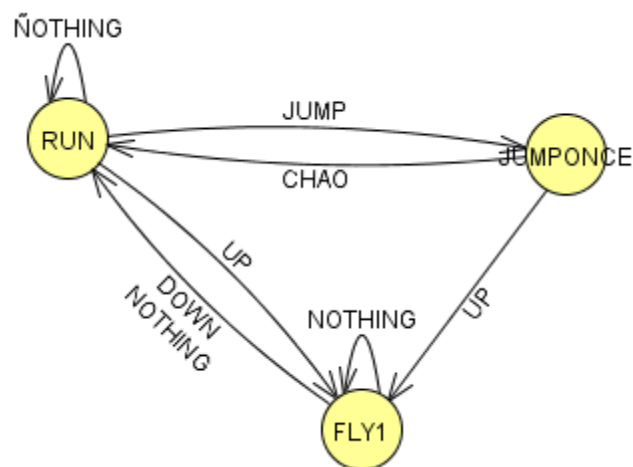
- **RTC (Desenvolvido por Rúben Almeida) (PAG. 25):**
 - O RTC é utilizado em polled mode para contabilizar um registo temporal do momento onde são batidos novos records,
 - Peso Relativo: 3%
- **UART (PAG. 33):**
 - Usado na comunicação entre máquinas. Para o estabelecimento de um modo multiplayer,
 - Peso Relativo: 7%
- **Game.c:**
 - Controla o flow do jogo,
 - Inicialização e atualização de variáveis,
 - Estruturas relevantes:
 - Boot_Resources:
 - Peso Relativo: 18%.
- **Proj.c:**
 - Função Main(),
 - Inicialização do modo gráfico,
 - Retorno ao modo texto predefinido,
 - Peso Relativo: 1%.
- **Graphics.c:**
 - Detecção de Colisões (José Guerra),
 - Trabalho de edição de imagem (José Guerra),
 - Highscores (Rúben Almeida),
 - Projeteis (adaptado em conjunto),
 - Inimigos,
 - Double Buffering.
 - Peso Relativo: 24%

- **Statemachine.c (Desenvolvido por José Guerra):**

- Implementação das máquinas de estado que controlam o flow do jogo,
- Peso Relativo: 8%.



Máquina de estado que controla o estado do jogo



Máquina de estado que controla o estado da personagem

- **Bitmaps.c:**

- Importado do trabalho de Henrique Ferrolho
- (<http://difusal.blogspot.com/2014/07/minix-posts-index.html>).
- Adaptado por José Guerra,
- Peso Relativo: 4%

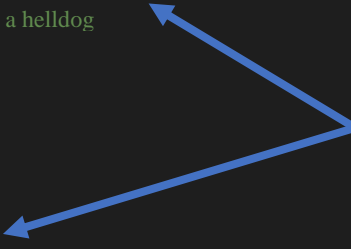
Memória dinâmica:

- **Vetor.c:** (Adaptado por Rúben Almeida):

- Importado das bibliotecas fornecidas em PROG2 do MEEC, de autoria do Prof.Dr. Luís Teixeira e da Prof. Dra. Ana Paula Rocha.
- (<https://moodle1718.up.pt/course/view.php?id=585> - acedido em 2018/12/29).
- Armazenamento da informação relativa a inimigos e projéteis.
- Peso Relativo: 8%

Estrutura:

```
enum Element_type
{ Is_Projectil, //if element is a fireball(projectile)
  Is_Enemy    // if elements is a enemy, axe_taurus, hell_dog, or ghost
};
enum Enemy_type
{ Is_Axe_taurus, //if enemy type is a axe_taurus
  Is_Fantasma,  //if enemy type is a ghost
  Is_Helldog   // if enemy is a helldog
};
typedef struct
{ enum Enemy_type type;
  union xwz {
    struct Axe_taurus *e1;
    struct Fantasma *e2;
    struct Helldog *e3;
  } union_enemies;
} enemies;
```



Recurso a tipos enumerados interligados com Unions de modo a assegurar uma fácil integração de elementos do mesmo estilo

