U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

[ ]

Invaders

An homage to the 1978 classic Space Invaders

Masters in Informatics and Computer Engineering – FEUP

Developed for the Computer Labs Class

Class 2 - Group 8

4th of January of 2021

Mário Manuel Seixas Travassos – up201905871@fe.up.pt

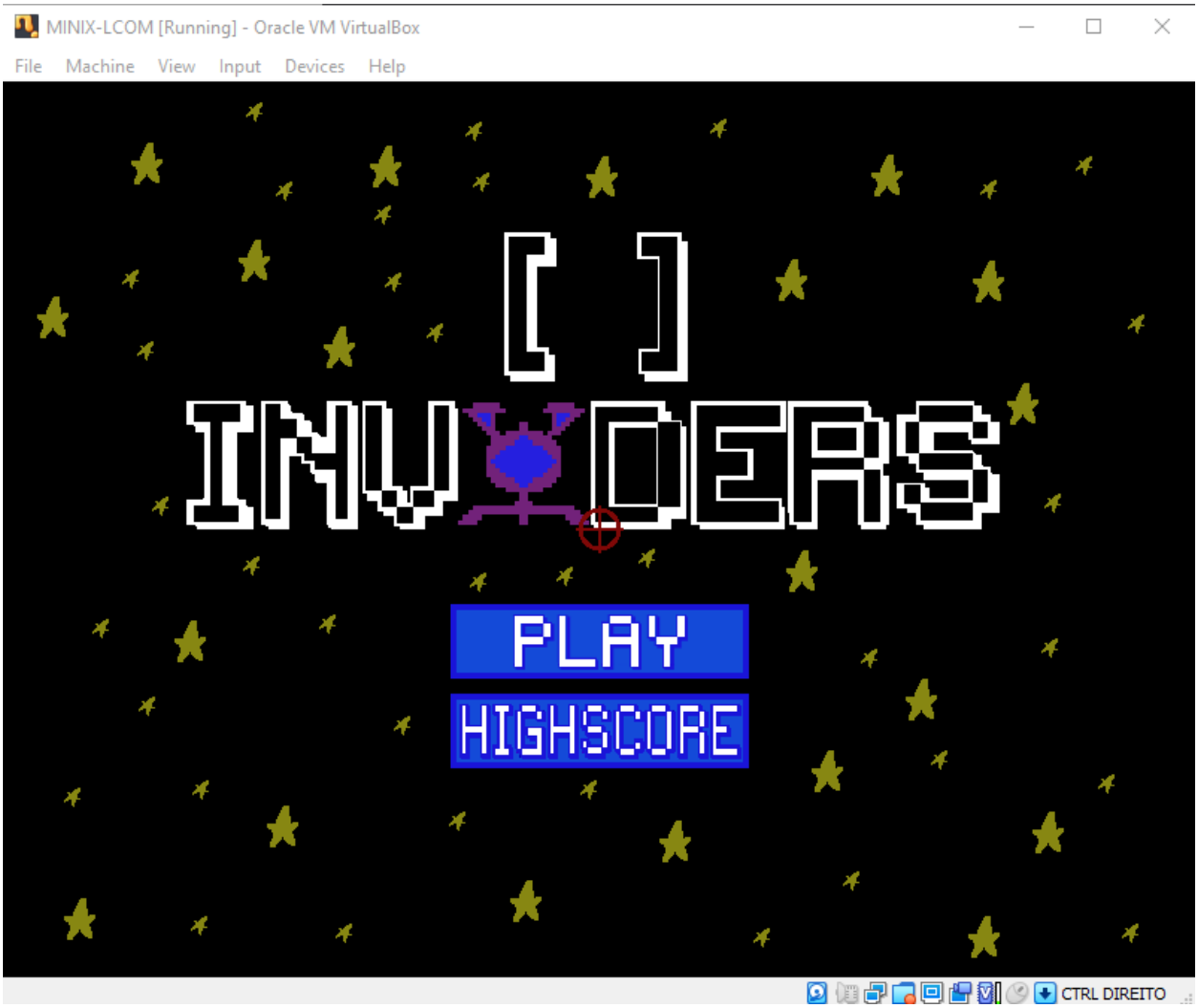
Index

Instructions and Small Description of the Project: 3
Project Status: 8
Devices used: 8
Graphics Card: 9
Timer: 11
Keyboard: 12
Mouse: 13
RTC/Real Time Clock: 14
Code Organization and Implementation Details: 15
Modules: 15
Video and Vbe_funcs implementation details: 16
Timer implementation details: 17
Keyboard implementation details: 18
Mouse implementation details: 19
RTC/Real Time Clock implementation details: 20
Sprite implementation details: 21
Logic implementation details: 23
Game implementation details: 27
Menu: 28
Highscore: 29
Proj: 31
Some additional notes: 32
Function call graph: 34
Conclusions: 35

Instructions and Small Description of the Project:

From inside of the src folder on the proj directory, one should write the command “make” to compile the game. Afterwards, running the command “lcom_run proj” should start the game.

Upon starting the game, the player will be met with the following screen:



The player may then move the mouse (represented by the red crosshair) and press the mouse’s left button on top of one of the “Play” or “Highscore” buttons, depending on if the player wants to play the game, or view the highscore leaderboards.

In case the player wants to quit the game, that must be done from this menu screen, by either pressing the mouse’s right button, or the keyboard’s “esc” key.

