THE BATTLESHIP

Final Project LCOM
T13_G03

Group members:

Alexandre Gonçalves Ramos (up202208028)
Filipa Jacob Fidalgo (up202208030)
Francisco Miguel Pires Afonso (up202208115)
Leonor Barreto Guimarães do Couto (up202205796)

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Index

1. Introduction

Our game is inspired by the classic Battleship Game, a strategic guessing game for two players. It is played on a grid where each player's fleet of warships is marked.

The game consists of two parts: firstly, players choose where to place their warships on the grid, secondly, players take turns, with each turn lasting 10 seconds, to decide where they want to "shoot" at the opponent's ships.

The objective is to destroy the opponent's fleet. The positions of the fleets are hidden from the opponent, adding to the strategic challenge.

Fig. 1 - Classic Battleship Game

2. User's Instructions

Game Objective

The objective of the game is to strategically locate and destroy the opponent's fleet of warships before they destroy yours. Players take turns "shooting" at the opponent's grid to find and sink all their ships. The first player to eliminate the opponent's fleet wins the game.

Main Menu

When the game starts the user first sees this menu. This menu changes the background according to the time and has the following options that the user can choose using the mouse:

- Local - This option allows the user to play the game locally with a friend;
- Remote - This option should allow the user to play remotely using the serial port, but it isn't fully working;
- Help - This option takes the user to the Help Menu, where they can see the instructions of the game;
- Quit - This option ends the program.

Fig. 2 - Game's main menu

Help Menu

This menu helps the user understand the game and also has instructions on how to play. The ESC button is the button used to leave this menu and go back to the Main Menu.

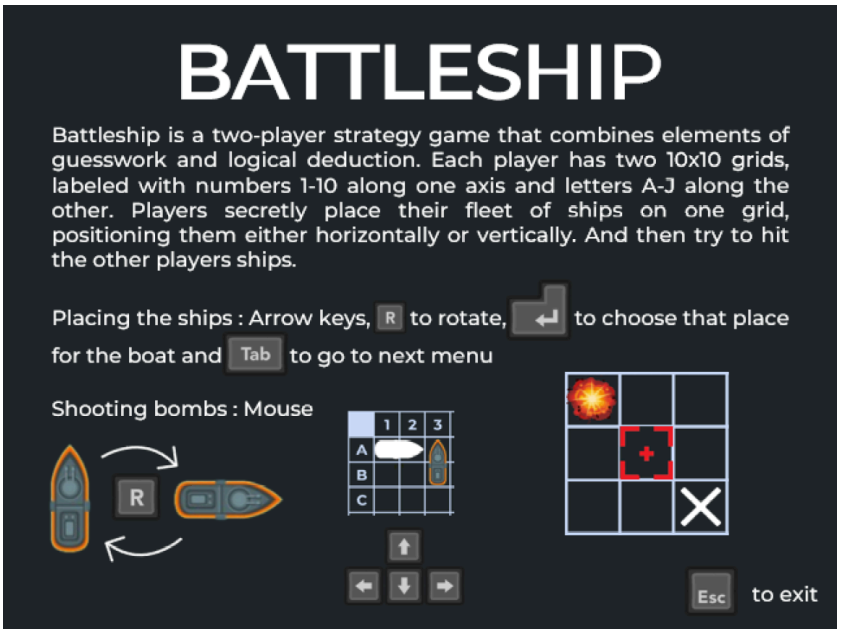


Fig. 3 - Game's help menu

Placing Ships

To start the game the players have to choose the place of their ships. The ships are selected with the keys A, B, C, D, and E and then placed using the keyboard arrows and R to rotate the ship. It's mandatory to place all the ships to continue the game.

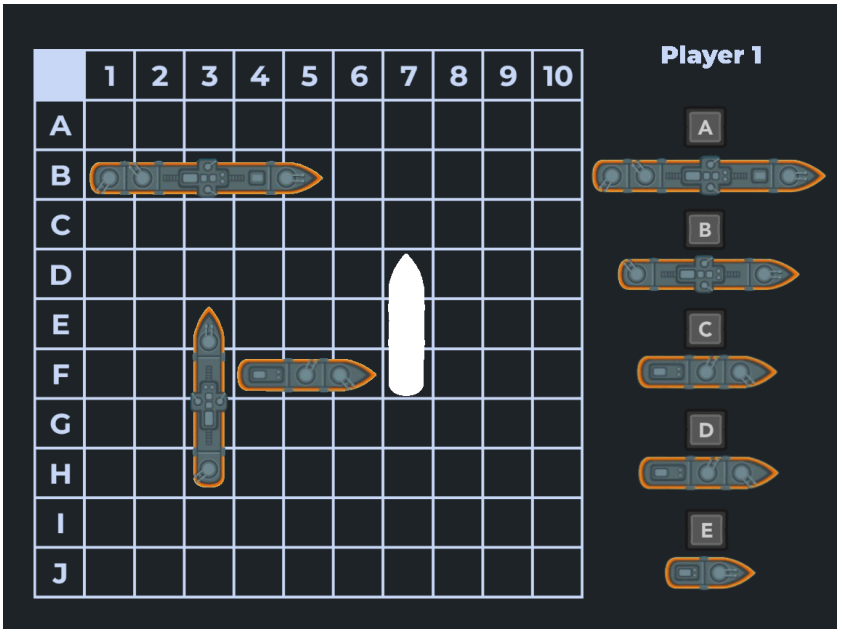
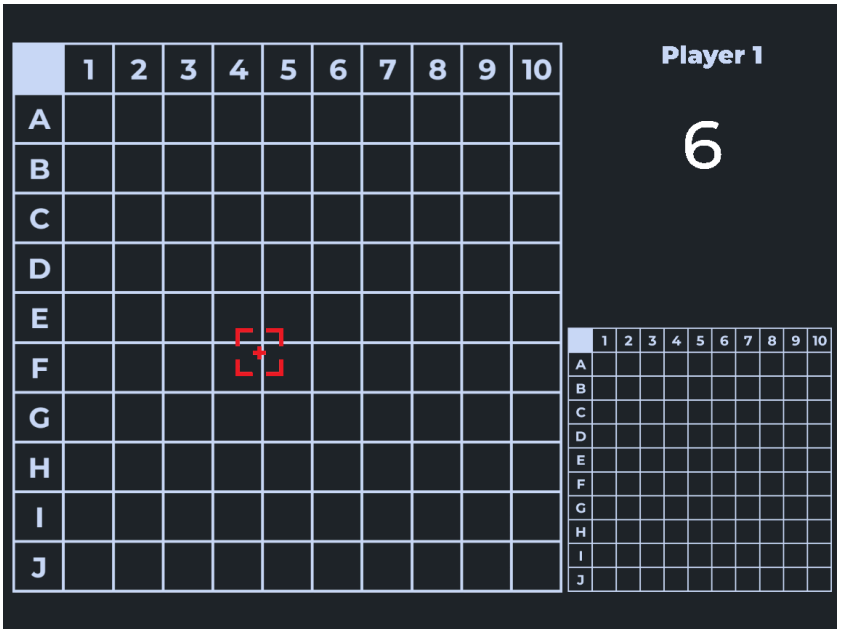


