FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

LCOM - L.EIC018 - **Computer Laboratory Project Report**
FEUP - 2023/2024

Agalag

Class 18 - Group 1
Regent - Professor Pedro Alexandre Guimarães Lobo Ferreira Souto
Practical Classes - Professor Nuno Miguel Cardanha Paulino

Authors:
João Sousa - up202207285@up.pt
Mariana Mota Azevedo- up202005658@up.pt
Sara da Silva Azevedo - up202006902@up.pt
Tiago Carvalho - up202103339@up.pt

Index:

Game's Presentation

The game we've created for the Computer Laboratory course is an adapted version of Galaga. **Agalag**, as we named it, is a fixed shooter arcade game where the player controls a spaceship that moves horizontally on the bottom of the screen and shoots at the enemies that appear above.
The game has 3 levels, with each one having a certain time duration and the higher the level, the faster the enemies respawn. The **goal of the game** is for the player to kill the maximum number of enemies during the duration of the three levels without dying, accumulating the maximum number of points. The player loses one of its three life hearts each time it gets hit by an enemy shot. The game ends when the last level ends or if the player loses all its lives.
Before playing, in the **Main Menu**, the player can read all of the instructions mentioned above and the commands needed to play the game and also view a leaderboard to check their score position.

1.User Instructions

Anywhere in the game, the "**ESC**" key can be pressed to leave the game environment and the "**Backspace**" key can be pressed to go back to the Main Menu.

Main Menu

The Main Menu appears when the player launches the game. It shows the current real date and time of the day, the game's title, four buttons and also shows a Sun, but only during the day (between 06h00 and 20h00).
The four buttons are:

- **Play**, to start playing the game;
- **Instructions**, to view the Instructions Menu;
- **Leaderboard**, to view the Leaderboard Menu;
- **Quit** to leave the game environment.

Image 1: Main Menu Screen

On this menu, the player can move the **Mouse** to move the cursor. **Hovering** over a button changes its color. **Clicking** on a button with the mouse's left button leads the player to the corresponding screen. Pressing the "**Q**", "**Backspace**" or "**ESC**" keys leaves the game environment.

Instructions Menu

On the **Instructions Menu**, the player can find detailed instructions and commands on how to play the game. To go back to the Main Menu, the player can press the "**Backspace**" key or use the **mouse** to click on the "**Back**" button.

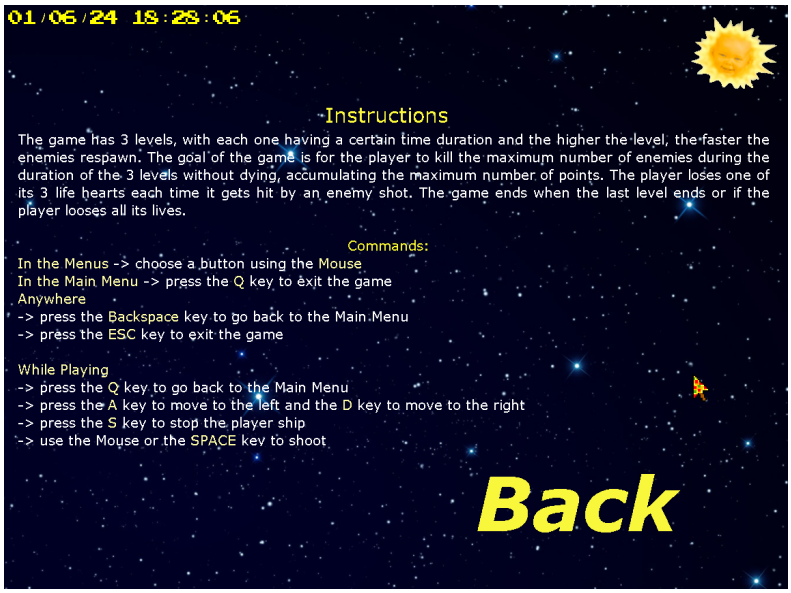

Image 2: Instructions Menu Screen

Leaderboard Menu

On the leaderboard page, the player can view some of their previous results. On the right the name of the user and on the left the score that he had. To exit the page to the main menu, the player can press the "Backspace" key.

