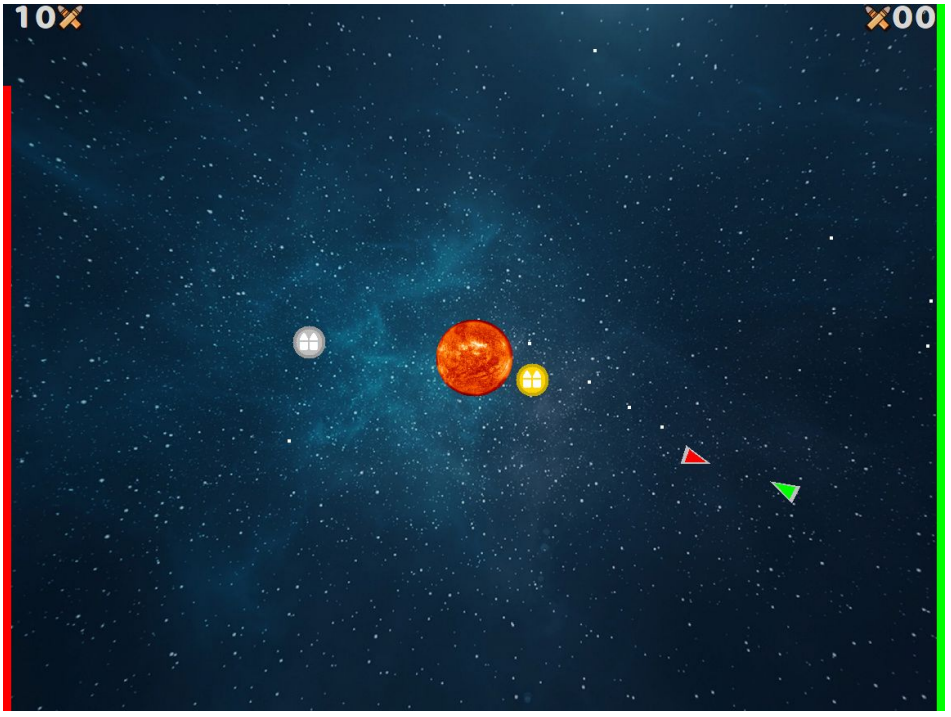
U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Collision Course



Laboratório de Computadores
2019/2020

Turma 6 - Grupo 06

Relatório final

Daniel Monteiro up201806185@fe.up.pt
Ana Teresa Cruz up201806460@fe.up.pt

Índice

Desambiguação

No contexto deste relatório, os termos "jogador", "player" e "Player 1/2" têm significados diferentes:

- O termo "jogador" refere-se à pessoa que está a controlar um dos "players";
- O termo "player" refere-se a um dos triângulos no ecrã, controlado diretamente pelo "jogador";
- Os termos "Player 1" e "Player 2" referem-se especificamente a um dos "players". O "Player 1" tem a sua barra de vida na esquerda do ecrã, e vice-versa em relação ao "Player 2"

Instruções de utilização

Main Menu

O jogo começa no Main Menu, onde são apresentadas as várias opções disponíveis aos jogadores. A primeira opção apresentada é Play que, quando selecionada, os jogadores começam a jogar. A opção seguinte é Help, na qual são explicados os controlos dos jogadores. A última opção é a Options, na qual os jogadores podem personalizar o jogo, podendo selecionar algumas settings.

No canto inferior esquerdo os jogadores podem ver a hora atual.

A partir do Main Menu,, os jogadores podem sair do jogo selecionando a opção no canto inferior direito.

A imagem acima é um screenshot de como o jogo começa, o Main Menu. Em cima tem o nome do jogo: Collision Course; de seguida apresentam-se as opções faladas em cima. Pode-se verificar que a screenshot foi tirada às 15 horas, 44 minutos e 9 segundos.

