# Lab-Chase

The TRON Light Cycle Clone

**Licenciatura em Engenharia Informática e Computação**
**Final Report LCOM**
**T05G02**

| | |
|---|---|
| up202008552 | Afonso Abreu |
| up202005277 | André Sousa |
| up201905429 | Pedro Moreira |
| up202008307 | Pedro Fonseca |

# Index

# 1. User Instructions

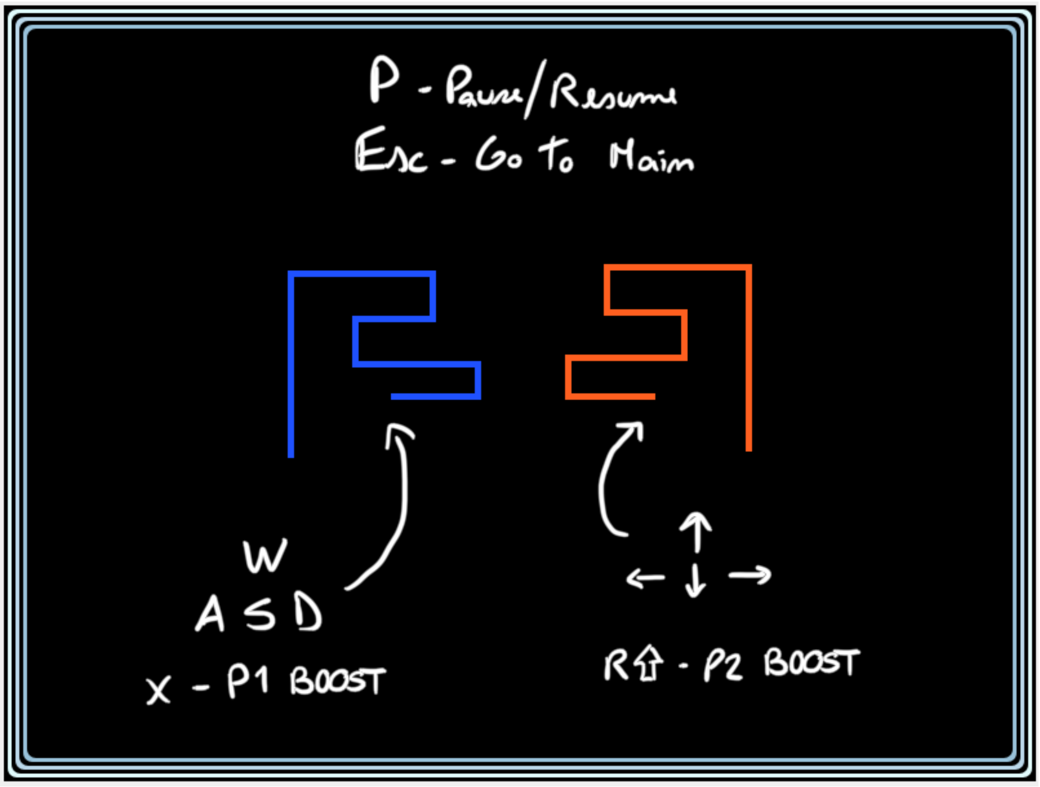
## 1.1 What is Lab-Chase?

Lab-Chase is a simple-looking, top-down, two-player game where the goal is to survive longer than the other player. To accomplish that task, you must prevent the front end of your trail of light from crashing into other entities (the enemy player, the edge of the map, or your's or the opponent's trails left behind). If you crash into any of the previously mentioned entities, you will lose the game. The front end of the trails is also equipped with a boost which allows for a brief period of faster speed. Boosts may be used as many times as desired with a short charge time in between.