Should the player chose to access the highscore leaderboards, he shall be met with a similar screen to this one, depending on the highscores saved on the “highscores.txt” file, inside of the proj directory:

On this screen, the highest scores achieved by the players can be seen. Each row of the image is a different score, each of which is accompanied by the name inputted by the player that achieved them on the left, the time of day during which it was achieved (in the format hh:mm:ss) in the middle, and the score itself on the right (caps out at 99999).

The player can exit out of this screen by pressing the right mouse button, or the keyboard’s “esc” key.

If the player choses to press the “Play” button while on the main screen, he shall be met with a screen similar to this one:

This is the main game screen. On the bottom of the screen we can see a small UI, comprised of the player’s hp on the left (caps out at 99, and if it becomes 0, the player loses the game), the number of projectiles available to be shot by the player in the middle (if it’s 0, the player is unable to shoot any more projectiles, until one of his already shot projectiles is destroyed, by colliding with an entity, or the edges of the screen), and on the right, the player’s current score. These values will naturally be updated as the game goes on.

We can also see a few other entities. Two kinds of enemies exist, each with their own separate point values. There is also a special enemy at the top of the screen, which is generated randomly, and which yields a high amount of points when destroyed. The two types of normal enemies will shoot projectiles randomly, and if the player gets hit by them, he loses one of his lives.

The player can move by pressing the “A” and “D” keys on the keyboard, “A” to move left, and “D” to move right. By pressing the left mouse button, the player will shoot towards the position of his mouse, represented by the red crosshairs. These projectiles will be destroyed upon reaching the edges of the screen, or colliding with the enemy. If the latter occurs, the player will gain points equal to that enemy’s value. Destroying all normal enemies will make a new wave appear after an intentional small 1 second wait time, to allow the player to process what happened. At this point, the player will gain 1 life.

The player loses when he runs out of lives, when his spaceship collides directly with an enemy object, or when an enemy object touches the ground. Alternatively, the player may end the game prematurely should he choose to do so, by pressing the keyboard’s “esc” key, or the right mouse button at any time.

If the game has ended, either by virtue of the usual game mechanics, or by player interference, if the current player’s score is greater than the lowest score saved on the leaderboards, the player will be given the option to choose a name to save his new highscore, on the following screen:

On this screen, the player may input his name, by typing on the keyboard at least 1 and up to 5 alphabetical characters. The player may erase one character by pressing the backspace, and save the name by pressing the enter key after saving is name. Doing so will save the current score, the current time, and the inputted name on the leaderboards

Alternatively, he may choose to not save the new highscore, and just quit this screen by pressing either the “esc” key on the keyboard, or the right mouse button, in which case the player will not save his new score, and the leaderboards will remain unchanged.

We chose to save the new highscore with the name “DEMO”, and this is the resulting highscores screen:

Upon closing the game, the highscores.txt file will be updated with the highscores values, to be accessed in later sessions of play.

Project Status:

Devices used:

Graphics card	Used to draw the images, sprites and letters on the screen	N/A
Timer	Used to update the display and the game and menu logic at a constant framerate, and to perform certain actions at regular intervals.	YES
Keyboard	Used to register player inputs, and perform the appropriate actions, be they simple movement commands, or more complex actions like writing text.	YES
Mouse	Used to register player inputs, and perform the appropriate actions, whether they be controlling where the mouse is at all times or performing the correct actions based on the player's mouse clicks.	YES
RTC	Simply used to read the current time at specific intervals.	NO

Graphics Card:

Used to draw pixels on the screen, and, building upon that, the images, sprites and letters on the screen, at the desired positions, and on the desired buffers.

The mode chosen was mode 0x115, with a resolution of 800x600, and a direct color mode with 24 bits per pixel, configured to run with a linear framebuffer.

All sprites used by the game were drawn by hand by me, the only exception being the font used for both the images, and for displaying the scores, names, and other changing variables. These letters and numbers come from the "Back to 1982" font, a royalty free font found at https://www.dafont.com/back-to-1982.font. The letters, numbers and symbols used in the game are not entirely equal to those of the font, since they had to be manually altered (and sometimes redrawn!) when scaling them up or down, and when changing the colors, due to what I'd like to call "GIMP being annoying and not doing what I tell it to", but what is probably just my lack of experience with the program.

The drawing and editing of sprites was done using GIMP, a free and open-source image and graphics editor, and the png files' conversions to xpm were done on the website anyconv.com, at https://anyconv.com/pt/conversor-de-png-para-xpm/ .

A simple double buffering system was implemented, along with the addition of a separate, and aptly named background buffer ("background_buffer" on the code), where we draw static sprites which are not to be altered during the execution of a certain state of the program, but which may be altered, when changing states.

What follows is a quick explanation of our double buffering system. Upon changing state (menu to game, menu to highscores menu, game to new highscore screen, etc.), the background_buffer's contents are replaced with the background of the new screen, and with every other sprite we would like to remain unchanged (effectively becoming part of the background).

After interacting with the game and the menus, and changing the positions of the moving objects, or upon changing the names and numbers to be displayed on the screen, the contents of the background buffer are copied over into our double buffer ("double_buffer" on the code), and then we write on the double buffer the moving objects, with the desired images, and on the desired position. At the end of this, we move all data on the double buffer into the video memory ("video_mem" on the code), updating it.

This system effectively saves us the trouble of having to draw the background and every single object it contains on the double buffer, and then draw on that same double buffer the moving objects, every single frame, which would be a terrible hit to the program's performance, not to mention unnecessary, since the background and its contents are static, and therefore there should be no need to draw them twice.

The functions which interact with the graphics card can be found on a few separate files (vbe_funcs.c, video.c, sprite.c, and a few others here and there). This will be elaborated upon later.