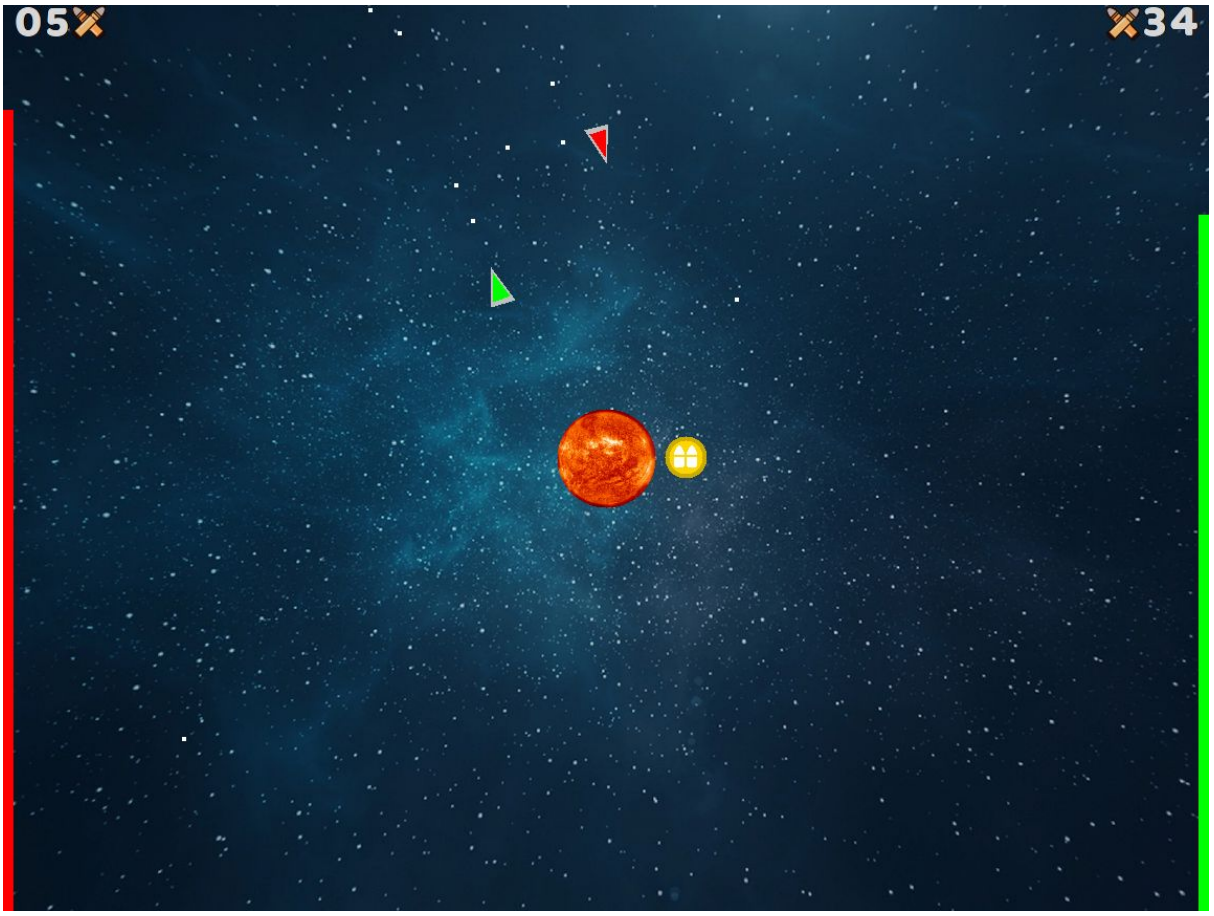
Play

O objectivo do jogo, como em qualquer outro jogo *shooter* tradicional, é matar o outro player, reduzindo a barra de vida do adversário para 0 (sendo que no início do jogo, as barras de vida estão cheias e são da altura do ecrã). O Player 1, o triângulo vermelho, é controlado com o teclado, enquanto que o Player 2 é controlado com o rato. Detalhes acerca da forma como estes dispositivos controlam os players podem ser encontrados na subsecção Help Menu. Em qualquer momento, é possível sair do jogo ao premir a tecla "Esc". Quando o jogo acaba, os jogadores são reencaminhados para para o Main Menu.

Os players começam o jogo em órbita do sol, em posições diametralmente opostas. No decorrer do jogo, o jogador pode afetar o player em três formas diferentes: virar, acelerar e disparar. O player pode virar e acelerar em qualquer momento. No entanto, este apenas poderá disparar se tiver munições, que podem ser recarregadas se apanhar as munições em órbita (representada na imagem abaixo como uma "bola" dourada.). Quanto mais perto do sol as munições estiverem, mais valiosas são e mais perigoso é apanhá-las.

Um player perde vida se colidir com uma bala ou com o outro player, daí o nome do jogo, ou se bater num dos lados do ecrã. Um player perde o jogo imediatamente se colidir com o sol.

A característica definidora do jogo será, provavelmente, a simulação física. O sol, no meio do ecrã, exerce uma força gravitacional em todos os players, balas, e munições, que é fielmente simulada em cada frame do jogo. Esta componente gravitacional adiciona toda uma nova dinâmica ao jogo, que o torna mais desafiante e até talvez educativo para o jogador que, ao jogar, desenvolve alguma intuição para a forma como a mecânica orbital funciona.



A imagem acima é um screenshot de um jogo em progresso. No lado esquerdo do ecrã está a informação pertinente ao Player 1 (o triângulo vermelho): no canto superior esquerdo pode-se ver que o número de balas que este player tem é 5 e a barra vermelha quase cheia indica que este player tem a vida quase toda. Uma análise rápida indica que o Player 1 tem mais vida do que o Player 2, enquanto que este último (o triângulo verde) tem mais balas do que o primeiro. O Player 1 pode resolver o seu défice de balas se apanhar as munições douradas em órbita perto do sol.