Image 3: Leaderboard Menu Screen

Game Screen

When the player clicks on the “**Play**” button on the Main Menu, the game screen is shown and they can start playing the actual game.
On the **game screen** there is the player's ship at the bottom with the enemies that are currently alive above it, as well as the surrounding environment and any shots fired by the player or the enemies.
Additionally, there are at the **top of the screen** the current Level, the countdown of seconds to end the current level, the current player points (obtained by shooting enemies), the player’s currently available shots and current life hearts and the Sun that only appears at certain hours.

Image 4: Game Screen

While on the Game Screen, the player can press the “**Q**” or “**Backspace**” key to go back to the Main Menu. They can make the player’s ship move to the left by pressing the “**A**” key and to the right by pressing the “**D**” key, and pressing the “**S**” key makes the player’s ship stop moving. To shoot at the enemies the player can press the “**Spacebar**” key or click the **mouse’s left button**.
If the player gets hit by an enemy shot, they will **lose a life heart** and an **explosion** is shown. If an enemy gets hit by a player’s shot, they will also **explode**. More enemies will continuously **respawn** with the passing of time, at a **spawn rate** defined by the current level, stopping respawning when there is a maximum of 32 enemies on the screen.

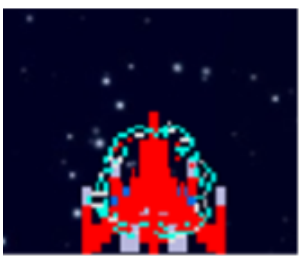
Image 5: Player Explosion

End Screen

When the player loses all of its life hearts or the time of the last level ends, the Endscreen is shown.
On this screen there is the current real date and time of the day, the Sun that only appears during the day, a Game Over message, information about how many points the player conquered while playing the game and a Back Button.
While on this screen, the player can press the "**Backspace**" key or use the mouse's left button to click on the "**Back**" button to go back to the Main Menu.

Image 6: End Screen

2.Project Status

Implemented Functionalities

- Showing the current real time and date on various menus
- Showing a Sun on various screens only at a certain real time (between 06h00 and 20h00)
- Have a mouse Cursor that can be moved and click on buttons
- Hovering over a button with the Cursor changes its color
- An Instructions Menu with instructions and commands to use the game
- A Leaderboard Menu with previous scores
- Drawing non-animated sprites for various game elements and be able to update their positions
- Using animated-sprites for the enemies and explosions
- Using double-buffering via copy for smoother display
- Show the game's current status on the game screen
- Have the player lose life hearts when shot by an enemy
- Having enemies respawn at a certain spawn rate defined by the current level
- Have 3 levels with different time durations and enemy spawn rates

- Have the player be able to move and shoot the enemies using the keyboard and the mouse
- Use the keyboard to navigate through the menus and to move the player while playing
- Calculate the player's score based on the number of killed enemies
- Collision check between the shots and enemies/player and between the enemies/player and the screen borders

Devices Usage

The devices used in the project are described in the following table:

Device	What For	Interruptions
Timer	Controlling frame rate Game time countdown Game events (enemy respawn and shots and animations control)	Yes
Keyboard	Gameplay (move ship and fire) Menu quit and go back	Yes
Mouse	Move cursor and select menu buttons Shot fire while playing	Yes
Video Card	Screens and Game Elements Display Sprites, Animations, Collision Detection Double Buffering	No
RTC	Display real time and date Display Sun during certain real time of day	Yes