## 1.2 How to play

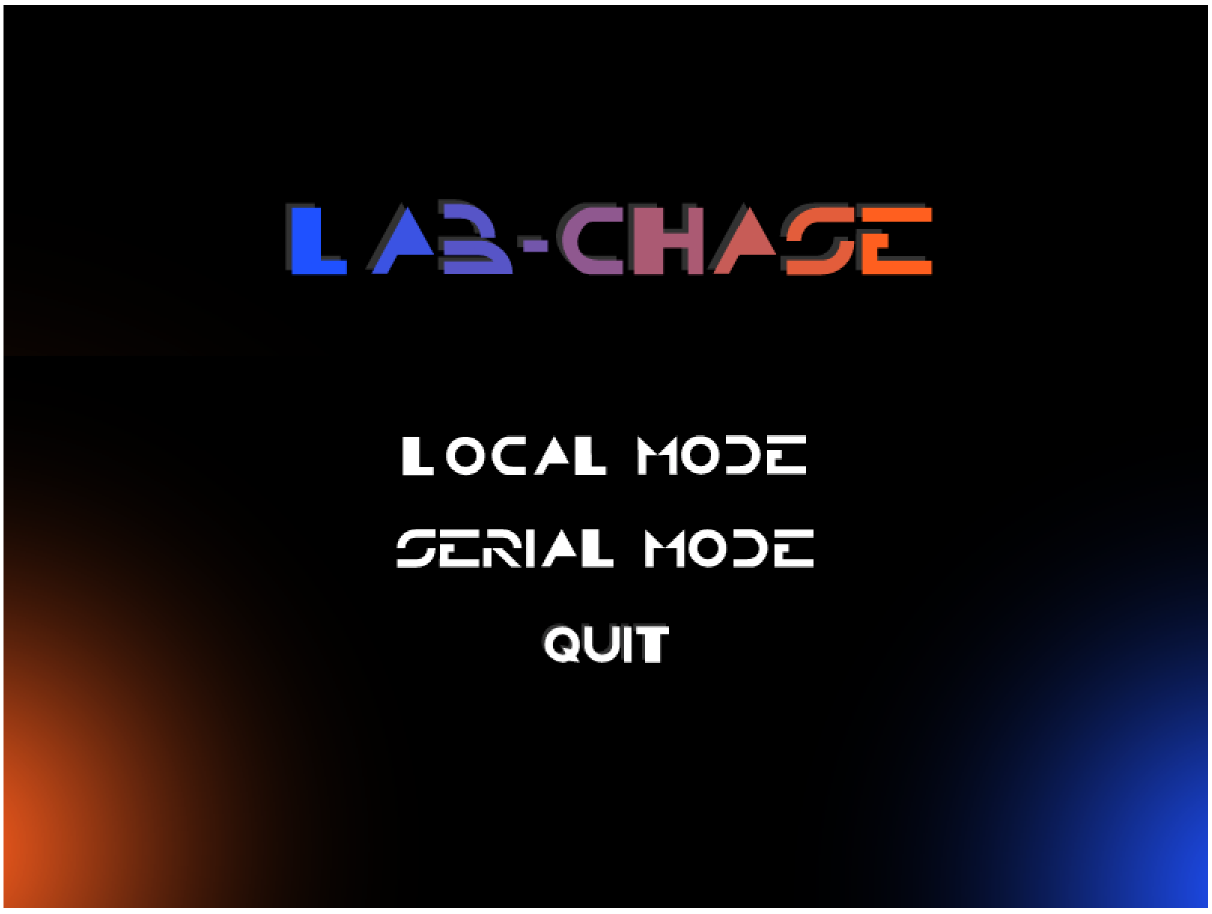
The controls for the game are as follows:

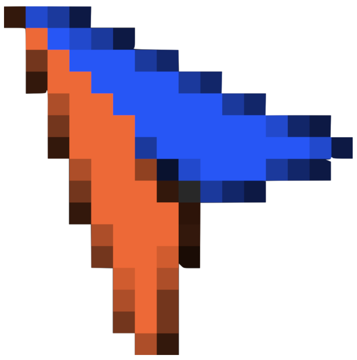
**WASD**: Controls player one, represented by the colour blue
**X**: Player one boost
**ARROWS**: Controls player two, represented by the colour orange
**Right Shift**: Player two boost
**P**: Pauses/Resumes the game, if playing in local mode
**Esc**: Goes back to the main screen while in-game



## 1.3 Main Menu

On startup, users are presented with the main menu. We created the image using other software and then converted it into a readable XPM so that we could use our lab functions to properly print it on screen.