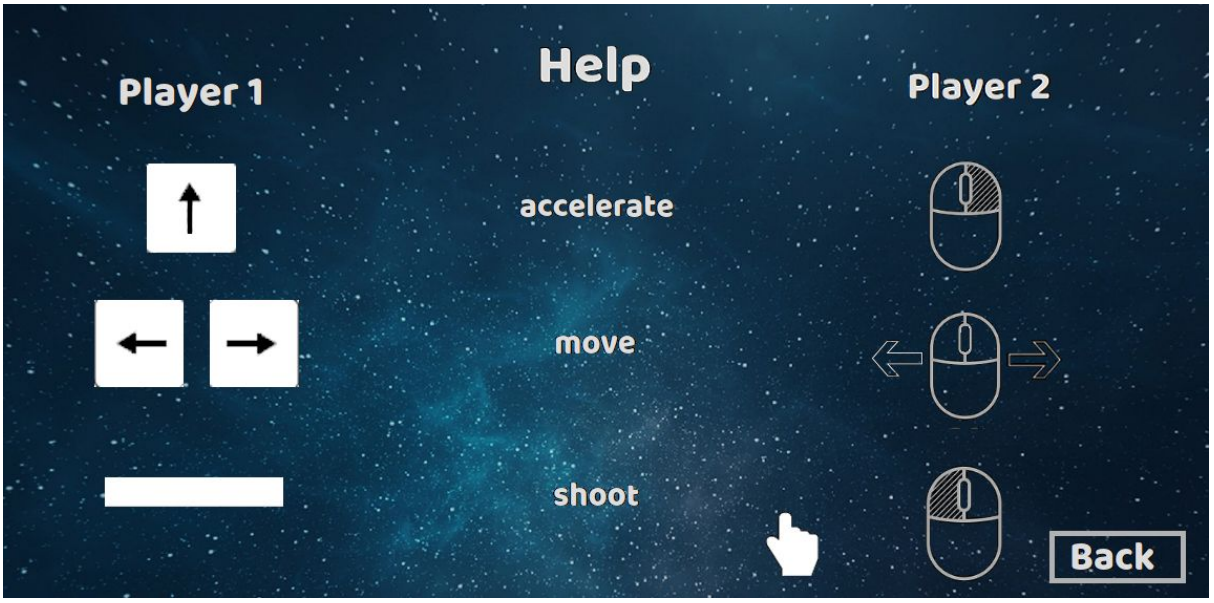
Help Menu

A segunda opção do Main Menu, o Help Menu, esclarece aos jogadores como executar os comandos para jogar, sendo estes acelerar, disparar ou virar.

No lado esquerdo são especificados os comandos do Player 1, o jogador com o teclado. Para acelerar o jogador deve premir a seta para cima; disparar premir a barra de espaço; e para virar o jogador pode premir tanto a seta da direita como a da esquerda.

No lado direito são especificados os comandos do Player 2, o jogador com o rato. Para acelerar o jogador deve premir o botão direito do rato; disparar premir o botão esquerdo do rato; e virar o jogador deve deslocar o rato para a direita ou esquerda.

Através do botão Back, o jogador consegue voltar ao Main Menu.



A imagem acima é um screenshot do Help Menu que contém imagens explicativas de como executar os comandos do jogo.

Options Menu

A terceira opção do Main Menu, o Options Menu permite aos jogadores personalizarem o jogo através da seleção de algumas settings.

Os jogadores podem personalizar 4 settings, sendo estas:

1. Bouncy walls: se as balas devem ou não retornar depois de saírem do ecrã;
2. Gravity: se o jogo é afetado pela gravidade ou não;

3. Infinite ammo: se os jogadores têm número de balas infinito ou finito;
4. Wall damage: se um jogador deve perder vida ao ir contra os limites do ecrã.

Se o jogador quiser que o jogo tenha uma das settings deve selecionar o botão Yes. A opção selecionada passa a ter uma moldura verde, em vez da cinzenta.

Através do botão Back, o jogador consegue voltar ao Main Menu.

A imagem acima é um screenshot do Options Menu onde se pode ver as settings discutidas em cima. A partir desta imagem pode-se concluir que no jogo, as balas podem sair do ecrã; o jogo seria afetado pela gravidade; os jogadores teriam um número finito de projéteis (no máximo 99); por fim, os jogadores perderiam vida ao ir contra os limites do ecrã.

Estado do Projeto

Foi implementado tudo o necessário para a completa funcionalidade do jogo pensado, no entanto não foi implementada a capacidade dos jogadores poderem escolher quais as teclas do teclado para executarem os comandos do jogo.

Dispositivos Utilizados

Dispositivo	Utilização	Interrupções	Ficheiros
Timer	Controlo do frame rate, tanto no jogo como nos menus	✓	timer.h timer.c
Teclado	Controlo do Player 1	✓	keyboard.h* keyboard.c*
Rato	Controlo do Player 2 e navegação de menus	✓	mouse.h* mouse.c*
Gráficos	Desenho de imagens (XPMs), círculos e triângulos	✗	video_card.h video_card.c
Real Time Clock	Leitura da hora do dia, para além do uso do alarme do RTC	✓	rtc.h rtc.c

* Nota: O keyboard.c e o mouse.c têm como "backend" o kbc.h e o kbc.c

Timer

O timer, mais especificamente as interrupções do mesmo, são usadas para controlar o frame rate do jogo. O frame rate, por sua vez, influencia outros componentes do projeto, nomeadamente, a simulação física.

Por volta da linha 500 do ficheiro core_game_loop.h, é possível ver o código que lida com interrupções do timer.

```
// ...
if (msg.m_notify.interrupts & TIMER_MASK)
{     /*subscribed interrupt*/
  timer_int_handler();
  if (timer_get_counter() % 2 == 0) {
    if (state != playing) {
      game_ended_countdown--;
    }

    game_loop();
  }
}
//...
```

Gostaríamos de evidenciar que a função game_loop(), que processa um único frame do jogo, é apenas chamada uma vez por cada 2 frames, o que significa que o frame rate do jogo é 30 FPS.

Teclado

A função principal do teclado é controlar o Player 1 durante o jogo, embora também tenha um papel marginal na lógica de alto nível do jogo (premir a tecla "Esc" termina imediatamente a partida, retornando ao Main Menu). As setas laterais controlam a orientação do Player 1, enquanto que a seta da frente faz com que o Player 1 acelere e a barra do espaço faz com que o mesmo dispare balas.