Timer

The timer was used to set and control the frame rate and Timer0's **interrupts** and interrupt counter (**timerIntCounter**) were used to draw the screens and control the game events, like countdown the game time, gradually introduce more enemy ships as time passes following the level's enemy spawn rate, generate shots coming from the enemies along the time, control the explosions and enemies sprite animations.
The timer is used in functions **proj_main_loop(int argc, char *argv[])** in proj.c, **handle_game_loop()** in game.c, **updateEnemiesAndExplosions()** in enemy.c and **updateSpaceshipExplosionSprite()** in spaceship_exposion.c..
The timerIntCounter is reseted each time a level begins in **init_level(int levelIndex)** from **game.c.**

The code developed to work with the Timer's Controller interface can be found in the **timer.c** file inside the **devices** folder. From there are used the functions all the functions but timer_display_conf();

Keyboard

The keyboard is used for the gameplay to control the player's spaceship movement using the "A" and "D" keys to move left and right, respectively, to shoot using the "Space" key and to stop the spaceship using the "S" key. The keyboard keys are also used for navigation in the menus as explained in the Instructions.
It is used in functions such as **handle_pressed_key()** in game.c and **proj_main_loop(int argc, char *argv[])** in proj.c, as the Keyboard's Interrupts are used to obtain information about which key was pressed.
The code developed to work with the keyboard's and KBC's interface can be found in the **keyboard.c** and **KBC.c** files inside the devices folder.

Mouse

The mouse position and button presses are used to hover and select the button options displayed in the menus and to shoot the enemy spaceships during the gameplay. For that, the function **handle_mouse_move()** in game.c was used in proj_main_loop() in proj.c when receiving a mouse packet through interrupts to update the cursor, change the game state or fire shots, according to the packet information.
The code developed to work with the mouse's and KBC's interface can be found in the **mouse.c** and **KBC.c** files inside the devices folder.

Video Card

VBE Function **0x4F02 "Set VBE Mode"** was used to set the video card's graphics submode to be **0x14C**, with a screen resolution of **1152 x 864**, **Direct Color Mode** and a **32-bit** format (8:8:8:8) for colors, allowing **16.8 million** colors. VBE function **0x4F01** "**Return VBE Mode Info**" was used to obtain the needed information to map the virtual memory to the physical memory and function "**BIOS Set Video Mode**" was used to return to text mode. VBE Functions **0x4F07** "**VBE Set/Get Display Start Control**" and **0x4F80** "**Set Display Start during Vertical Retrace**" were also used in the context of Double Buffering, to only swap the buffers when vertical retrace occurred.
The GIMP software was used to convert all the images to XPM format to be used as **Sprites**. The video card was used for **drawing** all the screens of the game (Main Menu, Game Play, Instructions Menu, Leaderboard Menu, End Screen) and other elements. **Double buffering** via copy was used to make the animations smoother and remove flickering[1] and **Sprites** were used to to represent the game elements, with some of them (enemies and explosions) containing **animations** and **collision detection** between the shots and the player/enemies and between the player/enemies/shots and the screen borders.
The videocard's functions and submode information are used in **proj_main_loop()** in proj.c, **draw_sprite()** and **check_borders_collisions()** in sprite.c, **updateCursor()** in cursor.c, **moveEnemyFire()** in enemyFire.c and **moveFire()** in fire.c[2]

The code developed to work with the videocard's interface can be found in the **videocard.c** file inside the devices folder.

RTC

In all the menus and the end screen, the **real time and date** are displayed. A **Sun** is also displayed in all the screens if the current real time is between 06h00 and 20h00.
In **proj_main_loop()** in proj.c the RTC's Interrupts are used to update the "**timeType timeData**" and "**dateType dateData**" data structures with the current real time of the day and date. Then that information is used in **draw_game_screen()** and **draw_end_screen()** in game.c and **draw_menu()** and **draw_time()** in menu.c.
The code developed to work with the RTC's interface can be found in the **RTC.c** file inside the devices folder.

