U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

LCOM 2022/2023

Project Report

Work done by GroupT14-05:

Adriano Machado – up202105352@up.pt

José João – up202108818@up.pt

Luís Cunha – up201709375@up.pt

Rodrigo Ribeiro – up202108679@up.pt

1. Introduction

Space Invaders, a video game released in 1978, challenges players to defend Earth from waves of alien invaders. The game's simple yet addictive gameplay, combined with its retro aesthetics, has made it an iconic and enduring title in the gaming industry. Recreating Space Invaders in Minix not only pays homage to this classic game but also allows us to explore the intricacies of the operative system, implementing all the needed devices.

Throughout this report, we will discuss the key aspects of our academic endeavor, highlighting the fundamental concepts and techniques employed to bring Space Invaders to life within the Minix environment. We will provide insights into the challenges faced during the development process and the solutions devised to overcome them.

The objectives of this project encompass:

1. Explore in detail the Minix operating system, its programming environment, the LCF library, and all the needed devices.
2. Understanding the principles of game design and the mechanics of Space Invaders.
3. Implementing essential game components, such as game logic, graphics rendering, and user input handling.
4. Exploring techniques for optimizing performance and ensuring smooth gameplay.

By documenting our experiences and sharing the knowledge gained from programming Space Invaders in Minix, we hope to inspire and guide fellow students and enthusiasts in their own programming adventures. This report serves as a testament to the capabilities of Minix as a learning tool and as a testament to the enduring charm of a classic video game.

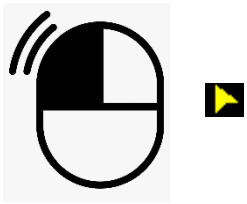
Let the journey begin as we embark on an exploration of the fusion between game development and operating systems, unveiling the inner workings of Space Invaders within the Minix universe.

2. User Instructions

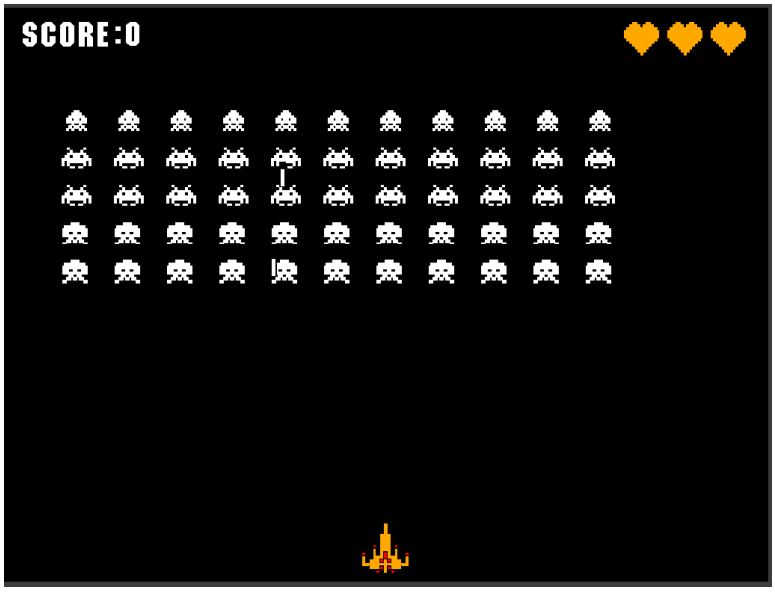
2.1. Main Menu

At the beginning of the program, the menu is displayed, consisting of three buttons: "Play Game," which initiates the game; "Highscores", where the personal highest scores are showcased with their corresponding characteristics, such as the date and time when they were achieved; and "Quit Game", which allows the user to exit the game and return to the terminal.

The navigation in the menu is done using the mouse, and to select the desired option the user must press the Left Button.