Fig. 4 - Game's placing ships menu

Game Screen

This is the actual game, where the players can place bombs to hit the other player's ships, with a time limit of 10 seconds. If a player can sink one of the other player's ships it's their turn again, if not it's the other player's turn. The game ends when a player can hit all the other player's fleet of ships.



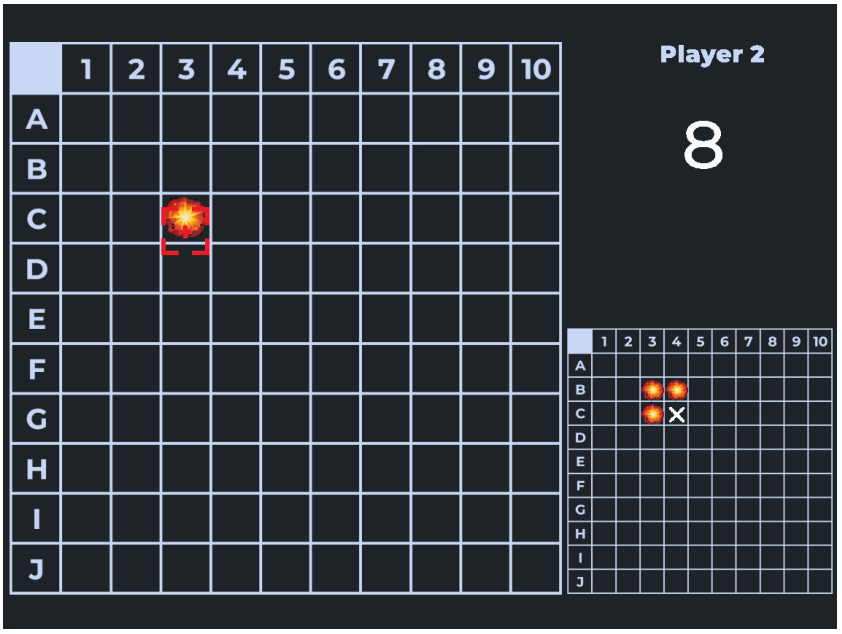


Fig. 5, 6 & 7 - Game Screen

3. Project Status

Devices table

Devices	Function	Used	Interrupts
Timer	Limit game time and frame-rate	Yes	Yes
Keyboard	Place the ships	Yes	Yes
Mouse	Choose the place for the bombs and menu navigation	Yes	Yes
Video Card	Show the entire game interface	Yes	No
RTC	Change main background according to the time	Yes	No
Serial Port	Make the communication between the two players in remote	Yes	Yes

Timer

The timer is used to limit the time each player has to choose where he wants to shoot (10 seconds) and to control the frame rate since we use double buffering. The implementation of the timer's settings, functions, and interruptions is in the file timer.c.
Important files and functions:

- timer.c (in controllers/timer folder):
  - `int (timer_set_frequency)(uint8_t timer, uint32_t freq)`: This function sets the frequency of a timer specified by 'timer' to the value 'freq'.
  - `int (timer_subscribe_int)(uint8_t *bit_no)`: Subscribes to timer interrupts, setting a bit number for the interrupt line.
  - `int (timer_unsubscribe_int)()`: Unsubscribes from timer interrupts, removing the interrupt policy.
  - `void (timer_int_handler)()`: Handles timer interrupts by incrementing a global counter.
  - `int (timer_get_conf)(uint8_t timer, uint8_t *st)`: This function is intended to get the configuration of a timer specified by 'timer'.
  - `int (timer_display_conf)(uint8_t timer, uint8_t st, enum timer_status_field field)`: This function is intended to display the configuration of a timer specified by 'timer', based on the requested field 'field'.

