LabWars

Final report

LCOM 2019/20

T5G03

up201806429@fe.up.pt Diogo Miguel Ferreira Rodrigues
up201806554@fe.up.pt Telmo Alexandre Espirito Santo Baptista

06/01/2020

Table of contents

1 User instructions **3**
- 1.1 How to play . . . 3
- 1.2 Main menu . . . 3
- 1.3 Single player . . . 4
  - 1.3.1 Campaign . . . 5
  - 1.3.2 Zombies . . . 6
- 1.4 Multiplayer . . . 7
- 1.5 Chat . . . 8

2 Project status **10**
- 2.1 Timer . . . 10
- 2.2 Keyboard . . . 10
- 2.3 Mouse . . . 10
- 2.4 Video card . . . 11
- 2.5 Real-Time Clock (RTC) . . . 11
- 2.6 Serial port . . . 11

3 Code organization/structure **12**
- 3.1 `libs` . . . 12
  - 3.1.1 `classes` . . . 12
  - 3.1.2 `graph` . . . 12
  - 3.1.3 `uart` . . . 12
  - 3.1.4 `utils` . . . 12
  - 3.1.5 others . . . 12
- 3.2 `proj` . . . 12
  - 3.2.1 `campaign` . . . 12
  - 3.2.2 `chat` . . . 12
  - 3.2.3 `ent` . . . 12
  - 3.2.4 `hltp` . . . 12
  - 3.2.5 `proj_func` . . . 13
  - 3.2.6 others . . . 13

4 Implementation details **14**
- 4.1 Object-oriented programming . . . 14
- 4.2 XPM and XPM2 . . . 14
- 4.3 Communication protocols . . . 14
  - 4.3.1 Non-Critical Transmission Protocol . . . 14
  - 4.3.2 High-Level Transmission Protocol . . . 14
- 4.4 Path-finding . . . 15

1 User instructions

1.1 How to play

In all game modes, the controls are the same:

- **WASD** to move North, East, South and West respectively.
- **Mouse left-click** to fire a bullet.
- **Ctrl+'+'** and **Ctrl+'-'** to zoom in and out.
- **ESC** to escape game mode (go back).

1.2 Main menu

On startup, users are greeted by a `Loading...` message, briefly followed by the main screen.

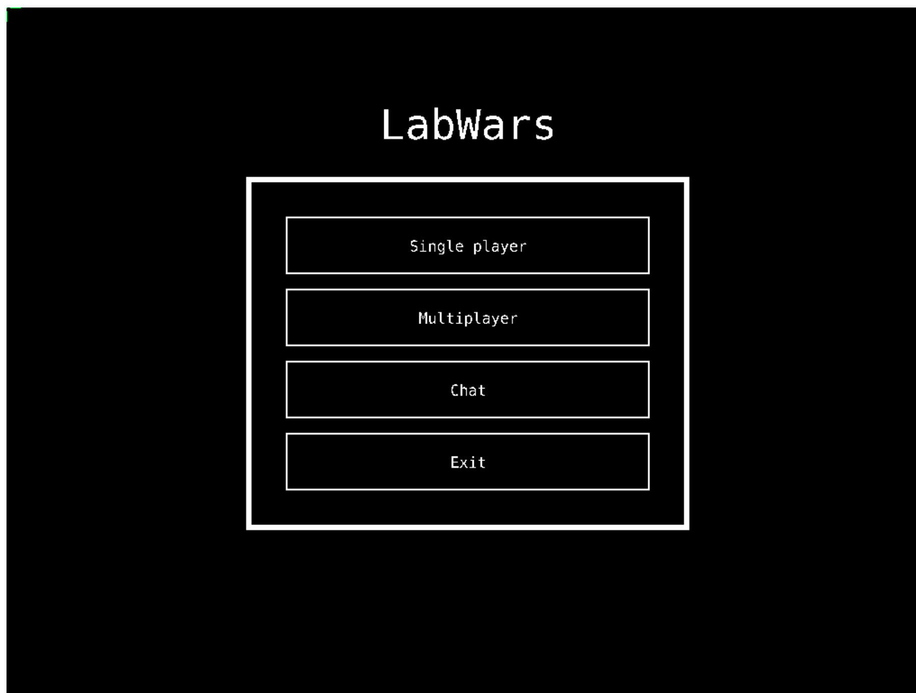


Figure 1: Main menu

Using the mouse movement and clicks, the user can select one of the avaliable options:

- **Single player**: Go to single player selection menu, to select one of the single player game modes.
- **Multiplayer**: Go to multiplayer mode, allowing to select more options.
- **Chat**: Exchange text messages with another connected computer.
- **Exit**: Exit the game

The user can also exit the game by presing **ESC**.

1.3 Single player

Upon entering into single player mode, the user is presented with a menu from which he can choose one of the options.