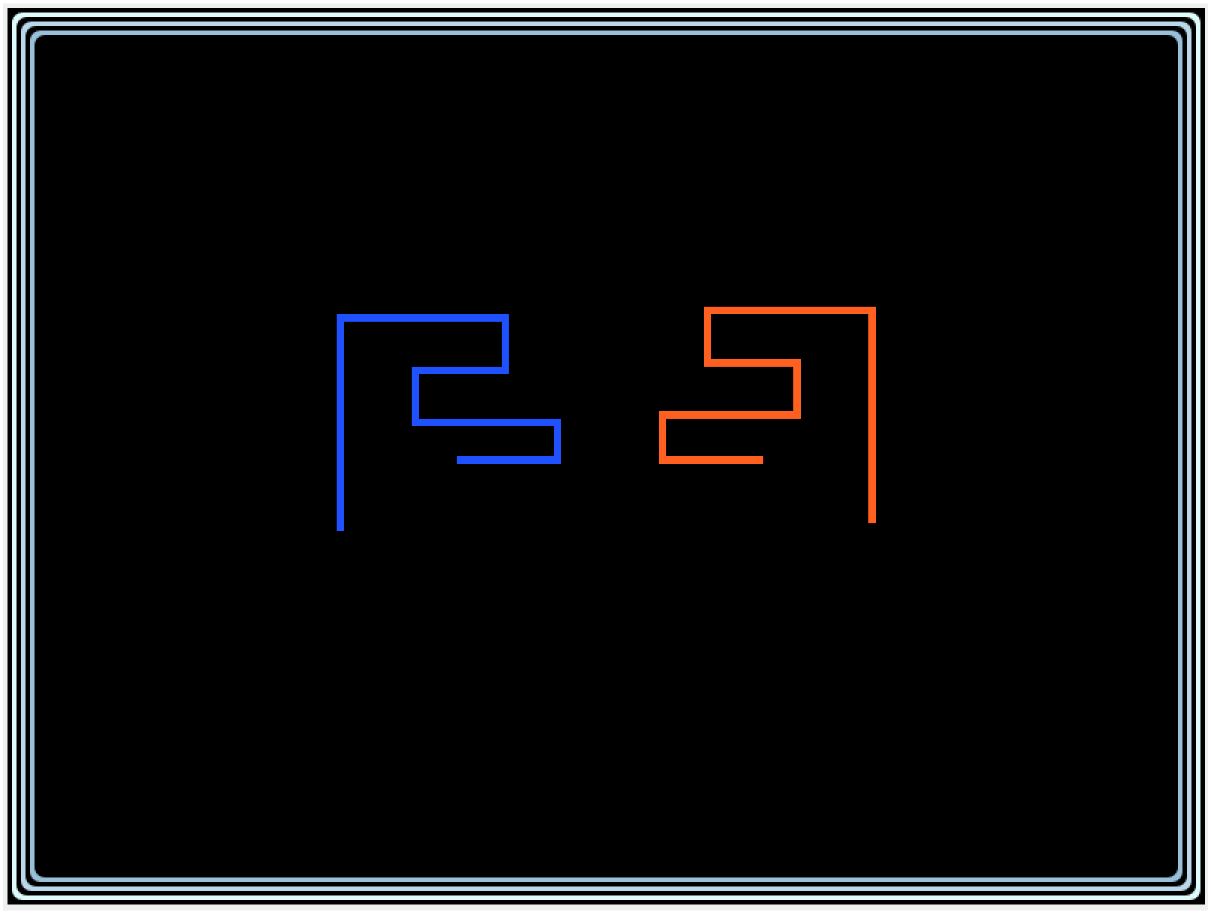


Using the mouse movement and clicks, the user with our custom cursor can pick one of the following options:

- **Local Mode**: Starts the game on the local computer with both players playing on the same keyboard.
- **Serial Mode**: On click, waits for another computer that is connected through the serial port to choose the same option and then starts the game. Each player plays on a separate computer.
- **Quit**: If clicked by the mouse, the game stops running.

