Faculdade de Engenharia da Universidade do Porto

L.EIC – Licenciatura em Engenharia Informática e Computação

L.EIC018 – Laboratório de Computadores

Prof. Sara Fernandes | Prof. Pedro Souto

Turma 13, Grupo 4

Nuno Miguel Carvalho Saavedra Machado, up202206186
Gonçalo Farinha Nunes, up202205538
Vítor Manuel Pereira Pires, up202207301

June 2024

Contents

I – Introduction

This report presents a comprehensive overview of the final project undertaken by our team in the LCOM (Laboratório de Computadores) 2023/24 curricular unit. In the subsequent sections, we will describe: the overall structure of the project, including its objectives and scope; how to use our project (user instructions); the project status; the organization and structure of our codebase, highlighting key modules and data structures employed; and some implementation details.

Project Overview

Let's begin by exploring the concept of the game.

What is Target Shooting?

Target Shooting offers a fun and challenging experience for all players, testing their speed, precision, and strategy. **Targets** appear on the screen, and players must quickly aim and shoot (using their mouse) to earn points. Moreover, watch out for **dynamite**! Players must detonate it before it hits the ground to avoid losing points. This twist adds another layer of strategy to the game, as players must decide whether to go for the targets or focus on the dynamite to protect their score. **The challenge continues until the countdown reaches zero**. The game has the option for **singleplayer** and **multiplayer** mode (where players can test who can score the most points before time runs out).

User Instructions

Main Menu

Figure 1 – Main menu in day mode (left) and night mode (right)

Once the program starts, the players are presented with a menu containing three options: `SINGLEPLAYER`, `MULTIPLAYER` AND `QUIT`.

- **`SINGLEPLAYER`**: solo game experience, where you can practice your shooting skills.

- **`MULTIPLAYER`**: face off against another player in a thrilling one-on-one contest; see who can accumulate the highest score before the clock hits zero.

- **`QUIT`**: it ensures that the program closes gracefully (i.e., the system resources are properly freed).

The user can navigate between options with the `UP` and `DOWN` arrow keys and select the current option by pressing `ENTER`.

The background of the menu is an image that adjusts according to the time of day when the program is being executed.

In-Game

Figure 2 – Singleplayer Game Screen

Once in the game, the player can control an aim with the mouse and must try to take the aim to the targets; once he can do it, he must shoot at the target pressing the left button of the mouse to win points. Shots in the middle of the target mean more points.

However, players must also keep an eye out for dynamite that appears on the screen. Failing to shoot (and detonate) the dynamite deducts points from their score.

One last thing, the player has the possibility to slow down everything on the screen thanks to one powerful power-up. When the icon of a white hourglass appears in the lower left corner of the screen the player can scroll down to activate slow time power-up. This power-up will regenerate with time.

Multiplayer

Figure 3 – Waiting screen.

Figure 4 – Multiplayer game.

The game also has multiplayer mode. To play it, the two players must be with the game open on the menu, each one in his own machine. Then, one of them must select the MULTIPLAYER option. This will lead the player one to a waiting screen. After this, the second player can also select the MULTIPLAYER option. Once he does this, the game will start.

In this mode, the rules are nearly the same for the single mode, with the differences that the two players battle for a higher score, and they can't use the slow time power-up.

Game Over

Figure 5 – Game Over Screen (Singleplayer)

Figure 6 – Game Over Screen (Multiplayer), winning player

When the timer expires, the game over screen is displayed, showing the player's score achieved during the game. If the player presses the ESC key, they are returned to the Main Menu.

In multiplayer version, it is indicated if the player won or lost.

II – Project Status

Devices Table

Device	Functionality	Interrupts
Timer	Control frame rate and countdowns.	Y
Keyboard	Navigate through menu.	Y
Mouse	Clicking in the game targets and mouse wheel action.	Y
Video Card	Displays all visual elements using triple buffering technique	N
RTC	Reading the real time.	N
Serial Port	Communication between VM's.	Y

We have implemented all the features mentioned in the previous section.
In this section, we will describe the code in each module and its relative weight in the project.

Devices

Devices are in a separate module: `devices/[device_name].`

Timer

- Timer controls the game's frame rate (sets a fixed frame rate on the display update) and the in-game countdown.
- In this project, it is used a frame rate of 60 Hz in timer 0 (minix default configuration for timer 0).

