

**­**

**­**

**Laboratório de Computadores**

#### Turma 12 Grupo 4:

* João Maria Correia Rebelo ([up202107209@up.pt](mailto:up202007658@fe.up.pt))
* Carlos Filipe Oliveira Sanches ([up202107694@up.pt](mailto:up20200@fe.up.pt))
* João Pedro Pereira Carvalho da Silva ([up202](mailto:up20200@fe.up.pt)208936[@up.pt](mailto:up20200@fe.up.pt))

# Index

[Index 2](#_Toc168245558)

[Image Index 3](#_Toc168245559)

[Introduction 4](#_Toc168245560)

[Instructions 5](#_Toc168245561)

[Menu 5](#_Toc168245562)

[Arena 6](#_Toc168245563)

[Movement 7](#_Toc168245564)

[Projectile 7](#_Toc168245565)

[Collisions 8](#_Toc168245566)

[Building 8](#_Toc168245567)

[Highscore 9](#_Toc168245568)

[End of game 9](#_Toc168245569)

[Project Status 10](#_Toc168245570)

[Table of features 10](#_Toc168245571)

[Device Table 10](#_Toc168245572)

[Devices 11](#_Toc168245573)

[Timer 11](#_Toc168245574)

[Keyboard 11](#_Toc168245575)

[Mouse 11](#_Toc168245576)

[RTC 11](#_Toc168245577)

[Video Graphics 11](#_Toc168245578)

[Serial Port 12](#_Toc168245579)

[Code Organization/Structure 13](#_Toc168245580)

[Models 13](#_Toc168245581)

[View 13](#_Toc168245582)

[Controller 14](#_Toc168245583)

[Dev\_interface 15](#_Toc168245584)

[Implementation details 16](#_Toc168245585)

[Conclusions 17](#_Toc168245586)

# Image Index

[Figure 1 - Atari Combat (1977) 4](#_Toc168247741)

[Figure 2 – Menu 5](#_Toc168247742)

[Figure 3 - Arena 6](#_Toc168247743)

[Figure 4 - Movement Instructions 7](#_Toc168247744)

[Figure 5 - Projectile 7](#_Toc168247745)

[Figure 6 - Bullet colliding with the player 8](#_Toc168247746)

[Figure 7 - Building 8](#_Toc168247747)

[Figure 8 - Highscore 9](#_Toc168247748)

[Figure 9 – Function Call Graph 10](#_Toc168247748)

# Introduction

Tank Wars, is a 2D game inspired by the classic Atari Combat, the game is based on the movement and firing of projectiles from a tank, combined with dynamic construction of obstacles in order to face an enemy tank. This new building mechanic allows for greater unpredictability in the combat arena scenario. The first player to hit the opponent wins, increasing their score.