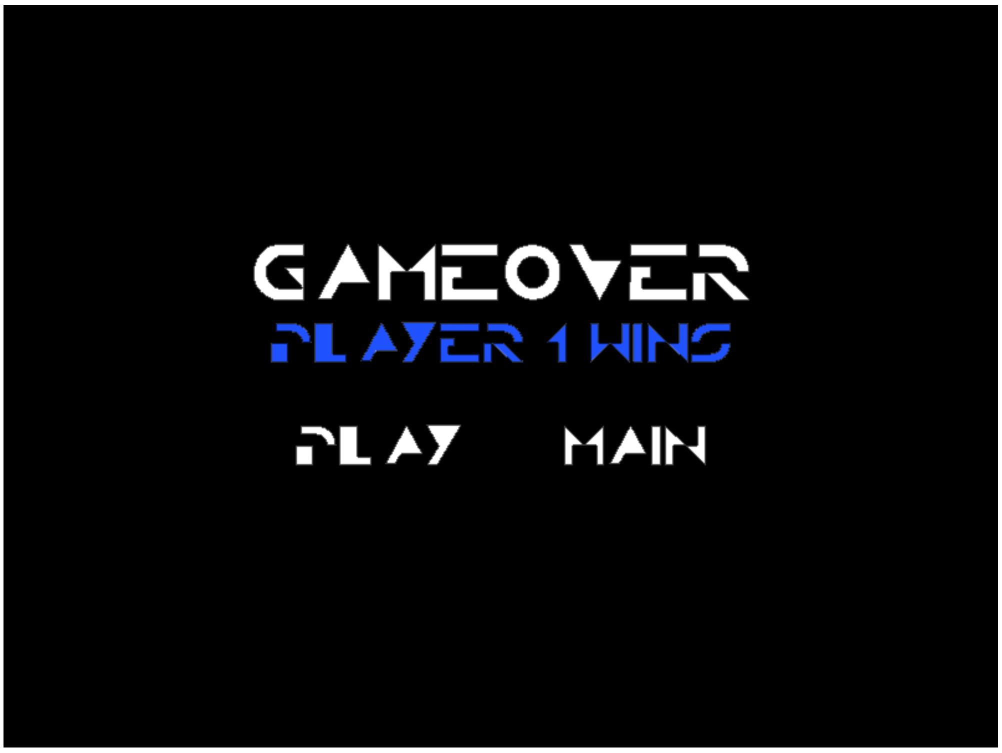
## 1.4 Gameplay

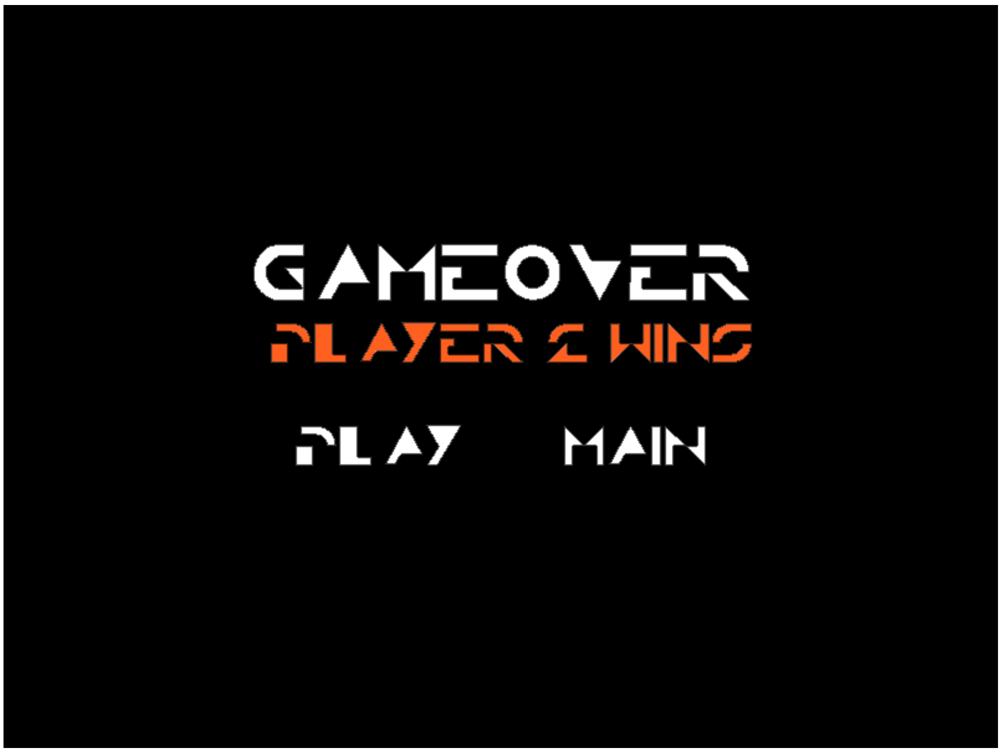
Inside the game we are initially presented with a black or white (depending on the time of the day) screen with two coloured rectangles representing the players. Using the keyboard, the players are able to move their streams of light across the screen which do not stop over time. One aspect of the game is that the speed of the streams of light in the serial mode is much slower than in the local mode. This isn't a hardware limitation. It was done solely for demonstration purposes to give enough time for one user to switch between machines before the players crash.



## 1.5 Gameover

If one of the players loses, users will be presented with a game over screen with the information on who won. On this screen, the cursor is available to choose: **PLAY** or **MAIN**.





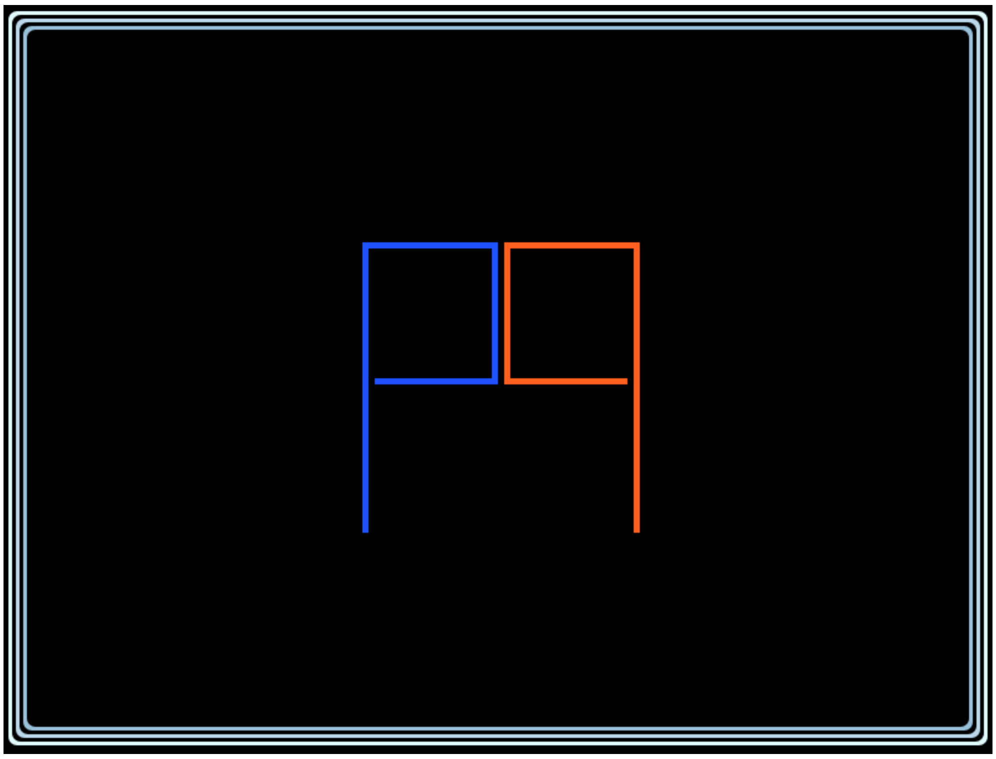
**Play**: Allows the user to restart the game
**Main**: Returns the user back to the main screen