Keyboard

The keyboard is used to place the user ships. To move them the user must use the arrows (up, down, left, and right), to rotate the boats the user must use the R key and use Enter key to select that position. The implementation of the keyboard's settings, functions, and interruptions is in the file kbd.c. There are other functions in files like mainMenu.c, local.c, and gameLocal.c that check the inputs of the keyboard in that state of the game.
Importante files and functions:

- kbd.c (in controllers/keyboard folder):
  - `int kbd_subscribe_int(uint8_t *irq)`: Subscribes to keyboard interrupts.
  - `kbd_unsubscribe_int()`: Unsubscribes to keyboard interrupts.
  - `read_output(uint8_t port, uint8_t *scancode, uint8_t mouse)`: Reads the output from the specified port and checks for errors.
  - `restoreKBD()`: Restore the keyboard.
- mainController.c (in handler folder):
  - This file has the verification if the interrupt received is a keyboard interrupt.
- menuController.c (in handler folder):
  - `int checkKeyboardInputMainMenu(uint8_t scancode)`: This function is called by mainController.c if the interrupt is a keyboard interrupt and the current menu is the main menu.
- placingController.c (in handler folder):

- `void checkKeyboardInputPlacing(uint8_t scancode)`: This function is called by mainController.c if the interrupt is a keyboard interrupt and the actual menu is local mode.

- gameLocalController.c (in handler folder):
  - `int checkKeyboardInputGameLocal(uint8_t scancode)`: This function is called by mainController.c if the interrupt is a keyboard interrupt and the actual menu is game local.

Mouse

The game uses the mouse position to place the bombs in the game and to choose the button option in the Main Menu and the pop-up menus to go on with the game after the player places their ships. The implementation of the mouse's settings, functions, and interruptions is in the file mouse.c.

Important files and functions:

- mouse.c (in controllers/mouse folder):
  - `void (mouse_ih)()`: Mouse interrupt handler.
  - `bool (mouse_output)()`: Processes the mouse output.
  - `int (mouse_subscribe_int)(uint8_t *irq_set)`: Subscribes to mouse interrupts.
  - `int (mouse_unsubscribe_int)()`: Unsubscribes to mouse interrupts.
  - `void (create_mpacket)()`: Creates a mouse packet from the byte array.
  - `int (mouse_unsubscribe_int)()`: Unsubscribes to mouse interrupts.
  - `int (mouse_data_report)(uint8_t command)`: Sends a mouse command and waits for an acknowledgment.
  - `int (write_command)(uint8_t port, uint8_t cmd)`: Writes a command to the specified port.
- mainController.c (in handler folder):
  - This file has the verification if the interrupt received is a mouse interrupt.
- menuController.c (in handler folder):
  - `int checkMainClick()`: This function is called by mainController.c if the interrupt is a mouse interrupt and the current menu is the main menu.
- placingController.c (in handler folder):
  - `void checkPlacingClick()`: This function is called by mainController.c if the interrupt is a mouse interrupt and the actual menu is placing menu.
- gameLocalController.c (in handler folder):
  - `int checkGameLocalClick()`: This function is called by mainController.c if the interrupt is a keyboard interrupt and the actual menu is game local.

Video Card

The video card is used to show the graphic interface of the game, that is the menus, board, boats, buttons, etc. To draw the images we used video card and sprites of the images.

Important files and functions:

- graphics.c (in controllers/video_graphics folder):
  - `void swap_buffers()`: Swaps the contents of the double buffer with the actual video memory.
  - `int (_vg_init)(uint16_t mode)`: Initializes the video graphics mode.
  - `int (enter_video_mode)(uint16_t mode)`: Enters the specified video mode.
  - `int (vg_draw_pixel)(uint16_t x, uint16_t y, uint32_t color)`: Draws a pixel on the screen at the specified coordinates.
  - `int (bitsPerPixel)()`: Gets the bits per pixel for the current video mode.
  - `void (printXPM)(uint16_t x, uint16_t y, uint8_t* pixmap, xpm_image_t *img)`: Prints an XPM image at the specified coordinates.
  - `int (vg_draw_xpm)(xpm_map_t xpm, uint16_t x, uint16_t y)`: Draws an XPM image at the specified coordinates.
- sprite.c (in controllers/sprite folder):
  - has functions that create sprite objects from xpm's, that load the needed sprites, that draw and destroy sprites.
- assetDraw.c (in view folder):
  - has several draw functions that call the sprite draw function.
- mainController.c (in handler folder):
  - calls the function that initializes the video card and swap buffers.
- menuDraw.c (in view folder):
  - `void drawMainMenu()`: function that draws the main menu
  - `void drawMainCursor()`: function that draws the main menu cursor
- helpMenuDraw.c (in view folder):
  - `void drawHelpMenu()`: function that draws the help menu
- gameLocaDraw.c (in view folder):
  - `void drawGameLocal()`: function that draws the game local
  - `void drawGameCursor()`: function that draws game cursor
- placingDraw.c (in view folder):
  - `void drawLocalMode()`: function that draws the local mode
  - `void drawLocalCursor()`: function that draws the local mouse

RTC

We use the RTC to extract the current hour. The implementation of the rtc's settings, functions, and interruptions are in the file rtc.c.
Important files and functions:

- rtc.c (in controllers/rtc folder):
  - `int RTCBinary(uint8_t *bcd)`: Check if the number of the RTC is in binary-coded decimal (BCD).
  - `void toBinary(uint8_t * time)`: Transform the RTC from BCD to binary.
  - `int readRTC(uint8_t * time, uint8_t port)`: Read a register from the RTC.
  - `int timeRTC()`: Test the results of the rtc to find the adequate background for the main menu.
- mainMenuDraw.c (int view folder):
  - void drawMainMenu() calls timeRTC() and chooses the correct background.

