Instituto Superior Técnico

MEEC

Programação de Sistemas

Projeto

Multiplayer Pacman

Alexandre Rodrigues, 90002

Índice

1.1 Arquitetura

1.1.1 Nós

Este projeto implementa uma arquitetura servidor/cliente simples. O servidor aguarda a conexão dos clientes, comunicando com estes de modo a receber e aplicar o *input* do jogador, e enviar o resultante estado do jogo.

1.1.2 Módulos

O cliente e o servidor estão ambos divididos em três módulos, sendo estes o módulo principal de jogo, o módulo de mensagem e o de conexão.

O primeiro dos três implementa a lógica que rege o jogo *Pacman*, como as interações entre os personagens e *delays* nas ações.

O módulo de mensagem implementa funções que definem o protocolo de comunicação entre o cliente e o servidor. Por exemplo, enquanto o módulo de mensagem do servidor implementa a função *message_send_board*, o módulo do cliente implementa a função análoga *message_recv_board*.

Por último, o módulo de conexão lida, como seria de esperar, com a conexão entre o cliente e o servidor. No cliente, por exemplo, implementa a função que faz a ligação inicial ao servidor e a que recebe as mensagens do servidor. No lado do servidor trata ainda de guardar informação relevante aos clientes atualmente conectados.

1.1.3 Threads

Enquanto que o cliente tem apenas duas *threads*, o servidor tem uma *thread* principal e tantas outras quantos os clientes que estiverem conectados.

O cliente tem uma *thread* responsável por desenhar a *board* e o estado do jogo em geral, bem como adquirir o *input* do jogador e enviá-lo para o servidor, e uma *thread* responsável por receber mensagens do servidor (*recv_from_server*), atualizando o estado do jogo comforme o necessário.

A *thread* principal do servidor processa o jogo em si, alterando o estado deste consoante os movimentos enviados pelos clientes, e enviando o resultado para os clientes. Cada cliente que se conecta causa a criação de uma *thread* que recebe exclusivamente as mensagens do respetivo cliente (*recv_from_client*), atualizando a informação utilizada pela *thread* principal.

1.2 Organização do código

O código está dividido em três categorias principais, sendo estas código que diz respeito ao cliente, ao servidor, e a ambos. A diretoria do projeto está organizada da forma seguinte:

```
Project
├── bin
├── board-drawer
├── res
└── src
    ├── common
    │   ├── message.c
    │   ├── message.h. .4 pacman.c
    │   ├── pacman.h
    │   ├── UI_library.c
    │   ├── UI_library.h
    │   ├── utilities.c
    │   ├── utilities.h
    │   ├── vector.c
    │   └── vector.h
    ├── client
    │   ├── client_connection.c
    │   ├── client_connection.h
    │   ├── client_message.c
    │   ├── client_message.h
    │   ├── client.c
    │   └── client.h
    └── server
        ├── server_connection.c
        ├── server_connection.h
        ├── server_message.c
        ├── server_message.h
        ├── server.c
        └── server.h
```

As diretorias *client* e *server* têm o código que diz respeito ao cliente e ao servidor, respetivamente. Cada uma delas implementa os respetivos módulos.

Alguns destes módulos dependem de funções, estruturas ou definições da diretoria *common*. Por exemplo, *message.h* define *MessageTypes*, que os módulos de mensagem utilizam para identificar e processar corretamente mensagens entre eles. *pacman.c* implementa a *board* do jogo, usada igualmente pelo cliente e servidor. *vector.c* e *utilities.c* implementam funções mais genéricas, como operações vetoriais, alocação de memória com verificação, ou conversões de formato.

O código está comentado nos ficheiros *header*, cada função sendo sucintamente explicada pelo comentário que a antecede. Os ficheiros de código também têm comentários ao longo das funções e estruturas quando código não é evidente. De qualquer dos modos, o código foi escrito de modo explicito e verboso, quer seja no nome das funções ou variáveis, com o intuito de facilitar a sua leitura.

