

OPERATION M.I.N.I.X.

André Júlio Moreira 201904721

Nuno Miguel da Silva Alves 201908250

Índice

1	Introdução	3
1.1	O que é OPERATION M.I.N.I.X.?	3
1.2	Objetivos	4
2	Livro de Instruções	5
2.1	Menu Principal	5
2.2	Menu <i>Multi Player</i>	6
2.3	Interface Jogo-Utilizador	7
2.3.1	<i>HUD (Heads-Up Display)</i>	9
2.3.2	Escolha de Armas	10
2.4	Menus de Fim de Jogo	11
3	Estado do Projeto	12
3.1	Timer	12
3.2	Teclado	13
3.3	Rato	13
3.4	Placa Gráfica	14
3.5	RTC	14
3.6	UART	15
4	Organização/Estrutura do Código	16
4.1	Código Próprio	16
4.1.1	Timer	16
4.1.2	KBC	16
4.1.3	Placa Gráfica	16
4.1.4	Sprite	17
4.1.5	RTC	17
4.1.6	Filas (Queues)	17
4.1.7	UART	17
4.1.8	Controlador de inimigos	18
4.1.9	Controlos	18
4.1.10	Mapa	18
4.1.11	Gestor de Jogo	19
4.1.12	Gestor de Níveis	19
4.1.13	Rendering	19
4.1.14	HUD	20
4.1.15	Armas	20
4.1.16	Menu de Inventário	20
4.1.17	Protocolo de Comunicação	21
4.1.18	Pickups	21
4.1.19	Menu Principal	21
4.1.20	Funções Auxiliares	22
4.1.21	Entidade Jogador	22
4.2	Código Externo Apropriado	23

4.2.1	Raycasting	23
4.2.2	Rendering de Inimigos/Pickups e Scaling de Sprites	24
4.2.3	Deteção de Colisões	25
4.2.4	Tabelas Virtuais e Polimorfismo	25
4.3	Gráficos de Chamadas	26
5	Detalhes de Implementação	30
5.1	Implementações Interessantes	30
5.1.1	Deteção de Colisões	30
5.1.2	Tabelas Virtuais e Polimorfismo	33
5.1.3	Raycasting	33
5.1.4	Rendering de Inimigos/Pickups e Scaling de Sprites	35
5.1.5	Escurecimento de Paredes e Sprites	37
5.1.6	Ordenação de Sprites	38
5.2	Implementações de Temas Abrangidos em Aula	39
5.2.1	Page Flipping	39
5.2.2	Protocolo de Comunicação	39
5.2.3	Filas UART e FIFO	42
5.2.4	Armas e máquinas de estado	42
6	Conclusão	45
7	Bibliografia	46
8	Apêndice	47
8.1	Anexos	47
8.1.1	Anexo A - Instruções de Instalação	47
8.1.2	Anexo B - Imagens e fontes	47
8.1.3	Anexo C - Links	47
	Glossário	49

1 Introdução

1.1 O que é OPERATION M.I.N.I.X.?



Figura 1: Um momento inoportuno

OPERATION M.I.N.I.X. é um jogo, *pseudo-3D*, desenvolvido no âmbito da unidade curricular **LCOM** por André Júlio Moreira e Nuno Miguel da Silva Alves.

O objetivo do jogo é escapar da prisão **El Come** onde o jogador foi encarcerado injustamente. Enquanto o jogador foge da prisão terá de ”*incapacitar*” os guardas prisionais e tentar escapar antes que a bomba **Ex Amis Finalis**, localizada no centro da prisão, expluda.

Para isso, o jogador tem ao seu dispor um número variado de armas e bens espalhados pela prisão que o vão ajudar a sobreviver.

No modo *Multi Player* dois jogadores têm de lutar até à morte. O sobrevivente ganhará o jogo.

1.2 Objetivos

Todos os objetivos desta unidade curricular foram cumpridos:

- Foram usados todos os periféricos desenvolvidos em aula bem como aqueles que foram referidos apenas nas aulas teóricas:
 - Timer
 - Rato
 - Teclado
 - Placa Gráfica
 - RTC
 - Uart
- Foi implementado ao longo do projeto um código modularizado, separando os respetivos módulos em diferentes pastas, agrupados com os do seu tipo. Por exemplo, a pasta `system/lib` continha todos os módulos de mais baixo nível que permitiam a interação com os periféricos.
- Foi usado o `git` como o `Version Control System`, tendo sido usado várias funcionalidades, como `commits` e `branches` para tornar a implementação de certas `features` mais fáceis, sem estar possivelmente a danificar o progresso que tínhamos até ao momento.

2 Livro de Instruções

2.1 Menu Principal



Figura 2: O Menu Principal

No menu principal (figura 2) são apresentadas várias opções.

Para iniciar o jogo no modo *Single Player* basta mover o cursor do rato [5] para o botão *Singleplayer* [1] e pressionar o botão esquerdo do rato. A mesma operação poderá ser efetuada se o utilizador preferir iniciar o jogo no modo *Multi Player* pressionando, ao invés, no botão *Multiplayer* [2], dando assim acesso ao *Menu Multi Player* (2.2).

Para desligar o jogo, move-se o cursor até ao botão *Exit* [3] e pressiona-se o botão esquerdo do rato. É também possível desligar o jogo, a qualquer momento, pressionando a tecla '**ESC**'.

No canto inferior esquerdo [4] é apresentado a data e a hora do sistema, atualizadas aos segundos.

2.2 Menu *Multi Player*



Figura 3: O Menu *Multi Player*

No Menu *Multi Player* (figura 3), se o utilizador quiser cancelar a procura de conexão, poderá pressionar no botão **Exit** [1] ou esperar por volta de 30 segundos. Neste processo, o utilizador deverá voltar ao **Menu Principal** (2.1). Se existir uma conexão possível com outro jogador iniciar-se-á instantaneamente o jogo no modo multi jogador.



Figura 4: Visão Jogador nº 1

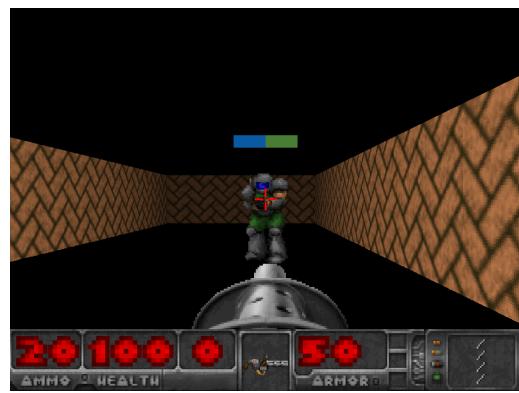


Figura 5: Visão Jogador nº 2

2.3 Interface Jogo-Utilizador

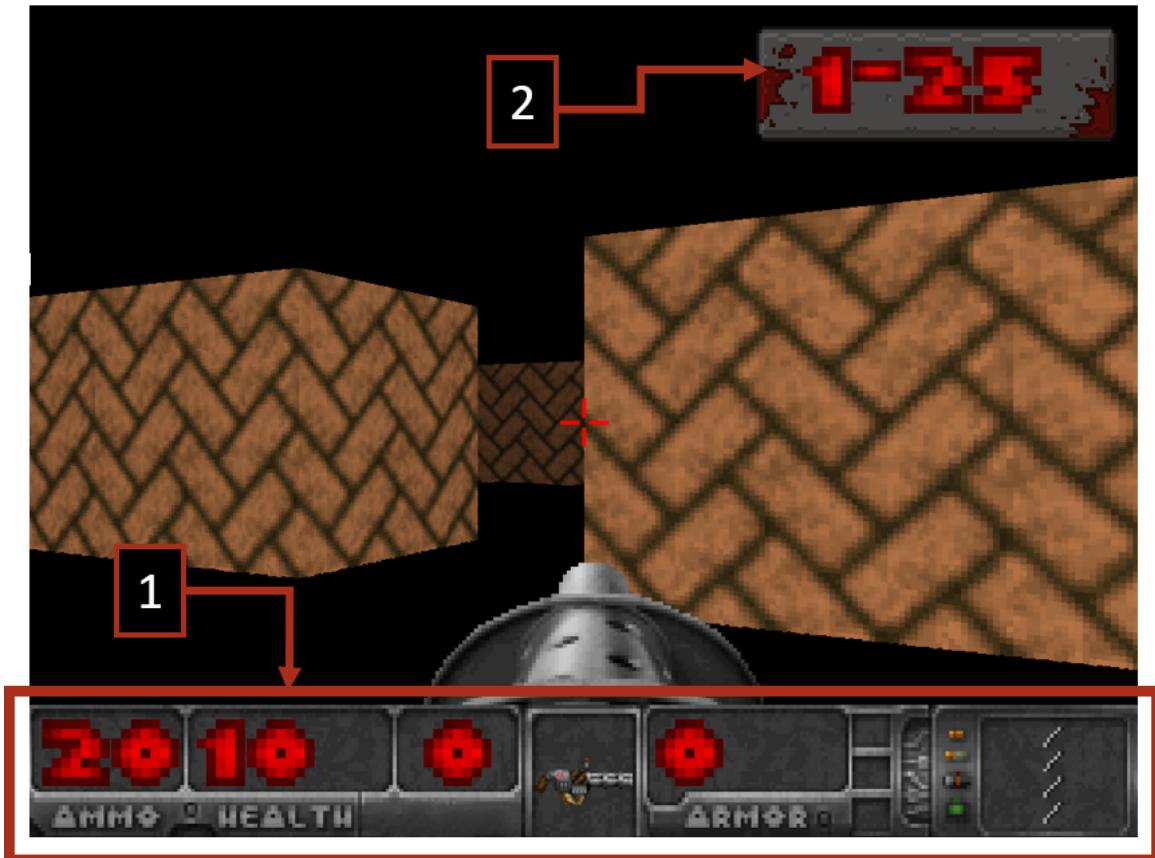


Figura 6: Interface Gráfica Jogo-Utilizador

Para movimentar o jogador usam-se as teclas '**W**' para andar em frente, '**S**' para recuar, '**A**' para deslocar para a esquerda e '**D**' para deslocar para a direita. Pode-se ainda segurar as teclas '**CTRL**' e '**Shift**' para andar mais devagar ou mais rápido, respetivamente. Para rodar a câmera usa-se o rato **deslizando** para a direita ou para a esquerda.