Serial Port

We use the serial port to allow two players to play on different computers. The implementation of the serial port's settings, functions, and interruptions are in the file serialport.c.
Important files and functions:

- serialport.c (in controllers/serialPort folder):
  - `void (mouse_ih)()`: Mouse interrupt handler.
  - `bool (mouse_output)()`: Processes the mouse output.
  - `int (mouse_subscribe_int)(uint8_t *irq_set)`: Subscribes to mouse interrupts.
  - `int (mouse_unsubscribe_int)()`: Unsubscribes to mouse interrupts.
  - `void (create_mpacket)()`: Creates a mouse packet from the byte array.
  - `int (mouse_unsubscribe_int)()`: Unsubscribes to mouse interrupts.
- gameRemoteController.c (in handler folder):
  - `int received(uint8_t val)`: Function that checks the content of the byte that was received and what should do in the game

4. Code Organization/Structure

Controllers (40%)

Timer Module

This module contains the functions developed in Lab 2 that could be reused for the project. These include the functions to set the timer's frequency, subscribe to and unsubscribe from timer interrupts, handle the interrupts, and retrieve and display the timer configuration.

Main data structures:

- `int timer_hook_id`: A global variable that stores the hook ID used for managing interrupt subscription and unsubscription. It is essential for identifying the timer's interrupt requests.
- `int global_counter`: A global variable that keeps track of the number of timer interrupts that have occurred. This counter is incremented in the interrupt handler and is used for measuring time intervals and tracking elapsed time.

The relative weight of the module in the project : 3%

Keyboard Module

This module contains the functions developed in Lab 3 that could be reused for the project. These include the functions to subscribe, unsubscribe, and handle keyboard interrupts.

Main data structures:

- `int hook_kbc`: A global variable that stores the hook ID used for managing interrupt subscription and unsubscription. It is essential for identifying the keyboard's interrupt requests.

The relative weight of the module in the project : 7%

Mouse Module

This module contains the functions developed in Lab 4 that could be reused for the project. These include the functions to subscribe, unsubscribe, and handle mouse interrupts, process mouse packets, and cursor movement.

Main data structures:

- `struct packet mpacket`: A structure representing a mouse packet containing information about mouse button states, movement, and overflow flags.
- `uint8_t byteIdx`: A variable tracking the index of the current byte being processed from the mouse.

- `uint8_t bytes[3]`: An array storing the three bytes of data received from the mouse, which together form a complete mouse packet.
- `uint8_t mouse_scancode`: A variable holding the most recent scancode received from the mouse.
- `int x, y`: Variables representing the current cursor position on the screen.
- `int mouse_hook_id`: A global variable storing the hook ID used for managing mouse interrupt subscription and unsubscription.

The relative weight of the module in the project : 7%

Video Card Module

This module contains the functions developed in Lab 5 that could be reused for the project. These include the functions to initialize the video graphics mode, manage the frame buffer, and draw pixels and images on the screen.

Main data structures:

- `uint8_t *video_mem`: A pointer to the frame buffer's virtual memory address, where pixel data is stored for display.
- `uint8_t *second_video_mem`: A pointer to a secondary buffer for double buffering, used to prevent flickering during screen updates.
- `unsigned h_res, v_res`: Variables storing the horizontal and vertical resolutions of the screen in pixels.
- `unsigned bytes_per_pixel`: Number of bytes required to represent a single pixel, calculated based on the bits per pixel of the video mode.
- `vbe_mode_info_t mode_info`: A structure containing information about the current video mode, retrieved using VBE functions.
- `struct minix_mem_range addr`: A structure defining a memory range for allowing memory mapping, used during initialization.
- `int r`: Variable used for error handling during system privilege control.

The relative weight of the module in the project : 9%

Sprite Module

The sprite module manages the creation, destruction, loading, and drawing of sprites used in the graphical user interface of the project. It provides functions to handle game sprites, buttons, cursors, and various pop-ups, facilitating the display of different elements on the screen during gameplay.

Main data structures:

- `Sprite`: A structure representing a sprite, containing information such as the sprite's image data, dimensions, position, and type.
- Various sprite pointers: Pointers to instances of Sprite representing different game elements such as menus, buttons, cursors, boats, numbers, and pop-ups.
- `enum xpm_image_type type`: An enumeration representing the type of XPM image being used.

The relative weight of the module in the project : 4%

RTC Module

The RTC (Real-Time Clock) Module provides functionalities for interacting with the system's real-time clock hardware. It includes functions for reading the current time from the RTC chip, converting the time format from binary-coded decimal (BCD) to binary, determining the current time of day and other related operations.

The relative weight of the module in the project : 3%

Utilities Module

This module contains the functions developed in Lab2 of the practical classes on the timer, which were subsequently used in the following Labs and also reused in our Project.

Main data structures:

- `int count`: An integer variable used to keep track of the number of times the util_sys_inb function has been called.

The relative weight of the module in the project : 1%

Serial Port Module

The Serial Port provides the connection needed to send and receive information allowing the players to play on different computers. It uses functions to subscribe, unsubscribe, send a byte and receive a byte, so the “instances” of the game can communicate with each other.