Keyboard

- Keyboard was used to control the flow of the game in the Main Menu and Game Over Screen, detecting keystrokes and navigating through menu options.

Mouse

- Mouse is one of the key devices for the game, mouse packets are used to track player's mouse position in the screen and to shoot at targets. Mouse wheel detection was also implemented, it is implemented in function `mouse_enable_scrolling().`

Video Card

- The video card mode used was 0x14C, with a resolution of 1152 x 864 with a direct color mode implementation with 32 bits (4 bytes) per pixel.
- Tripple buffering was also implemented using page flipping technique. After every draw, `vg_page_flipping()` is called.
- All moving elements in the game are animated sprites, clicking in them triggers the animation.

- SET_VBE_MODE (0x4F02) was used in the project.
- VBE function call 0x4F07 – That sets the display start used for page flipping technique.

RTC

- Used to read real time and show it live during the game.
- We make sure that date is consistent before attempting to read, by making a bit check on UIP (update in progress flag)

Serial Port (UART)

- Used to allow communication between VM's.
- Used both polling and interrupts, with FIFO.

III – Code organization / Structure

	Module	Path	Description	Weight (%)
Devices	timer	`devices/timer/`	- Timer implementation.	4
	keyboard	`devices/kbc/`	- Keyboard implementation. - Implementation of functions to process scan codes from keyboard.	5
	mouse	`devices/kbc/`	- Mouse implementation. - Implementation of functions to process and parse mouse packets. - Implementation of enable scrolling, set sample rate and enable data reporting.	7
	graphics	`devices/graphics/`	- Video card implementation. - Implementation of triple buffering and page flipping. - Implementation of an optimized function to draw the background.	8
	rtc	`devices/rtc/`	- RTC implementation. - The guarantee to get the correct time was made.	3
	uart	`devices/uart`	- Serial Port (UART) implementation. - It has a transmitter (`tx_queue`) and a receiver queue (`rx_queue`); - It has functions to configure the serial port, subscribe interrupts, either send or receive characters in interrupt mode, and to use the `UART FIFOS`.	5

			- Furthermore, it handles our program's communication protocol by sending and interpreting the bytes according to the specified communication rules, ensuring reliable data transmission and reception between the components of our system.	
	menu	`game/`	- Menu implementation. - Just deals with choosing an option and drawing the menu.	5
	game	`game/`	- This module deals with every logic game. - Defined here a few variables to save all the game state, since position - Implementation of functions to update times and positions for all the objects in the game, until functions to check collisions and draw all elements of the game. - Implementation of several features like slowing time, check collisions, add and remove points to score. - Implementation of functions to draw all elements in game. - It also has support for all functions if the game is in multiplayer mode.	20
	gameover	`game/`	- Game over implementation. Just draw the game over screen.	1
	wait_menu	`game/`	- Wait menu implementation. Just draw the waiting screen. - It is used when one player is waiting for another player to play a game in the multiplayer mode.	1
	position	`game/`	- Definition of a struct with x and y coordinates. - Definition of an enum Direction (LEFT, RIGHT, UP or DOWN). - These are useful for all objects of the game.	1
	player	`game/`	- Defines a struct `Player`, with the position of the player, the score of the player and the ability to shoot of the player. - Player has functions to get its position, his score or the ability to shoot. - Player also has setters for all the variables. - Player is used in the game module.	2

target	`game/`	- Defines a struct `Target`, with the position of the target, an activity status of the target, a fall counter of the target (used for the fall animation) and the direction of the target. - Target has functions to get its position. - Targets are used in the game module.	2
dynamite	`game/`	- Defines a struct `Dynamite`, with the position of the dynamite, an activity status of the dynamite, a fall counter of the target (used for the fall animation) and the direction of the target. - Dynamite has functions to get its position. - Dynamite is used in the game module.	2
sprite	`game/`	- Sprites implementation. - Functions to load sprites, delete and to draw them in video card. - The loading of sprites only happens at the beginning of the program. - We also have here arrays of Sprite elements that work as an animation so we can have animated sprites.	3
font	`resources/font/`	- All the letters and numbers xpm's used in the game.	1
sprites	`resources/sprites/`	- All the icons and images xpm's used in the game.	3
proj	`./`	- The main function. Here, we do all the devices interrupt subscriptions, and we have the project main loop the driver receive loop (the only place in the code where we receive the devices interrupts). In the end of execution, we unsubscribe all the device interrupts.	10
event_handler	`./`	- Functions to handle the different device interrupts (timer, keyboard, mouse, serial port / UART) depending on the program state. - These functions return the next program state. - Definition of the enum with all the program states.	15
utils	`./`	- Some utility functions like `util_sys_inb()`, `util_get_LSB()` and `util_get_MSB()`.	1