Para disparar a arma selecionada basta pressionar o **botão esquerdo** do rato, se o jogador tiver balas. Se o jogador ficar sem balas e ainda tiver "recargas", essas serão usadas para recarregar a arma selecionada.

No canto superior direito [2] encontra-se o temporizador da bomba. Quando este temporizador atinge zero, o **Menu de Fim de Jogo** (2.4) surgirá e o jogador perde o jogo.



Figura 7: *Pickups* que podem ser apanhados do chão.



Figura 8: *Pickups* de armadura (esquerda) e de vida (direita).



Figura 9: *Pickup* de recargas para a *Minigun*.



Figura 10: *Pickup* de recargas para a *Service Pistol*.

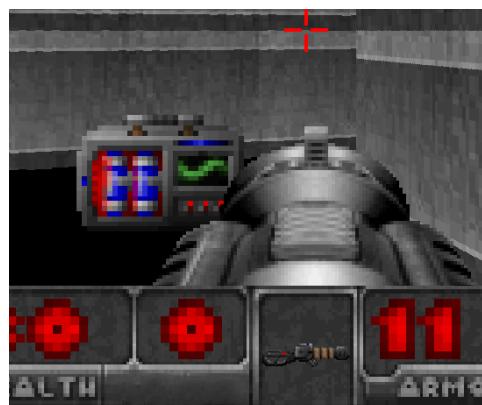


Figura 11: *Pickup* de recargas para a *BLMG 5100*.

2.3.1 **HUD** (*Heads-Up Display*)

O **HUD** (figura 6)[1] apresenta várias informações importantes.

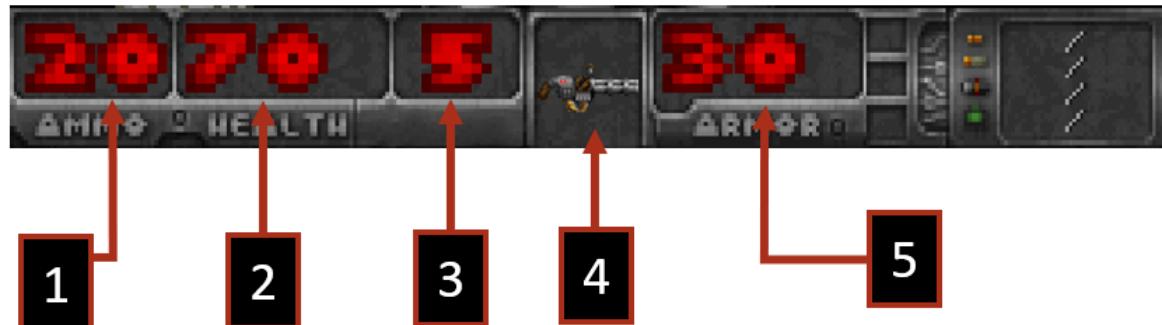


Figura 12: O **HUD**

- 1 - Apresenta a quantidade de balas da arma selecionada ao dispor do jogador.
- 2 - Apresenta a quantidade de vida do jogador.
- 3 - Apresenta o número de "recargas" de balas da arma selecionada ao dispor do jogador.
- 4 - Apresenta o ícone da arma selecionada.
- 5 - Apresenta a quantidade de armadura do jogador.

2.3.2 Escolha de Armas

Para escolher uma das 3 armas disponíveis usam-se os números do teclado, nomeadamente as teclas '**1**', '**2**' e '**3**', que escolhem a *Minigun*, a *Service Pistol* e a *BLMG 5100* respetivamente.

Como alternativa, pode-se escolher uma arma através do menu de inventário, segurando a tecla '**Q**' enquanto movimenta o rato (figura 13).

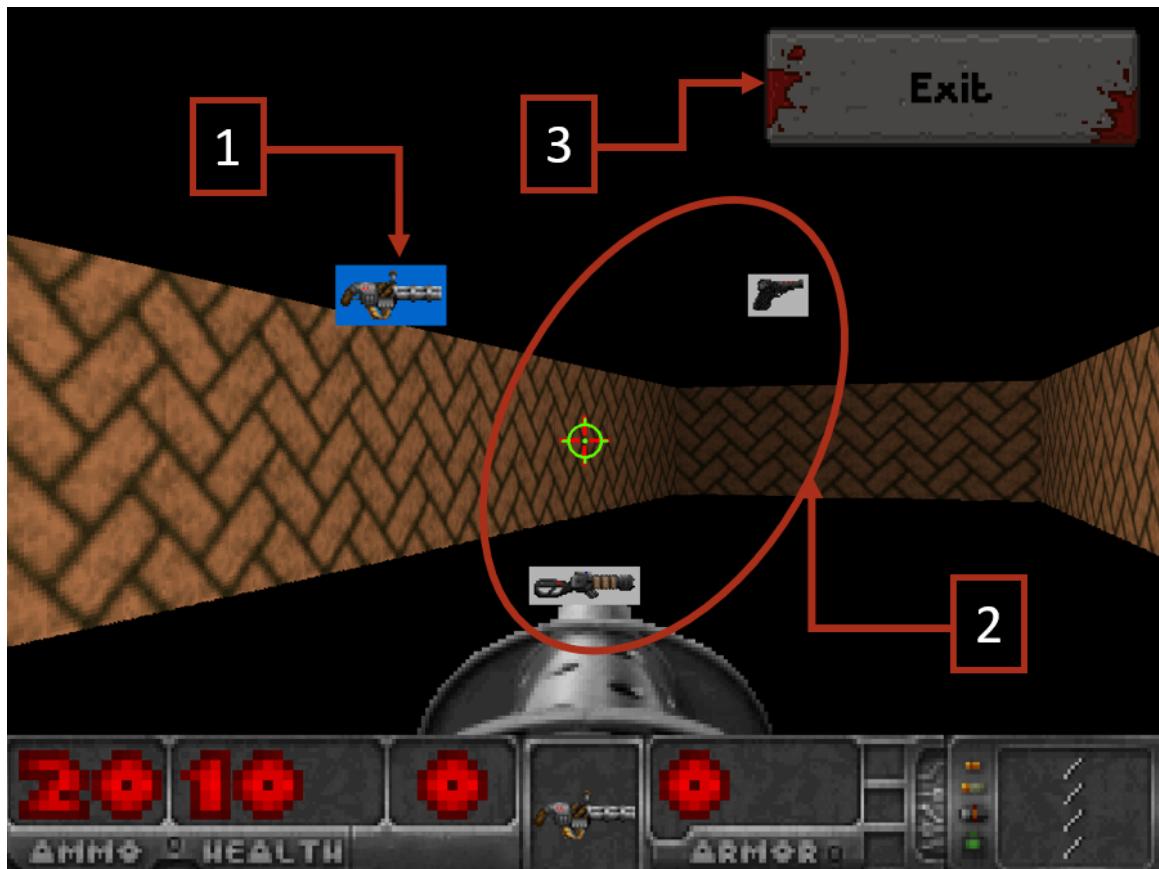


Figura 13: O Menu de Inventário

A arma selecionada apresenta um fundo **azul** [1]. As armas no inventário apresentam um fundo **cinzento** [2]. Para selecionar uma arma, no inventário, basta mover o **cursor do rato** para cima do ícone da arma a selecionar e pressionar com o **botão esquerdo do rato**.

No canto superior direito [3], no lugar do temporizador da bomba, aparece um botão de **saída**. A utilização deste botão implica a **saída** do jogo atual para o **Menu Principal** (2.1) se o jogo estiver no modo *Single Player*, ou, se estiver no modo *Multi Player*, para um **Menu de Fim de Jogo** (2.4) para os dois jogadores.

2.4 Menus de Fim de Jogo



Figura 14: Jogador Ganhou



Figura 15: Jogador Perdeu

Os **Menus de Fim de Jogo** (Figuras 14, 15 e 16) aparecem quando o jogo termina. Na parte superior [1] aparece uma mensagem explicitando se o jogador **ganhou** (Figura 14) ou **perdeu** (Figura 15). Além disso, quando existem mais níveis para jogar aparece um botão a dizer *Next*, caso tenha ganho, ou para tentar novamente caso tenha perdido, botão **Restart**.

Como no **Menu Multi Player** (2.2), na parte inferior [2], aparece um botão para voltar ao **Menu Principal** (2.1). Além disso, também apresenta um botão de **Restart** para reiniciar o modo *Multi Player*.



Figura 16: Jogador foi Desconectado

Na possibilidade do outro jogador desligar o jogo subita e prematuramente, é apresentado, após 8 segundos sem receber informação, um **Menu de Desconexão** com funcionalidade igual aos outros **Menus de Fim de Jogo**.

3 Estado do Projeto

Todas as funcionalidades apresentadas no **Livro de Instruções** (2) foram implementadas com sucesso. Na tabela 1 são demonstrados todos os periféricos implementados no projeto.

Periféricos Implementados		
Nome do Periférico	Utilização	Uso de Interrupções
Timer	Definir uma <i>frame rate</i> ; Atualizar estado do jogo; Temporizador de animações, estados de armas, entre outros.	Sim
Teclado	Deslocar o jogador; Abrir Menu de Inventário.	Sim
Rato	Rodar o jogador; Disparar tiros; Navegar menus.	Sim
Placa Gráfica	<i>Display</i> do Jogo e Menus.	Não
RTC	Ler a data e hora do sistema; Atualizar a data e a hora a cada segundo; Temporizador da bomba.	Sim , periódicos e de atualização
UART	Conectar dois jogadores no modo <i>Multi Player</i> .	Sim , adaptadas para FIFOs

Tabela 1: Periféricos implementados.

3.1 Timer

São utilizadas as interrupções do **Timer** com o propósito de:

- Definir uma *frame rate*, neste caso *60fps*.
- Atualizar animações.
- Processamento de máquinas de estado para as armas, *networking* e o jogo em si.
- Atualizar físicas como por exemplo velocidades e colisões.

O **Timer** está implementado no ficheiro `system/lib/timer.c`

3.2 Teclado

São utilizadas as interrupções do **Teclado**¹ com o objetivo de:

- Controlar o deslocamento do utilizador com as teclas '**W**', '**A**', '**S**', '**D**', '**CTRL**' e '**Shift**', como explicado na secção 2.3.
- Abrir o menu de inventário segurando na tecla '**Q**', como explicado na secção 2.3.2.
- Mudar a arma usando as teclas '**1**', '**2**' e '**3**' como explicado na secção 2.3.2
- Forçar a saída do jogo pressionando na tecla '**ESC**', como explicado na secção 2.1.