The relative weight of the module in the project : 3%

Model (13%)

Boat Module

This module manages the creation, manipulation, and rendering of boats in the game. It defines functions for creating boats, updating their positions, rotating them, fixing their positions on the game board, and rendering them. It also includes functions for loading boat sprites, checking for overlapping boats, and determining if all boats are fixed on the board. The module interacts with the game's matrix data structure to determine boat positions and overlaps.

Main data structures:

- `bool oneIsSelected`: Boolean flag indicating whether at least one boat is currently selected in the game.

The relative weight of the module in the project : 6%

Game Module

This module encompasses the core game logic and data management. It defines functions for saving player data, adjusting boat positions for display purposes, clearing the game board, creating the game board matrix, and updating damage inflicted on boats. Additionally, it includes structures representing players and their respective game boards.

Main data structures:

- `Spot matrix[MATRIX_SIZE][MATRIX_SIZE]`: Represents the game board matrix of spots.
- `Player p1`: Represents Player 1.
- `Player p2`: Represents Player 2.
- `Boat *boat_2p`: Pointer to Player 1's 2-pixel boat.
- `Boat *boat_3p1`: Pointer to Player 1's first 3-pixel boat.
- `Boat *boat_3p2`: Pointer to Player 1's second 3-pixel boat.
- `Boat *boat_4p`: Pointer to Player 1's 4-pixel boat.
- `Boat *boat_5p`: Pointer to Player 1's 5-pixel boat.

The relative weight of the module in the project : 7%

View (23%)

Asset Draw Module

This module handles the drawing of various assets onto the screen during gameplay, such as the game board, player names, cursor, crosshair, and end-game messages. It contains functions to draw different elements onto the screen using sprites loaded from external resources. It also includes logic to display the time elapsed during gameplay.

The relative weight of the module in the project : 3%

Game Local Draw Module

This module handles the drawing of various parts of the local game mode screen, such as the game board, player names, secret boards, cursor, crosshair, and time left. It contains functions that call others that draw different elements onto the screen using sprites loaded from external resources.

Main data structures:

- `bool cursor`: A variable that enables or disables the mouse cursor.
- `bool keyboard`: A variable that enables or disables the keyboard input.
- `bool changeTurn`: A variable that turns true if the 10 seconds of the player's turn comes to an end and makes switching turns by timeout possible.
- `int winner`: A variable that stores the player that wins the game.
- `enum GAMESTATE gamestate`: A variable that stores which player is playing in the current turn.

The relative weight of the module in the project : 7%

Game Remote Draw Module

This module is responsible for drawing the game remote onto the screen. It has 3 main states: placing the boats, the player's turn to play, and waiting for the play of the other player.

Main data structures:

- `bool boatsDrawn`: Boolean indicating whether the boats have been drawn.

The relative weight of the module in the project : 3%

Help Draw Module

This module is responsible for drawing the help menu onto the screen. It contains a single function that draws the help menu sprite onto the screen.

The relative weight of the module in the project : 1%

Menu Draw Module

This module handles the drawing of various elements related to the main menu of the game. It includes functions for drawing the main menu background, title, buttons, and cursor.

The relative weight of the module in the project : 4%

Placing Draw Module

This module handles the drawing of various elements related to the placing phase of the game. It includes functions for drawing the placing menu, the cursor, and the text associated with different states of the placing phase.

Main data structures:

- `enum PLACINGSTATE state`: Enum representing the current state of the ship placing phase.
- `enum PLACINGSTATE previousState`: Enum representing the previous state of the ship placing phase.
- `bool cursorActive`: Boolean indicating whether the cursor is active.
- `bool keyboardActive`: Boolean indicating whether the keyboard is active.
- `bool drawn`: Boolean indicating whether the boats have been drawn.

The relative weight of the module in the project : 5%

Handlers (24%)

Main Controller Module

This file serves as the main controller for managing user inputs and updating the game state accordingly.

Main data structures:

- `enum MODE menu`: Enum representing the current mode of the game.
- `uint8_t irq_mouse, irq_kbd, irq_timer`: Variables holding the IRQ lines for mouse, keyboard, and timer interrupts.
- `uint8_t scancode`: Variable storing the scancode received from the keyboard.

The relative weight of the module in the project : 5%

Game Local Controller Module

This file contains the controller functions specifically designed for the local game mode.

Main data structures:

- `int winner`: Variable storing the winner of the game.

The relative weight of the module in the project : 5%

Game Remote Controller Module

This file contains the controller function specifically designed for the remote game mode.

Main data structures:

- `bool cursorIsActive`: Variable used to store the state of the cursor.
- `bool keyboardIsActive`: Variable used to store the state of the keyboard.
- `int c_attack`: Variable used to store the column of the player's attack.
- `int l_attack`: Variable used to store the line of the player's attack.
- `enum THISPLAYER player_status`: Variable used to store the current state of the game.
- `bool second`: Variable used to check if this player is the first to play or not.

The relative weight of the module in the project : 3%

Help Controller Module

This file contains the controller functions specifically designed for the help menu.

The relative weight of the module in the project : 2%

Menu Controller Module

This file contains controller functions for handling user input in the main menu.