		- Definitions of auxiliary structs: mousePacket (to save a whole packet from mouse) and player2_info_t (to send the data about the player 2, sent via serial port.	
queue	`devices/uart/queue/`	- Implementation of a circular queue. - It is used for serial port implementation. - It has functions to enqueue and dequeue elements, to get the front element and to know if the queue is empty or full.	2
		TOTAL:	100%

Function call graph

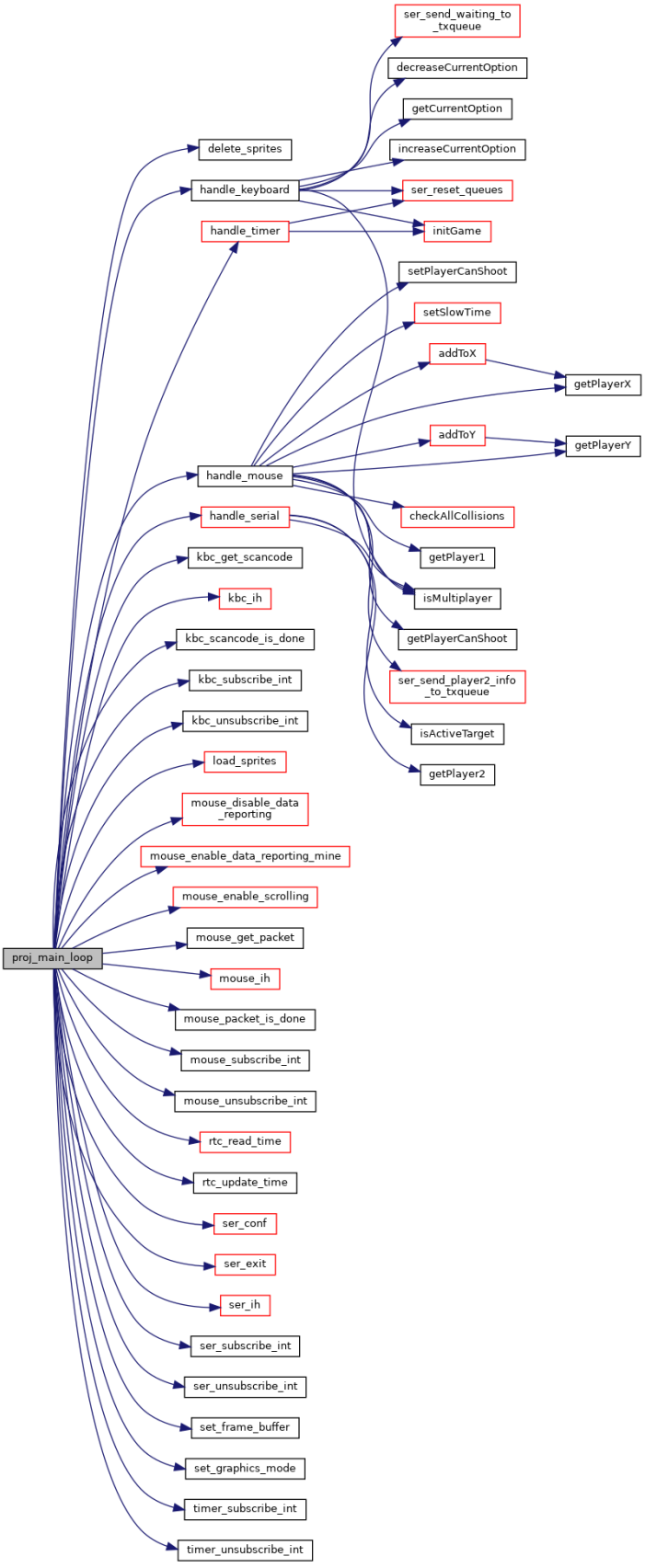

Figure 7 – `proj_main_loop` call graph

IV – Implementation Details

Triple Buffering and Page Flipping

Triple buffering uses an additional buffer to enhance rendering performance and smoothness. In triple buffering, we use three buffers: one **front buffer** and two **back buffers**.

- **Rendering**: the graphics processor renders frame n+1 to the first back buffer.