O **Teclado** está implementado no ficheiro `system/lib/kbc.c` e os dados recebidos são tratados na função `kbd_parse_data()` do ficheiro `system/controls/controls.c`.²

3.3 Rato

São usadas interrupções de **Rato**³ com o objetivo de:

- Controlar os menus através do movimento do rato e do botão esquerdo. (secção 5.1.1)
- Controlar a visão do jogador para rodar a câmera do jogador ao movimentar o rato.
- Interagir com as armas ao disparar com o botão do lado esquerdo do rato. (secção 5.2.4)

O **Rato** está implementado no ficheiro `system/lib/kbc.c` e os dados recebidos são tratados na função `mouse_parse_data()` do ficheiro `system/controls/controls.c`.

¹`kbc_ih()` de `system/lib/kbc.c`

²Nota: As definições do **KBC** (ou seja do **Rato** e do **Teclado**) estão no ficheiro `/system/lib/i8042.h`.

³`mouse_ih()` de `system/lib/kbc.c`

3.4 Placa Gráfica

O modo utilizado é o 0x14C, de resolução 1152x864, modelo de cor **direto** e 32 *bits* por *pixel*.

A **Placa Gráfica** é usada para:

- Desenhar *pixels* no ecrã.⁴
- Animações e movimento de sprites.⁵
- Desenho das Paredes. (secção 5.1.3)
- Desenho de menus e **HUD**

As seguintes funcionalidades foram implementadas com o uso da **Placa Gráfica**.

- **Double Buffering**, com um terceiro *buffer*. (secção 5.2.1)
- **Page Flipping**. (secção 5.2.1)

A **Placa Gráfica** está implementada no ficheiro `system/lib/video_graphics.c`.⁶

3.5 RTC

São usadas interrupções⁷ do **RTC** com dois objetivos diferentes:

- Interrupções de **atualização** que são usadas para saber quando vai mudar a data, tendo o objetivo de atualizar o texto da data do menu.⁷
- Interrupções **periódicas** para atualizar um temporizador, de segundo a segundo, a cada 2 interrupções de 500ms. Este temporizador tem a função de terminar o jogo quando chegar a 0.⁷

O **RTC** está implementado no ficheiro `system/lib/rtc_controller.c`.⁸

⁴`set_px()` de `system/lib/video_graphics.h`

⁵`update_anim()` de `system/lib/sprites.c`

⁶Nota: As definições da **Placa Gráfica** encontram-se em `system/lib/video_cmd.h`

⁷`rtc_ih()` de `system/lib/rtc_controller.c`

⁸Nota: As definições do **RTC** encontram-se em `system/lib/rtc.h`.

3.6 UART

A **UART** é usada através de interrupções para envio e receção, em conjunto com as suas FIFO's, com o auxílio de filas implementadas no programa (secção 5.2.3). Está configurada com um **Baud rate** de 115200b/s⁹. São usadas interrupções do **UART**¹⁰ com dois objetivos diferentes:

- Interrupções de **Transmitter Holding Register Empty** para verificar quando se pode enviar nova informação, com auxílio à fifo de envio e a filas implementadas no nosso programa. (secção 5.2.3)
- Interrupções de **Receiver Holding Register Empty** para verificar quando existem dados para ser recebidos. Após uma interrupção, os dados são lidos da fifo da **UART** para uma fila, para depois serem processados. (secção 5.2.3)

Este periférico é crucial para tornar possível uma versão de 2 jogadores. Após a sincronização dos 2 pc's são enviados 3 bytes e recebidos 3 bytes por interrupção de timer (em condições normais). (secção 5.2.2)

A **UART** está implementada no ficheiro `system/lib/uart.c`.¹¹

⁹`uart_set_baud_rate()` de `system/lib/uart.c`, usado em `init_game()` no ficheiro `system/game/level_manager.c`

¹⁰`uart_ih()` de `system/lib/uart.c`

¹¹Nota: As definições da **UART** encontram-se em `system/lib/rs232.h`.

4 Organização/Estrutura do Código

4.1 Código Próprio

4.1.1 Timer

O timer foi usado para definir um **frame rate** do jogo, atualizar estado das animações, estado das armas, atualizar posições do jogador. Encontram-se neste módulo funções para configurar o timer, ativar e desativar interrupções.

- `system/lib/timer.c`

Importância: 7.2%

Contribuição: André 50% - Nuno 50%

4.1.2 KBC

Módulo importante para se fazer a ligação do mais alto nível do projeto aos periféricos do **Rato** e do **Teclado**. Contém as funções de baixo nível que permitem a configuração do rato e teclado, ativar e desativar interrupções bem como ler dados tanto do teclado como do rato.

- `system/lib/kbc.c`

Importância: 8.4%

Contribuição: André 50% - Nuno 50%

4.1.3 Placa Gráfica

Módulo importante para se fazer a ligação do mais alto nível do projeto à **Placa Gráfica**. Contém as funções de mais baixo nível que permitem a configuração dos modos da placa gráfica, chamadas à **VBE**, para obter informação e para permitir o **Page Flipping** e para desenhar sprites, pixéis e retângulos no ecrã.

- `system/lib/video_graphics.c`

Importância: 8.8%

Contribuição: André 50% - Nuno 50%

4.1.4 Sprite

Foi criado no módulo dos **Sprites** uma estrutura customizada `sprite_t` bem como uma estrutura `animated_sprite_t` para satisfazer as necessidades do projeto. Neste módulo é feita toda inicialização e carregamento de sprites, alocando e libertando a memória necessária. Também contém funções de atualização de estado de animações.

- `system/lib/sprites.c`

Importância: 3.2%

Contribuição: André 50% - Nuno 50%

4.1.5 RTC

Módulo importante para se fazer a ligação do mais alto nível do projeto ao periférico do **RTC**. Contém as funções de baixo nível que permitem a configuração do rtc, ativar e desativar interrupções e ler datas.

- `system/lib/rtc_controller.c`

Importância: 6.2%

Contribuição: André 70% - Nuno 30%

4.1.6 Filas (Queues)

Módulo onde se encontram funções para inicializar, interagir e apagar dados das `queues`. Foi criada uma estrutura `queue_t` baseada em listas ligadas.

- `system/lib/queue.c`

Importância: 2.6%

Contribuição: André 0% - Nuno 100%

4.1.7 UART

Módulo importante para se fazer a ligação do mais alto nível do projeto ao periférico **UART**. Contém as funções de baixo nível que permitem a configuração da **UART**, ativar e desativar

interupções, ler e enviar dados.

- `system/lib/uart.c`

Importância: 8.3%

Contribuição: André 30% - Nuno 70%

4.1.8 Controlador de inimigos

Módulo que faz a gestão dos movimentos e ações dos inimigos no modo de jogo *singleplayer*. Existem duas configurações possíveis para estes inimigos, uma em que vão escolhendo novas posições aleatórias periodicamente e outra em que anda entre posições pré-definidas. Assim que um jogador esteja no seu campo visual, este irá perseguí-lo e tentar atacar o jogador.

- `system/ai/enemy_controller.c`

Importância: 3.8%

Contribuição: André 85% - Nuno 15%

4.1.9 Controlos

Módulo que descodifica os dados obtidos pelo módulo **KBC**. Foi criada uma estrutura de dados `controls_t` que contém informação de que teclas estão a ser premidas, movimentos e estado do rato.

- `system/controls/controls.c`

Importância: 3.8%

Contribuição: André 70% - Nuno 30%

4.1.10 Mapa

Módulo que carrega informação de um mapa guardado na pasta de dados (`data/maps`), cuja extensão de ficheiro será `.mxmap`. Foi criada a estrutura `map_t` para guardar a informação numa *array 2D* que contém o identificador da textura e as dimensões respetivas do mapa.

- `system/file/map.c`

Importância: 1%

Contribuição: André 5% - Nuno 95%

4.1.11 Gestor de Jogo

Módulo que tem como objetivo inicializar e terminar o programa e os diferentes modos de jogo. Chama as funções de mais baixo nível necessárias a inicializar corretamente o jogo, por exemplo configuração dos dispositivos, subscrição de interrupções, carregamento de sprites, entre outras.

- `system/game/game_manager.c`

Importância: 4%

Contribuição: André 80% - Nuno 20%

4.1.12 Gestor de Níveis

Módulo que tem como função escolher os níveis a carregar e no fim libertação de memória, inicializar as definições do jogo, como quais as armas a serem usadas, inimigos e as suas características.

- `system/game/level_manager.c`

Importância: 2.5%

Contribuição: André 65% - Nuno 35%

4.1.13 Rendering

Módulo que contém as funções de mais alto nível relacionadas com gráficos. É neste que estão implementadas funções relacionadas com a técnica de **raycasting**, como desenhar paredes e preparar a renderização de inimigos. Também se encarrega de desenhar *sprites* específicos como a arma, renderizar números, ordenar os *sprites* dos inimigos e também a função mais alto nível `display(double range)`.

- `system/graphics/render.c`

Importância: 7.3%

Contribuição: André 20% - Nuno 80%

4.1.14 HUD

Módulo que desenha o HUD no ecrã. Vai buscar informação ao jogador e imprime números conforme a vida, armadura, número de balas, e qual a arma a utilizar no momento. Também mostra um temporizador de ronda atualizado pelo **RTC**.

- `system/graphics/hud.c`

Importância: 1%

Contribuição: André 100% - Nuno 0%

4.1.15 Armas

O módulo das armas tem definida as estruturas `gun_t` e `gun_instance_t` que representam as armas. Estão implementadas também funções para criar um conjunto de armas, inicializar as respetivas animações, atualizar os estados das armas (pronta a disparar, a recarregar, entre outros), recarregar a arma e adicionar pacotes de balas.

A *struct gun_t* tem alguns parâmetros que foram preparados para outras possíveis `features` que acabaram por não ser implementadas, como um modo diferente de disparo, `burst fire`, e o `spread`, que consistia em dar à direção dos tiros alguma instabilidade.

- `system/guns/gun.c`

Importância: 3.2%

Contribuição: André 60% - Nuno 40%

4.1.16 Menu de Inventário

Módulo que tem como função renderizar o menu quando a tecla **Q** se encontra premida, também permitindo interagir com os botões e chamar as funções correspondentes. Existem botões para escolher a arma a usar e também para sair do jogo.