Por volta da linha 480 do ficheiro core_game_loop.h, é possível ver o código que lida com interrupções do teclado.

```
// ...
if (msg.m_notify.interrupts & KEYBOARD_MASK)
{     /*subscribed interrupt*/
  kbc_ih();
  if (keyboard_ih2(&k_packet)) {
    handle_player1_input(k_packet);

    if ((k_packet.n_bytes == 1) && (k_packet.bytes[0] == BREAK_ESC_KEY)) {
      esc_key_pressed = true;
    }
  }
}
// ...
```

Rato

O rato é usado tanto para controlar o Player 2 no contexto do jogo como para controlar o mouse pointer no contexto dos menus. No contexto do jogo, a orientação do Player 2 é controlada com

movimentos laterais do rato, a aceleração é controlada com o botão direito do mesmo e disparar é controlado com o botão esquerdo. No contexto dos menus, o jogador deve mover o rato, movendo assim o mouse pointer nos menus. Para selecionar uma opção o jogador deve fazer com que o mouse pointer esteja dentro do retângulo que moldura a opção desejada e premir o botão esquerdo do rato.

Por volta da linha 490 do ficheiro core_game_loop.h, é possível ver o código que lida com interrupções do rato.

```
// ...
if(msg.m_notify.interrupts & MOUSE_MASK)
{     /*subscribed interrupt*/
  mouse_ih();
  if(mouse_parse_packet(&m_packet)){
    handle_player2_input(m_packet);
  }

}
// ...
```

Gráficos

A funcionalidade gráfica do Minix é utilizada constantemente durante todo o projeto. Mais especificamente, o modo utilizado é o 0x14c, com uma resolução de 1152x864 (e capaz de mostrar $2^{24} = 16777216$ cores, ignorando o canal alfa).

Este projeto faz uso de double buffering, com algumas modificações (mais detalhes na secção "Detalhes de implementação"), tanto como nos menus como no jogo em si.

As palavras que aparecem no jogo são "pré-feitas". No entanto, os números que aparecem no ecrã são gerados automaticamente pelo jogo (ver numbers.h e numbers.c).

Se há algo que se possa inferir pelo nome deste projeto é que o jogo tem colisões. Mais especificamente, o jogo lida com colisões entre triângulos, triângulos e círculos, triângulos e pontos e, por último, colisões de círculos, pontos e triângulos com os lados do ecrã. A função de alto nível que lida com as colisões chama-se, apropriadamente, collision() e está definida em core_game_loop.h. Dado o número de funções de deteção de colisão de baixo nível, decidimos compilar essas funções na tabela abaixo.

triangle.h	circle.h	ammo.h	player.h

check_pixel	local_pixel	ammo_hits_top	player_hits_top
		ammo_hits_bottom	player_hits_bottom
		ammo_hits_left	player_hits_left
		ammo_hits_rigth	player_hits_rigtht
			players_colide
			player_touches_circle

Por último, apresenta-se uma imagem da função draw() do ficheiro core_game_loop.c, que lida com o processo de desenhar um frame no ecrã.

```
void draw() {
  // "Resets" the double buffer
  memcpy(double_buffer,
         background_buffer,
         get_hres() * get_vres() * get_bytes_per_pixel());

  draw_life_bars(double_buffer);
  if (!get_infinite_ammo()) {
    draw_ammo_numbers(double_buffer);
  }
  draw_projectiles(double_buffer);
  draw_players(double_buffer);
  draw_ammo(double_buffer);
  draw_message(double_buffer);

  // Copies the data from the double buffer to the frame buffer
  memcpy(get_buffer_base_ptr(),
         double_buffer,
         get_hres() * get_vres() * get_bytes_per_pixel());
}
```

Real Time Clock (RTC)

Funcionalidades diferentes do RTC são utilizadas no Main Menu e no jogo em si. No Main Menu, o RTC é utilizado para indicar as horas. No jogo em si, a capacidade que o RTC tem de gerar alarmes é utilizada para controlar o "spawning" de munições. Em ambos os casos, as interrupções do RTC são utilizadas.

Por volta da linha 390 do ficheiro core_game_loop.h, é possível ver o código que lida com a inicialização do RTC para o jogo.

```
rtc_read_reg(RTC_REGISTER_C); // Clearing some possible flag

// Saving these for posterior restoration
uint8_t rtc_reg_b = rtc_read_reg(RTC_REGISTER_B);
uint8_t rtc_seconds_alarm = rtc_read_reg(RTC_SECONDS_ALARM);
uint8_t rtc_minutes_alarm = rtc_read_reg(RTC_MINUTES_ALARM);
uint8_t rtc_hours_alarm = rtc_read_reg(RTC_HOURS_ALARM);

//I want these interrupts
rtc_write_reg(RTC_REGISTER_B, rtc_reg_b | RTC_REG_B_UIE | RTC_REG_B_AIE);

// Sets the alarm to 6 seconds later
rtc_write_reg(RTC_SECONDS_ALARM,
              decimal_to_bcd((bcd_to_decimal(rtc_read_reg(RTC_SECONDS)) + 6) % 60));

rtc_write_reg(RTC_MINUTES_ALARM, 0xff); // Don't care
rtc_write_reg(RTC_HOURS_ALARM  , 0xff); // Don't care
```

Por volta da linha 510 do ficheiro core_game_loop.h é possível ver o código que lida com interrupções do RTC para o jogo.

```
// ...
if (msg.m_notify.interrupts & RTC_MASK)
{     /*subscribed interrupt*/
  if (rtc_ih()) { // It's an alarm interrupt if it returns true

    // Delay the alarm by 12 seconds
    rtc_write_reg(RTC_SECONDS_ALARM, decimal_to_bcd((rtc_get_second() + 12) % 60));

    if (!get_infinite_ammo()) {
      spawn_ammo();
    }
  }
}
// ...
```