3.Code Organization/Structure

proj.c

This module serves as the entry point for the game and manages the initialization, subscription, unsubscription and handling of various device interrupts, essential for gameplay. It sets up interrupt handlers for the timer, keyboard, mouse, and real-time clock (RTC), allowing the game to respond to the passing of time or any user input and update the game's state according to that. Additionally, the module contains the main game loop, where interrupts are processed and game events are handled in real time.
The main data structures in this module include IRQ masks and variables related to device subscriptions and game state. The IRQ masks (timer_irq_set_mask, keyboard_irq_set_mask, mouse_irq_set_mask, rtc_irq_set_mask) are used to enable/disable interrupts from respective devices. These masks help with interrupt handling and ensure that the correct interrupt sources are processed during gameplay. Additionally, the module contains various variables such as the cursor and game instances to track game state and manage the game elements. These variables enable the coordination of game events, including updating the cursor position, handling keyboard and mouse input and rendering game screens based on the current state.
This module is present in 100% of the project.

game.c

This module contains the game logic, managing various game elements such as levels, players, enemies, shots, and the user interface interactions. It handles the game's progression, handles player input, updates game states, and generates different screens based on the current game state. The module is responsible for initializing levels, resetting the game state, generating enemies, handling game loop iterations, updating player status and managing transitions between different game screens. It also contains functions for drawing game elements on the screen, including the player's spaceship, enemies, explosions, and user interface components like menus and buttons.

The main data structure in this module is the Game struct, that contains attributes such as the current game state (currentState), the current level (currentLevel), and the countdown timer for the current level (levelCountdown). There is also a struct Level to store the information about each level's maximum time and enemy spawn rate. There is also an enumerated type "gameState" for the different game states, used to know what to draw and how to handle different events depending on the current state. The module also uses several arrays to store instances of game elements. These include arrays for storing enemy entities (enemyArray[]), enemy projectiles (enemyFireArray[]), enemy explosions (enemyExplosion[]), and player-fired projectiles (fireArray[]). These arrays facilitate the organization, tracking, and interaction of various game entities during gameplay. There is also an array of Levels with the definition of each of the game's 3 levels.
This module is present in 100% of the project.

Game Elements

cursor.c

This module is responsible for handling the game's cursor. It includes functions for creating, drawing, updating, and destroying the cursor. The createCursor function initializes a new cursor with a given sprite and a position, drawCursor renders the cursor on the screen, updateCursor changes the cursor's position while ensuring it stays within screen boundaries, and destroyCursor frees the memory used by the cursor.
The main data structure is the Cursor, which represents the game cursor. It has attributes such as position (x,y) and its current sprite (curr_sprite). The Sprite structure, used within the cursor, stores the cursor's graphical representation, like its dimensions and pixel data.
This module is present in 50% of the project.

enemy.c

This module is responsible for creating, animating, moving, and destroying enemy entities in the game. It contains functions for creating enemies with four different sprites, drawing the current sprite, setting initial positions, updating enemies positions and states and handling their destruction. The updateEnemiesAndExplosions function also manages the animation and movement logic for enemies and triggers enemy explosions when necessary.
The main data structure is the Enemy, which contains attributes like position (x,y), sprites (sprites array), current sprite (curr_sprite), sprite count (sprite_quant), current sprite index (curr_sprite_no), alive status (alive), and movement direction (direction). The EnemyExplosion and Fire arrays are used to manage enemy explosions and enemy fire.
This module is present in 40% of the project.

enemyExplosion.c

This module is responsible for managing the creation, animation, drawing and destruction of enemy explosion effects in the game. It includes functions for creating an enemy explosion with multiple sprites, updating the explosion sprite, drawing the current explosion sprite, and handling the destruction of the explosion object when it is no longer needed.