As for the colisions, we check them using a simple AABB algorithm, which can be found better explained in the Logic module detailed description, since the implementation created does not actually interact with the graphics in any relevant way.

Major Functions:

- graphics_init() – Initializes the 0x115 graphics mode, and retrieves its information (defined in “video.c”)

- draw_pixel() – draws a pixel on a desired position, in the desired buffer (defined in “video.c”)

- copy_from_background() – copies the background buffer’s contents into the double buffer (defined in “video.c”)

- copy_to_background() – copies the double buffer’s contents into the background buffer (defined in “video.c”)

- update_vram() – updates the video memory by copying the double buffer’s contents into it (defined in “video.c”)

Minor Functions:

- load_images() – loads all images to be used by the program, and stores them in the appropriate containers (defined in “sprite.c”)

- draw_xpm() – draws an xpm image on the desired buffer (defined in “sprite.c”)

Timer:

Used to update the display and the game and menu logic at a constant framerate, and to perform certain actions at regular intervals, such as moving the enemies, or changing their animations. These regular intervals may be affected by actions happening in the game.

The file “proj_timer.c” contains the simple functions created during lab2, for subscribing and unsubscribing timer interrupts, and the interrupt handler.

Each of the game, menu and highscore modules (in game.c, menu.c and highscore.c, respectively) contain one or several functions for performing appropriate actions, depending on the value of the timer counter. Which of these functions is called depends on what the program state is.

Major Functions:

- game_loop() – updates display and logic during the game (defined in “game.c”)

- menu_loop() – updates display and logic inside the menu screen (defined in “menu.c”)

- highscore_menu_loop() – updates display and logic inside the highscore menu screen (defined in “highscore.c”)

- new_highscore_loop() – updates display and logic while on the screen where the player chooses a name for his new highscore (defined in “highscore.c”)

Minor Functions:

- proj_timer_subscribe_int() – subscribes timer interrupts (“timer.c”)

- timer_unsubscribe_int() – unsubscribes timer interrupts (“timer.c”)

- timer_int_handler() – timer interrupt handler (“timer.c”)

Keyboard:

Used to register player inputs, and perform the appropriate actions, usually simple movement commands during the game.

It is also used for registering a new highscore's name, which the player has the ability to type and alter a string's contents, upon ending the game with a new highscore.

The file "keyboard.c" contains altered versions of the functions created during lab3, for subscribing and unsubscribing keyboard interrupts, and the interrupt handler.

Once again, each of the game, menu and highscore modules contain one or several functions for performing appropriate actions, depending on the scancode read. Which of these functions is called depends on what the current program state is.

Major Functions:

- game_kbd_handler() – moves the player according to the scancode received (defined in "game.c")

- new_highscore_kbd_handler() – changes a string based on user input (defined in "highscore.c")

Minor Functions:

- menu_kbd_handler () – simply closes the menu screen, after pressing the "esc" key (defined in "menu.c")

- highscore_menu_kbd_handler() – simply closes the highscore leaderboards screen, after pressing the "esc" key (defined in "highscore.c")

- kbd_subscribe_int () – subscribes keyboard interrupts (defined in "keyboard.c")

- kbd_unsubscribe_int() – unsubscribes keyboard interrupts (defined in "keyboard.c")

- kbd_ih() – keyboard interrupt handler (defined in "keyboard.c")

Mouse:

Used to register player inputs (both the button presses and the mouse movement), and either shoot towards the mouse's position, or to navigate through the menus.

The file mouse.c contains altered versions of the functions created during lab4, for subscribing and unsubscribing timer interrupts, the interrupt handler, a function which writes commands to the kbc, to properly initialize the device, and a function that discards a read packet, to prevent the mouse and keyboard from bricking, among others.

The game, menu and highscore modules also contain one or several functions for performing appropriate actions, depending on the scancode read. Akin to the timer and keyboard, the function among these to be called depends on what the program state currently is.

Major Functions:

- parse_packet() – Function which parses a received packet (defined in "mouse.c")

- game_mouse_handler() – does the appropriate action when a player presses a mouse button (defined in "game.c")

- menu_mouse_handler() – changes the program state based on which button the player did, or did not click on (defined in "menu.c")

Minor Functions:

- highscore_menu_mouse_handler() – simply closes the highscore leaderboards screen, after pressing the right mouse button (defined in "highscore.c")

- new_highscore _mouse_handler() – simply closes the new highscore saving screen, after registering a right mouse button press (defined in "highscore.c")

- ms_subscribe_int () – subscribes mouse interrupts (defined in "mouse.c")

- ms_unsubscribe_int() – unsubscribes mouse interrupts (defined in "mouse.c")

- mouse_ih() – mouse interrupt handler (defined in "mouse.c")

- ms_issue_arg_command() – issues a command to the kbc, as an argument to the 0xD4 command (defined in "mouse.c")

RTC/Real Time Clock:

Used only to read the current time of day, at specific program states, during the program's execution. Also contains a function which parses the time read into the format that is used in the program.

Major Functions:

- rtc_read_register() – writes the register we want to read from into the rtc's address register (0x70), and updates the variable passed as a parameter with the read value (defined in "rtc.c")

- parse_time() – parses the time into the format hhmmss, which is used in the program (defined in "rtc.c")

Minor Functions:

- read_time() – reads the time and updates a global array with the read values (defined in "rtc.c")

Code Organization and Implementation Details:

Modules:

- Video and Vbe_funcs
- Timer
- Keyboard
- Mouse
- RTC/Real Time Clock
- Sprite
- Logic
- Game
- Menu
- Highscore
- Proj

Video and Vbe_funcs implementation details:

This module is responsible for initializing the video mode, drawing pixels on the video memory and on the other buffers. It is split into two files: vbe_funcs.c and video.c

VBE functions 0x01 and 0x02 can be found on the aptly named vbe_funcs.c file. These functions are the same as the ones created during lab5.