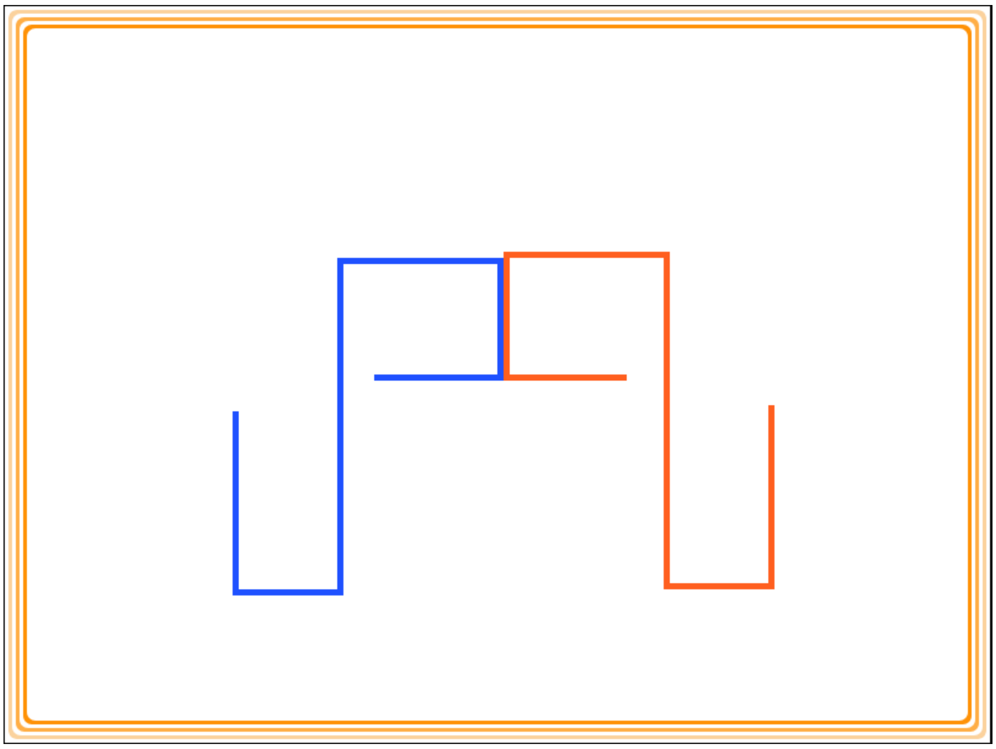
## 1.6 Pause

The pause screen is entered when the ‘P’ key is pressed. Places the game in pause and only works in Local Mode. To return to the game the user must press the ‘P’ key again.

## 1.7 Day/Night Mode

With the use of RTC, between 07:00 and 19:00, the background while in-game turns black (dark mode). At any other time, the background while playing is white (light mode).





# 2. Project Status

All functionalities previously presented were completely implemented. All the objectives mentioned in our project proposal were achieved.

The I/O Devices used in this project are as follows:

| **Device** | **What for** | **Interrupt/Polling** |
|---|---|---|
| Timer | Frame rate, Passively moving players | INTERRUPT |
| Keyboard | Both players movement | INTERRUPT |
| Mouse | Selecting options in menu | INTERRUPT |
| Video Card | Menus, In-game drawing | None |
| RTC | Day/Night mode | None |
| Serial Port | Multi-Computer Game Mode | Sending: POLLING, Receiving: INTERRUPT |

To handle all subscribed interrupts, each device has a **unsubscribe_interrupt** function implemented.

## 2.1 Timer

Timer 0 is used to make interruptions at a rate of 60Hz. The timer is essential in most of the game being especially important in moving both players even if it's not their intention (once every 5 interrupts from the timer) since the difficulty factor of this game is that the players can't be standing still.

As previously mentioned, we are using the timer through interrupts therefore we had to implement the functions:

- Ø **timer_subscribe_int**: used to subscribe the timer 0 interrupts.
- Ø **timer_unsubscribe_int**: used to unsubscribe the timer 0 interrupts.
- Ø **timer_int_handler**: used to handle the interrupts from timer 0 (each time the timer 0 raises an interrupt, the interrupt handler will increment a global variable (totalInterrupts)). Every time totalInterrupts is a multiple of 20 (serial game mode) or 5 (single computer mode), the main loop calls the function that moves the players (**passive_move_players**).