- `system/inventory/menu.c`

Importância: 2.2%

Contribuição: André 90% - Nuno 10%

4.1.17 Protocolo de Comunicação

Módulo em que se encontra código de mais alto nível que envolve a **UART**, para estabelecer um protocolo de envio e receção de dados inerentes ao modo de jogo *multiplayer*, bem como iniciar a comunicação entre dois computadores através de sincronização.

- `system/networking/networking.c`

Importância: 7.3%

Contribuição: André 40% - Nuno 60%

4.1.18 Pickups

É neste módulo que estão implementados os **pickups**. Estes têm auxílio a tabelas virtuais de funções, explicado mais à frente. Estão implementadas também os vários tipos de **pickups**, as suas inicializações, libertar a memória alocada por estas operações e ainda verificar as colisões dos jogadores com estes elementos, através de um mapa de colisões.

- `system/pickups/pickups.c`

Importância: 3.8%

Contribuição: André 0% - Nuno 100%

4.1.19 Menu Principal

Módulo que tem como objetivo renderizar o menu principal e permitir ao utilizador a interação com o seu conteúdo, averiguando o clique nos menus e chamando as funções respetivas.

- `system/game/main_menu.c`

Importância: 2.3%

Contribuição: André 95% - Nuno 5%

4.1.20 Funções Auxiliares

Módulo que contém funções auxiliares, algumas relacionadas com matemática, **raycasting**, chamadas ao sistema, entre outras. Também contém novos tipos de dados como **vetores**, **linhas gráficas**, **data**, entre outros.

- **system/utils/utils.c**

Importância: 5.3%

Contribuição: André 50% - Nuno 50%

4.1.21 Entidade Jogador

O módulo do jogador contém uma estrutura de dados relativa ao jogador **player_info_t** que contém informações como posição, arma a ser utilizada, controlos. Neste módulo também existem funções para inicializar um jogador, inicializar as animações caso seja um inimigo e libertar a memória alocada por estas operações. É aqui também que existe a função **void(player_int_ih)(player_info_t * ply)**, que é de tamanha importância, pelo que vai atualizar a posição do jogador, atualizar animações caso seja um inimigo e verificar se o jogador está a tentar disparar e chamar a respetiva função.

- **system/player/player.c**

Importância: 5.8%

Contribuição: André 70% - Nuno 30%

4.2 Código Externo Apropriado

4.2.1 Raycasting

- **int**(display_map)(**double** range);

Função 1: De system/graphics/render.c

Esta função realiza o processo explicado na secção 5.1.3 e a projeção de texturas nas paredes.

Alterações feitas:

- Criação de **structs** e adaptação ao projeto, como a utilização da **struct player_info**.
- Separação em diferentes funções.
- Otimizações, como explicadas na secção 5.1.3.
- Implementação de um limite de visão explicado na secção 5.1.5.

Este código foi apropriado de: <https://lodev.org/cgtutor/raycasting.html> e <https://lodev.org/cgtutor/raycasting3.html>.

- **void**(shoot_ray)(vector_t* pos, vector_t* ray_dir, ray_hit_t* ray_hit, **double** range);

Função 2: De system/utils/utils.c

Esta função realiza o algoritmo DDA como referido na secção 5.1.3.

Alterações feitas:

- Separada da função **display_map()** e utilizada noutras contextos como verificar se um tiro atinge um inimigo ou se um inimigo consegue ver o jogador.
- Criação da **struct ray_hit** do ficheiro **system/utils/utils.h** usada para guardar o resultado do algoritmo.

Este código foi apropriado de: <https://lodev.org/cgtutor/raycasting.html>.

4.2.2 Rendering de Inimigos/**Pickups** e Scaling de Sprites

- **int**(render_object)(**render_t** * render, **vector_t** * rel_position, **double** range);

Função 3: De system/graphics/render.c

Esta função calcula o início da posição e largura de um *sprite* no mapa.

Alterações feitas:

- Diferente forma de verificar se o sprite está no ecrã, usando produtos internos e o **FOV** do jogador.
- Criação de **structs** como **render_t** para facilitar a ordenação de *sprites* como explicada na secção 5.1.6.

Este código foi apropriado de: <https://lodev.org/cgtutor/raycasting3.html>.

- **int**(draw_walls)(**int32_t** highest_wall, **int32_t** lowest_wall);

Função 4: De system/graphics/render.c

Apenas a linha de código `wall_line->texture.tex_y += wall_line->texture.step;` foi apropriada de: <https://lodev.org/cgtutor/raycasting.html>. Esta linha de código efetua o *scaling* vertical da textura da parede.

- **int**(draw_rect_sprite)(**pair_t** * end_pos, **pair_t** * offset, **pair_t** * bounding_box, **sprite_t** * sprite, **double** sprite_dist, **double** factor);

Função 5: De system/lib/video_graphics.c

Esta função projeta o *sprite* de um inimigo/**pickup** num plano numa posição do ecrã com uma largura definida.

Alterações feitas:

- Guardar a última cor utilizada de forma a reduzir o número de acessos à textura do *sprite*.
- Desenhar horizontalmente de forma a reduzir **cache misses**.

Este código foi apropriado de: <https://lodev.org/cgtutor/raycasting3.html>

4.2.3 Deteção de Colisões

```

• bool check_circle_rect_collision(circle_t * circle , rectangle_t
* rect);

void circle_rect_distance(circle_t * circle , rectangle_t * rect ,
vector_t * distances);

```

Função 6: De system/utils/utils.c

Estas funções verificam se existe colisão entre *colliders* circulares e retangulares.

A única alteração feita foi a criação das structs **circle** e **rectangle** do ficheiro system/utils/data_types.h.

Este código foi apropriado de: https://www.lazyfoo.net/tutorials/SDL/29_circular_collision_detection/index.php

Na função **check_player_pickup_collision()** de system/pickups/pickups.c também foi apropriada a verificação entre *colliders* circulares do mesmo site.

4.2.4 Tabelas Virtuais e Polimorfismo

```

• struct pickup_vtable;
struct pickup;
static inline bool pickup_handler(player_info_t * player ,
pickup_t * pickup);

```

Função 7: De system/pickups/pickups.h

Estas estruturas emulam uma **Tabela Virtual** como explicado na secção 5.1.2.

Alteração feita: adição do membro **void * pickup_details** à struct **pickup** que armazena informação adicional a ser usada na ”*função virtual*” **pickup_handler** que depende do tipo de *pickup*.

Este código foi apropriado de:

<https://stackoverflow.com/questions/8194250/polymorphism-in-c>.

4.3 Gráficos de Chamadas

A função `proj_main_loop` contém a chamada à inicialização do jogo, no modúlo `level_manager`, ao qual está inerente a subscrição de *interrupts* importantes como **timer**, **rato**, **teclado**, **UART** e **RTC**, incializando também a placa gráfica. Além disso também inicializa e carrega todos os *sprites* necessários para o decorrer do jogo.

Chama a função `interrupt_handler` que vai conter o loop do *driver receive*. Este vai verificar o estado do jogo em que se encontra e chamar algumas funções de receber dados da **UART**, caso esteja em *multiplayer*, chamar as funções de atualização dos jogadores e *bot's* e ordenar a renderização do ecrã, aquando as interrupções do **timer**. Quando receber interrupções de outros periféricos, os respetivos *handlers* serão chamados.

Por fim, chama a função `free_game()` definida em `level_manager`, que irá libertar toda a memória alocada pelo projeto.

Pode-se averiguar mais facilmente o *flow* de chamadas de funções no doxygen.