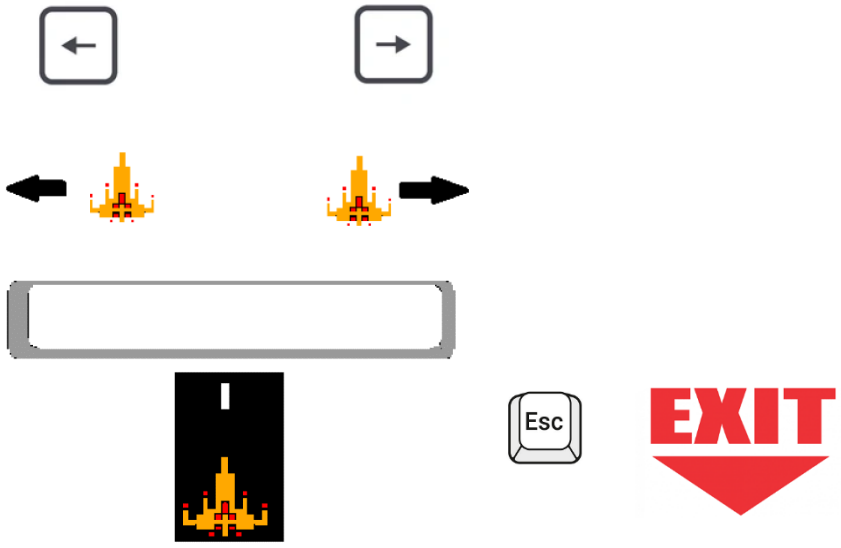


2.2. Playing the Game

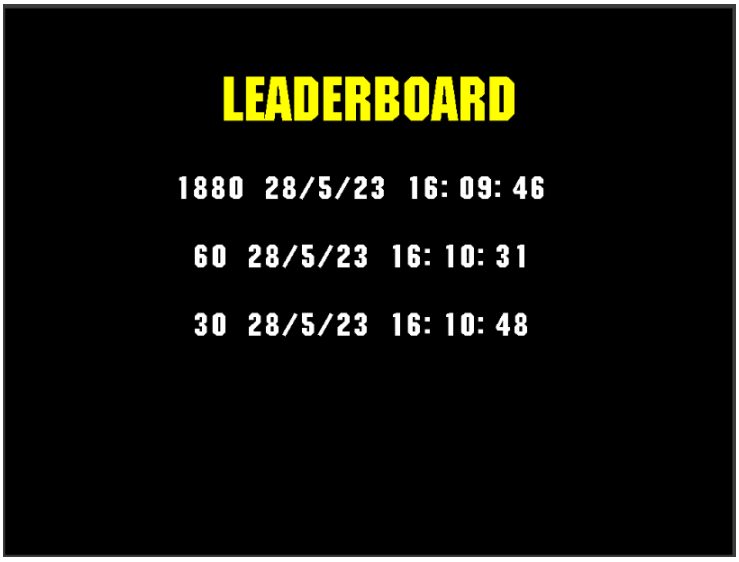


The game begins with three lives, meaning that you can only sustain damage from three enemy ship bullets. The game does not have levels; instead, the focus is on achieving a high score. For each enemy ship you defeat, you earn 10,30 or 50 points that are added to your high score. However, be cautious because every set of 55 destroyed monsters is followed by another set of 55 monsters. Your score will be saved, and if it ranks among the top 10, it will be displayed in the "Highscores" option. The game has been designed to be single player only.

The player is controlled using the keyboard arrow keys, and shoots using space bar. Pressing ESC returns to the main menu.

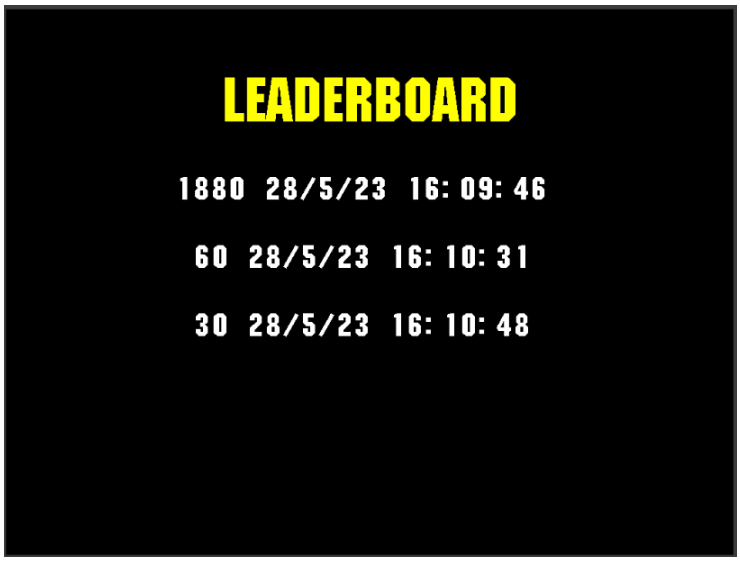


2.3. Highscores Screen



This screen shows the top 10 scores achieved by a player in a gaming session. The idea was to save the scores in a separate text file, so that we could keep track of old scores. However, probably because file permissions on Minix that we couldn't change, we were unable to read and write from files. So, the scores between gaming sessions will not be registered.