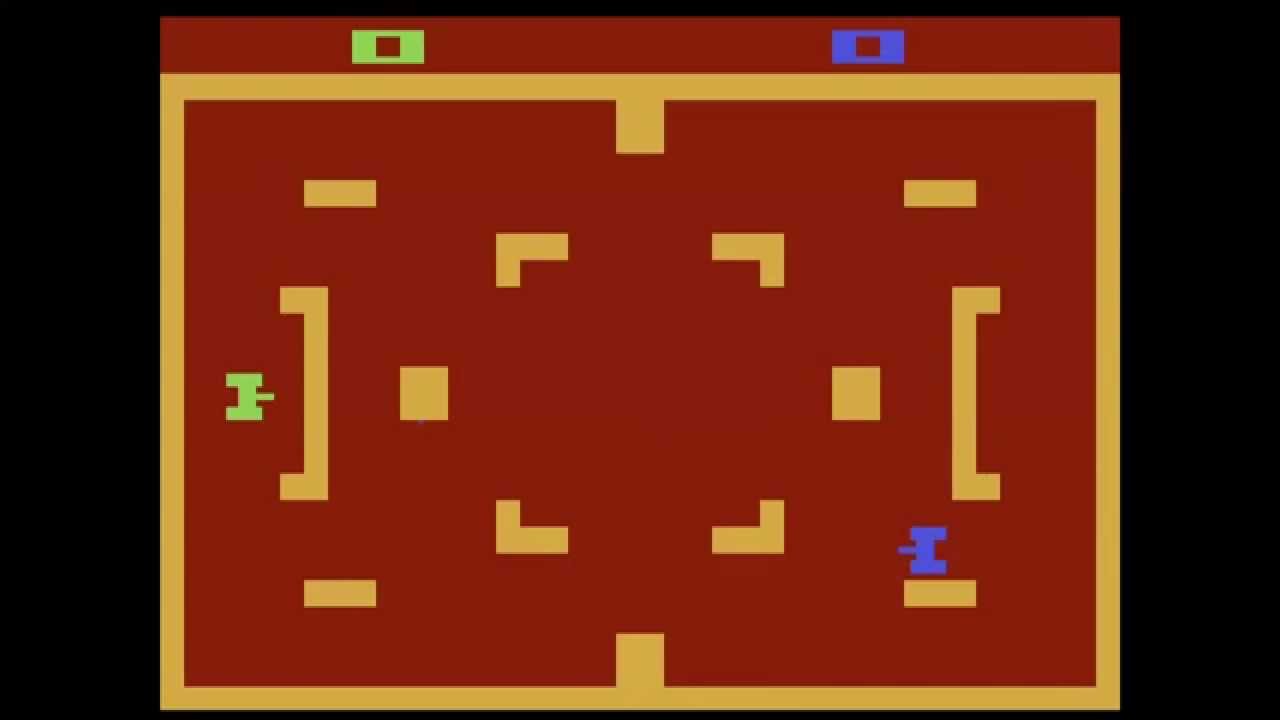


Figure 1 - Atari Combat (1977)

# Instructions

## Menu

When you start up, you can select the single player option or the multiplayer option. Each of these initializes a new game. If the player has a recorded highscore (he scores for each game he wins and accumulates points until he loses a game) the menu unlocks a highscore option that allows you to consult the record made and times the moment in which it occurred.



Figure 2 – Menu

In order to make the code modular, we use the MVC (model, view, controller) architecture. According to this structure, in the Models folder we have all the models of game elements, in the view folder we have the functions responsible for designing each element and in the controllers folder we have all the processing of the same elements and the logic inherent to each of them.

## Arena

The arena is the most important element of the game as it encapsulates all the information related to tanks, projectiles, obstacles, buildings, dynamics, score.