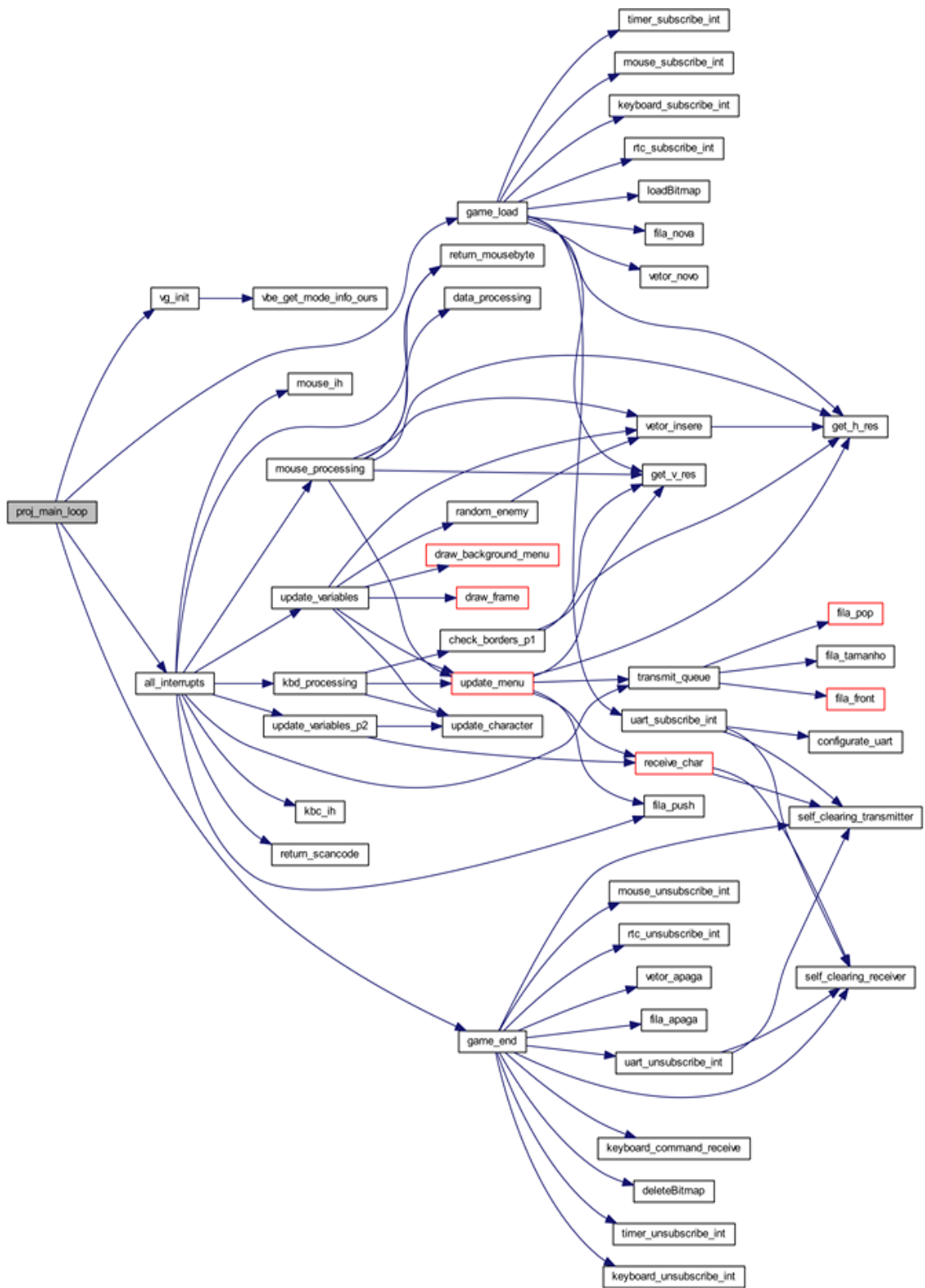
• **Fila.c: (Adaptado por Rúben Almeida):**

- Importado das bibliotecas fornecidas em PROG2 do MEEC, de autoria do Prof.Dr Luís Teixeira e da Profa. Dra. Ana Paula Rocha,
- (<https://moodle1718.up.pt/course/view.php?id=585> - acedido em 2018/12/29),
- Armazenamento da informação relativa à transmissão de dados pela UART,
- Peso Relativo: 3%

Estrutura:

```
struct fitem
{
    unsigned char* string;
    struct fitem *proximo;
};
typedef struct fitem filaItem;
/**
 * \brief registo para suportar uma fila de strings com base em lista
 * este registo contem um apontador para a cabeca da lista de strings
 * e outro para a cauda
 */
typedef struct
{
    filaItem *cabeca;
    filaItem *cauda;
} fila
```

Function call graph



Implementation Details:

- **Desenho do Fundo em movimento:** De forma a reproduzir a sensação de movimento, o desenho do fundo em `void draw_background (boot_resources *start)` é efetuando recorrendo a um bmp com duas propriedades:
 - A imagem com dimensões superiores às que o modo gráfico permite representar,
 - O fundo escolhido ser uma imagem com propriedades de simetria face às extremidades laterais. Esta propriedade, assegura uma transição suave sem “saltos” gráficos criando uma experiência mais imersiva no jogador.