2.4. Gameover Menu



O ecrã de game over surge sempre que o jogador perde, exibindo a pontuação obtida, juntamente com a opção de voltar ao menu ou jogar novamente.

3. Project Status

Device	Functionalities	Interruptions
Timer	Frame Rate. Event management.	Yes
RTC	High scores screen, date/time display	Yes
Keyboard	Control the spaceship, shoot bullets and exit	Yes
Mouse	Menu Navigation	Yes
Video Card	Rendering and displaying visual elements in the program.	No

3.1. Timer

We used Timer 0 to obtain a fixed 60 FPS (Frames per second). While on game mode (*game_handle_timer()* function) the timer is used to animate the monsters (alternating between 2 images 2 times per second), move them, move the bullets, draw every element of the game and update their position. Of course collision detection is dependent on the timer as well, with the function *playerBulletCollision()*. While on the 3 other menus, the timer is only used to draw them. The functions related to the timer are contained in the file **timer.c**.

```
int (timer_set_frequency)(uint8_t timer, uint32_t freq);

int (timer_subscribe_int)(uint8_t *bit_no);

int (timer_unsubscribe_int)();

void (timer_int_handler)();
```

```
void (game_handle_timer)(){
    if(timer_counter % 30 == 0){
        animateMonsters(map->monsters);
    }
    moveMonsters(map);
    moveBullets(map);
    drawMap(map);
}
```

3.2. Keyboard

While in the game, the keyboard is used to move the player using the arrow keys (*movePlayer* function), to shoot using the space bar (*fireBullet* function) or go back to the main menu using ESC. The handler receives a *enum Keys key*, obtainable with the *get_key* function, that returns a Key after receiving the make code through the keyboard interrupt handler (*kbd_bytes[2]).*

```
void (game_handle_keyboard)(enum Keys key){
    if(key == Make_Arrow_left){
        movePlayer(map->player, LEFT);
    }
    else if(key == Make_Arrow_right){
        movePlayer(map->player, RIGHT);
    }
    else if(key == Make_Spacebar){
        fireBullet(map->bullets[0], map->player->drawableObject, UP);
    }
    else if(key==Make_Esc){
        addScore(map->player->score);
        resetMap(map, false, true, true, true);
        changeState(MENU);
    }
}
```

While on any menu the keyboard is only used to detect the pressing of the ESC key to exit the game. The functions related to the keyboard implementation can be found on **keyboard.c**.

```
int (keyboard_subscribe_int)(uint8_t *bit_no);

int (keyboard_unsubscribe_int)();

int (keyboard_get_status(uint8_t *st));

void (keyboard_int_handler)();

bool keyboard_parse_output();
```

3.3. Mouse

The mouse is only used on the main menu and game over menu to select among the available options using a cursor. When there is a mouse interruption, the mouse packet is built using the *mouse_build_packet()* function.

Then the menu_handle_mouse() and game_over_handle_mouse() functions update the cursor position and select the desired option if that's the case.

```
void game_over_handle_mouse(){
    updateCursor(gameover->cursor);
    game_over_option(gameover);
}
```

```
void menu_handle_mouse(){
    updateCursor(menu->cursor);
    selectOption();
}
```

```
int mouse_subscribe_int(uint8_t *bit_no);
/**...
int mouse_unsubscribe_int();
/**...
int mouse_get_status(uint8_t *st);
/**...
void mouse_ih();
/**...
bool mouse_parse_output();
/**...
uint16_t twoComplement(uint8_t* number, uint8_t msb);
/**...
void mouse_build_packet();
/**...
int write_KBC_command(uint8_t port, uint8_t cmd);
/**...
int write_mouse_cmd(uint8_t cmd);
/**...
int disable_data_report();
/**...
int enable_data_report();
```

The functions related to the implementation of the mouse can be found in the **mouse.c** file, and the information about the cursor, such as image and position, on **cursor.c**.