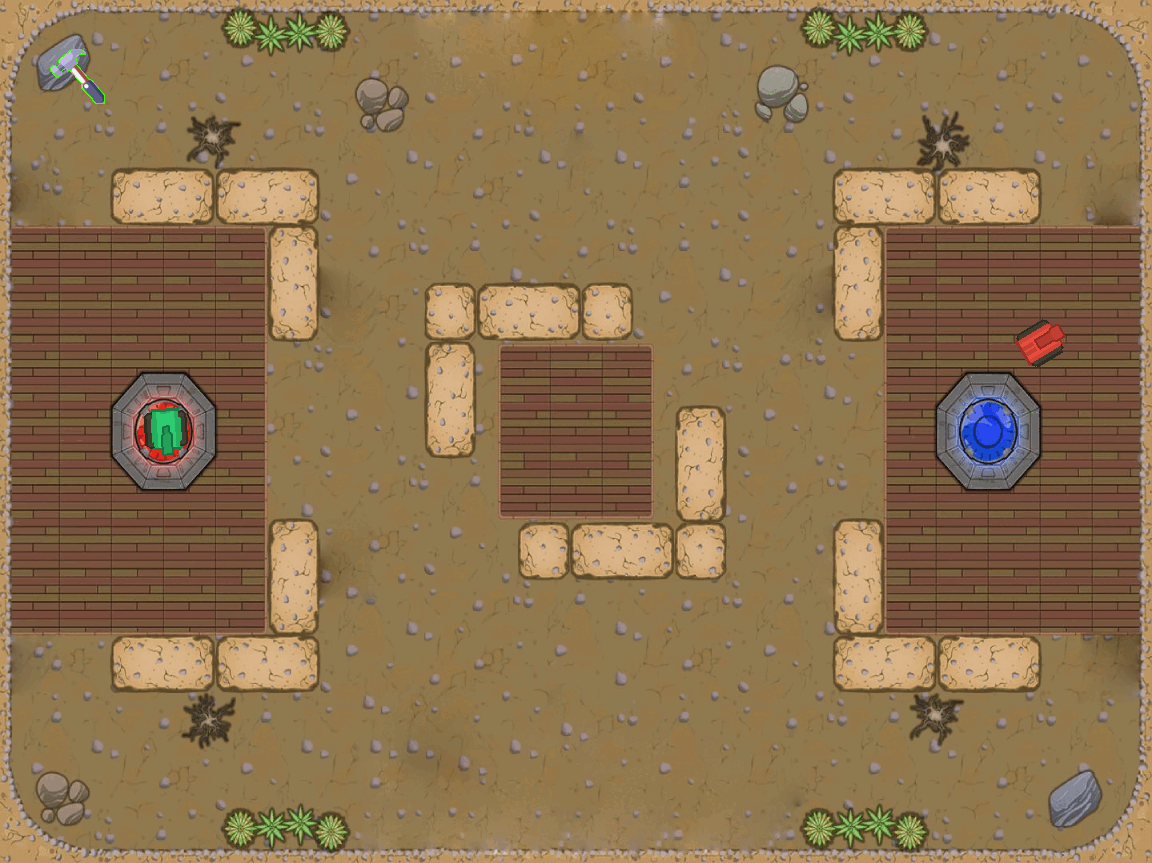


Figure 3 - Arena

## Movement

The player's movement is controlled by the keyboard that allows him to rotate on himself or walk back and forth, this allows the tank to explore the entire arena based on a position and an Angle. The enemy tank moves under a "patrol" logic, explores waypoints over time looking for the enemy tank, whenever a game is initialized, we add to a list of waypoints (already existing) more randomly generated waypoints, the idea is to always create a certain randomness in the enemy's movement in order to be “Smarter”.

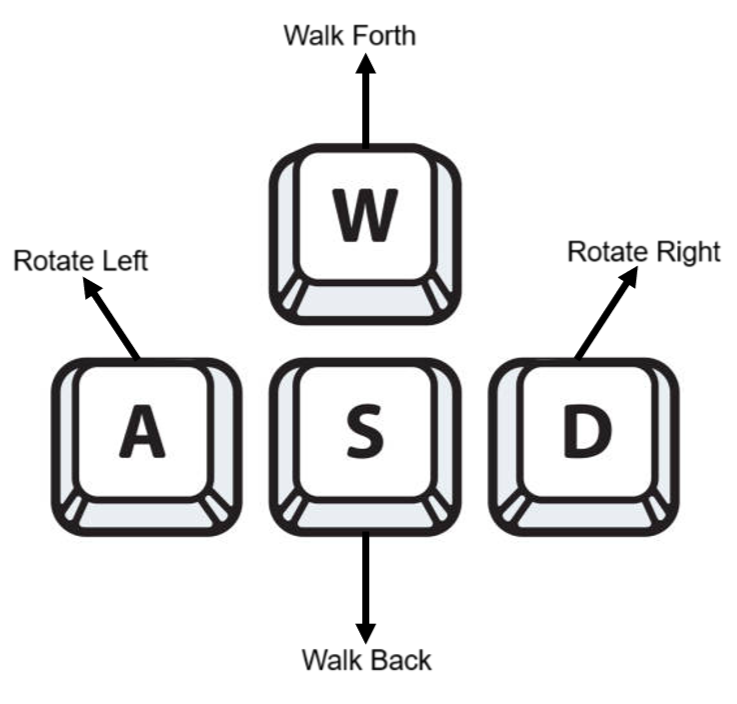


Figure 4 - Movement Instructions

## Projectile

In the implementation of the projectile, we apply the angle of rotation of the tank to the projectile so that it follows it. Each tank has a cooldown for firing.

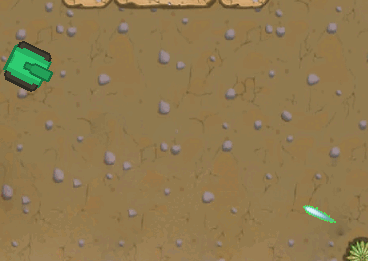


Figure 5 - Projectile

## Collisions

In order to detect collisions at each moment of arena processing, we process a dynamic hitbox for each element of the game, in order to understand if there is overlap between the elements, this translates into certain events in the game (bullet disappearing or until the end of the game).