The main data structure is the EnemyExplosion struct, which contains the properties and behavior of an enemy explosion effect within the game. This structure includes some elements like: sprites, an array of pointers to sprites representing different frames of the explosion animation; (x,y), the coordinates of the explosion on the screen; curr_sprite, a pointer to the current sprite being displayed; sprite_quant, indicating the total number of sprites in the explosion animation; curr_sprite_no, the index of the current sprite being displayed; and alive, a flag indicating whether the explosion is still active or not.
This module is present in 10% of the project.

enemyFire.c

This module manages the enemy fire shots and detects collisions between the player's spaceship and these shots. It includes functions for creating enemy fire, drawing them on the screen, moving them, checking for collisions with the player's spaceship, and destroying them when necessary. Additionally, it maintains data structures to store enemy fire instances and related information.
The main data structure in the code is EnemyFire, representing individual enemy fire projectiles. It encapsulates attributes like position, velocity, sprite information, and a flag indicating if the projectile is active or not. The code manages the instances of EnemyFire using an array of pointers called enemyFireArray. Additionally, pointers to the player's spaceship (player) and an explosion animation (explosionPlayer) facilitate interaction and collision detection between enemy fire and the player's ship.
This module is present in 20% of the project.

fire.c

This module focuses on managing player-fired projectiles and detecting collisions with enemy ships. It includes functions for creating player-fired projectiles, drawing them on the screen, moving them, checking for collisions with enemy ships, and destroying them when necessary. Additionally, it maintains data structures to store player-fired projectiles, enemy ships, and associated explosion animations.
The main data structures manage the player and enemy interactions within the game environment. The Fire struct represents projectiles fired by the player's spaceship, containing attributes such as position, velocity, and sprite information. These instances are organized using the fireArray[] array, facilitating the handling of multiple projectiles during gameplay. The enemyArray[] array stores pointers to Enemy instances, enabling the tracking and interaction of enemy ships with player-fired projectiles. The enemyExplosion[] array manages pointers to EnemyExplosion instances, responsible for rendering explosion animations upon the destruction of enemy ships. Together, these data structures allow dynamic player-enemy interactions in the game.
This module is present in 20% of the project.

menu.c

This module is responsible for rendering the different menu screens and handling the user interactions. It includes functions for drawing the menu elements, such as background images, buttons, and time indicators. Besides, it maintains data structures to store sprites representing menu items, including buttons and visual elements like a sun. The menu system allows users

to navigate between different menu screens, such as the main menu, instructions, and leaderboard.
The main data structures consist of various sprites representing graphical elements displayed on the menu screen. These include pointers to sprites such as background, gameName, gameOver, and the sun. Additionally, buttons for different menu options are represented by sprites like playButton, leaderboardButton, instructionsButton, and quitButton, each having regular states denoted by suffixes "2" and normal, respectively. Visual separators like twoDots and slash are also managed as sprites. Furthermore, arrays of sprite pointers, such as numberArray[], are utilized to represent numerical digits used in displaying time and date information.
This module is present in 70% of the project.

playerShip.c

This module manages the player's spaceship, including functions for creating, drawing, updating, and destroying the player's ship. It also handles the score updates and spaceship explosions.
The main data structure is the PlayerShip struct, which represents the player's spaceship. This structure contains attributes such as position (x,y), current lives (currLives), current score (currScore), a pointer to the current sprite (curr_sprite), and a boolean flag indicating whether the spaceship is alive (alive) or not. Also, the code uses an instance of Spaceship_Explosion, represented by explosionPlayer, to manage explosion animations associated with the player's spaceship. This structure includes sprites representing different stages of the explosion.
This module is present in 30% of the project.

spaceship_explosion.c

This module manages the explosion animation associated with a spaceship's destruction in the game. It includes functions for creating the explosion animation, updating its frames, drawing the current frame, and destroying the animation when it's no longer needed. Besides, it utilizes a timer interrupt counter to control the rate of explosion frame updates, improving the visual effect of the explosion.
The main data structure is the Spaceship_Explosion, which represents the explosion animation associated with a spaceship's destruction. This structure contains attributes such as the explosion's position (x,y), an array of pointers to sprites representing different frames of the explosion (sprites), the current sprite being displayed (curr_sprite), the total number of sprites in the explosion animation (sprite_quant), the current sprite index (curr_sprite_no), and a boolean flag indicating whether the explosion is active (alive). The Spaceship_Explosion structure facilitates the creation and management of explosion animations.
This module is present in 10% of the project.