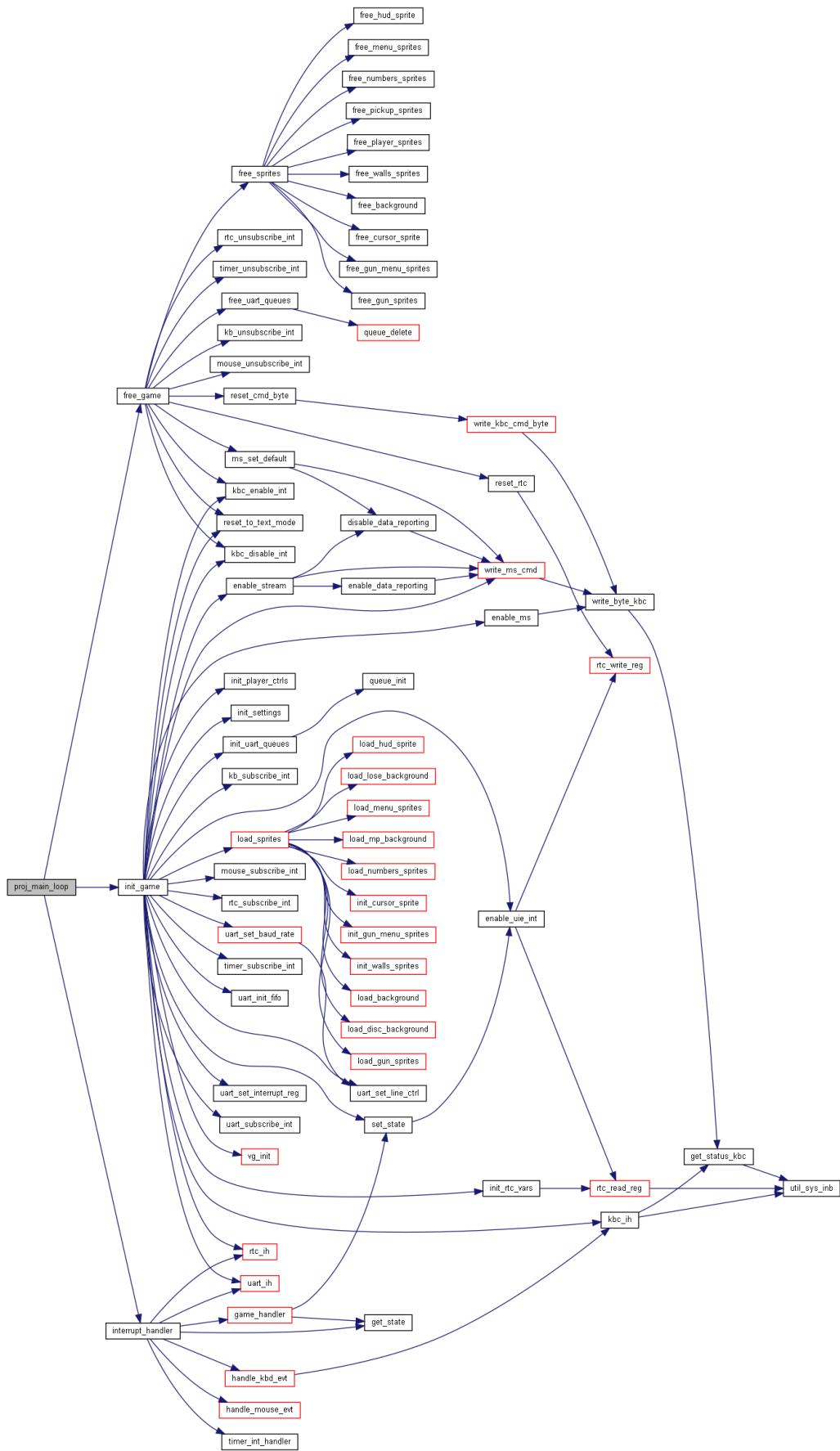


Figura 17: Gráfico de chamadas da função proj_main_loop

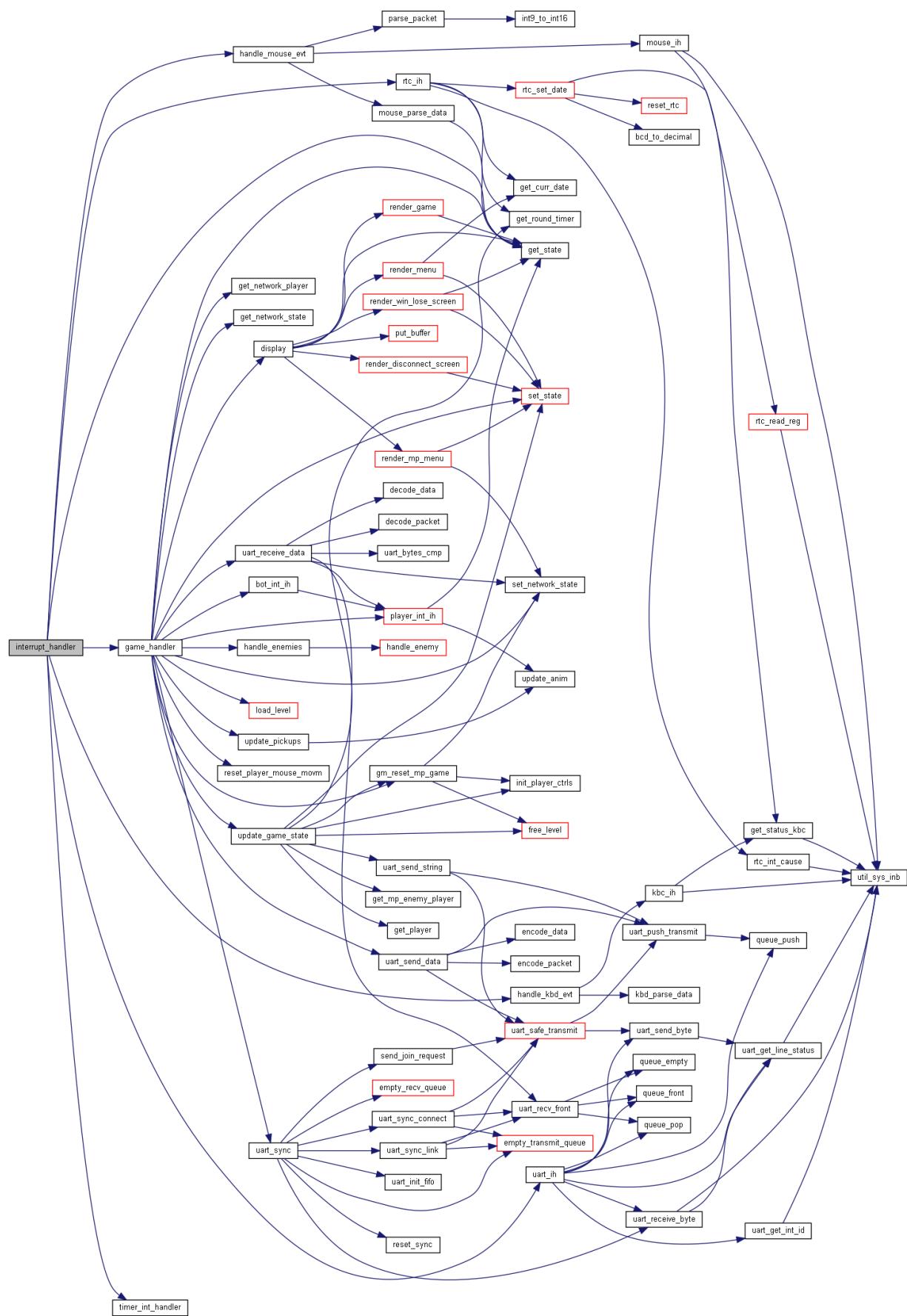


Figura 18: Gráfico de chamadas da função `interrupt_handler`
LCOM 2020/2021

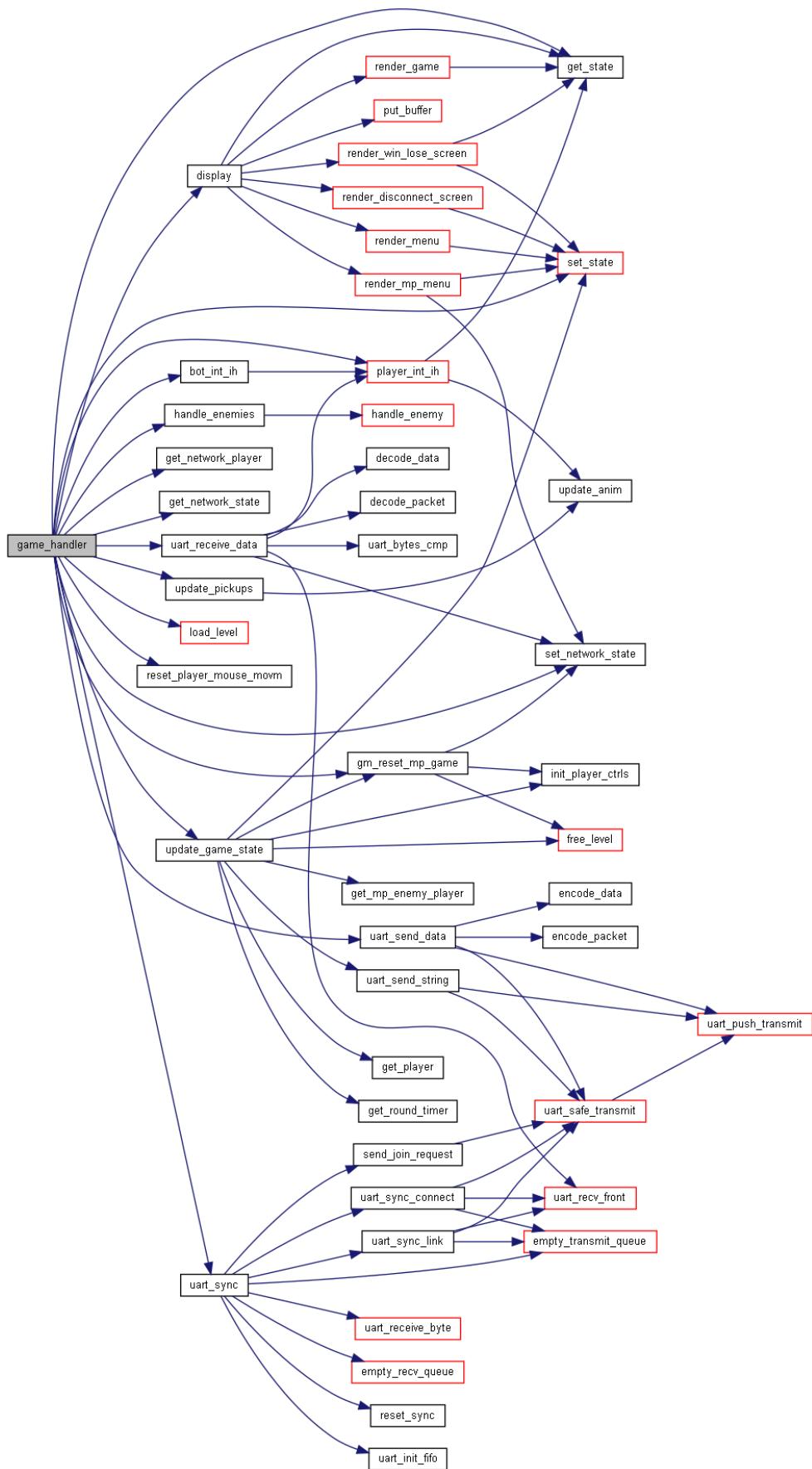


Figura 19: Gráfico de chamadas da função `game_handler`

5 Detalhes de Implementação

5.1 Implementações Interessantes

5.1.1 Deteção de Colisões

- **Colisões Rato-Botão**

Para permitir a interação com menus, verificando colisões entre o rato e botões, utiliza-se a função:

```
bool ( ms_detect_collision )( vector_t ms_pos , pair_t button_pos ,
coordinate_t button_size );
```

Função 8: De system/controls/controls.c

Esta função vai verificar se o rato, valores da posição enviado pelo argumento `ms_pos`, se encontram dentro dos limites do botão, caracterizado pelos argumentos `button_pos`, que indica o canto superior esquerdo e a dimensão do botão, argumento `button_size`. Por motivos de eficiência, esta função é chamada apenas quando o jogador se encontra num menu e pressiona o botão esquerdo do rato.

- **Colisões Jogador-Paredes**

Para verificar se o jogador colide com uma parede, é associado um `collider` ao jogador e usa-se uma verificação de colisão entre `colliders` circulares e rectangulares.

Para deslizar o jogador ao longo da parede foi calculado o **vetor de distância** entre o **ponto de colisão na parede** e a **posição do centro do collider** do jogador, como foi explicado em [4.2.3](#), denominado `distance`. Seguidamente, calcula-se quanto maior é o raio do `collider` do jogador em relação a `distance`, denominado `factor`.

$$factor = \frac{raio_do_collider}{\|\overrightarrow{distance}\|}$$

Finalmente, aumenta-se a **distância entre o ponto de colisão da parede e o jogador** até ser igual ao tamanho do raio do `collider`, usando o `factor`.