Figure 6 - Bullet colliding with the player

## Building

The dynamic object building mechanic also contains a 10-second obstacle duration, the obstacles allow the user-controlled tank to apply different strategies both attack and defense throughout the game. The hammer cursor also indicates based on its color if the player can build (green) or not (red)



Figure 7 - Building

## Highscore

The highscore is recorded at the end of each game and is based on the current score (saved in the arena) which is incremented every time the user wins a game, and accumulates in consecutive wins, if the player is defeated this value is 0. If the current score is higher than the previously recorded highscore value, the value is updated, and this timed update can be consulted in the highscore menu (available in the home menu). We use a highscore.txt that stores the values. At the start of the game, we read the value and after each update we write to it.



Figure 8 - Highscore

## End of game

The game ends when one of the tanks hits its opponent and an "explosion" occurs. At the end of the game, the player's current score is updated and if it is higher than the recorded highscore, it will be replaced with a record of the hour, minute and second in which it occurred

# Project Status

## Table of features

|  |  |  |
| --- | --- | --- |
| Functionality | Devices | Implementation status |
| Movement | Keyboard e video graphics | Complete |
| Projectile | Keyboard e video graphics | Complete |
| Enemy | Video graphics | Complete |
| Highscore | Video graphics e RTC | Complete |
| Menu | Keyboard e video graphics | Complete |
| Building | Mouse e video graphics | Complete |
| Hitbox | Video graphics | Complete |
| Multiplayer | Serial port, timer | Incomplete |

## Device Table

|  |  |  |
| --- | --- | --- |
| Devices | Functionality | Interruptions |
| Timer | Updating and designing game and mouse elements and events and frame-rate | Yes |
| Keyboard | Used in menu navigation and tank movement | Yes |
| Mouse | Used in the menu and in building obstacles in the menu | Yes |
| Video Graphics | Draw each element of the menu and the game and in the rotation of the elements | No |
| RTC | Timed highscore record stored on highscore.txt | No |
| Serial Port | Used to send and receive arena status between players | Yes |

# Devices

## Timer

- We use timer 0 to generate periodic interruptions that allow certain operations to be performed in a regular and controlled manner, for example, when updating the mouse and game events.

## Keyboard

- The Keyboard is used both in-game to move the tank and in the menu to select options. Based on the makecode generated by each key.

## 

## Mouse

- Throughout the project, the mouse has been used based on its position on the screen and is essential to the mechanics of building obstacles in the game. The lb is also used in the menu to confirm the option. When you hover over the options, the menu reacts with a drawing that represents the option. The mouse was used in:

## RTC

- The RTC was used to read the time when the highscore was recorded, this is reflected in the highscore menu.

## Video Graphics

- We use a resolution of 1152x864 using vbe mode (0x14C) and direct color mode using 3 bytes for the color.

- Transform PNGs into XPM and load from xpm to the screen for all game elements and static structures.

- The project uses triple buffering in order to make the screen refresh quickly. We use the third buffer for background changes, the second buffer to prepare the frames, and the first to refresh the screen.

## Serial Port

- We didn’t fully implement the mutiplayer mode but we have the developed the code for serial port on the dev\_interface folder and for the serial port controller. We are using a FIFO buffer, which we use for the handshake between the players. We intended to use the serial port to send over the Arena between users, since the arena encapsulates all the game events and data by passing it, we could update both users. We would need to develop a way to serialize the arena data so that it could fill the buffer size. (we took inspiration for the fifo buffer implementation from here: <https://blog.stratifylabs.dev/device/2013-10-02-A-FIFO-Buffer-Implementation/>).

# Code Organization/Structure

- We have the relative weight of each .c file, it was fun to conclude that the MVC architecture was really followed because the weight of the controllers is way higher than the others which makes sense taking into consideration it encapsulates most of the logic.

## Models

-This folder is responsible for all the models of the elements of the game

Files:

* arena.c (1%)

This file is responsible for defining and managing the game arena. It includes creating the arena and placing elements within it.

* bullet.c (1%)

This file handles the creation, movement, and collision of projectiles fired by tanks. It defines the trajectory and behavior of the bullets in the game.

* tank.c (1%)

This file contains the modeling of the tanks.

* menu.c (1%)

This file manages the game's main menu, including available options, navigation between them, and the logic to select and start the game.

* obstacle.c (1%)

This file models the dynamic obstacles built within the arena.