Por volta da linha 218 do ficheiro Menu.c pode-se ver o código para gerar o tempo no Main Menu.

```
  draw_numbers(double_buffer,30,515,rtc_get_hour());
  draw_transparent_image(double_buffer,80,514,doispontos_img,true);
  draw_numbers(double_buffer,100,515,rtc_get_minute());
  draw_transparent_image(double_buffer,150,514,doispontos_img,true);
  draw_numbers(double_buffer,170,515,rtc_get_second());
```

Estrutura e Organização do Código

A ordem pela qual os módulos vão ser descritos será de baixo nível para alto nível: começaremos pelos módulos que fornecem interfaces para os periféricos utilizado pelo projeto, passaremos pelos módulos que implementam componentes do jogo, e acabaremos com os módulos que implementam a lógica dos menus do jogo. As funções de cada módulo não serão enumeradas. Para esse tipo de análise, esperemos que a documentação fornecida com o projeto seja suficientemente informativa.

timer.h

Este módulo contém funções relacionadas com o manuseamento das interrupções do timer. Este módulo foi importado de um dos labs anteriores.

Peso relativo	Proporção Ana/Daniel (%)
0%	N.A.

kbc.h

Este módulo contém funções relacionadas com o manuseamento do KBC. Este módulo foi importado de um dos labs anteriores e serve de "backend" para o os módulos keyboard.h e mouse.h.

Peso relativo	Proporção Ana/Daniel (%)
0%	N.A.

keyboard.h

Este módulo contém funções relacionadas com o manuseamento das interrupções do teclado. Este módulo foi importado de um dos labs anteriores e ligeiramente modificado para este projeto.

Peso relativo	Proporção Ana/Daniel (%)
0%	N.A.

mouse.h

Este módulo contém funções relacionadas com o manuseamento das interrupções do rato. Este módulo foi importado de um dos labs anteriores.

Peso relativo	Proporção Ana/Daniel (%)
0%	N.A.

video_card.h

Este módulo contém funções relacionadas com o manuseamento dos gráficos, para além da inicialização do modo gráfico do Minix. Este módulo foi importado de um dos labs anteriores.

Peso relativo	Proporção Ana/Daniel (%)
0%	N.A.

rtc.h

Este módulo contém funções relacionadas com o manuseamento do RTC e das suas interrupções.

Peso relativo	Proporção Ana/Daniel (%)
5%	0/100

utils.h

Este módulo contém algumas funções úteis para outros módulos. Gostaríamos de evidenciar que o código da função bcd_to_decimal tem como origem este post do stackoverflow:
https://stackoverflow.com/questions/28133020

Peso relativo	Proporção Ana/Daniel (%)
5%	0/100

vector2.h

Este módulo contém funções que manipulam vetores bidimensionais (soma de vetores, multiplicação por um escalar, etc). Esta abstração é especialmente útil para este projeto, dado

que lidamos com acelerações, velocidades e posições bidimensionais.

Peso relativo	Proporção Ana/Daniel (%)
5%	0/100

numbers.h

Este módulo contém uma única função que desenha um número de dois dígitos no ecrã. Esta função é utilizada para mostrar o número de balas que cada player tem e as horas no Main Menu.

Peso relativo	Proporção Ana/Daniel (%)
5%	0/100

triangle.h

Este módulo contém funções relacionadas com o desenho e a colisão de triângulos.

Peso relativo	Proporção Ana/Daniel (%)
5%	90/10

circle.h

Este módulo contém funções relacionadas com o desenho e a colisão de círculos.

Peso relativo	Proporção Ana/Daniel (%)
5%	100/0

projectiles.h

Este módulo lida com a inicialização e gestão das balas presentes (dentro e fora) do ecrã.

Peso relativo	Proporção Ana/Daniel (%)
5%	0/100

player.h

Este módulo contém várias funções importantes para a gestão das structs "player" declaradas neste ficheiro. As funcionalidades presentes incluem o processamento de input, deteção de vários tipos de colisões, cálculo dos vértices do triângulo, entre outras.

Peso relativo	Proporção Ana/Daniel (%)
5%	20/80

ammo.h

Este módulo lida com a inicialização e gestão das "bolas" de munições presentes no ecrã.

Peso relativo	Proporção Ana/Daniel (%)
5%	0/100

core_game_settings.h

Este módulo contém variáveis que influenciam aspetos estéticos e mecânicos do jogo. Este módulo acaba por servir como uma interface entre o Options Menu e o jogo em si.

Peso relativo	Proporção Ana/Daniel (%)
5%	0/100

core_game_loop.h

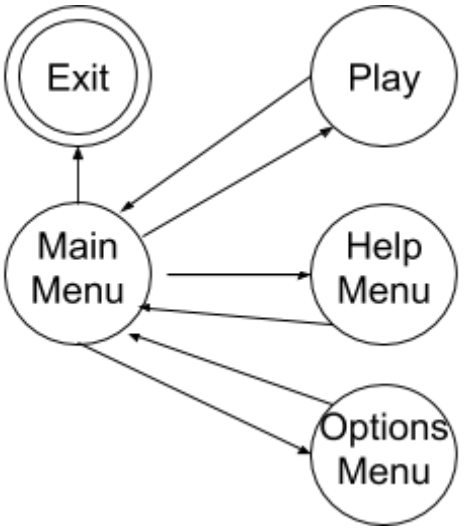
Este módulo contém a lógica de alto nível do jogo. A função core_game_loop é a função chamada pelos menus para começar o jogo.