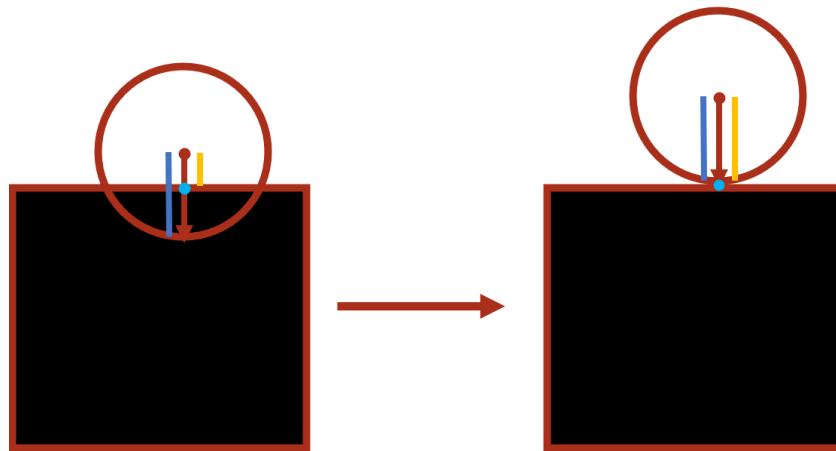


Figura 20: Visualização gráfica da colisão entre parede e jogador. O laranja representa o tamanho do vetor distance e o azul representa o tamanho do raio.

Este método está implementado na função seguinte:

```
vector_t (player_wall_collision)(player_info_t * ply,
vector_t * new_pos);
```

Função 9: De system/player/player.c

- Colisões **Pickup**-Mapa (**Tabela de colisões**)

No âmbito de minimizar o tempo de execução na deteção de colisões entre o jogador e os *pickups*, foi desenvolvida uma espécie de **Tabela de Colisões**.

1	1			
1			2	2
			2, 3	2, 3
	4		3	3

Tabela 2: Exemplo de uma **Tabela de colisões**

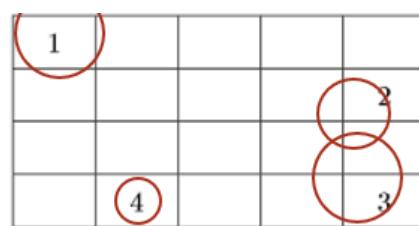


Figura 21: Mapa que gera a **Tabela de Colisões** na tabela 2

A ideia é minimizar o número de verificações de colisão entre jogadores e *pickups*. Com esta implementação, apenas é necessário verificar colisões entre *pickups* e jogadores que estejam no mesmo bloco da **Tabela de Colisões**, sendo que um bloco tem 1 unidade de largura e comprimento.

A tabela de colisões é atualizada sempre que é adicionado um *pickup* no mapa, que tem associado um *collider* circular. Neste processo, verificam-se os blocos diretamente

adjacentes ao bloco onde o *pickup* foi posicionado (inclusive). Se existir colisão entre o *collider* do *pickup* com qualquer um desses blocos, então é adicionado o *ID* do *pickup* à **Tabela** nessa posição.

Uma desvantagem deste método é que os *colliders* dos *pickups* só podem ter 1 unidade de raio no máximo, pelo facto de se estar apenas a verificar os blocos adjacentes.

Este método está implementado nas seguintes funções:

```
collision_table_t collision_table ;
size_t (add_pickup)(pickup_t * pickup) ;
int (add_collision)(pickup_t * pickup , size_t pickup_index) ;
```

Função 10: De system/pickups/pickups.c

Para remover colisões da **Tabela de Colisões** um processo similar é usado.

```
void (remove_collision)(vector_t * position , size_t
pickup_index) ;
```

Função 11: De system/pickups/pickups.c

- **Colisões Pickup-Jogador**

Para verificar se o jogador apanha um *pickup* usa-se uma verificação simples de colisões entre círculos. Trata-se de comparar a **soma dos raios** dos *colliders* circulares com a **distância entre os centros deles**.

Para evitar usar raízes quadradas, que são operações muito custosas, comparam-se os quadrados desses valores.

$$\|\overrightarrow{\text{PosiçãoJogador}} - \overrightarrow{\text{PosiçãoPickup}}\|^2 \leq \text{raio}^2$$

Este método está implementado na seguinte função:

```
size_t (check_player_pickup_collision)(player_info_t * player
) ;
```

Função 12: De system/pickups/pickups.c

5.1.2 Tabelas Virtuais e Polimorfismo

Para facilitar o armazenamento e uso de **pickups** foi criado um sistema de **polimorfismo** simples. Da **struct** mãe **pickup** derivam as **structs** **ammo**, **medkit** e **armor**, possibilitando guardar todos os **pickups** numa **array** com o mesmo tipo de **pickup_t** e usar o índice na **array** como *ID* do **pickup**.

Na **struct pickup** usa-se uma **Tabela Virtual** que por sua vez seleciona a função a ser chamada aquando de obter um **pickup** e cada **struct** derivada define a sua **Tabela Virtual** e a função.

Esta funcionalidade está implementada na sua íntegra no ficheiro **system/pickups/pickup.h**, com a exceção da definição das **Tabelas Virtuais** no ficheiro **system/pickups/pickup.c**.

5.1.3 Raycasting

Raycasting é a técnica de *rendering* que é utilizada para desenhar as paredes e gerar um **efeito 3D**. Consiste em **projetar** geometricamente as paredes num plano de câmera imaginário frontal ao jogador e **perpendicular** ao vetor de direção do jogador.

O passo inicial da técnica consiste no lançamento de **raios** por cada coluna do ecrã verificando quando e onde atinge uma parede. Usando o algoritmo **DDA** (ver Apêndice 8.1.3) para percorrer os blocos do mapa, que é uma matriz, é possível saber a posição exata onde o raio atinge a parede.

O algoritmo **DDA** está implementado na função:

```
void(shoot_ray)(vector_t* pos, vector_t* ray_dir, ray_hit_t*  
ray_hit, double range);
```

Função 13: De **system/utils/utils.c**

Com essa informação é calculada a **distância perpendicular** entre o plano da câmera até à parede que por si é também usada para calcular a altura da parede. (<https://lodev.org/cgtutor/raycasting.html> 2020).

Este método está implementado na função:

```
int (display_map) (double range);
```

Função 14: De system/graphics/render.c

Inicialmente, as colunas foram desenhadas no *buffer* da placa gráfica **verticalmente**. No entanto, aproximar o jogador à parede gerava um atraso no jogo. Este problema é efeito das operações de **cache miss**, que são muito custosas. Como o *buffer* é guardado na memória como uma linha contínua de *pixels*, e como para aceder aos *pixels* de uma coluna é necessário ”*saltar*” blocos de memória, os *pixels* que se pretendem alterar não estão guardados na cache.

Uma primeira solução foi desenhar as colunas num *buffer* auxiliar rodado a 90° em relação ao ecrã, de forma que as paredes fossem desenhadas **horizontalmente**. No término do desenho das paredes, o *buffer* seria rodado para a orientação do ecrã, usando um algoritmo *cache-efficient* (ver Apêndice 8.1.3). Esta solução chegou então a ser implementada inicialmente mas foi necessário encontrar outra pois não foi tão eficiente como o que era desejado.

A solução atual passa por calcular as alturas e posições de todas as colunas e guardar essas informações na array `g_data.lines` de tipo `line_t` no ficheiro `system/lib/video_graphics.h`. Após calcular todas as paredes, é percorrido o ecrã **linha a linha**, pixel a pixel, para evitar ao máximo **cache miss**'s e, se no *pixel* percorrido existir uma parede a ser desenhada, são escritas as cores correspondentes de acordo com a textura da parede.

Esta solução está implementada na função:

```
int (draw_walls) (int32_t highest_wall, int32_t lowest_wall);
```

Função 15: De system/graphics/render.c

Também foi tentada a implementação de *floor casting*, ou seja o desenho do chão e do teto como descrito em <https://lodev.org/cgtutor/raycasting2.html>, mas não permaneceu na versão final por atrasar demasiado o jogo.

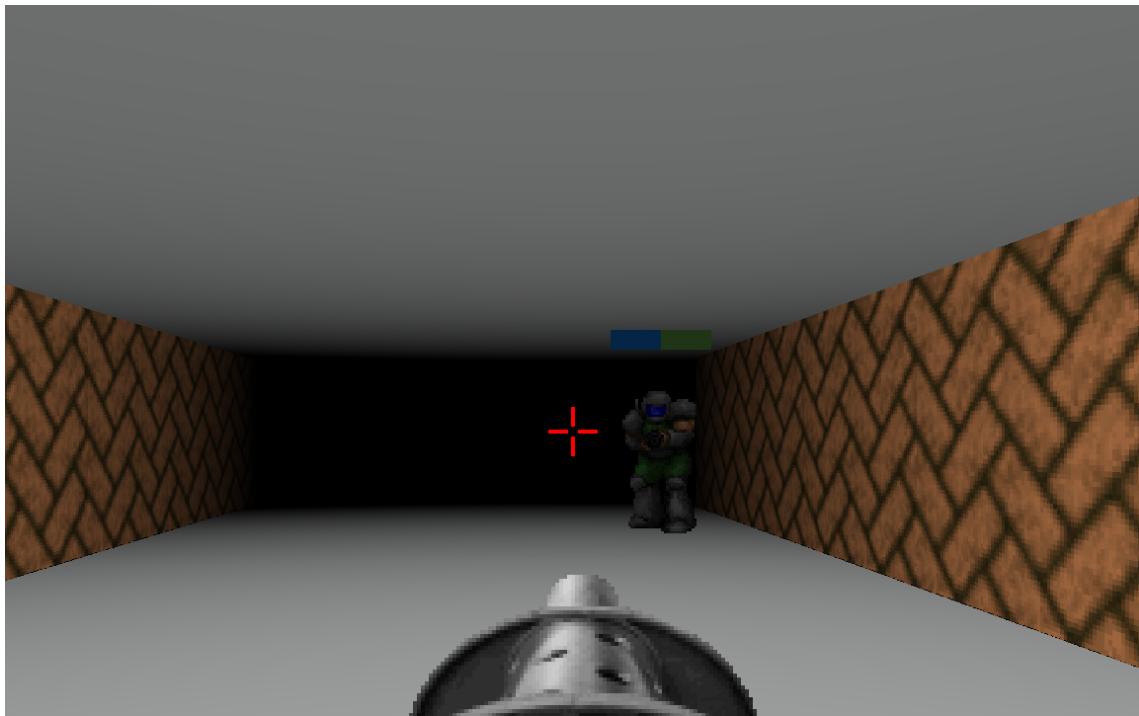


Figura 22: Desenho de chão e teto.