The function which initializes the 0x115 graphics mode is inside the file video.c. On this file, there is also a function which draws a pixel on the correct position inside a memory buffer identified by an enum passed as a parameter, described below.

On this same video.c file we can also find functions which handle the copying of data in between buffers (from the background buffer to the double buffer, from the double buffer to the background buffer, and from the double buffer to the video memory)

The video.c file also contains an enumerated type for specifying which buffer we're drawing in, when calling this module's drawing function, and other module's drawing functions, the enum buffer, shown below:

```
/**
 * @brief Enumerated type for specifying the buffer we're writing in
 */
enum buffer {
  b_buf = 0,  /*!< background_buffer */
  d_buf = 1,  /*!< double_buffer */
  vram = 2    /*!< video_mem */
  };
```

Relative weight of module: 8%

Developed by: Mário Travassos

Timer implementation details:

This module is responsible for updating all aspects the game at regular intervals, intervals which can be affected by the game.

The file “proj_timer.c” mostly contains functions similar to those created during lab2, if not equal.

Other modules regularly interact with the timer counter incremented by this module’s interrupt handler.

Relative weight of module: 1%

Developed by: Mário Travassos

Keyboard implementation details:

This module is responsible for registering the player's keyboard inputs, and checking if the data read is valid, and can be used to interact with the game.

The file "keyboard.c" mostly contains functions similar to those created during lab3, with some quality of life changes, in order to prevent the keyboard from bricking.

Other modules regularly interact with the scancodes read by this module's interrupt handler.

Relative weight of module: 1%

Developed by: Mário Travassos

Mouse implementation details:

This module is responsible for registering the player's mouse inputs, and checking if the data read is valid, and can be used to interact with the game.

The file "mouse.c" mostly contains functions similar to those created during lab3, with some quality of life changes, in order to prevent the mouse from bricking.

Other modules regularly interact with the packets read and parsed by this module's interrupt handler.

Relative weight of module: 6%

Developed by: Mário Travassos

RTC/Real Time Clock implementation details:

This module is responsible for reading data from the real time clock, and parsing it, so it can be used on the remainder of the game.

To do this, functions were created on the file “timer.c” which allow us to interact with the rtc’s registers, to read the time, and parse it into a format which we can use on the rest of the program.

Relative weight of module: 1%

Developed by: Mário Travassos

Sprite implementation details:

This module is responsible for loading all xpm images to be used during the game, and for drawing those same sprites. The module's functions are contained inside of the "sprite.c" file.

To do this, an object Image was created, which holds both an image's information and an image's color map:

```c
/**
 * @brief Struct that holds an image's information and color map
 */
typedef struct Image{
    xpm_image_t image_info; /*!< the image's info */
    uint8_t *color_map;     /*!< the image's color map */
} Image;
```

The "load_images()" loads all images into a few global arrays of Image objects, one for holding the game images (player object, crosshair and the game background), the menu images, the enemy related images, the number images and the letter images. To improve the readability of the code, some enumerated types were added to the file "sprite.h". These are to be used when accessing the image arrays.

```c
/**
 * @brief Enumerated type for specifying which enemy animation we're accessing
 */
enum enemy_images {
    type1_A = 0,    /*!< type 1 animation A */
    type1_B = 1,    /*!< type 1 animation B */
    type2_A = 2,    /*!< type 2 animation A */
    type2_B = 3,    /*!< type 2 animation B */
    special_A = 4,  /*!< special enemy animation A */
    special_B = 5,  /*!< special enemy animation B */
    enemy_proj = 6  /*!< enemy projectile sprite */
};


/**
 * @brief Enumerated type for holding the menu's images
 */
enum menu_images{
  menu_back = 0,    /*!< the menu's background image */
  title_1 = 1,      /*!< the first title image */
  title_2 = 2,      /*!< the second title image */
  p_butt = 3,       /*!< the play button's image */
  h_butt = 4,       /*!< the highscore button's image */
  high_back = 5,    /*!< the highscore's background image */
  new_high = 6      /*!< the new highscore screen's background image */
};


/**
 * @brief Enumerated type for holding the game's non-enemy images
 */
enum game_images{
  game_back = 0,    /*!< the game's background image */
  player_img = 1,   /*!< the player's image */
  crosshair_img = 2 /*!< the crosshair's image */
};


/**
 * @brief Enumerated type for holding the game's non-enemy images
 */
enum letter_images{
  letter_a = 0,  /*!< a letter image */
  letter_b = 1,  /*!< b letter image */
  letter_c = 2,  /*!< c letter image */
  letter_d = 3,  /*!< d letter image */
  letter_e = 4,  /*!< e letter image */
  letter_f = 5,  /*!< f letter image */
```

Another important function of this module is the "draw_xpm()" function. This function is a slightly improved version of the one created during lab5, allowing us to draw on a specific buffer of our choice (using the enumerated type on the video module, inside "video.h").