3.4. Video Card

For this project we used the 0x105 mode (1024x768), with indexed mode. We chose it because we didn't need many colors in our game, and the Minix palette would be enough. We only used a single buffer, even though the initial idea was to implement a double buffer to reduce flicking. However, we didn't have enough time to do this implementation.

Every image and text in the game was generated using xpms, and these files are stored on the xpms directory, with the loadXpms() function, which is responsible for the loading of xpms and store them on different arrays (game, letters, numbers, menus, and symbols), making it easier to use the images in our game. The letters, numbers, symbols, and titles present in the menus were made using a free space invaders font generator. The monsters and spaceship were taken from

free png websites and transformed to xpms by us. The functions related to the graphics implementation can be found on **graphics.c**.

```
int (video_set_mode)(uint16_t mode);

int (video_exit_mode)();

int (map_phys_mem) (uint16_t mode);

int (video_get_index)(uint16_t x, uint16_t y);

int (video_draw_pixel)(uint16_t x, uint16_t y, uint32_t color);

int (video_draw_hline)(uint16_t x, uint16_t y, uint16_t len, uint32_t color);

int (video_draw_rectangle)(uint16_t x, uint16_t y, uint16_t width, uint16_t height, uint32_t color);

int (video_draw_xpm)(xpm_image_t xpm,uint8_t* img_colors, uint16_t x, uint16_t y);

int (erase_screen)();
```

3.5. RTC

The RTC is used exclusively on the high scores screen to display when has the score been done. The function *rtc_update_time ()* is used to update the useful information on the Date struct. This information is kept in the rtc.c file and exported to other parts of the program when needed.

```
/**...
int rtc_subscribe_int(uint8_t *bit_no);
/**...
int rtc_unsubscribe_int();
/**...
int rtc_read_register(uint8_t reg, uint8_t *data);
/**...
int rtc_write_register(uint8_t reg, uint8_t data);
/**...
int rtc_startup();
/**...
int rtc_startdown();
/**...
int rtc_convert_bcd(uint8_t *data);
/**...
int rtc_update_date();
/**...
void rtc_int_handler();
/**...
typedef struct Date {
    uint8_t seconds; ///< Seconds (0-59)
    uint8_t minutes; ///< Minutes (0-59)
    uint8_t hours;   ///< Hours (0-23)
    uint8_t day;     ///< Day of the month (1-31)
    uint8_t month;   ///< Month (1-12)
    uint8_t year;    ///< Year (0-99)
} Date_t;
```

4. Code Organization/Structure

4.1. Controllers Modules Directory (Game, Menu and Entities) – 25%

The controllers directory contains everything that is related to controlling the different elements of our game. On the game controller we handle the different keyboard keys and update the necessary elements when there is a timer interruption in game mode. The controllers present on the entities directory are generally responsible for the player movement, monsters and bullets (position update for example), and the collision control can be found on the bulletController module.

The menu directory contains the menus controllers, involving timer, mouse and keyboard in the main menu, and only keyboard and timer on leaderboard and game over menus. The logic behind the options selections is present in this modules aswell.

The handleInterrupt module was a way to organize the different functions of devices depending on the current state. What is game related was implemented mainly by Adriano e José, while what is related to the menus was implemented by Rodrigo and Luís.

4.2. Devices Modules Directory – 25%

The devices directory contains every file related to the device integration in our project. The implemented devices were timer, keyboard, mouse, graphics and RTC, based on the labs. We didn't implement the serial port. The integration of this devices was done mainly by José and Adriano.

4.3. Model Modules Directory – 30%

The model directory contains all the important classes used in our game, with their respective constructors and destructors.

4.3.1. Game Models – 20%

Related to the game we have the bullet, monster and player modules, and a map module that basically contains every element present in our game. With the loadGame() function we create everything we need and with drawMap() we draw these elements on the screen. There are some other useful functions as well.