Peso relativo	Proporção Ana/Daniel (%)
25%	15/85

game.h

Este módulo contém as funções necessárias para tratar dos inputs dos jogadores e para começar o jogo propriamente dito. Foi feita uma máquina de estados, sendo estes main_menu, help_menu, options_menu, exit_program e core_game (representados por um enum

em game.c). O estado muda sempre que o jogador seleciona uma opção, passando para o estado respetivo. O estado inicial é o Main Menu, podendo passar para o Help Menu, Options Menu ou Play; sendo que qualquer um deste pode voltar para o estado inicial. O estado final é o exit_program atingido quando o jogador seleciona a opção Exit. Em baixo apresenta-se um autómato para a máquina de estados.



Peso relativo	Proporção Ana/Daniel (%)
15%	100/0

Menu.h

Este módulo contém as funções necessárias para desenhar os menus, carregar os XPMs e desenhar os botões. Tem também uma função de colisão entre pixel e o botão, usada para detetar botões premidos.

Peso relativo	Proporção Ana/Daniel (%)
10%	100/0

proj.h

Este módulo contém a função main, que inicializa o modo gráfico do Minix, cede o controlo à função letsplay e restaura o ecrã ao seu estado inicial.

Peso relativo	Proporção Ana/Daniel (%)
0%	0/100

Grafo de Chamadas da Função letsplay

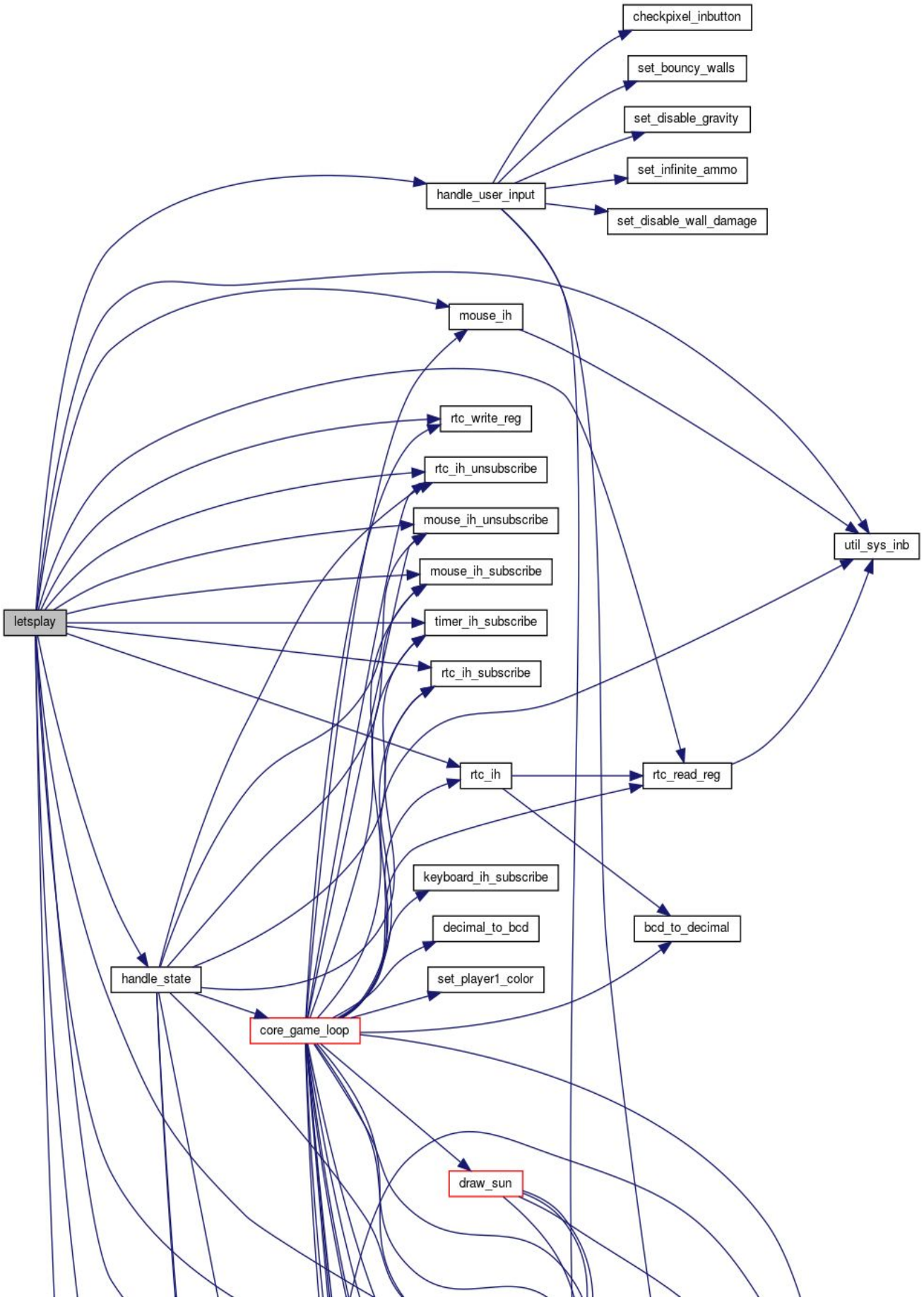
Infelizmente, não fomos capazes de gerar o grafo de chamadas da função letsplay. No entanto, dado que a nossa função main tem este aspeto (imagem abaixo), consideramos que a apresentação do grafo de chamadas da função letsplay é uma alternativa aceitável.