1.3 Estruturas de dados

O projeto possui algumas estruturas de dados comuns a ambos os lados servidor e cliente, bem como algumas que são análogas mas diferentes, devido aos requesitos diferentes do cliente e do servidor. Por exemplo, observando a estrutura *Player*, que guarda informação relevante a um jogador no contexto do jogo, do lado do cliente:

```
typedef struct _Player
{
```

```
    unsigned int player_id;
    Color color;
    unsigned int score;
    int powered_up;
    Vector* pacman_pos;
    Vector* monster_pos;
} Player;
```

Observamos que esta é diferente da estrutura análoga do lado do servidor:

```
typedef struct _Player
{
    unsigned int player_id;
    Color color;
    int powered_up;                     // Stores 0, 1 or 2, depending on how
        many monsters the pacman can still eat
    unsigned int score;                 // The number of things eaten
    Vector* pacman_pos;
    char pacman_move_dir;               // Movement direction
    struct timespec pacman_last_move_time; // Time of last movement
    Vector* monster_pos;
    char monster_move_dir;
    struct timespec monster_last_move_time;
} Player;
```

Como seria de esperar, o servidor tem uma implementação mais complexa da mesma estrutura, pois é este que é responsável pelo processamento dos dados. Guarda a mais a direção de movimento do jogador, bem como o tempo em que o último movimento foi executado, para ambos os personagens.

A estrutura *Game* é o método principal de passar informação às diversas funções (de modo a evitar variáveis globais), contendo toda a informação que diz respeito ao jogo. Mais uma vez existem discrepâncias na sua implementação dado que no lado do cliente:

```
typedef struct _Game
{
    int server_socket;
    unsigned int player_id;
    Board* board;
    unsigned int n_players;
    Player** players;
    unsigned int n_fruits;
    Fruit* fruits;
} Game;
```

E no lado do servidor:

```
typedef struct _Game
{
    Board* board;
    unsigned int max_players;
    unsigned int n_players;
    Player** players;
    unsigned int n_fruits;
    Fruit** fruits;
} Game;
```

O cliente necessita de saber o seu *player_id* e, por conveniência, dado que o cliente só comunica

com uma outra entidade, a socket do servidor. Já o servidor guarda também o número máximo de jogadores, que o cliente não precisa de saber.

Quanto às frutas, estas são implementadas *client-side* por:

```
typedef struct _Fruit
{
    unsigned int fruit_type; // Cherry or lemon
    int is_alive;            // Is it on the board? (Or waiting to respawn)
    Vector* pos;
} Fruit;
```

E do lado do servidor:

```
typedef struct _Fruit
{
    unsigned int fruit_type; // Cherry or lemon
    int is_alive;            // Is it on the board? (Or waiting to respawn)
    struct timespec eaten_time; // Time it was last eaten
    Vector* pos;
} Fruit;
```

Mais uma vez, como o servidor implementa o jogo em si, precisa de guardar o instante em que a fruta é comida para fazer o seu *respawn*, algo que não acontece no cliente.

A *board* do jogo é implementada de forma sucinta e privada:

```
// The game board
typedef struct _Board
{
    Vector* board_size;
    unsigned int** board;
} Board;
```

Por fim, os vetores são também definidos, simples e privadamente por:

```
typedef struct _Vec
{
    int x;
    int y;
} Vector;
```

1.4 Protocolo de comunicação

A comunicação entre o cliente e o servidor é feita através de mensagens discretas, identificadas pelo seu *MessageType*:

```
typedef uint16_t MessageType;
enum _MessageType {
    // Terminates a message, used for checking allignment
    MESSAGE_TERMINATOR = UINT16_MAX,
    // Player color
    MESSAGE_COLOR = 0,
    // The game board
    MESSAGE_BOARD,
    // ...
```

```
    // Resto das mensagens em common/message.h
    // ...
    // Alerts the server is full
    MESSAGE_SERVER_FULL
    };
```

message.c também implementa diversas funções de envio e recebimento de dados de tipos específicos e explicitos quanto ao tamanho e sinal. Tem-se por exemplo:

```
int message_send_uint16_t(int socket, uint16_t message)
{
    uint16_t m_net = htons(message);
    return send_all(socket, (void*)&m_net, sizeof(uint16_t));
}
int message_recv_uint16_t(int socket, uint16_t* message)
{
    int ret = recv_all(socket, (void*)message, sizeof(uint16_t));
    *message = (uint16_t)ntohs(*message);
    return ret;
}
```

A partir destas funções, são implementadas funções mais abstratas nos módulos de comunicação específicos ao cliente ou servidor. Tomando o exemplo simples (o qual é seguido pela maioria das funções do mesmo tipo) da função (do lado do servidor, em *server_message.c*) que envia o *player_id* ao seu cliente respetivo:

```
void message_send_player_id(int socket, unsigned int player_id)
{
    message_send_uint16_t(socket, (uint16_t)MESSAGE_PLAYER_ID);
    message_send_uint32_t(socket, (uint32_t)player_id);
    message_send_uint16_t(socket, (uint16_t)MESSAGE_TERMINATOR);
}
```