## View

- This folder is responsible for all the functions and logic concerning drawing elements

File:

* views.c (4%)

This file is responsible for all functions and logic related to rendering elements on the screen. It draws the arena, tanks, projectiles, obstacles, and menus, ensuring everything is correctly presented to the player.

## Controller

- This folder is responsible for processing all the element’s interactions and events

Files:

* arenaController.c (1%)

This file manages interactions and events within the arena, including updates and state changes of elements during the game.

* buiderController.c (1%)

This file handles the dynamic construction of obstacles during the game, using the mouse to place new elements in the arena.

* collisionController.c (1%)

This file is responsible for detecting and handling collisions between tanks, projectiles, and obstacles, updating the game state as necessary.

* bulletController.c (1%)

This file controls the behavior of projectiles, from firing to impact, managing their trajectory.

* enemyController.c (5%)

This file manages the behavior of enemy tanks, including movement, attacks, and “smart” movement.

* gameController.c (1%)

This file oversees the overall game state, coordinating the interaction between all components and ensuring the game progresses coherently.

* tankController.c (2%)

This file controls tank actions, including movement, firing, and interaction with the environment and other tanks.

* timerController.c (1%)

This file manages time-based events, ensuring actions and updates occur at regular intervals.

* kbController.c (1%)

This file handles keyboard input, processing player commands to control the tank and navigate menus.

* mouseController.c (2%)

This file deals with mouse inputs, including building obstacles and selecting options in the menu.

* rtcController.c (1%)

This file manages the real-time clock, updating the highscore chrono.

* serialPortController.c (1%)

This file manages communication through the serial port, facilitating multiplayer interaction and other functions dependent on serial communication. (not 100% implemented).

## Dev\_interface

- This folder is responsible for all the code we have developed for the labs across the semester

Files:

* timer.c (2%)

This file implements the timer used to generate periodic interrupts, fundamental for synchronizing game actions.

* kbc.c (1%)

This file manages the keyboard controller, capturing and processing keystrokes to control the game.

* mouse.c (2%)

This file implements the mouse interface, including detecting movements and clicks for game interactions.

* vídeo\_gr.c (7%)

This file handles the initialization and management of the game's graphics mode, ensuring all visual elements are correctly displayed.

* rtc.c (1%)

This file manages the system's real-time clock, synchronizing time-based events in the game.

* serial\_port.c (2%)

This file implements communication through the serial port, essential for multiplayer mode and other functionalities that require data exchange between devices.

Uma imagem com captura de ecrã, texto, file, diagrama

Descrição gerada automaticamente

# Implementation details

-In the development of Tank Wars, we applied several key concepts from our coursework with some creative adaptations. Layering was essential for the game's architecture, ensuring a clear separation of concerns between the model, view, and controller components. This MVC architecture facilitated modular development and easier debugging. Event-driven programming was crucial, particularly in handling user inputs from the keyboard and mouse, as well as managing periodic updates via timer interrupts.

-The Real-Time Clock (RTC was significant in our project. The RTC was employed to timestamp high score records accurately, adding a real-time element to the game stored in the highscore.txt. This required understanding how to interface with the RTC and retrieve current time values. The UART was intended for multiplayer functionality, allowing arena status to be sent and received between players. Although we did not fully implement the multiplayer mode, we developed the code for serial communication using a FIFO buffer. This required learning about data serialization and managing buffer sizes to ensure efficient communication.

-Additionally, we ventured into topics not covered in lectures, such as collision detection and dynamic hitboxes. Collision detection was vital for determining interactions between tanks, projectiles, and obstacles. We developed a system to calculate and update hitboxes dynamically, ensuring accurate collision responses. This involved learning new algorithms and techniques to handle the physics of collisions effectively. We have also spent some time developing the arithmetic for the movement and projectile logic based on trigonometry.

# Conclusions

-The fundamental lesson we took from this project is that developing software requires a good amount of modularity and code organization. In the beginning of the project, we spent some time developing a good code architecture that led to an easier long-term implementation of features.

-We consider C to be a less syntax heavy language making the project more about focusing on the logic itself rather than language conventions and paradigms of programming.

-We didn´t fully implement the serial port which makes us sad about it, because it would take our project to another level. We took more time polishing other features than focusing on this one which led to a certain delay.