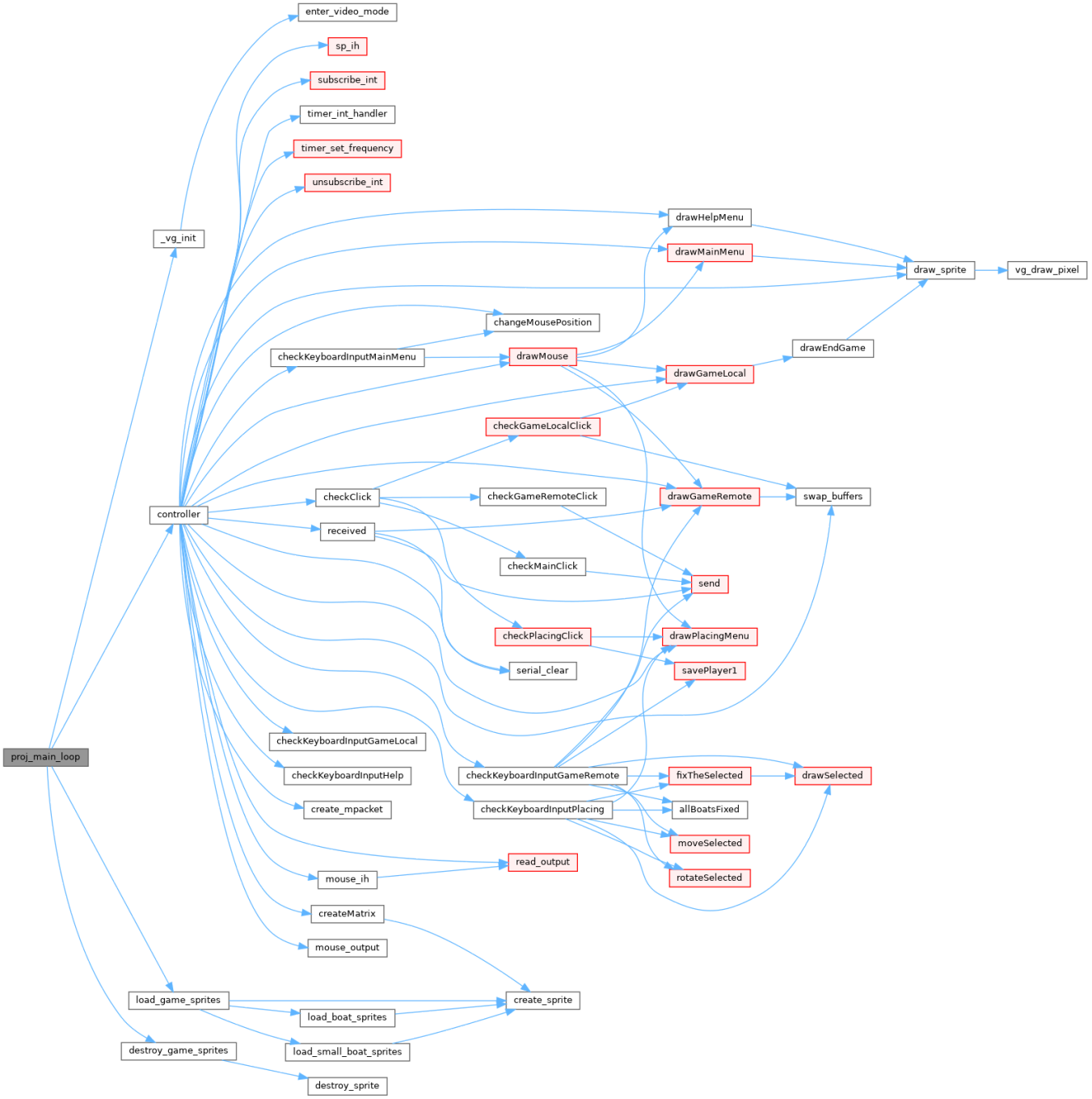
The relative weight of the module in the project : 3%

Placing Controller Module

This file contains controller functions for handling user input during the placing phase of the game.

The relative weight of the module in the project : 6%

Function Call Graph



5. Implementation Details

Structures

To simplify the creation of objects and their characteristics and sprites, we created three structures.

The Spot structure contains the normal and small variants of the hit and miss sprites, keeps track of the fixed pixel position, coordinates on the board, and occupied and attacked status.

The Boat structure contains all the variants between normal, small, selected, and rotated sprites and keeps track of the boat's ID, health, damage, height, width, first spot, fixed rotated, and selected status.

The Player Structure groups all of the player's boats and a matrix of his board's spots.

States

To represent different menu changes and game steps we implemented two state machines.

The Mode state is used throughout the program to determine where we are in the program life-cycle.

It contains the states:

- MAINMENU: The main menu of the game where the user can navigate to the help menu, start a new local game, or quit;
- HELP: A simple menu that explains the controls and objectives of the game;
- PLACING MENU: The mode where each user will select their boats;
- GAME LOCAL: Local mode for the game where the two players play on the same computer;
- GAME REMOTE: Remote mode where each player plays on their separate computer;
- QUIT: The final state of the program.