## 2.2 Keyboard

In this game, the keyboard is used to move the front-end of each player's light trail every time the player desires. The keyboard works with interrupts, which happen when a random key is pressed and therefore we implemented the following functions:

- Ø **keyboard_subscribe_int**: used to subscribe to the keyboard interrupts, in exclusive mode.
- Ø **keyboard_unsubscribe_int**: used to unsubscribe the keyboard interrupts.
- Ø **kbc_ih**: used to get the key code associated with the keyboard interrupt.

## 2.3 Mouse

The mouse was configured to do interrupts on mouse movement and mouse clicks and is used to navigate the menus through its movement and choose the desired option by clicking on the left mouse button.

Since we are using the mouse with interrupts, we had to implement the following functions:

- Ø **mouse_subscribe_int**: used to subscribe to the mouse's interrupts in exclusive mode.
- Ø **mouse_unsubscribe_int**: used to unsubscribe the mouse's interrupts.
- Ø **mouse_ih**: used to get the byte associated with each interrupt.
- Ø **parse_mouse_info**: used to convert the raw bytes received from the mouse into a more manageable data structure (struct packet provided in the lab).

## 2.4 Video Card

The video card is used to display the game, which includes the menus (main, pause and game over), the mouse and the gameplay itself to the user. We chose to run the game in mode 0x115 because this mode has 24 bits per pixel (1 byte per colour) so the range of colours available is wide. Furthermore, since we are using the function **xpm_load** provided in the labs, the mentioned mode is compatible with XPM images that have 3 bytes per pixel given the fact that this function was programmed to handle XPM that only have 1 or 3 bytes per pixel.

To use the video card properly, we use the following functions:

- Ø **new_vg_init**: uses the function **vbe_get_mode_info** provided in the lectures to get the information about a specific graphic mode. After that, the function maps the Video RAM in the process's address space, so that our game can change what is displayed on the screen. Finally, this function calls another function called **set_mode** that sets the graphic mode of the video card.
- Ø **set_mode**: programs the video card to operate in the mode specified.
- Ø **vg_draw_pixel**: colours the pixel in the specified coordinates with a given colour.
- Ø **vg_draw_rectangle**: draws a rectangle in the specified position filled with a given colour.
- Ø **vg_draw_hline**: draws a horizontal line with a given length starting in the specified position and coloured with a given colour.
- Ø **find_color**: collision detection is implemented by this function which checks if a certain position is occupied or not (if the colour in that position is black/white depending on whether the game is in light or dark mode, then there is no collision).
- Ø **draw_img**: draws XPM images on the given coordinates. This function is used to draw the menus and the cursor.

## 2.5 RTC

The RTC is used only when the program starts to get the current time to decide if the game should be in light or dark mode. To do this, the program reads the current hour set on the computer. The hour variable is then passed at the start of the game (**start_game**) and runs the game in light mode if the hour is between 7h-19h and dark mode otherwise.

For the RTC to work, the following functions are used:

- Ø **rtc_subscribe_int:** subscribes interrupts from the RTC.
- Ø **rtc_unsubsribe_int:** unsubscribes the RTC interrupts.
- Ø **rtc_read_reg:** reads the register and returns in another variable.
- Ø **read_hours:** reads the current hours.

## 2.6 Serial Port

The serial port was used to create a game mode in which the two players are playing on different computers. The serial port is programmed to operate with half-duplex communication, with a bit rate of 115200 baud, to reduce the latency as much as possible. Furthermore, we make use of FIFOs in both transmitting and receiving data. Finally, we use pooling to send data to the other device and interrupts to receive it. Regarding the data sent, each time a movement key is pressed, the direction of the movement (an integer which can be any number between 0 and 4, inclusive) is sent to the other computer.

We chose not to use the centralized approach, to minimize the latency in the movements of the players. Therefore, each computer has almost the same code and sends the input received to the other one so that the processing of each movement is done by both computers. However, this requires a more refined synchronization between the computers, so to address that problem, when one user chooses to play the multi-computer mode, the computer sends a specific character to the other one and waits for a response (function **wait_for_connection**). When the response arrives, both computers start the game.

The functions to make the Serial Port work are as follows:

- Ø **serial_subscribe:** Subscribes the interrupts sent by the serial port (in exclusive mode).
- Ø **serial_unsubscribe:** Unsubscribes the interrupts from serial port
- Ø **send_character:** Sends a byte through the serial port, using polling to check if the Transmitter Holding Register is empty
- Ø **read_character:** Reads a character from the Receiver Buffer Register. Used by the interrupt handler to get the received byte

- Ø **serial_ih:** Serial port interrupt handler
- Ø **clear_DLAB:** Sets the DLAB flag (bit 7 of the LCR) to 0
- Ø **write_to_IER:** Writes a given byte to the Interrupt Enable Register
- Ø **write_to_LCR:** Writes a given byte to the Line Control Register
- Ø **read_from_LCR:** Reads the word present in the Line Control Register
- Ø **write_to_FCR:** Writes a byte in the Transmitter Holding Register
- Ø **write_to_THR:** Writes a byte in the Transmitter Holding Register
- Ø **read_LSR:** Reads a byte from the Line Status Register
- Ø **read_RBR:** Reads a byte from the Receiver Buffer Register

# 3. Code organization/structure

## 3.1 Modules

### 3.1.1 keyboard

Provides functions specifically designed for the management of the keyboard and the manipulation of its interrupts.