E a sua análoga encarregue do recebimento (no cliente, em *client_message.c*):

```
void message_recv_player_id(int socket, unsigned int* player_id)
{
    message_recv_uint32_t(socket, (uint32_t*)player_id);
}
```

Como se observa, a função de envio envia não só os dados. Envia primeiro o *MessageType*, de modo a que o cliente possa identificar o conteúdo, seguido dos dados da mensagem, e finalmente um terminador de mensagem. Este último permite já fazer uma validação preliminar dos dados quanto ao seu comprimento como se poderá observar de seguida.

A função de recebimento está encarregue exclusivamente de receber os dados. Isto porque a identificação e verificação do alinhamento da mensagem é feito em *recv_from_server* (em *client_connection.c*), uma versão resumida da qual está abaixo:

```
void* recv_from_server(void* _game)
{
    // Cast to game
    Game* game = (Game*)_game;
    int server_socket = game_get_server_socket(game);

    // Probe for new messages
    while (1)
```

```
    {
        // Determine message type
        MessageType mt;
        int ret = message_recv_uint16_t(server_socket, (uint16_t*)&mt);
        // ...
        switch (mt)
        {
        // ...
        case MESSAGE_PLAYER_ID:
        {
            unsigned int player_id;
            message_recv_player_id(server_socket, &player_id);
            game_set_player_id(game, player_id);
            break;
        }
        // ...
        default:
            break;
        }

        // Receive the terminator
        message_recv_uint16_t(server_socket, (uint16_t*)&mt);
        if (mt != MESSAGE_TERMINATOR)
            message_misaligned();
    }
    client_quit();
    return NULL;
}
```

A função recebe o tipo de mensagem, escolhendo através do *switch* a função adequada para interpretar os dados que se seguem. Depois da mensagem ser lida verifica o alinhamento e, caso não esteja alinhada, o cliente interrompe a execução. Na prática, nunca há um desalinhamento dos dados, sendo que isto foi implementado de modo a ajudar com o densenvolvimento das diversas mensagens.

A função *recv_from_client* do servidor é quase idêntica, bem como todo o processo detalhado anteriormente. No entanto, neste contexto não faria sentido interromper a execução caso exista desalinhamento de mensagens. Se isso fosse feito, qualquer cliente malicioso poderia enviar um pacote de dados demasiado longo ou curto e parar o servidor. Neste caso, um desalinhamento dos dados resulta então na terminação da conexão com o cliente que o causou.

Há que salientar que um desalinhamento nunca ocorre em situações normais, pois as mensagens foram desenhadas de forma a serem discretas e de tamanho explicito, sendo que não há risco do uma das partes ler menos ou mais informação do que o que a mensagem contém.

O servidor tem ainda outra forma de comunicação única, a qual utiliza para atualizar os clientes com o estado do jogo. A função *send_to_all_clients* envia uma mensagem idêntica para todos os clientes atualmente conectados a comando da *thread* principal. Um resumo dessa função encontra-se abaixo:

```
void send_to_all_clients(Game* game, MessageType message_type, void* extra_data)
{
    pthread_mutex_lock(&client_array_lock);
    // For every client
    for (unsigned int i = 0; i < n_clients; ++i)
    {
        // Send the message defined by message_type
        switch (message_type)
        {
```

```
            // ...
            case MESSAGE_PLAYER_LIST:
                message_send_player_list(client_array[i]->socket, game);
                break;
            case MESSAGE_PLAYER_DISCONNECT:
            {
                unsigned int* player_id = (unsigned int*)extra_data;
                message_send_player_disconnect(client_array[i]->socket, *player_id);
                break;
            }
            // ...
            default:
                break;
            }
        }
        pthread_mutex_unlock(&client_array_lock);
    }
```

1.5 Validação de dados

Na transação de informação, é especialmente importante protejer o servidor de informação possívelmente prejudicial. Deste modo, é necessário verificar que os dados que chegam ao servidor são válidos, e agir de acordo com o resultado. Um exemplo de verificação de dados está representado abaixo, um excerto da função *receive_from_client* (funcionalmente idêntica à analisada acima) do ficheiro *server_connection.c*:

```
    case MESSAGE_MOVE_PAC:
    {
        char move_dir;
        message_recv_movement(client->socket, (char*)&move_dir);
        if (move_dir == 'w' || move_dir == 'a' || move_dir == 's' || move_dir == 'd'
            || move_dir == (char)0)
            player_set_pac_move_dir(player, move_dir);
        break;
    }
```

Caso um cliente enviasse uma direção de movimento inválida, poderiam haver consequências graves para o funcionamento do servidor, a não ser que o movimento também fosse verificado mais à frente, antes dos personagens serem movidos, por exemplo. Ao verificar assim que a mensagem é recebida, e ignorando-a caso não seja válida, assegura-se a proteção do servidor de dados inválidos e simplifica-se o processo de *error-checking*.