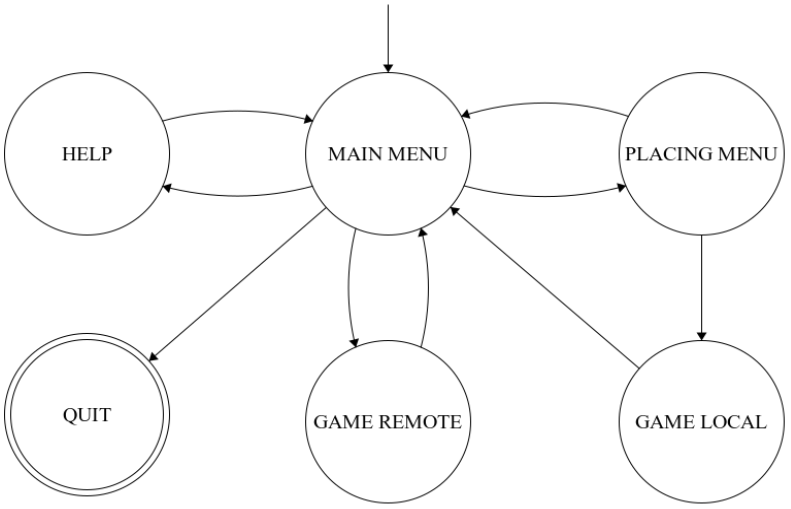


Fig. 8 - Mode state

The Placing state is used in the boat selection phase, it follows the natural progression of choosing the boats for player 1, then player 2, then finally advancing to the game. At any point, the user can quit.

It contains the states:

- PLAYER 1: State where player 1 places his boats;
- ASKING PLAYER 2: State where player 1 confirms his choice;
- PLAYER 2: State where player 2 places his boats;
- ASKING GAME: State where player 2 confirms his choice.
- ASKING QUIT: State that returns the program to the main menu.

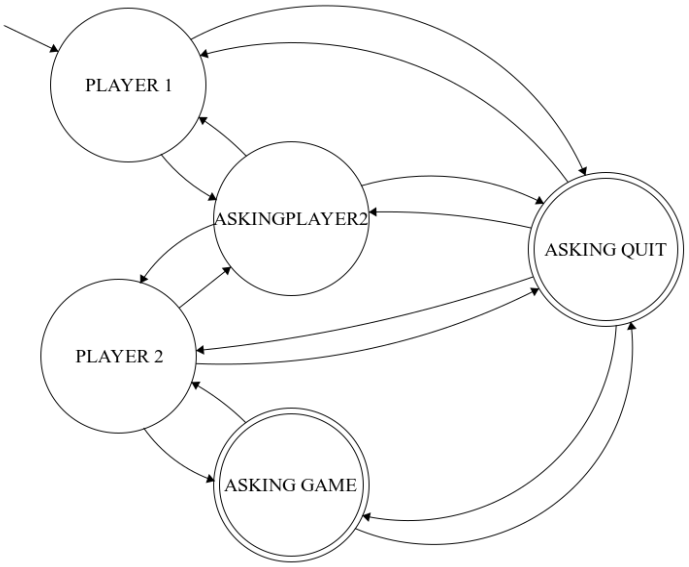


Fig. 9 - Placing state

The Local game state is used throughout the game to determine where we are in the game life-cycle.

It contains the states:

- TURN 1: State where player 1 attacks the opponent's boats;
- TURN 2: State where player 2 attacks the opponent's boats;
- END: State that returns the program to the main menu. The players can switch to this state by pressing ESC or by finishing the game.

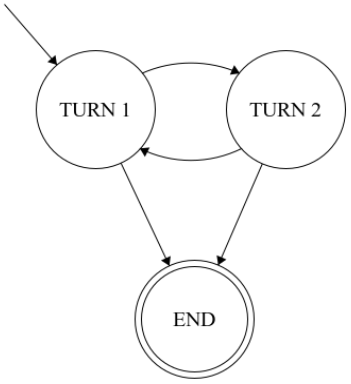


Fig. 10 - Local game state

The Remote game state is used throughout the game to determine where we are in the game life-cycle.

It contains the states:

- PLACING: The state where the player places their boats.
- CONNECTING: The state where the first player who places all boats stays until the other player ends placing the boats.
- WAITING: The state where a player waits for the attack of the other player.
- MY TURN: The state where a player attacks the other player's boats.
- EXIT: The state that returns the program to the main menu.

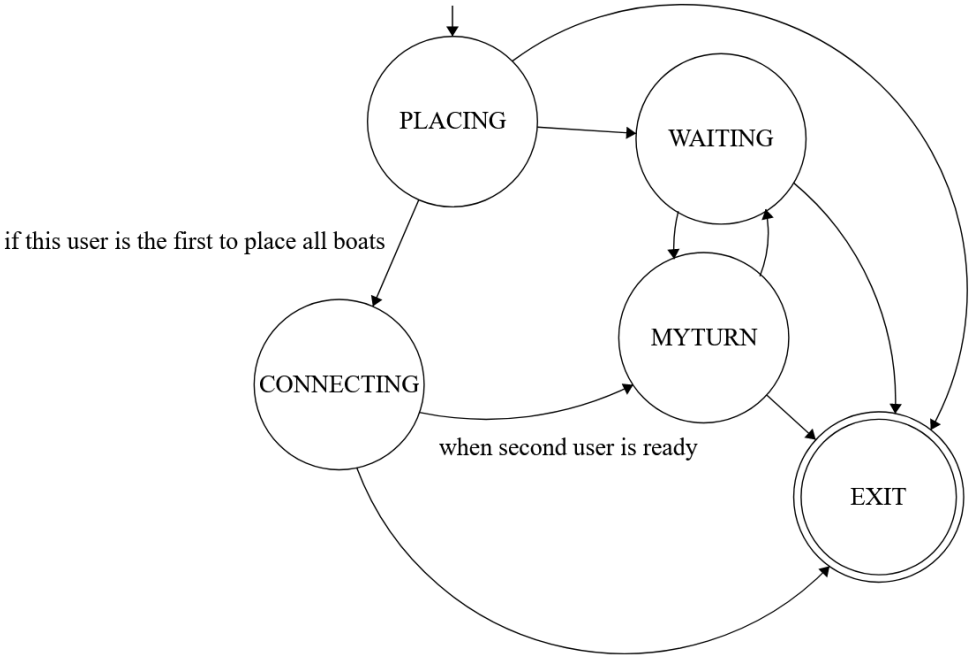


Fig. 11 - Remote game state

Board

To save the placed boats, hits, and misses, we created a matrix of spots.