leaderboard.c

This module manages and displays a leaderboard for the game. It includes two primary functions: update_leaderboard and draw_leaderboard. The update_leaderboard function updates the leaderboard file by adding or replacing player scores, guaranteeing that it maintains a sorted list of the top scores. It reads existing records from a file, inserts the new score if it's higher than the existing ones, and then writes the updated list back to the file. The draw_leaderboard function reads the scores from the leaderboard file and visually displays them on the screen, generating both player names and their corresponding scores. The main data structures are the Record structure, an array of Record structures, and file pointers. The Record structure contains a player's name and score. An array of Record structures, declared as Record records[MAX_RECORDS + 1], is used to temporarily store the leaderboard entries read from the file and to manage updates before writing them back. File pointers, defined as FILE *file, are used for reading from and writing to the leaderboard file to ensure proper access to the stored data. Additionally, temporary strings and integer variables, such as char p_name[4] and int counter, are employed for intermediate storage and iteration purposes within the functions.

This module is present in 10% of the project.

Sprites

sprite.c

The module is responsible for the creation, animation, movement, and destruction of sprites in the game environment. It includes functions for initializing game elements such as players, enemies, explosions, and menu options, as well as handling border collisions, sprite movement, drawing sprites, and cleanup operations.

The main data structure in this module is the Sprite, which represents a game sprite and includes its properties such as position, speed, size, and a pixel map. Additionally, external structures like Cursor, PlayerShip, Spaceship_Explosion, Enemy, EnemyExplosion, Fire, and EnemyFire are used to represent specific game elements with distinct properties.

This module is present in 100% of the project.

Devices

utils.c

This module provides utility functions for handling the Least Significant Byte(LSB) and Most Significant Byte(MSB).

The main data structures include function parameters and local variables for data manipulation. The "val" variable in the "util_sys_inb" function stores a 32-bit value for extracting desired bytes using bitwise operations. Function parameters such as "val," "lsb," "msb," and "value" facilitate data interaction between function calls and the code. Pointers ("lsb," "msb," "value") allow modification of data values outside the function scope, providing results are reproduced to the caller.

This module is present in 10% of the project.

RTC.c

This module implements the functionality to interact with the Real-Time Clock (RTC). The module includes functions to read and update the time and date stored in the RTC, as well as handling interrupts to ensure accurate timekeeping.
The main data structures include variables, time and date structures, function parameters, status registers, control registers, and error-handling mechanisms. Variables such as rtc_hook_id, timeData, dateData, and timerIntCounter are used to store various values and track system states. Structures like timeType and dateType represent time and date information with fields for seconds, minutes, hours, day of the month, month, and year. Function parameters, such as address and byte, facilitate communication with the RTC by passing input values and receiving output values. Constants representing status registers (RTC_ADDR_REG_PORT, RTC_DATA_REG_PORT) define addresses used to read from or write to the RTC. Control registers (REGISTER_A, REGISTER_B) and IRQs (RTC_IRQ_LINE) are used for handling RTC operations and interrupts. Error codes returned by functions enable error detection and debugging.
This module is present in 10% of the project.

timer.c

This module contains the functions developed in the Lab2 practical classes that have been adapted for the game. The main data structures were explained earlier in the device details.
This module is present in 90% of the project.

keyboard.c/KBC.c

These modules contain the functions developed in the Lab3 practical classes that have been adapted for the game. The main data structures were explained earlier in the device's details.
This module is present in 70% of the project.

mouse.c

This module contains the functions developed in the Lab4 practical classes that have been adapted for the game. The main data structures were explained earlier in the device details.
This module is present in 60% of the project.

videocard.c

This module contains some of the functions developed in the Lab5 practical classes that have been adapted for the game. The main data structures were explained earlier in the device details.
This module is present in 70% of the project.