Outro exemplo de leitura e interpretação de erros está relacionado com o método escolhido para terminar a *thread* que aceita conexões dos novos clientes. Quando o servidor fecha por meio da janela *SDL2*, há que alertar as outras *threads* do processo. Isto é feito por meio de um sinal enviado através de *pthread_kill*, de modo a que o *accept* (na *thread* que gere as conexões) ou o *read* (nas *threads* que comunicam com os clientes) desbloqueiem. Após isto, estas funções retornam -1, mas é importante analisar mais profundamente (através da variável *errno*) o erro, determinando se se trata da interrupção ou de outro erro indesejado. Um excerto relevante da função *connect_to_clients* de *server_connection.c* encontra-se abaixo:

```
    // ...
    // Tell the kernel to listen on this socket
    listen(listen_socket, 5);
```

```
// Accept all incoming connections
int client_socket;
while (1)
{
    // Accept connections - blocks until a client connects
    client_socket = accept(listen_socket, NULL, NULL);
    if (client_socket == -1)
    {
        // If accept was interrupted by a signal (means server is shutting down)
        if (errno == EINTR)
        {
            break;
        }
        else
        {
            perror("ERROR - Accept failed");
            exit(EXIT_FAILURE);
        }
    }
    // ...
```

1.6 Funcionalidades Implementadas

1.6.1 Controlo dos personagens pelo cliente

O cliente controla o Pacman segurando o botão esquerdo do rato e movendo o ponteiro de acordo com a direção que se quer deslocar (se quiser que o Pacman se desloque para cima, coloca o ponteiro acima do Pacman). O monstro é controlado com as teclas W, A, S e D.

De modo a determinar a direção em que o Pacman se deve mover, o cliente tem de comparar as posições do Pacman e do ponteiro. De forma geral, o Pacman tenta deslocar-se no sentido que minimiza a sua distância ao ponteiro, ou seja, ao longo do eixo que apresenta a maior distância entre os dois. Por outras palavras, se o ponteiro estiver "mais para a direita" (face às outras direções) do Pacman, este move-se para a direita. Este processo é realizado pela função *handle_user_input* de *client.c*, o excerto relevante da qual está abaixo:

```
static char last_move_pac = -1;
static char curr_move_pac = -1;
int mouse_x = 0, mouse_y = 0, board_x = 0, board_y = 0;
// If the user is pressing LMB
if (SDL_GetMouseState(&mouse_x, &mouse_y) & SDL_BUTTON(SDL_BUTTON_LEFT))
{
    get_board_place(mouse_x, mouse_y, &board_x, &board_y);
    Player* player = player_find_by_id(game, game->player_id);
    int pac_x = player_get_pac_pos_x(player), pac_y =
        player_get_pac_pos_y(player);
    // Compute the distance between the pacman and the tile the mouse points
        to
    // The pacman will move in the direction which shows the biggest
        discrepancy, or not move if null distance
    if (board_x == pac_x && board_y == pac_y)                // If the mouse
        is over the pacman
    {
        curr_move_pac = (char)0;
    }
    else if (abs_int(board_x - pac_x) > abs_int(board_y - pac_y)) // If
```

```
            further in the x direction
        {
            if (board_x > pac_x)
                curr_move_pac = 'd';
            else
                curr_move_pac = 'a';
        }
        else                                                    // If further
            in the y direction
        {
            if (board_y > pac_y)
                curr_move_pac = 's';
            else
                curr_move_pac = 'w';
        }
    }
    else   // If the user isn't pressing mouse 1
    {
        curr_move_pac = (char)0;
    }
```