The main board on the left side of the screen displays the board of the opponent. It features vacant spots, hits, misses, and sunken boats. We achieved this by analyzing the matrix at the start of every turn and handling each spot independently.

Small Board

While planning his next move, the player can also keep track of his board to assess the current damage. Since the game is played locally, both players are assumed to be looking at the screen at all times (except for boat placing menus), so the player's access to his board is limited by the hits and misses so as to not give advantage to the opponent.

We reutilized the logic from the normal board, applying it to different (smaller) sprites.

RTC - Main Menu

After we read the hour from the RTC, the main menu background changes between 3 different backgrounds, light, sunset, and dark. This background is always being reloaded, so when the hour changes the background changes accordingly.

Fig. 12 - Main Menu Backgrounds

Placing Boats

The game allows for the rotation and placement of the selected boat. After the user selects a position, every spot in the board is checked and the operation is not allowed in case of a collision. The boat remains selected and the user must find another arrangement.

Destroyed Boats

When a player hits every spot occupied by a certain boat, its position is revealed.

This was achieved by creating a damage and health attribute to the boat structure. Every time a hit occurs, the damage to the boat is increased until it reaches the health of the boat, at which point the sunken ship starts being displayed on the board.

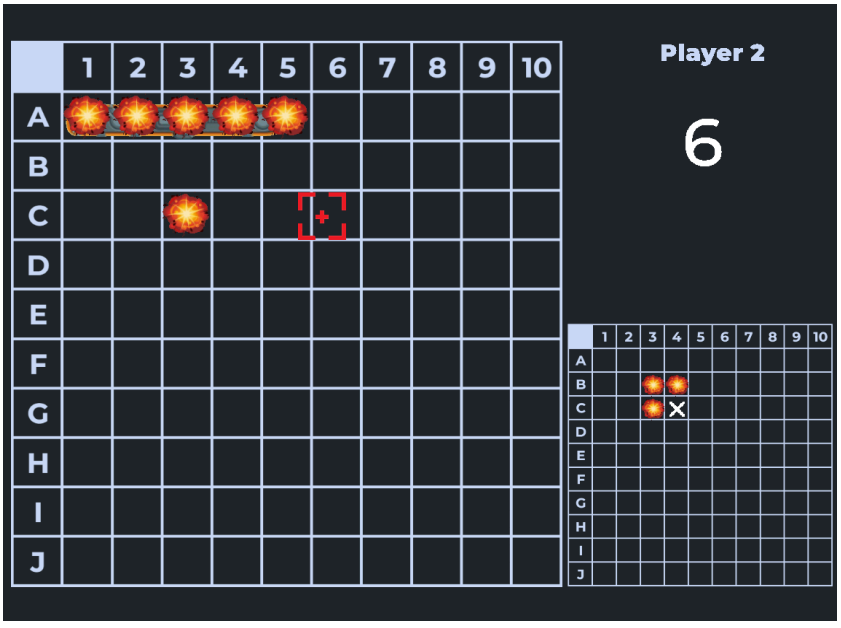


Fig. 13 - Game with a destroyed boat

Player Turn Countdown

To prevent the game from being stalled indefinitely until one of the players makes a move, we implemented a timer so that after 10 seconds from the start of the player's turn, it hands over the baton to the opponent. For this, we created a boolean changeTurn that becomes true when these 10 seconds pass, allowing the game to know the end of this time. Additionally, we created a draw function that displays the remaining time until the end of the current turn.

Accessibility Controls

To help navigate the buttons in each menu, the user can press keys that will take the mouse to the intended location:

- L for the local mode
- H for the help menu
- ESC for the quit button

Double Buffering

At a fixed rate of once every 5-timer interrupts, two frames are generated and loaded into two buffers. This Technique known as double buffering allows us to prevent visual artifacts and to enforce frame generation in certain moments, such as drawing the countdown timer or making the cursor smoother.

6. Conclusions

Setbacks

Initially, we set out to create one local and one remote. We managed to partially configure the serial port, initially it works correctly, but when the turn changes to the other player, the first player's Minix stops receiving interruptions from the serial port. And unfortunately, due to time constraints and difficulty in correcting this error, we decided to focus our efforts on other aspects of the project.

Next Steps

If we were to continue the development of the project, we would work on continuing to implement the remote functionality and to make it work smoothly.

Goals Achieved

Aside from the serial port, our group was able to achieve every project requirement. This includes the use of the timer, keyboard, mouse, video card, RTC, and double buffering.

What we learned

This project is part of one of the most difficult classes we have had until now, because it was the one that challenged us the most, from learning how to use gitlab to trying to connect two machines through the serial port, and to which we had to give up a lot of time. To put the project together, we needed to properly learn everything so it all worked out. Specifically, the project learned how to better work with MINIX, to better program in C and to develop low-level software. We also learned to work with the RTC and the Serial Port (even if not finished) which were challenging in their way.