The other functions of this module are either functions responsible for changing the animation we're drawing next of different game entities (this will be touched upon later), and specific functions, which access the image arrays, and draw certain sprites, by calling the "draw_xpm()" function.

```
//Function which draws all of the UI elements
int draw_ui(){

    //Draw the player hp
    if( draw_hp() != 0 ){
        return 1;
    }

    //Draw the available ammo
    if( draw_ammo() != 0 ){
        return 1;
    }

    //Draw the points
    if( draw_points() != 0 ){
        return 1;
    }

    return 0;
}
```

For example, for drawing the UI during the game, we have the “draw_ui()” function, which calls three other functions, “draw_hp()”, “draw_ammo()” and “draw_points()”, which are responsible for drawing separate members of the UI, and each of which call the “draw_xpm()” function, drawing on the double buffer. I believe this “separation of tasks” not only improves the general readability of the code, but also allows us to easily build upon it, and, say, add a 4th UI element at will, with little need for changing already existing code.

Relative weight of module: 23%

Developed by: Mário Travassos

Logic implementation details:

This is by far the most complex module in the game. It controls the logic surrounding not only the game, but also the other modules. Its functions are declared on the “logic.c” file.

The development all functions in this module was done following the same concepts of modularity and easy readability applied to the Sprite module’s functions.

```
/**
 * @brief Enumerated type for controling the game state
 */
typedef enum state{
  menu_open = 0,        /*!< open the menu */
  menu = 1,             /*!< while the menu is open */
  highscore_open = 2,   /*!< open the highscore screen */
  highscore = 3,        /*!< while the highscore screen is open */
  game_start = 4,       /*!< start the game */
  game_ok = 5,          /*!< while the game is running */
  game_over = 6,        /*!< the game just ended */
  new_hs = 7,           /*!< getting new highscore info */
  save_new_hs = 8,      /*!< save the new highscore*/
  close_program = 9     /*!< close the program */
} state;
```

For starters, it contains an enumerated type for controlling the program state. One could say this is the heart of the project as a whole. It allows the program to functionally work as a state machine. Upon receiving an interrupt from any device, after calling the respective device’s interrupt handler, we immediately evaluate the program state variable’s value, and compare it to the enum’s, to decide what is happening and what should happen next.

Besides that extremely important function, this module is also responsible for holding data related to objects created by the game on startup, and during it. A few functions on this module also allow for easy addition and deletion of game objects as necessary, and special care was taken in order to make sure no allocated memory was left unfreed.

6 structs in total have been created, one for each type of object:

- logic_data – controls the directions the objects are moving, and holds other important game information

- Cursor – holds information related to the cursor’s position and image

- Player – holds information related to the player, varying from its position to the hit points, score, and image, among others

- Enemy – holds information to each enemy’s information, varying from its position, to its speed, value, image, etc.

- Player_projectile – holds information related to each of the player’s projectiles. These are wildly different from the enemies’ projectiles, and much more complex. Details about their generation and movement will be described below

- Enemy_projectile – holds information related to each of the enemy’s objects

An example of one of these structures can be found on the image below.

```
/**
 * @brief Struct that holds an enemy's data
 */
typedef struct Enemy {
    int x, y;               /*!< the enemy's current position */
    int x_speed, y_speed;   /*!< the enemy's speed */
    int width, height;      /*!< the width and height */
    unsigned int value;     /*!< the enemy's point value */
    struct Image *img_1;    /*!< the enemys 1st image */
    struct Image *img_2;    /*!< the enemys 2nd image */
    horizontal direction;   /*!< the enemy's horizontal direction */
    bool version1;          /*!< true if we're drawing img_1 next */
} Enemy;
```

Two other enumerated types were created to aid with interacting with the game. One of them – enemy_type - was created to make differentiating between normal enemies and special enemies simpler, on the functions that interact with Enemy objects (the normal and special enemies contain the same data, but act differently from one another, so this was done to prevent code duplication). The other one – horizontal – simply contains values relating to the direction the objects are moving, to make the maths involved in moving them simpler.

Besides the entity generation function, which simply allocate memory for every object, and initialize their member variables to the values stored on the defines inside "logic_defines.h", and the entity generation function, which free the memory previously allocated, this module also contains functions that alter the object's positions, but those are quite simple. We simply add the directional velocity to every object. There are also a few functions related to the menu and the highscores, but those are quite rudimentary.

The really interesting functions in this module are those which check for collisions between objects, and the one which generates the player projectiles in the direction of the mouse, using Bresenham's Line Algorithm.

For detecting collisions, we use a simple AABB algorithm for detecting colisions between entities. For the projectiles, the last two points of the projectile are considered to be the whole entity, which could potentially lead to some issues, when moving objects move and stand on top of the tail of a projectile, for example. I believe that this situation is quite rare, and if it happens, it should be barely noticeable, and shouldn't impact gameplay too much. For the other objects, we consider them to be boxes, of (x + object width) length, and (y + object height) height, and if any of these boxes overlap, we consider them to have collided. Again, this may cause issues with collisions being detected between an object and the transparent edges of another object, namely the enemies, but while testing I did not come across it often, and when I did, it was not a cause for much concern.

For generating the player projectiles, we use Bresenham's Line Algorithm, appropriated from the code on the following url: http://members.chello.at/~easyfilter/bresenham.html .

We had to make a few changes, namely to the return type, how the points are saved, and the loop itself, but other than that, the algorithm is the same as the one found on the link written above.