Simetria do fundo

Esta propriedade gráfica encontra-se intimamente ligada a um detalhe do código, cuja robustez da implementação, oferece a facilidade de ser aplicada sobre qualquer fundo. **O desenho em espaço modular.**

- **Através do desenho em espaço modular,** garantimos que independentemente da dimensão do fundo, podemos imprimir uma velocidade arbitrária ao fundo, pois quando as extremidades forem alcançadas, o desenho gráfico voltará ao início. A existência de uma imagem com propriedades simétricas **garante assim que esta operação modular aos olhos do utilizador seja impercetível.**

```
void draw_background(boot_resources *start)
{
    unsigned char *temp = start->background_sprite.bmp->bitmapData;

    const int half_width = (start->background_sprite.width / 2);
    const int half_width_pixel = half_width * (get_bits_per_pixel() / 8);

    start->background_sprite.x = (start->background_sprite.speed + start->background_sprite.x) % half_width;
    temp += start->background_sprite.x * (get_bits_per_pixel() / 8);
}
```

Desenho em espaço modular

- **Desenho linha a linha da porção de imagem representada:** Partindo da definição de formato **bmp** e da função **memcpy**, somos capazes de desenhar nas dimensões adequadas ao formato gráfico.

```
for (int i = 0; i < start->background_sprite.height; i++)
{
    memcpy(get_double_buffer() + i * half_width_pixel, temp + 2 * i * half_width_pixel, half_width_pixel);
}
```

- **Mecanismo de simulação da ação da gravidade:** De forma a garantir um maior grau de realismo ao jogo, *Demon Slayer* suporta um mecanismo que, quando em voo, a personagem por predefinição perde altitude de forma regular, até a um mínimo estabelecido pela posição do chão (FLOOR_POSITION), aí a personagem evolui para o estado normal de corrida. Esta predefinição pode ser facilmente contrariada pelo recurso à tecla “W”, cuja funcionalidade, atrás descrita, resume-se ao aumento da altitude da mesma.
- **O teste de Colisão Pixel-a-Pixel só é implementado entre a personagem e os inimigos:** De forma a garantir que não ocorra uma sobrecarga computacional, dado os recursos que esta técnica pressupõe, optou-se por restringir esta técnica, somente, entre a personagem e os inimigos.
- **Implementação de um modelo de comunicação com a UART tipo polling:** Apesar das tentativas iniciais de implementar a UART em modo de interrupts, como previamente referidos, não fomos capazes de concluir com sucesso essas demandas, desta forma, podem-se verificar alguns problemas acrescidos de delay e de quebra de performance no jogo no modo multiplayer.
- **Animação especial de “flash” intermitente da personagem ao colidir com inimigos:** Com o objetivo de estabelecer uma certa nostalgia ao utilizador, implementa-se em *Demon Slayer* uma das animações gráficas mais características deste estilo de jogos, a intermitência do jogador ao “perder uma vida”.
- **Uso de filas em conjunto com FIFOs:** De forma a atingir um grau de proximidade superior na relação Software/Hardware, dada o recurso da UART em modelo de FIFOs, implementam-se filas para garantir a integridade dos dados, quer transmitidos, quer recebidos.
- **Utilização do mouse no modo standard Vs Rage mode:** De forma a conferir unicidade e dinamismo ao Rage Mode, o modo como os packets do mouse são efetuados divergem. **No modo standard de jogo**, mesmo que o utilizador mantenha premido o botão de disparo, os mesmo não serão efetivados. Essa limitação, planeada, deixa de existir em **Rage Mode**.



Stream de tiros exclusiva de **Rage Mode**

- **A biblioteca garante que a personagem não adquiere comportamentos impressionantes de desaparecimento do ecrã, ou semelhantes:** A implementação de funções como **void check_borders_p1(boot_resources *start)**, ou sucessivas verificações de posições são o garante que os comportamentos usuais e irritantes que marcam este estilo de jogos estão, em *Demon Slayer*, controlados.

Conclusions:

Demon Slayer consegue ser uma forma prática de demonstração dos conhecimentos adquiridos na cadeira de LCOM. A interação homem-máquina que um jogo consegue alcançar demonstrou ser uma forma fácil de demonstrar a cada semana os progressos que eram alcançados.

Consegue-se, apesar das dificuldades que um projeto desta índole origina, implementar todos os I/O devices abordados em LCOM, sendo esse na opinião dos autores a maior demonstração de sucesso. Neste documento estão, desta forma, descritas as funcionalidades, mas também a forma como estas foram implementadas, de modo a que o leitor consiga estabelecer o link de abstração necessária para a implementação das mesmas com maior facilidade e de forma mais pedagógica.

Por último, gostaríamos de estabelecer algumas recomendações para colmatar dificuldades com que nos deparamos quer ao longo das aulas laboratoriais, quer no projeto.

- Apresentação de sugestões como um jogo deva ser organizado,
- No mini-teste, optar pela programação de um periférico não utilizado, ou, invés, procurar estabelecer outras funcionalidades que não colidam com os objetivos das Labs, potencialmente criadoras de dúvidas durante a realização do mesmo.
- Referir, com exemplos, inclusive, a existência de bibliotecas na internet úteis nos desenvolvimentos de determinados projetos, por exemplo, a **Bitmap.c** por nós utilizada.
- Utilização do sistema de controlo de versões **Github**. O sistema SVN possui documentação precária e esteve na origem de conflitos de resolução morosa durante o projeto que não seriam verificados no sistema Git.

Os autores:

José Barbosa da Fonseca Guerra,

Rúben Filipe Seabra de Almeida.