Developed by:

- ➢ Pedro Moreira

### 3.1.2 main

The project's main function of the game is located on this module and is used as a connector between the different modules. Is responsible for subscribing to all used I/O devices and checking their interrupts, updating the game state as needed, checking for mouse clicks, saving the different menus images, and overall running the game.

Developed by:

- ➢ Afonso Abreu (mouse input)
- ➢ André Sousa (**load_images**, Serial Port integration)
- ➢ Pedro Moreira (RTC integration)
- ➢ Pedro Fonseca (Graphic Design, Game States)

### 3.1.3 mouse

Group of functions that enable the reception of data from the mouse as well as the reading of the mouse status.

Developed by:

- ➢ Afonso Abreu (mouse movement and clicks)
- ➢ Pedro Fonseca (initial draft)

### 3.1.4 rtc

Provides functions that deal with the rtc interrupts, the reading and writing from and into registers, and also the reading of the current time (minutes/hours).

Developed by:

- ➢ Pedro Moreira

### 3.1.5 serial port

Collection of functions that enable and disable the serial port interrupts and deals with the reception and sending of data.

Developed by:

- ➢ André Sousa (development)
- ➢ Pedro Moreira (initial draft)

### 3.1.6 timer

Enables and disables the timer interrupts, sets a frequency for the game and it does that using its interrupt handler. Similar to the lab2 version.

Already developed.

### 3.1.7 utils

A group of functions useful for the other modules. They enable data sending and getting the most and least significant bits of a certain byte for example.

Already developed.

### 3.1.8 video_gr

Deals with everything that is displayed on the screen. Takes care of drawing XPMs, gets the colour of a pixel in a determined position, saves the screen resolution, draws the cursor, deals with the collisions and the movement of the players (passive and active) and sets the intended mode for the game.

Developed by:

- ➢ Afonso Abreu (cursor drawing and location handling, collision detection)
- ➢ André Sousa (initial draft, player movement)
- ➢ Pedro Moreira (light/dark mode)
- ➢ Pedro Fonseca (XPM image drawing)