Our implementation is as follows:

The function takes 4 parameters, the starting x and y values (the place the player shoots from), and the final x and y values (the current mouse position).

```
//Generates a line based on the first and last points of a line
//code apropriated from http://members.chello.at/~easyfilter/bresenham.html
int *bresenham_line_algorithm(int x0, int y0, int xf, int yf){

    int dx =  abs(xf-x0);
    int sx = x0<xf ? 1 : -1;
    int dy = -abs(yf-y0);
    int sy = y0<yf ? 1 : -1;
    int err = dx+dy, e2; /* error value e_xy */

    //alocate memory for the points of the line (2 times the number of points)
    int *to_return = (int *) malloc(2*PROJECTILE_SIZE*sizeof(int));

    //loop to generate PROJECTILE_SIZE number of points
    for( unsigned int i = 0; i < PROJECTILE_SIZE; i++){

        //initialize the x and y value of the projectile's point number i
        to_return[2*i] = x0;
        to_return[2*i + 1] = y0;

        e2 = 2*err;

        if (e2 >= dy) {
            err += dy; x0 += sx;
        } /* e_xy+e_x > 0 */

        if (e2 <= dx) {
            err += dx;
            y0 += sy;
        } /* e_xy+e_y < 0 */
    }

    return to_return;
}
```

This implementation of the function returns a pointer to the start of an array of 2*PROJECTILE_SIZE integers, to be generated by the algorithm. The define can be found on the "logic_defines.h" file, and it can be considered to be the projectile length. The reason why we multiply it by two is because we want to store both the x and y coordinates of each point.

After allocating memory for the entire array, we generate 2* PROJECTILE_SIZE points, which we will store in the array, for accessing later.

Another function of note is "create_player_projectile()". This function initializes an object of type Player_projectile, which only contains the directional velocities of the projectile, and a pointer to the line array (generated by Bresenham's Algorythm shown above).

This function allocates memory for the Player_projectile object whose pointer we're returning, generates the projectile's line array, and then calculates the angle between the first and final points of this array, using the "atan()" function from the "cmath" library.

Knowing this angle, we calculate the directional velocities of the projectile, and floor them, in order to get an integer. We're more than content with having this small inaccuracy.

An image of the "create_player_projectile()" function can be found on the next page.

```c
//Creates a projectile object
Player_projectile *create_player_projectile(int16_t mouse_x, int16_t mouse_y){

    //Alocates memory for the projectile
    Player_projectile *to_return = (Player_projectile *) malloc(sizeof(Player_projectile));

    //check if we could not successfully alocate the memory
    if(to_return == NULL){
        return NULL;
    }

    //Generates the projectile's line
    to_return -> line = bresenham_line_algorithm(player_obj -> shoot_x, player_obj -> shoot_y,
    mouse_x, mouse_y);

    //calculate angle
    double x0 = to_return->line[0];
    double y0 = to_return->line[1];
    double xf = to_return->line[PROJECTILE_SIZE * 2 - 2];
    double yf = to_return->line[PROJECTILE_SIZE * 2 - 1];
    double dx = (xf-x0);
    double dy = (yf-y0);
    double theta = atan(dy/dx);

    //When dx < 0, our angle is obligatorily higher than 90º and -90º < atan < 90º
    //Therefore, we must add Pi to our calculated value
    if( dx < 0 ){
        theta = theta + M_PI;
    }

    //calculate vx and vy for the projectile (vx = v*cos(theta), vy = v*sin(theta))
    to_return -> vx = (int) floor((double) PROJECTILE_SPEED * cos(theta));
    to_return -> vy = (int) floor((double) PROJECTILE_SPEED * sin(theta));

    return to_return;
}
```

Relative weight of module: 33%

Developed by: Mário Travassos

Game implementation details:

The functions in this module initialize the logic_data struct used to control the important game variables, and calls functions in “logic.c” and “sprite.c” which initialize the remaining game entities, and draw them as we see fit, respectively.

The Game module also contains functions which receive as parameters the data read from each respective interrupt, and call the appropriate functions in “logic.c” and “sprite.c” as necessary, one for each type of interrupts (timer, keyboard and mouse).

Probably the most important function of that type is the “game_loop()” function. This function effectively serves as the game itself, and it consists of several if statements, to compare the value of the timer counter with other values, to perform some tasks at regular intervals. As mentioned before, and akin to the original Space Invaders, a few of those intervals can change according to the game state. For instance, after killing a certain amount of enemies, the enemies wait less and less time to move, until a point where they move every half a second! Inside the game loop we also use the “rand()” function to generate pseudo-random numbers, to simulate randomness when creating certain objects that are supposed to be created at unexpected intervals.

Relative weight of module: 10%

Developed by: Mário Travassos

Menu:

The menu module is responsible for handling almost everything related to the main menu of the game. While much more simple than its brothers Game and Highscore, it is no less important.

It contains a structure that holds information about the buttons that exist on it, which is initialized when opening the menu, after drawing the menu background on the background buffer.

Just like the Game module, this module contains functions which receive as parameters the data read from each respective interrupt, and call the appropriate functions in “logic.c” and “sprite.c”.

Relative weight of module: 5%

Developed by: Mário Travassos

Highscore:

The Highscore module is the one that I had the most fun creating, and playing around with.

One half of the functions on “highscore.c” serve only to access and interact with the highscore leaderboards, from the main menu, and those are quite rudimentary.

The really interesting ones are other half, which are functions which access and modify data written in the aptly named “highscores.txt” file, to load and save the highscores, and functions which allow the player to type a name upon getting a new highscore, to tag it and claim it as theirs.

These last few functions, while not sounding specifically interesting, were a really good way for me to get used to working with c strings in a more direct way, directly modifying them as the player saw fit, and working on them was an extremely good learning experience.