O movimento do monstro é implementado de forma ligeiramente mais complexa. Regra geral, jogador espera poder segurar a tecla A para se mover para a esquerda, carregar na tecla W para se mover para cima e, ao largar o W, continuar a mover-se para a esquerda sem ter de pressionar a tecla A novamente. Para que isto aconteça, é implementado um *stack* de 4 teclas, de modo a que, qualquer que seja a combinação de teclas premida, ao largar a última tecla, a penúltima (desde que continue premida) dita a direção de movimento. O excerto relevante (presente na mesma função que o excerto acima) está abaixo:

```
    // Stores the WASD keys in order of them being pressed
    // They are removed once released
    static char wasd_stack[4] = {0,0,0,0};
    // Stores which of they keys is currently pressed as a 1
    // W is idx 0, A is idx 1,...
    // WASD
    static unsigned int wasd_pressed[4] = {0,0,0,0};
    static unsigned int n_pressed = 0;

    const Uint8* keys_pressed = SDL_GetKeyboardState(NULL);

    // If the state of the a key differs from the last frame or in other words
    //  if the user pressed or released a key this frame
    if (keys_pressed[SDL_SCANCODE_W] != wasd_pressed[0]) // Update the stack and
         pressed arrays
        update_key(wasd_stack, wasd_pressed, &n_pressed, 'w',
            keys_pressed[SDL_SCANCODE_W]);

    else if (keys_pressed[SDL_SCANCODE_A] != wasd_pressed[1])
        update_key(wasd_stack, wasd_pressed, &n_pressed, 'a',
            keys_pressed[SDL_SCANCODE_A]);

    else if (keys_pressed[SDL_SCANCODE_S] != wasd_pressed[2])
        update_key(wasd_stack, wasd_pressed, &n_pressed, 's',
            keys_pressed[SDL_SCANCODE_S]);

    else if (keys_pressed[SDL_SCANCODE_D] != wasd_pressed[3])
        update_key(wasd_stack, wasd_pressed, &n_pressed, 'd',
            keys_pressed[SDL_SCANCODE_D]);
```

```
// The monster's last and current movement directions
static char last_move_mon = (char)-1;
static char curr_move_mon = (char)-1;
// Update the current movement direction
if (n_pressed)                                  // If any keys are pressed,
    movement is the last pressed key
    curr_move_mon = wasd_stack[n_pressed - 1];
else                                            // Otherwise the monster stops
    curr_move_mon = (char)0;
// If there is a new movement direction (different from the last one sent)
```

O stack em si é gerido pela função *update_key*. A função é simples, adicionando uma tecla ao *stack* quando esta é premida, e removendo quando é solta. No entanto é algo extensa, pelo que será omitida do relatório. Econtra-se no mesmo ficheiro que a anterior.

1.6.2 Validação do número de jogadores

Devido ao fato do tabuleiro de jogo ser limitado, o servidor tem de estabelecer um limite de clientes. O número máximo de jogadores depende do número de células vazias do tabuleiro. O número de células C ocupadas por N jogadores vem:

$$C = 2N + 2max(0, N - 1) \tag{1.1}$$

Invertendo a equação, obtermos o número máximo de jogadores para um dado número de células como:

$$N_{max} = floor(\frac{C + 2}{4}) \tag{1.2}$$

O número máximo de jogadores é determinado na *main* de *server.c*:

```
// Read the board
unsigned int n_empty = read_board(game, "board-drawer/board.txt");

// Calculate maximum number of clients
// This magic formula apparently works, yay for no nested condition mess
game->max_players = floor(((float)n_empty + 2.0) / 4.0);
```

A leitura do tabuleiro retorna o número de células vazias, que é utilizado para calcular o número máximo de jogadores.

A função que aceita as novas conexões verifica se o servidor está cheio antes de guardar a informação relevante ao novo cliente (excerto de *connect_to_clients* em *server_connection.c*):

```
// Accept connections - blocks until a client connects
client_socket = accept(listen_socket, NULL, NULL);
// ...
if (game_is_full(game))
{
    puts("Server is full, denying connection request");
    message_send_server_full(client_socket);
    continue;
}
```

1.6.3 Colocação de novos jogadores no tabuleiro

De modo a colocar os personagens em sítios aleatórios, recorre-se à função *board_random_empty_space* de *pacman.c*:

```
if (!board_empty_space_exists(board)) // This prevents the mythical infinite
    loop
{
    *x = -1; *y = -1;
    return;
}
do
{
    *x = rand() % vec_get_x(board->board_size); // It aint efficient
    *y = rand() % vec_get_y(board->board_size); // but it works, it just
        works
}
while (board->board[*x][*y] != TILE_EMPTY);
```

Esta verifica primeiro se existe um espaço vazio, e depois gera coordenadas aleatórias até ser satisfeita a condição da célula se encontrar livre.