Function Call Graph

Below is the call graph of the project:

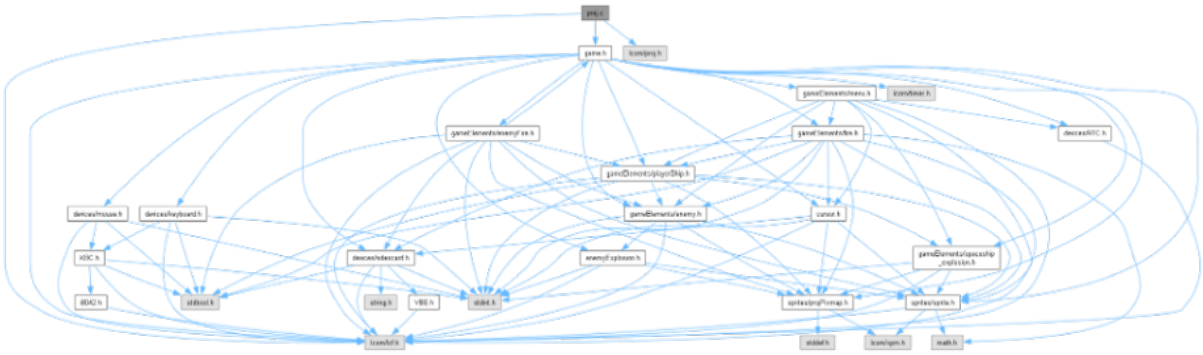

Image 7: Graph 1

The main function of the project, the one that calls driver_receive(), is **proj_main_loop()** on the **proj.c** module, which initializes the game, subscribes the needed devices' interrupts and handles those interrupts by calling other handler functions. (To see the graphs, click here).

4.Implementation Details

In the project, a **state-machine** approach was used to manage the different states of the game. Each state is linked to a specific set of actions, and transitions between states occur based on certain conditions. This approach helped to effectively organize the game's logic.
Object-oriented programming was made possible by the use of structs to represent various game elements, like the player and the enemies, among others.
The code was divided into separate modules, each responsible for specific functionalities such as game logic and input handling, using a **layered architecture** that also contributes to **modularity**.
We also applied **event-driven** programming principles to create an efficient system. Events like timer interrupts, mouse inputs, keyboard inputs, and RTC updates started related actions, allowing seamless interaction and dynamic behavior.
Additionally, the implementation included **error-handling** mechanisms. Error detection was used throughout the codebase to mitigate potential issues and improve the overall user experience.
By setting the used timer's frequency and using its interrupts to draw the game's visual elements, a **frame rate** of 60fps was obtained, being paired with **double-buffering** for obtaining quick and smooth transitions and animations.
The **RTC** was used to read from it the current real time and date to then display it and display other elements depending on the current time. The real time and date are stored in data structures created purposefully for that effect. There was care on assuring data was only read from the RTC after informing from which address the data was wanted and checking that there was no update in progress before reading.
In addition to the topics covered in the lectures, we explored areas not explicitly addressed in the course material. We explored advanced game mechanics such as **collision detection**, improving the gameplay experience.

5.Conclusions

After completing this project, we have gained a deeper insight into how each device works and the functionalities that make them up.
This understanding was essential for the successful implementation of the game. Throughout the report, we have highlighted the significant level of complexity of the project, emphasizing the importance of collaboration and communication among the group members.
The experience gained during this project has helped us strengthen our skills for future group work in other curricular units, as well as improving our skills on designing and planning the needed structures, modules and elements for a game.
We are happy with how Agalag turned out to be, as we implemented all the functionalities we had planned and even some more ideas that came to our minds along the way. However, despite creating the leaderboard and running it on the personal machine, we couldn't get it to work 100% in time for the delivery. Additionally, something we think that could add to the game in the future would be the possibility of having more than one player playing at the same time in different machines, as it would add a collaborative aspect to the game.
In conclusion, we found that it was interesting to think about where each device's functionality could be used in the context of a game and it made us wonder about other possible uses and the real different utilities of each functionality and what kind of different things can be done with each I/O device.