I’d like to point out the smart implementation of the typing mechanism. Initially I thought about comparing the scancode received from the keyboard interrupts with each of the scancodes of the desired keys, but that was just ugly code. Then I noticed that the scancodes on the same line simply change by 1. In other words, the scancode for T, for example, is just 1 unit higher than the scancode for R. Some helpful defines for this can be found on “scancode_defines.h”.

Knowing that, we could implement three global “maps” of sorts: arrays which, when receiving the scancode subtracted by a certain offset (the scancode of the key at the starting position of the array), would return us the corresponding character. A few code snippets can be seen further down.

While I am sure one can further improve this solution, I still think it was the most efficient solution I could come up with, given the time.

The “maps”:

```
const char make_line_1[] = {'q', 'w', 'e', 'r', 't', 'y', 'u', 'i', 'o', 'p'};
const char make_line_2[] = {'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l'};
const char make_line_3[] = {'z', 'x', 'c', 'v', 'b', 'n', 'm'};
```

The accessing of the “maps”:

```
//Keyboard handler for the new highscore screen
void new_highscore_kbd_handler(uint8_t scancode, char* store_in, int *str_pos){

  //if the pressed key is on the 1st line
  if(scancode >= MAKE_1_MIN && scancode <= MAKE_1_MAX){

    //if we have written less than 5 characters, we can still write more
    if( *str_pos < 5 ){

      //save the inputted character into the correct position of the string
      store_in[*str_pos] = make_line_1[scancode - MAKE_OFFSET_1];

      //increment the position
      (*str_pos)++;
    }
  }

  //else if the pressed key is on the 2nd line
  else if(scancode >= MAKE_2_MIN && scancode <= MAKE_2_MAX){

    //if we have written less than 5 characters, we can still write more
    if( *str_pos < 5 ){
      //save the inputted character into the correct position of the string
      store_in[*str_pos] = make_line_2[scancode - MAKE_OFFSET_2];

      //increment the position
      (*str_pos)++;
    }
  }

  //else if the pressed key is on the 3rd line
  else if(scancode >= MAKE_3_MIN && scancode <= MAKE_3_MAX){

    //if we have written less than 5 characters, we can still write more
    if( *str_pos < 5 ){
      //save the inputted character into the correct position of the string
      store_in[*str_pos] = make_line_3[scancode - MAKE_OFFSET_3];

      //increment the position
      (*str_pos)++;
    }
  }
```

Relative weight of module: 12%

Developed by: Mário Travassos

Proj:

The proj.c file contains a global variable program_state, which has values of the type enum state, and which controls the program state.

This contains our program's main function, which calls all other functions in the project as seen fit, and depending on the program state.

At the start of the execution, we prepare all devices, initialize arrays for use with the highscores, and at the end of the execution we reset all devices to their original state.

The interrupt loops consist of simple calls to each of the interrupt handlers, followed by a switch case which has cases for all possible program states. Depending on the program state, it calls different parts of the program. A quick look at a few functions of each module, and a short reading of the enumerated type state is enough to understand what will happen on each call.

Due to its stunningly small complexity, despite the size, and it coming pre-built in the files we were given to work on during the project, I decided to leave it out of the "relative weight" calculations.

Developed by: Mário Travassos

Some additional notes:

This project was developed with the intention of nothing being soft coded (asides from specific situations that will be touched upon later).

There are three big header files used for defines only: “logic_defines.h”, “video_defines.h” and “scancode_defines.h”.

The video and scancode defines files contain constant definitions used for drawing sprites on the screen and parsing the keyboard interrupts, respectively. These are not to be tampered with. Doing so will produce effects such as simple images being drawn in the wrong positions, or the keyboard scancodes not being parsed correctly. These two files are properly documented, so there should be no issue with understanding what each define is supposed to mean.

Most of the constants on the logic defines file, however, are free to be altered at will. In fact, the whole project was developed with the intention of the player having control over most of these values, and being able to change them at will (within certain reasonable bounds, which would, of course, be imposed upon the user). Furthermore, powerups were also a clever idea I was considering implemented, but one which ultimately did not come to fruition… Unfortunately, time constraints lead to these idea being scrapped. Nevertheless, changing the values of these constants, except for a few listed below, should result in no issues with the game, as the code was developed with that in mind.

Whilst “logic_defines.h” is not as well defined as the other two files mentioned before, the constant names should be explicit enough.

The highscore related defines control the number of highscores we save, load and display, and the number of letters on each of them. Changes to these were not tested, and effects should range from extra (or not enough) highscores displaying on the leaderboards, or the names having more letters, to completely breaking this system. Whatever the case, it might be fun to mess around with all of this, just be mindful of the possible consequences.

```
/*
        DEFINES FOR CONTROLING OBJECTS AND INITIALIZING OBJECT INSTANCES
*/

#define SIZEOF(x)   (sizeof(x)/sizeof(*x))  /**< @brief Macro to calculate the size of x*/

#define MAX_HIGHSCORES    5       /*!< maximum number of highscores we're allowing */
#define HS_NAME_MAX_SIZE  6       /*!< size of each highscore (COUNTING THE NULL CHARACTER) */


//UI elements defines
#define UI_BAR_HEIGHT        64
#define UI_ELEM_FIRST_Y     552
#define HP_UNITS_X          160
#define AMMO_UNITS_X        384
#define POINTS_UNITS_X      736

#define ENEMY_NUMBER     66
#define ENEMIES_PER_LINE (ENEMY_NUMBER / 6)
#define ENEMY_MOVES        6
#define START_MOVE_TIME  90
#define ENEMY_START_X    16
#define ENEMY_X_SPEED    16
// #define ENEMY_START_Y    32
#define ENEMY_START_Y    64
#define ENEMY_Y_SPEED    16
#define ENEMY_1_VALUE    20
#define ENEMY_2_VALUE    10
#define SPECIAL_VALUE   150

#define INIT_PLAYER_LIVES 3
#define PLAYER_SPEED     16
#define SHOOT_COOLDOWN   30 //60 ticks = 1 sec

#define MAX_PLAYER_PROJ  15
#define MAX_ENEMY_PROJ    5
#define PROJECTILE_SIZE  20
#define PROJECTILE_SPEED 10
```