Quando um cliente se conecta geram-se duas posições aleatórias, nas quais se colocam o Pacman e o monstro. O excerto seguinte é da função *player_create* de *server.c*:

```
// Put characters in a random (empty) position
int x, y;
board_random_empty_space(game->board, &x, &y); // Pacman
new_player->pacman_pos = vec_create(x, y);
board_set_tile(game->board, x, y, board_player_id_to_tile_type(player_id,
    1));
board_random_empty_space(game->board, &x, &y); // Monster
new_player->monster_pos = vec_create(x, y);
board_set_tile(game->board, x, y, board_player_id_to_tile_type(player_id,
    0));
```

1.6.4 Desconexão dos clientes

Quando um dos lados de um *socket INET* fecha a sua socket, a função *recv* retorna 0 no lado oposto. O servidor faz uso disto para determinar se um cliente se desconecta, como se pode observar no excerto seguinte, da função *recv_from_client* em *server_connection.c*:

```
// Determine message type
MessageType mt = MESSAGE_TERMINATOR;
int ret = message_recv_uint16_t(client->socket, (uint16_t*)&mt);
if (ret == 0)
{
    fprintf(stdout, "Client %d left!\n", client->player_id);
    break;
}
```

Ao ler o tipo de mensagem enviado pelo cliente, verificando que o *recv* retorna 0, o sevidor quebra o *loop* e destrói as estruturas *player* e *client* associadas ao cliente, notificando os clientes restantes da desconexão.

1.6.5 Movimento dos personagens

Para a tornar mais breve, nesta secção será abordado apenas o movimento do Pacman. O movimento do monstro é implementado de forma análoga, com as regras diferentes estabelecidas no enunciado do projeto.

A gestão do movimento do Pacman é feito pela função *handle_pacman_move* em *server.c*. Esta começa por verificar se a célula alvo (para a qual o Pacman tem o intuito de se mover) está *out of bounds*, ou *OOB*. Se for o caso, o movimento é feito considerando que o alvo é um tijolo, pois ambos os casos são funcionalmente idênticos.

```
unsigned int tile = board_get_tile(game->board, tgt_x, tgt_y);
if (board_is_oob(game->board, tgt_x, tgt_y))   // If the target is OOB (out
    of bounds)
    tile = TILE_BRICK;                          // Handle collisions as if
        the target were a brick (bounce back)
switch (tile)
{
// ...
```

Caso o alvo seja um tijolo (ou, como foi visto antes, *OOB*), verifica-se se é possível mover para a célula oposta e, caso esteja, a função é chamada recursivamente com a célula oposta como alvo.

```
case TILE_BRICK:          // Bounce back (if able)
tgt_x = curr_x; tgt_y = curr_y;
get_opposite_target_tile(&tgt_x, &tgt_y, player->pacman_move_dir);
                                  // Invert the movement direction
if (board_is_oob(game->board, tgt_x, tgt_y) || board_get_tile(game->board,
    tgt_x, tgt_y) == TILE_BRICK); // If the new target tile is OOB or a brick
    // Stay in place, do nothing
else
    handle_pacman_move(game, player, tgt_x, tgt_y);
                                                          // Handle the new
        target tile
break;
```

Caso a célula alvo esteja livre, o Pacman é simplesmente movido para a célula em questão. Existem duas representações da posição do Pacman, dentro da estrutura *Player* e no tabuleiro, sendo que ambas têm de ser atualizadas.

Quando um fruto se encontra presente na célula alvo, faz-se recurso à função *handle_fruit_eat*, que trata de mover o Pacman, incrementar a sua pontuação, torná-lo no *powered-up* Pacman, e eliminar o fruto (temporariamente) do tabuleiro.

A troca de posição é algo trivial, resumindo-se à troca dos vetores de posição dos dois personagens, e das células respetivas no tabuleiro.

Quanto às interações com monstros inimigos, utiliza-se a função *handle_character_eat*, que processa um personagem a comer outro, tendo em conta qual dos dois se move. Neste caso o Pacman é o personagem que se move. Se estiver também *powered-up*, este move-se para a célula do monstro, tem o seu *score* incrementado, estado de *power-up* decrementado, e o monstro é movido para uma posição livre aleatória. Caso contrário, o monstro permanece na mesma posição, o seu jogador tem a pontuação incrementada, e o Pacman desloca-se para uma posição aleatória.