Figure 2: Single player menu

- **Campaign**: campaign mode; kill all autonomous opponents.
- **Zombies**: zombies mode; kill as many zombies and survive as much time as possible.
- **Zombies Ranking**: Scoreboard with the highest scores obtained on zombies mode.
- **Back**: go back to main menu.

The user can also go back to main menu by pressing **ESC**.

1.3.1 Campaign

In campaign mode the goal is to kill all the opponents in the map as fast as possible, while sustaining as little damage as possible.

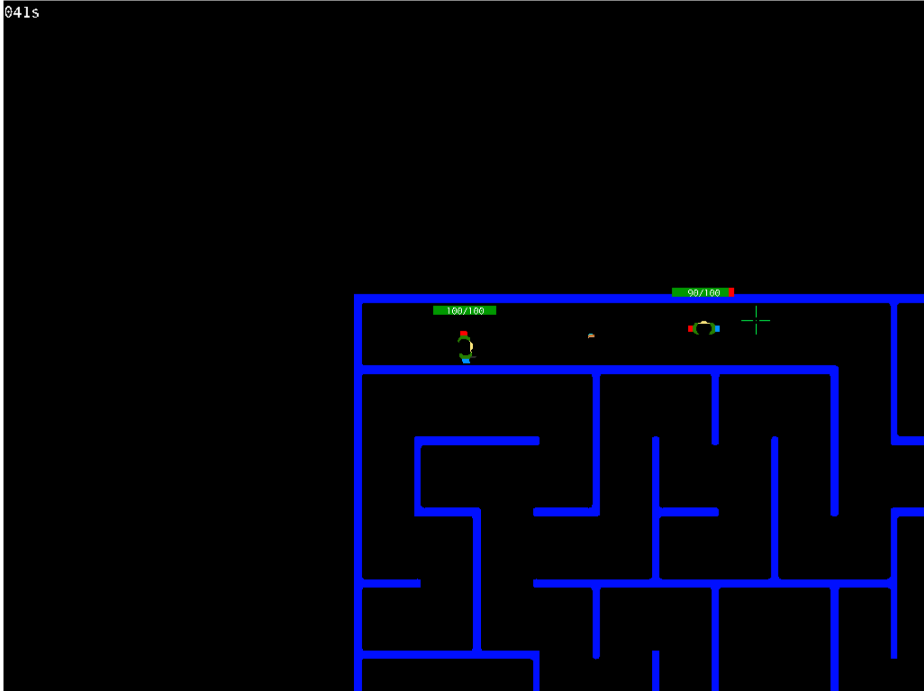


Figure 3: Campaign mode

1.3.2 Zombies

In zombie mode the goal is to kill as many zombies and survive as much time as possible. Zombies slowly follow the player and attack the player when in short range. Once the player kills a zombie, a new zombie spawns in a random part of the map, with more life than all previous zombies.

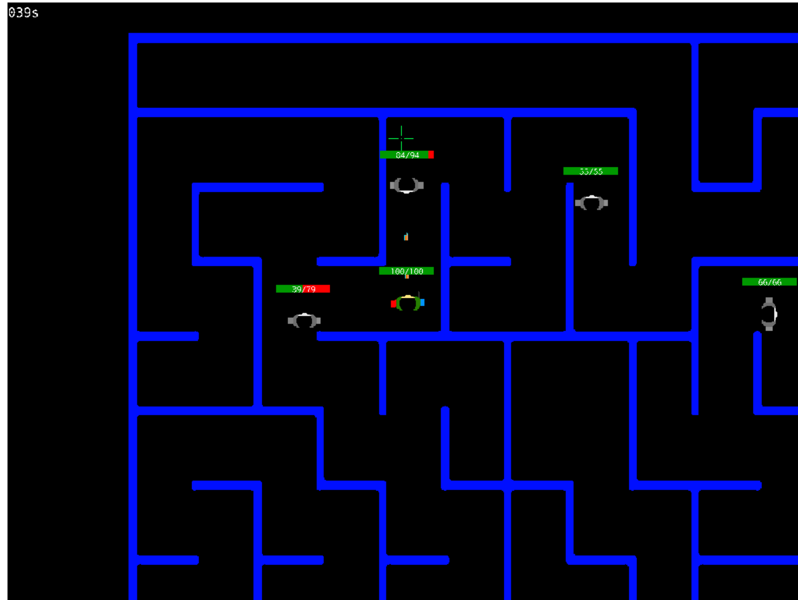


Figure 4: Zombies mode

After exiting zombie mode, the player can check the scoreboard by selecting **Zombies Ranking**.

Date	Score	Time Played
06/01/20 18:01:51	8	45
06/01/20 15:01:27	6	38
06/01/20 18:01:59	3	18

Figure 5: Zombies mode scoreboard

1.4 Multiplayer

In multiplayer mode, INCOMPLETE.