- **Additional rendering**: once frame n+1 is complete, the graphics processor starts rendering frame n+2 on the second back buffer.

- **Page flipping**: During the vertical retrace (we have used the vertical retrace VBE sub-function (`0x80`)), the system flips the front buffer with the back buffer containing the most recently completed frame. The previous front buffer becomes a back buffer ready for new frame data.

Using both triple buffering and page flipping we significantly reduce visual artifacts like screen tearing.

Draw Background

At first, we tried to use the same function for drawing both regular sprites (objects in the game) and the background. However, this caused the game to slow down significantly, making it challenging to play. The continuous call of `vg_draw_pixel()` was inefficient, especially when drawing the background (which occupies the entire screen and thus requires a lot of drawing operations).

Since the background occupies a continuous space in the buffer, we **optimized** the process by using a single `memset()` call in `draw_background()` in graphics module. This greatly improved the gaming experience.

“Object-Oriented” Programming

Object-oriented programming (OOP) is a programming paradigm based on the concept of objects (which contain both data members / fields / attributes [**state**] and code in form of procedures / methods [**behavior**]). Furthermore, OOP emphasizes data abstraction, polymorphism, inheritance, and encapsulation for flexible and secure code.

While C does not have built-in support for OOP, we can still achieve some OOP concepts through careful structuring of your code and adherence to the already mentioned principles.

We used `structs` to store the state of “objects” and created functions as methods of the class (where the first argument is a pointer to the state of the object on which the method will operate).

Load of resources

We implemented a Sprite “class” to consolidate resources for elements sharing identical visuals in our game. This class stores image dimensions and maps, enabling us to load resources only once at the start of the program and reuse them across multiple elements.

Benefits:

- **Space Efficiency**: we reduce memory usage, particularly beneficial when many elements share the same visual representation.
- **Time Efficiency**: loading resources only once minimizes redundant loading during gameplay, enhancing performance and responsiveness.

Animated Sprites

In the game, we have **animations**. For instance, when the player shoots in dynamite, an explosion occurs, and when the player shoots in a target, the target falls.

As we explained above, we didn’t want to have unnecessary sprite loadings, so we have created arrays of sprites with all the sprites of the images in the animation. Then, each object has a counter that is incremented at each of the timer interrupts if the animation is being shown, which allows us to know which frame of the animation must be shown.

This procedure is made in the functions `draw_targets()` and `draw_dynamite()` of the game module.

State Machine

We have used a state machine for this game with all the possible states of the game (MENU, WAIT, GAME, GAMEOVER, ENDGAME). This way we could use a unique `proj_main_loop()` that works the same for all the game states. We did this by using the same handler functions for all states and doing a few conditions inside the handlers to know which functions should be called (inside event_handler.c).

This approach offers several advantages, such as modularity and readability. Modularity allows each game state to be handled independently, making the code easier to expand. Also, the game logic becomes clearer and more intuitive. Basically, using this state machine helped us to write more organized and manageable code.

The implementation of the state machine is in the `event_handler` module.

Collision Detection

The **collisions** in our game only happen **between players** (aim position) **and targets or dynamite**.

We store the **center position** and **radius** of targets and dynamite (abstracting shapes into **simple geometric forms**). So, the only thing that we need to do is calculate the distance from the center. If this distance is smaller than the radius, it is because a collision has happened.

To further **optimize** collision checks with targets, we exploit their constrained (discrete) y-coordinate values (note that targets are divided into three distinct lines). By examining the y-coordinate of the player's aim position, we minimize the number of targets requiring collision checks, decreasing computational overhead.

Moreover, we enhance player engagement by implementing a **scoring mechanism that rewards accuracy**. Players receive more points for shots closer to the center of the target.

Figure 8 – Target scoring mechanism.

IntelliMouse and Scroll

We wanted to have scroll in our game for activating the power-up of slowing down the time, but the implementation of the PS/2 Mouse that we have seen in classes doesn't allow us to use this additional button.

So, we have implemented the Intellimouse extensions. We have done this by sending this sequence of commands to the mouse:

1. Set Sample Rate to 200
2. Set Sample Rate to 100
3. Set Sample Rate to 80

After the moment we do this (in the beginning of the program), the mouse works almost the same way, with the difference that the packets that the mouse sends now have 4 bytes instead of the 3 bytes of the normal version of the PS/2 Mouse. This new byte has information about the scroll, the value of the delta of scroll movement (between -8 and +7).