4.3.2. Menu Models – 10 %

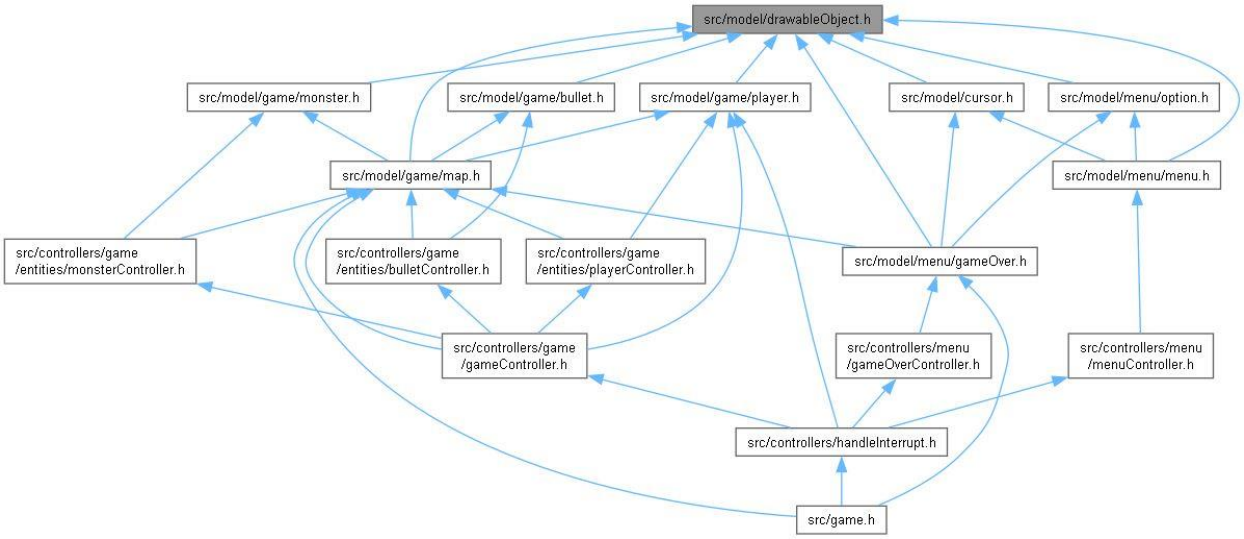
Related to the menu we have the gameOver leaderboard and menu modules, containing everything useful about these elements.

The drawableObejct module was used as a super class to everything that is drawable in our project to help on the implementation on some things. The cursor module stores the information related to the cursor. Direction and constants and useful enums to our functions. Everyone contributed equally to this part of the project.

Game Module – 10%

This module contains the main game loop, handling all the device subscription and unsubscription of interrupts, and the main structures of our game. We implemented a state machine, useful to determine the current program state. This part was mainly made by Adriano and José.

5. Function Call Graph



6. Implementation Details

6.1 Object oriented programming

Although the C programming language is not inherently object-oriented, we employed an object-oriented approach by utilizing structs. A notable example of this approach was the

'drawableObject' struct, encompassing the necessary attributes for rendering an object on the screen, such as position and xpm. For instance, in the game and menu components, we maintained an array of 'drawableObject' structures. Whenever a time interruption occurred, we iterated through the array and displayed the objects on the screen. This is just one of the numerous instances of object-oriented programming employed in our project.

6.2. Layering

We divided our project into several layers to simplify and better organize our code. The files within the "model" folder contain the logic of the game, while the files within the "controllers" folder handle interruptions and provide instructions on how to deal with them. This division of code makes it easier to understand the logic of our program.

6.3 Collisions

Collisions form the fundamental component of our game, which is why we prioritized addressing this aspect early in the project. Our collision system utilizes "hitboxes," where each object is represented by a rectangle.

7. Conclusions

In conclusion, undertaking this project in Minix and developing custom I/O devices has been an incredibly rewarding experience.

The hands-on involvement in creating and integrating these devices has deepened our understanding of the system and its inner workings. This project has provided a valuable opportunity to explore the intricacies of device drivers, low-level programming, and system interaction. The knowledge gained throughout the process will undoubtedly prove invaluable in future endeavors, fostering a deeper appreciation for the complexities involved in system development.

Overall, this project has been an enlightening journey that has enhanced our technical skills and broadened our understanding of operating systems and device integration.

I would like to express our sincere gratitude to Professor Nuno Cardoso and Monitor Bia for their invaluable support and unwavering availability throughout this project. Their guidance and expertise have been instrumental in the successful completion of this endeavor.