6.Annexes

Image 7: Graph 1

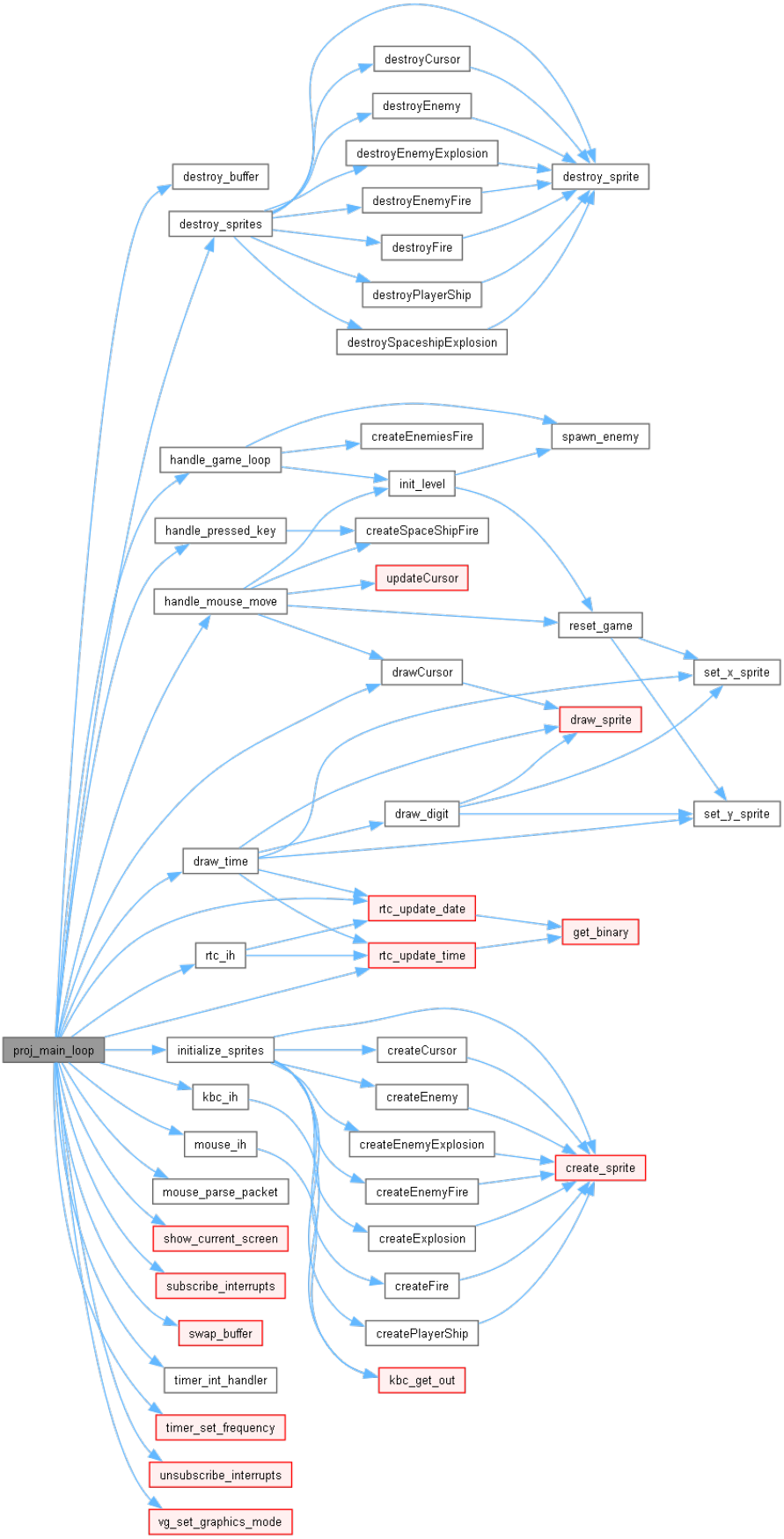

Image 7: Graph 2