5.1.4 Rendering de Inimigos/Pickups e Scaling de Sprites

O jogador inimigo contém 8 conjuntos de animações direcionais. Ao andar à volta do jogador, irão sendo selecionados diferentes *sets* de animações, para permitir a ilusão de um modelo 3D. Para escolher o índice da animação (de 0 a 7), tem de se calcular o ângulo entre o **vetor direção** do jogador inimigo e do **vetor** que une o jogador ao inimigo.

Note-se que o ângulo em graus é calculado da seguinte forma:

Seja o vetor \vec{dir} a direção do inimigo e $\vec{players_dir}$ a direção que liga o jogador ao inimigo.

Seja \det o determinante dos dois vetores:

$$\begin{vmatrix} dir_x & players_dir_x \\ dir_y & players_dir_y \end{vmatrix}$$

Seja int_prod o produto interno dos vetores, $\overrightarrow{\text{dir}} \cdot \overrightarrow{\text{players_dir}}$

Dados estes valores, o ângulo calculado em graus será

$$\frac{180}{\pi} * \text{atan2}(\det, \text{int_prod})$$

Adaptado de <https://en.wikipedia.org/wiki/Atan2>

Após descobrir o ângulo, divide-se o resultado por 360 e obter um índice de 0 a 7.

Para dar aos sprites desenhados uma ilusão de profundidade e distância, teve de se aplicar um algoritmo de **scaling**. Inicialmente, é calculada a largura que o sprite deve ter para criar essa ilusão, bem como a posição inicial no ecrã, conforme a sua posição no mapa em relação ao jogador. Tendo em conta este passo, existem duas situações possíveis. (<https://lodev.org/cgtutor/raycasting3.html>)

- O sprite original precisa de ser aumentado: Neste caso calcula-se a quantidade de pixeis do sprite que têm de ser replicados (**x**). Ao desenhar o sprite, repete-se o mesmo pixel **x** vezes, para que haja um aumento de tamanho.
- O sprite original precisa de ser diminuido: Neste caso calcula-se a quantidade de pixeis que se tem de ignorar (**step**). Ao desenhar o sprite, deve-se pintar de **step** em **step**, tornando o sprite mais pequeno.

Ao desenhar cada linha vertical do sprite, é necessário saber se está atrás de uma parede. Para isso, guarda-se a informação da distância à parede mais próxima do jogador num *buffer*. Assim, desenha-se apenas essa linha do sprite se a sua distância for menor que a da parede correspondente.

Métodos implementados em:

```
int(draw_rect_sprite)(pair_t * end_pos, pair_t * offset, pair_t *
bounding_box, sprite_t * sprite, double sprite_dist, double
factor);
```

Função 16: De system/lib/video_graphics.c

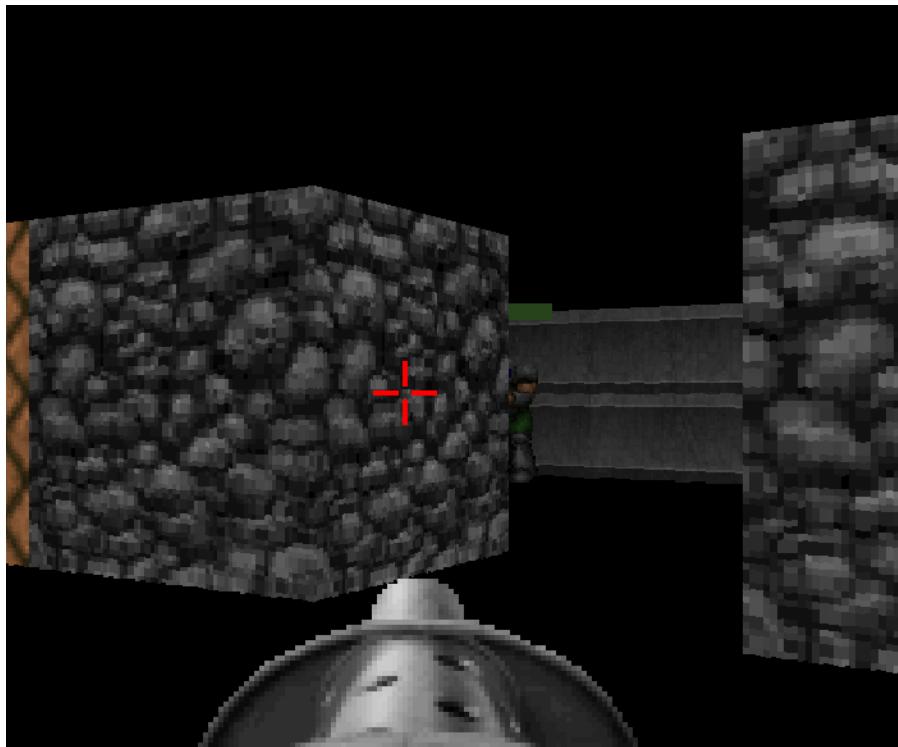


Figura 23: Apenas metade do jogador está visivel devido à parede

5.1.5 Escurecimento de Paredes e Sprites

O escurecimento de *pixels* é feito dividindo cada componente da cor por um valor. Isto é feito na função `int(darken_pix)(uint16_t x, uint16_t y, double factor);` no ficheiro `system/lib/video_graphics.c`.

O escurecimento das paredes e *Sprites* é feito à base da distância do jogador. Para além disso, se o raio do *Raycast* (5.1.3) ultrapassar o ponto de máxima escuridão então as paredes e/ou *Sprites* não serão desenhados.

Por motivos de eficiência, o escurecimento de sprites e de paredes teve de ser feito a partir de um bloco de distância do jogador, de forma gradual. Se o jogador se colocasse demasiado perto de uma parede ou de um jogador adversário, a operação de escurecimento iria demorar muito tempo e iria notar-se uma quebra dramática de *performance*.

5.1.6 Ordenação de Sprites

Este é um passo extremamente importante para que os sprites mais distantes, eventualmente, não sobreponham os sprites mais perto do jogador. Para isso tem de se ordená-los através da sua distância ao jogador.

Criou-se uma estrutura de dados `render_t` (`system/lib/video_graphics.h`) para caracterizar um sprite que fosse necessário renderizar com a ilusão de distância. Todos os objetos a ser renderizados, como jogadores e `pickup`'s, estão guardados em *arrays*, de fácil acesso. Esses mesmos que estiverem ativos irão ser processados num `render_t` e adicionados a uma *array*, fazendo uma inserção ordenada conforme a sua distância ao jogador, de forma decrescente. Por fim, todos os objectos `render_t` irão ser renderizados, percorrendo a *array* resultante das operações, dos mais distantes para os mais próximos.

Métodos implementados em:

```
void renders_insert_sorted(render_t * renders, render_t * value,
                           size_t * size);
size_t setup_renders(render_t * renders, double range);
```

Função 17: De `system/graphics/render.c`

5.2 Implementações de Temas Abrangidos em Aula

5.2.1 Page Flipping

Para implementar **Page Flipping** foi reservado espaço para 3 *buffers* na memória da placa gráfica e a cada um dos 3 foi associado um endereço virtual e uma *scanline*.

- O *buffer* a ser desenhado para o ecrã na placa gráfica é o `drawing_address`.
- O *buffer* a ser desenhado pelo processador com as funções `set_px()` do ficheiro `system/lib/video_graphics.h` é o `page_address`.
- O *buffer* restante, que é o `drawing_address` do frame anterior, é o `third_address`.

Numa operação de **Page Flip**, faz-se a chamada de Page Flipping da VBE à BIOS, enviando como parâmetro a *scanline* do `page_address`, com a opção de esperar pelo próximo *vertical retrace* ativada. Nesta operação trocam-se os *pointers* e os *scanlines* dos *buffers* da seguinte forma:

1. O novo *pointer/scanline* do `page_address` é o *pointer/scanline* antigo do `third_address`.
2. O novo *pointer/scanline* do `third_address` é o *pointer/scanline* antigo do `drawing_address`.
3. O novo *pointer/scanline* do `drawing_address` é o *pointer/scanline* antigo do `page_address`.

Este processo está implementado na função:

```
int (page_flip)();
```

Função 18: De `system/lib/video_graphics.c`

5.2.2 Protocolo de Comunicação

A conexão entre dois computadores é um processo de múltiplos passos, de forma a garantir que não seja possível ”*forçar*” conexão enviando bytes aleatórios. Por cada símbolo enviado é recebido um **símbolo de reconhecimento ’O’ (Acknowledgment Byte)**¹². Se o símbolo

¹²Um símbolo de reconhecimento enviado não implica receber um símbolo de reconhecimento de volta.

de reconhecimento não for recebido a tentativa de conexão **falhará**. Se a qualquer momento for recebido um byte com **erro** o processo será recomeçado.

O processo é o seguinte:

1. É enviado um símbolo de pedido de conexão '**J**'.
2. Por cada **símbolo de reconhecimento** recebido é enviado o próximo símbolo da palavra passe "**START**".
3. Após receber o último **símbolo de reconhecimento** é enviado o símbolo de pedido de resposta '**P**'.
4. Por cada **símbolo de resposta** da palavra "**BEGIN**" recebido é enviado um símbolo de reconhecimento.
5. Após enviar o último **símbolo de reconhecimento** o jogo começa e ambos os computadores ficam conectados.

Este processo está implementado nas seguintes funções:

```
int (uart_sync_link)();  
int (uart_sync_connect)();  
void (uart_sync)();
```

Função 19: De system/networking/networking.c

Ambos os computadores têm a sua versão do jogo a correr e apenas são trocadas informações de *inputs*.

Para facilitar a receção e envio de dados importantes ao jogo, foi criada a estrutura de dados **network_packet_t** (system/networking/networking.h). Esta estrutura contém informação do estado do teclado do adversário, estado do teclado e arma a usar no instante atual.

- Para enviar informação tem de se codificar o pacote de dados e depois codificá-lo em bytes. No primeiro byte é enviado uma flag de identificação de primeiro byte (bits 3, 5 e 7 ativos). Ao ler o pacote de dados rejeita-se o byte inicial se não contiver essa informação. Os bits 0, 1 e 2 são ativados conforme a arma a ser usada de momento. O

segundo byte contém o deslocamento do rato total na frame. O terceiro e último byte contém informação sobre o estado do teclado e também um bit que indica o sinal do deslocamento do rato.