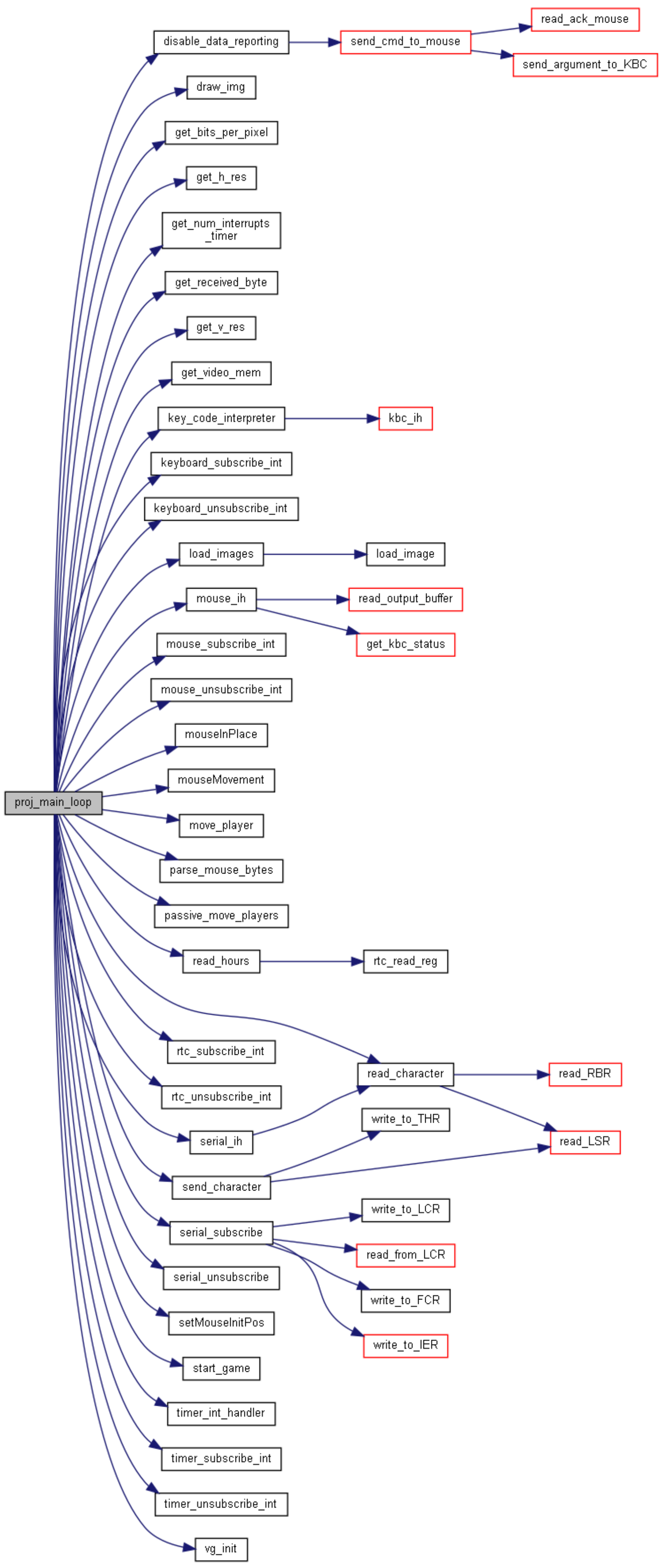
### 3.1.9 auxiliary_data_structures

Data structures used throughout the project to make the code more readable.

- ➔ **enum direction**: Enumerated type for specifying the direction of the movement
- ➔ **enum player**: Enumerated type for specifying the player
- ➔ **struct MovementInfo**: Struct to store the information regarding each movement in the game (direction and player)
- ➔ **enum screenState**: Enumerated type for specifying the current state of the screen/game
- ➔ **struct PlayerPosition**: Struct to store the player position and direction
- ➔ **struct mousePos**: Struct to store the position of the mouse
- ➔ **struct images**: Struct to store all the images, used in the game, that are loaded from xpm files

| Module | Relative Weight |
|---|---|
| Keyboard | 7% |
| Main | 33% |
| Mouse | 10% |
| RTC | 5% |
| Serial | 11% |
| Timer | 5% |
| Utils | 5% |
| Video_gr | 24% |

## 3.2 Function Call Graph



# 4. Implementation Details

## 4.1 Game states

This was a requirement for our project so that the user can navigate between the game and the various menus. Handling the current game state is done through a variable of type **enum screenState** that has the different screens as possible values, such as **MAIN**, **PAUSE**, **QUIT**, etc. Through conditional statements inside our game loop, we are able to set the current state and change the behaviour of our application, by, for example, disabling/enabling the cursor.

## 4.2 Mouse cursor

To implement the cursor in our game we had to adapt multiple functions from both the mouse and graphics labs since we needed the information from the mouse and also a way to print it on screen. During this time we faced some difficulties regarding the fact that the game crashed every time the cursor was out of bounds as well as the intense delay that the cursor had while drawing it. In order to fix these issues, we delimited bounds to our screen to make sure the mouse would stay in place and we modified some functions responsible for drawing images so that the efficiency could be higher. The fact that we used an XPM for our cursor also helped with its speed in-game.

## 4.3 RTC

Despite the fact that we did not learn the RTC in our classes, we found it to be easier than we expected as we were implementing it. The most difficult task in this department was converting BCD, used by the RTC, to decimal. Everything else went according to plan.

## 4.4 Collision

While playing the game there is a continuous necessity to check for collisions between every entity in the game (player, trail, border). To accomplish this we created a function called **find_color** whose purpose is to find the colour of a given pixel. This function works by looking at the video memory at that given coordinate by summing to the base pointer of the video memory the displacement y * h_res + x. If the value found by the function **find_color** is something other than 0x000000 (black) or 0xFFFFFF then an entity has been found.

## 4.5 UART

The UART was, without a doubt, the hardest device to program. Thus, we tried to keep its implementation simple, therefore, the serial port was programmed to only send interrupts when it received data from the other computer. When we need to send data through the serial port, we make use of polling: the program reads the status of the Transmitter Holding Register and, if empty, the data is sent; if it is not empty, the program waits 2 milliseconds before trying again. In the worst case, the program tries to send the information 5 times, after that the function gives up and returns an error.

Another difficulty of using the serial port was making sure that both computers were synchronized, therefore, in order to address that problem, when a computer starts the serial mode, the script sends a predefined message to the other computer and enters a loop waiting for the response (another predefined message). This way, there is no specific order regarding whose computer enters serial mode first, which gives more flexibility to the users.

# 5. Conclusions

We believe that making this game allowed us to better understand how I/O devices communicate with the computer by allowing us to put into practice the course's contents.

Luckily, everything we had planned for this project was accomplished. Needless to say, there were some modules harder to implement than others, such as the UART, because of the lack of know-how and theoretical foundation when starting to implement it.

If we were going to proceed to develop this project, our group would like to have implemented an option on the main screen that leads to a scoreboard that lists the longest times survived. Besides that, we would like to expand the light/dark mode to all of the menus to keep the game more coherent with the menus.

The group is really proud of what we accomplished, specifically the implementation of the UART that allows the two users to play on separate computers. Besides that, making the cursor run smoothly on the screen was a challenge and something that we are pleased by. Lastly, we are happy about our overall performance and the way the game turned out.