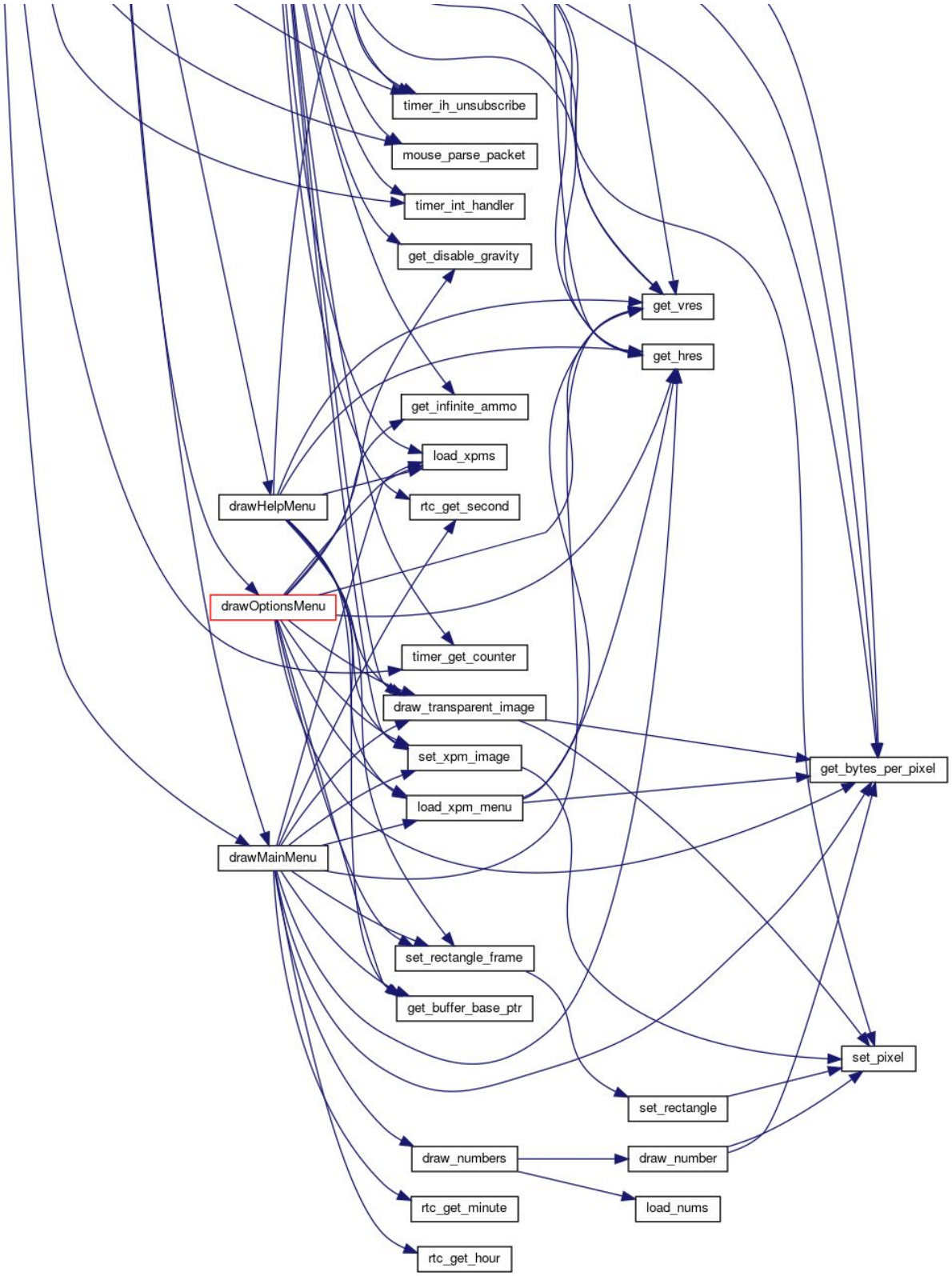
```
int(proj_main_loop)(int argc, char *argv[]) {
  if( vg_init(0x14c) == NULL){
    vg_exit();
    return 1;
  }

  letsplay();

  vg_exit();
  return 0;
}
```

Gostaríamos de evidenciar também que só há duas ocorrências da função driver_receive: uma ocorre na função core_game_loop, a função de alto nível do jogo (core_game_loop.c) enquanto que a outra ocorre na função letsplay, a função de alto nível dos menus (game.c). A afirmação anterior pode ser verificada através da execução do comando "grep -r driver_receive" na raíz do projeto.


checkpixel_inbutton
set_bouncy_walls
set_disable_gravity
set_infinite_ammo
handle_user_input
set_disable_wall_damage
mouse_ih
rtc_write_reg
rtc_ih_unsubscribe
mouse_ih_unsubscribe
util_sys_inb
mouse_ih_subscribe
letsplay
timer_ih_subscribe
rtc_ih_subscribe
rtc_ih
rtc_read_reg
keyboard_ih_subscribe
decimal_to_bcd
bcd_to_decimal
handle_state
set_player1_color
core_game_loop
draw_sun


timer_ih_unsubscribe
mouse_parse_packet
timer_int_handler
get_disable_gravity
get_vres
get_hres
get_infinite_ammo
load_xpms
drawHelpMenu
rtc_get_second
drawOptionsMenu
timer_get_counter
draw_transparent_image
set_xpm_image
get_bytes_per_pixel
load_xpm_menu
drawMainMenu
set_rectangle_frame
get_buffer_base_ptr
set_pixel
set_rectangle
draw_numbers
draw_number
rtc_get_minute
load_nums
rtc_get_hour

Detalhes de implementação

Double Buffering com Caching do Background

O processo normal de double buffering costuma começar com um memset ao double buffer, que depois é preenchido com vários elementos do jogo e, finalmente, é copiado para o frame buffer. O nosso jogo, no entanto, possui uma característica que permite optimizar este processo: o background é estático, ou seja, há muitos pixeis que não mudam de cor durante todo o jogo. Este background estático tem de ser desenhado todos os frames, o que é altamente ineficiente, visto que, no final, o resultado será sempre o mesmo.