- Para se receber a informação vai-se verificar se o primeiro byte contém o código de identificação, caso contrário irá ser ignorado. Em seguido vai-se montando uma array de dados para se descodificarem e para se poder descodificar o respetivo `network_packet_t` resultante, modificando a informação necessária no jogador adversário.

Além disso, para compactar a informação do deslocamento do rato numa frame, foi feita uma conversão de `double` para `uint8_t`, multiplicando-se por 128, impondo um teto a esse valor (para caber no byte). Além disso mandava-se um bit que indicava o sinal desse valor. No lado da receção, fazia-se a divisão para obter o valor original. Para evitar erros de cálculos, o valor aplicado ao jogador, no lado de envio, era truncado com a mesma operação, para que os dados em ambos os computadores fossem iguais.

1	Bits Arma								Código de confirmação							
	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
2	Deslocamento total do Rato															
3	LB	CTL	SFT	D	S	A	W	MSB	0	1	2	3	4	5	6	7

Tabela 3: Estrutura do packet enviado.

Codificação e descodificação dos dados implementada em:

```
int (encode_data)(uint8_t * data, network_packet_t * pkt);
```

Função 20: De system/networking/networking.c

```
int (decode_data)(uint8_t * data, network_packet_t * pkt);
```

Função 21: De system/networking/networking.c

```
int (encode_packet)(network_packet_t * pkt, player_info_t * ply);
```

Função 22: De system/networking/networking.c

```
int (decode_packet)(network_packet_t * pkt, player_info_t * ply);
```

Funcão 23: De system/networking/networking.c

Por fim, é preciso garantir que alguma falta de sincronização dos dois jogos não implique que um dos jogos termine e outro não. Para isso, quando num dos computadores o jogo é terminado, é enviada uma mensagem de 3 bytes indicando o final do jogo, e se ganhou ou perdeu. Esta mensagem é **WIN** se o jogador adversário tiver vencido e **LOS** caso o jogador adversário tenha perdido.

5.2.3 Filas UART e FIFO

Para que não se perca informação, foram ativadas as fifos da UART e usadas filas. Foi criada uma estrutura **queue** (system/queue/queue.c). Sempre que se recebem dados da uart, estas são adicionadas a uma fila de receção e, da mesma maneira, ao enviar são colocadas numa fila de envio, caso o **Transmitter Holding Register** não esteja disponível. Sempre que a *UART* emite uma interrupção de **Transmitter Holding Register Empty**, enviam-se todos os dados que estão na fila de envio. Mesmo que não seja possível enviar dados naquele instante, ao adicionar-se à fila, irão ser enviados mais tarde.

Além disso, aquando uma interrupção de frame, lê-se tudo o que está dentro da fila de receção. Desta forma, se houver algum atraso da *UART*, ao ler/enviar a informação toda, ambos os computadores irão voltar a estar sincronizados. (ver **uart_ih()** no ficheiro **system/lib/uart.c**)

5.2.4 Armas e máquinas de estado

Existem 2 tipos diferentes de armas:

- Automática: Pode deixar o lado esquerdo do rato em baixo durante todo o tempo.
- Semi-Automática: Após disparar uma vez tem de largar o lado esquerdo do rato e só depois conseguirá disparar novamente a arma.

Para isto ser possível, criou-se uma máquina de estados para ações do rato. O **enum** inerente a esta implementação é **key_state_t** (system/controls/controls.c). Na estrutura de

`controls_t` tem uma referencia a `lb_single_tap`. Esta variável irá indicar se a arma semi-automática pode ser disparada ou não. O estado do rato é atualizado da seguinte maneira

- Se o estado for `KEY_UP` e se estiver com o rato pressionado, irá ativar `lb_single_tap` e mudar o estado para `KEY_DOWN`.
- Se o estado for `KEY_DOWN`, `lb_single_tap` irá ser desativado, visto que já não foi apenas um clique, e se de facto o utilizador tiver deixado de pressionar o rato, o estado irá ser atualizado para `KEY_UP`.

Note-se que a única maneira de voltar a ativar `lb_single_tap` após ter sido desativado, é através do estado `KEY_UP`, que é obtido ao largar o botão do rato. Atualização de estados implementada em:

```
void ( update_player_controls )( player_info_t * ply );
```

Função 24: De system/player/player.c

Todas as ações relativas às armas foram implementadas com auxílio a uma máquina de estados. O `enum gun_state_t` contém diferentes estados, que restringe que ações podem ser realizadas:

- INVENTORY: A arma não está em uso
- INITIAL: A arma acabou de ser retirada do inventário
- CHARGING: A arma tem um tempo de espera até que possa ser usada depois de ser escolhida no inventário.
- READY: A arma está pronta a ser usada. Apenas neste estado o jogador consegue disparar a arma.
- COOLDOWN: A arma está em tempo de espera após ter sido disparada uma vez.
- NO_AMMO: A arma ficou sem balas.
- RELOADING: A arma está atualmente a ser recarregada, à espera que esta operação esteja completa.

Cada arma tem uma variável `last_op_tick` que identifica o tick da ultima mudança de estado. Dependendo do tempo de espera associado a cada estado, vai-se manter ou trocar de estado, substituindo novamente o valor de `last_op_tick`.

A atualização de estado da arma está implementado em:

```
void update_gun_state(gun_instance_t * gun, bool shot, bool player);
```

Função 25: De system/guns/gun.c

6 Conclusão

Em primeiro lugar, quase todos os objetivos do nosso trabalho foram completos. Apenas faltaram algumas pequenas mecânicas de jogo que gostaríamos de ter implementado no toca às armas. Além disso, faltou alguma organização no que toca a variáveis globais. Implementamos em alguns ficheiros variáveis `static` com funções de `getters`, no entanto ficou aquém daquilo que achávamos correto.

A unidade curricular de **LCOM** foi algo que desde o início captou o nosso interesse. Compreendemos a importância de aprender a interagir com periféricos, para percebermos as bases do computador. A parte mais complicada de programar em baixo nível foi fazer *debugging*. Sentiu-se falta de algumas ferramentas ao qual estamos habituados, como *breakpoints* e modo *debug*. No entanto, ganhou-se alguma experiência nesse ramo.

Algo que sentimos que faltou foi a integração e continuação da unidade curricular de **AOCO** e **MPCP**, pelo que teria sido bastante interessante aplicar e estender conhecimentos de ainda mais baixo nível e utilizar os mesmos em algo mais prático.

7 Bibliografia

- Tutoriais e Editor Online de L^AT_EX.** Obtido de Overleaf: <https://www.overleaf.com/>
- Vandevenne, L. (2020). **Raycasting.** Obtido de Lode's Computer Graphics Tutorial: <https://lodev.org/cgtutor/raycasting.html>
- Vandevenne, L. (2019). **Raycasting II: Floor and Ceiling.** Obtido de Lode's Computer Graphics Tutorial: <https://lodev.org/cgtutor/raycasting2.html>
- Vandevenne, L. (2020). **Raycasting III: Sprites.** Obtido de Lode's Computer Graphics Tutorial: <https://lodev.org/cgtutor/raycasting3.html>
- Vandevenne, L. (2020). **Raycasting IV: Directional Sprites, Doors, And More.** Obtido de Lode's Computer Graphics Tutorial: <https://lodev.org/cgtutor/raycasting4.html>
- Foo', L. (19 de junho de 2019). **Circular Collision Detection.** Obtido de Lazy Foo' Productions: https://www.lazyfoo.net/tutorials/SDL/29_circular_collision_detection/index.php
- LuKremBo (2019). **lofi type beat “onion”.** Obtido de <https://www.youtube.com/watch?v=KGQNrzqrGqw>

8 Apêndice

8.1 Anexos

8.1.1 Anexo A - Instruções de Instalação

Para poder instalar o jogo *Operation M.I.N.I.X.*, deve dirigir-se à pasta `g01/proj` e fazer os seguintes comandos. Devido à necessidade de usar caminhos absolutos para guardar os ficheiros de mapa, estes passos são extremamente importantes.

- `sh install.sh` (correr o ficheiro executável).
- Tal como indica a bash, deve dirigir-se para a pasta `minixop` criada na pasta `home/lcom/labs`.
- Garantir que a pasta `g01` e o seu conteúdo se encontra em `home/lcom/labs`

8.1.2 Anexo B - Imagens e fontes

- Sprites dos jogadores, armas, texturas do mapa e *backgrounds* dos menus:
 - [https://github.com/freedom/freedoom/tree/master/sprites](https://github.com/freedoom/freedoom/tree/master/sprites)
 - [https://github.com/freedom/freedoom/tree/master/flats](https://github.com/freedoom/freedoom/tree/master/flats)
 - [https://github.com/freedom/freedoom/tree/master/graphics](https://github.com/freedoom/freedoom/tree/master/graphics)
 - <https://opengameart.org/content/ocean-background>
- Botões de menu
 - <https://opengameart.org/content/horrorific-button>
- Fonte usada
 - <https://www.fontspace.com/messe-duesseldorf-font-f27861>
 - <https://cooltext.com/>

8.1.3 Anexo C - Links

- Tutorial raycasting usado: <https://lodev.org/cgtutor/raycasting.html>

- Raycasting extras interessantes:

<https://permadi.com/1996/05/ray-casting-tutorial-table-of-contents/>

- Algoritmo DDA: <https://www.javatpoint.com/computer-graphics-dda-algorithm>

- Transposição de matrizes *Cache-Efficient*: https://www.youtube.com/watch?v=huz6hJP1_cu&t=551s

Glossário

Collider é um modelo que representa a forma de um objeto usado em deteção de colisões. [25](#), [30–32](#)

FOV ou *Field of View* é o campo de visão do jogador expresso em graus. [24](#)

FPS é a unidade de **frame rate**. [12](#)

Frame rate é o numero de *frames* ou imagens por segundo. [12](#), [16](#), [49](#)

HUD ou *Heads-Up Display* é uma interface gráfica que transmite visualmente informações ao jogador sobre o jogo a decorrer. [1](#), [9](#), [14](#)

Pickup é um objeto ou item que pode ser apanhado pelo jogador. [1](#), [2](#), [8](#), [21](#), [24](#), [25](#), [31–33](#), [38](#)

Ray casting é uma técnica de *rendering* baseada em raios com diversos usos. [2](#), [19](#), [22](#), [23](#), [33](#), [47](#), [48](#)