The UI element defines are used for drawing the sprites on the

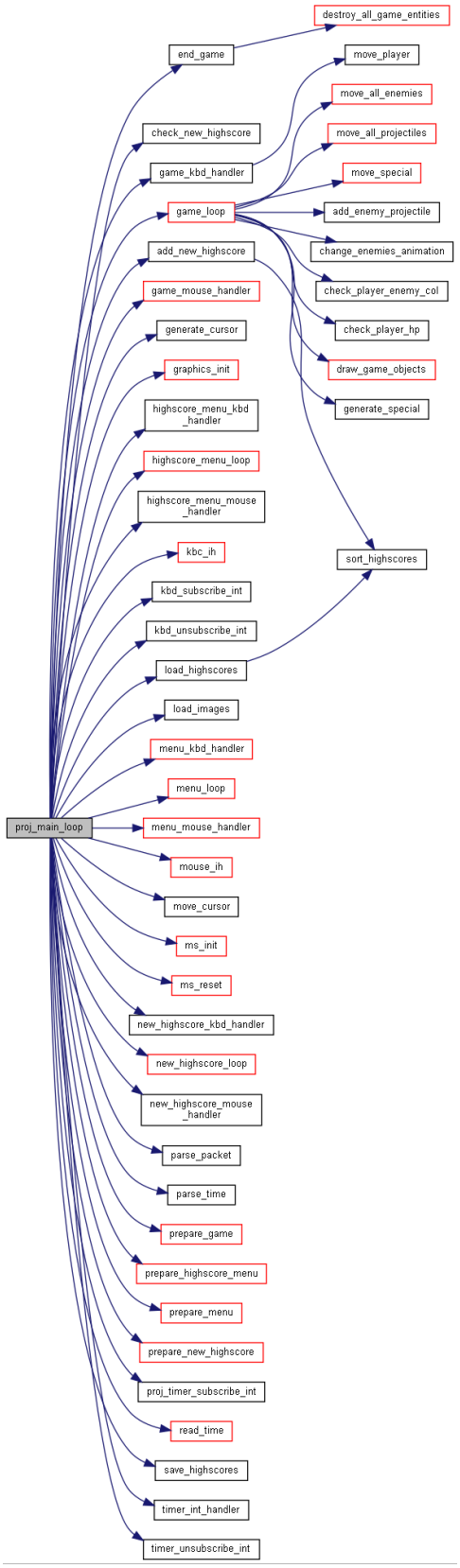
game UI. They are perhaps most fit to be on “video_defines.h”, but I must have forgotten to clean these up.

No experimentation was made with the enemy movement and spawning related constants, but the enemy values can be changed at will.

Changing the player speed will not break the game, but it might allow the player to go out of bounds, depending on the value chosen (the game only checks if the position of the player is equal to 0 or 800-width, when preventing the player from moving).

Other than that, the variables related to the projectiles and lives can be freely altered, and even any changes to the movement constants will not necessarily bring any issues other than enemies moving and/or spawning out of screen (becoming, as such, unreachable), if the values go too far above what their default values are. No crashes should occur as a result of altering these, for instance.

Function call graph:



Conclusions:

This is mostly a repetition on what I have already stated on the individual evaluation form.

The first weeks of lab work are extremely tough. The contents are really outside of our confort zone, and perhaps could be tackled in a more slow paced way. I can confirm that during the first weeks I was working 7 to 8 hours for lcom for a few days, and most of that was just looking at the computer screen reading the same sentences over and over again to decipher what was written on the pages. In addition, while I do enjoy it, I believe giving students a huge volume of unfiltered documentation provided by the manufacturer is quite terrifying for some, and can contribute for students losing their way early on.

With that being said, the whole concept of working with low level devices on a language that, although with some rigid rules, is extremely flexible, is something really interesting which should be preserved in the future. The modularity inherent to the idea of working in the devices during the labs to later use them in the project is also really good, and makes us think from the start on how we can make our lives easier in the future, during the creation of the project, and try to come up with ideas of how we can make our code as generic and flexible as possible, without compromising on efficiency. This is also a great mindset to get into when writing code in general.

From what I hear, and I can confirm it, despite the hard work required by this class, LCOM is one of the favourites of a few students, in part because we have free reign to do what we wish to for the final project, as long as it is within the scope of the class.

I can attest to the fact that despite all the lost sleep working for this class, I don't consider any of those hours wasted, because despite it being fun to make a big project like this work, and seeing all the pieces look together, all the time spent debugging stupid bugs that after all were not so stupid, and made sense, was oddly interesting. Failing, understanding why, succeeding, and right after that failing once more because of an unrelated reason, but that actually was related, we just didn't know it before, is an amazing way to learn.

I would also like to give my sincere thanks to the teachers, the monitors, the alumni and my colleagues, who were always willing to help solve problems and providing constructive criticism as to how we could improve. It made each step of this journey easier, and made developing this project an extremely fun experience as a whole.