A nossa modificação envolverá um terceiro buffer, o background_buffer. Este buffer é modificado apenas no momento de "startup", como se pode ver na imagem abaixo (linha ~420 no core_game_loop.c):

```
// Background
set_xpm_image(background_buffer, 0, 0, nebula_background_img);

// Sun
if (!get_disable_gravity()) {
  draw_sun(background_buffer, get_hres() / 2 - sun_img.width / 2, get_vres() / 2 - sun_img.height / 2, sun_img);
}

// Ammo symbols
if (!get_infinite_ammo()) {
  draw_transparent_image(background_buffer, 10 + NUMBERS_LENGTH, 1, ammo_img, false);
  draw_transparent_image(background_buffer, get_hres() - 10 - NUMBERS_LENGTH - ammo_img.width, 2, ammo_img, false);
}
```

Depois, a nossa função draw (no mesmo ficheiro) terá este aspeto:

```
void draw() {
  // "Resets" the double buffer
  memcpy(double_buffer,
         background_buffer,
         get_hres() * get_vres() * get_bytes_per_pixel());

  draw_life_bars(double_buffer);
  if (!get_infinite_ammo()) {
    draw_ammo_numbers(double_buffer);
  }
  draw_projectiles(double_buffer);
  draw_players(double_buffer);
  draw_ammo(double_buffer);
  draw_message(double_buffer);

  // Copies the data from the double buffer to the frame buffer
  memcpy(get_buffer_base_ptr(),
         double_buffer,
         get_hres() * get_vres() * get_bytes_per_pixel());
}
```

Como se pode ver, em vez de se fazer memset, o double_buffer é inicializado com o background_buffer, que é aqui utilizado para dar um "headstart" ao processo de desenho do double_buffer.

Uso de um pseudo-buffer circular para guardar um número variável de entidades

Apresentamos aqui uma estrutura de dados capaz de guardar um número variável de entidades. A estrutura-base sobre a qual a nossa é implementada é um array de structs. O único requerimento que essa struct tem é que é tenha um membro bool chamado "active", por exemplo.

Exemplo:

```
/**
 * @brief Struct that represents one of our game's bullets
 */
typedef struct projectile
{
    gravity_object grav; /**< @brief The position and velocity of the bullet*/
    bool active; /**< @brief You can think of this as whether or not the bullet exists*/
} projectile;
```

Este bool "active" determina se esse elemento do array "existe" ou não.

Um novo elemento do array é inicializado da seguinte forma: primeiro precisaremos de um "write pointer", que funciona da mesma forma que um write pointer de um buffer circular. O primeiro passo que executamos é incrementar o "write pointer" (caso esteja no último, volta ao início). Agora, é só uma questão de preencher os

valores da struct da forma mais conveniente, à exceção do membro "active", ao qual tem de ser atribuído obrigatoriamente o valor "true".

Exemplo:

```
void new_projectile(vector2 pos, vector2 speed) {
    if (proj_array == NULL) return;

    n = (n + 1) % n_projectiles; // n is a global variable

    proj_array[n].grav.position = pos;
    proj_array[n].grav.velocity = speed;

    // The projectile now "exists"
    proj_array[n].active        = true;
}
```

O processo de eliminação de um elemento é substancialmente mais trivial: basta atribuir o valor "false" ao membro "active" desse elemento.

Exemplo:

```
for(uint8_t i = 0; i < no_projectiles; i++){
  if(projects_array[i].active){
    //projetil acertar no jogador 1
    if (p1->alive) {
      if(check_pixel(projects_array[i].grav.position.x,projects_array[i].grav.position.y,
        p1->p1x,p1->p1y,p1->p2x,p1->p2y,p1->p3x,p1->p3y)){
          projects_array[i].active = false; // The projectile no longer exists
          player_deal_damage(1, 100);
        }
    }
```

A imagem acima também ilustra como é que os elementos desta estrutura de dados devem ser processados: apenas os elementos com "active" igual a "true" devem ser processados.

Esta estrutura de dados tem um problema: se o "write pointer" der uma volta completa, é perfeitamente possível que elementos ativos sejam reescritos. Esta limitação é mitigada com um array suficientemente grande.

Conclusões

Ao longo da cadeira sentimos que haviam alguns aspetos que poderiam ser melhorados. O principal aspeto a melhorar é os

handouts dos labs que, muitas vezes, estavam um pouco incompletos e ambíguos. Também sentimos que a informação relativa ao teste foi fornecida demasiado tarde. Gostaríamos também de mencionar a ironia que é a cadeira ter um setup automatizado de avaliação, e nem as notas do primeiro lab foram publicadas no momento de escrita deste relatório.

No entanto, é de igual importância destacar os aspetos positivos, como o projeto, que foi um trabalho desafiante, mas excelente para aplicarmos o que fomos aprendendo ao longo do semestre; os professores demonstraram-se disponíveis para tirar as dúvidas dos alunos (apesar de no nosso caso não termos exposto nenhuma). Também ao longo da cadeira fomos ganhando cada vez confiança a escrever programas não triviais em C.

Para concluir, sentimos que apesar da cadeira ter alguns aspetos a melhorar, é uma cadeira desafiante, na qual se aprende muitos métodos que com certeza nos serão úteis no nosso futuro.