Real-Time Clock

We use the RTC device to keep track of the real time and change the background image accordingly. We noticed that the default configuration was in BCD, so we had to make a bit check to see in which base the values were in and then make the appropriate conversion to see the real value. A possible alternative solution was to simply change the RTC configuration to binary manually but that was not advised because it could interfere with the OS itself. We also make sure that date is consistent before attempting to read, by making a bit check on UIP (update in progress flag), this represents one of the mechanisms to overcome possible inconsistencies.

Serial Port

The **UART** (Universal Asynchronous Receiver/Transmitter) is a hardware communication protocol used for asynchronous serial communication in computing devices. It is said to be asynchronous because it does not use a shared clock signal between the transmitter and the receiver to synchronize the data transmission. Instead, both ends of the communication agree on a common data rate (baud rate) in advance.

In our implementation of the serial port, we have used a transmitter (tx_queue) and receiver (tx_receiver) software queues in order to obtain an application-independent handler.

We implemented and used both interrupts and polling (with [hardware] FIFO) for transmitting data, while only FIFO interrupts for reading. We decided to use polling because the serial port may raise a transmitter empty interrupt while there are no bytes to send.

Application-Specific Communication Protocol

For the application-specific protocol defines several states that govern the communication flow, namely:

- WAITING: The default state where the system is idle, waiting for communication to start.

- PLAYER2_INFO: Indicates that the incoming data is related to Player 2's information (position, score and the index of the target hit (if any, or –1 otherwise).

Waiting Signal Transmission: The function `ser_send_waiting_to_txqueue` sends a waiting signal to the transmission queue, indicating the system is ready to receive instructions or data.

Data transmission: `ser_send_player2_info_to_txqueue` sends Player 2's information (mentioned above) to the transmission queue. Each piece of information is preceded by a specific identifier to indicate its type.

Data handling: `ser_handle_start` and `ser_read_data_from_rx_queue` process received data based on the current communication state. `ser_handle_start` handles the start signal received, setting the communication state accordingly; if the start signal indicates waiting, the

system transitions to the WAITING state, signaling that it's ready for communication. `ser_read_data_from_rx_queue` determines whether the incoming data is Player 2's information, keyboard scancodes, or other types of data; it also updates the appropriate data structures based on the received data.

Resetting Queues: ser_reset_queues resets the transmission and reception queues.

V – Conclusion

This project, along with the previous work in this course unit, allowed us to put into practice the knowledge gained about driver implementation. We faced some problems, primarily in debugging, which proved to be a significant challenge throughout the development process.

The biggest obstacle was integrating the UART with the rest of the project. Even though we had the driver implemented, ensuring its proper functionality and integration required substantial effort.

Delays during development led to partial implementation or the absence of certain planned (and additional) functionalities, like dynamic difficulty levels, capacity of moving the scenario or leaderboard. Additionally, we identified potential enhancements such as refining the user interface and incorporating more interactive features for end users, which were not initially defined but could benefit the game flow.

Our primary accomplishments include successfully implementing drivers and devising a state machine structure for the game, demanding innovative thinking and programming approaches. We managed to overcome many technical challenges and deliver a final product we can be proud of.

This project provided invaluable lessons, emphasizing the importance of independent research, self-reliance, and critical thinking in resolving complex problems. It not only reinforced our technical knowledge but also significantly contributed to our personal and professional development. For instance, by programming a full game in C, we learned more about memory management; by implementing the UART we learned how to develop a protocol, to test it and improve it.

Each team member contributed significantly to different phases of the project, collectively trying to overcome these challenges.

Appendix

Appendix I – Installation instructions

To run this project, for single player, you just need to have a Virtual Machine in Minix open, compile the project with “make clean && make” and run with “lcom_run proj”.

To play in multiplayer mode, you should have two open virtual machines, already correctly configured with the serial port.

References

Souto, Pedro F. (2024). “Computer Labs: The PC’s Real Time Clock (RTC)”. https://web.fe.up.pt/~pfs/aulas/lcom2324/at/10rtc.pdf

Souto, Pedro F. (2024). “Computer Labs: Lab5 Video Card in Graphics Mode”. https://web.fe.up.pt/~pfs/aulas/lcom2324/at/7video.pdf

Souto, Pedro F. (2024). “Computer Labs: The PC’s Serial Port”. https://web.fe.up.pt/~pfs/aulas/lcom2324/at/11ser.pdf