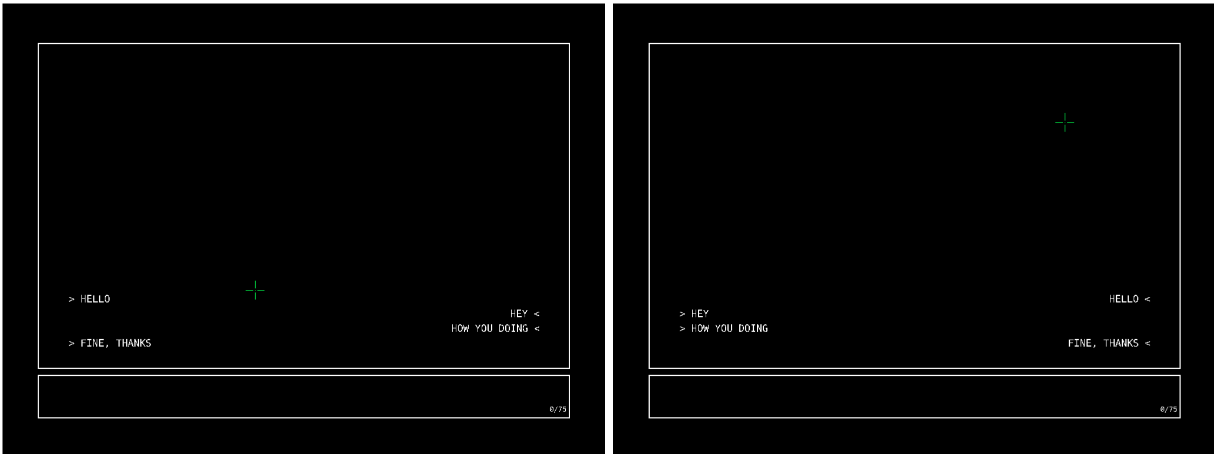
1.5 Chat

This chat tool was initially designed as a simple, text mode, test communication between different machines. We have however decided to include it as a functionality in the project for a number of reasons:

1. It was easy to develop the graphical part and integrate in the project.
2. Having a friendly functionality that uses the communication modules allows for faster debugging; in case the computers are not properly connected, or if during development something stops working we can immediately check if the communication modules also stopped working.
3. It served as a minimal insurance that our project would integrate the communication modules, in case we could not implement multiplayer mode.
4. It is a useful feature.

Figure 6: Chat environment

The chat can be used for exchanging messages of up to 75 characters directly writable with the keyboard. The character limit was imposed to prevent strings from rendering as wider than the input box, and the fact they should be directly writable with the keyboard simplifies the process of capturing scancodes, having as downside not allowing to write characters that require more than one key press (like exclamation or question marks in a Portuguese keyboard).



(a) Computer 1 chat

(b) Computer 2 chat

Figure 7: Two users interacting via chat

The user can exit the chat mode by pressing **ESC**.

2 Project status

All functionalities previously presented were fully implemented, with the exception of:

- **Campaign**: autonomous opponents were supposed to follow a pre-programmed path and shoot on sight at the player. Currently, they don't do either of those.
- **Multiplayer**: still working on it.

The I/O devices used in the project are presented in the following table.

Device	**What for**	Method
Timer	Frame rate, time since beginning of game	Interrupts
Keyboard	Player movement, writing chat messages	Interrupts
Mouse	Player orientation, shooting, selecting options in menus	Interrupts
Video card	In-game drawing, menus	None
RTC	Scoreboards	Interrupts
Serial port	Chat communication, multiplayer modes	Interrupts

To manage all interrupt subscriptions, the general function `unsubscribe_interrupt` was implemented and used.

2.1 Timer

Timer 0 is used to generate periodic interrupts at a rate of 60Hz, essentially controlling a large part of what the program does.

Timer interrupts regulate screen refreshing, which happens at a rate of 60Hz. In all game modes, timer interrupts serve not only the purpose of refreshing the screen, but also to process all the game data: collisions, movement, path-finding algorithms, etc.

To manage timer interrupt subscriptions, functions `subscribe_timer_interrupt`, `timer_int_handler` and `timer_get_no_interrupts` were implemented and used.

2.2 Keyboard

The keyboard was configured to issue interrupts on key presses and releases.

The keyboard is used to control some menus (namely using **ESC**), to input text in the chat, and to control player movement in the game and zoom in/out options.

To manage keyboard interrupt subscription, function `subscribe_kbc_interrupt` was implemented. To manage keyboard interrupts, functions `kbc_ih`, `keyboard_get_done`, `keyboard_get_scancode` and `update_key_presses` were implemented and used.

2.3 Mouse

The mouse was configured to issue interrupts on movement and button presses.

The mouse is used to control all menus (position and buttons), as well as allowing the player to aim at the opponents (position) and shoot bullets (buttons).

To manage mouse interrupt subscription, function `subscribe_mouse_interrupt` was implemented. To manage mouse interrupts, functions `mouse_ih`, `mouse_parse_packet` and `update_mouse`, among others, were implemented.

2.4 Video card

The video card was configured to work in graphic mode with direct color encoding, with resolution 1024x768 pixels and 8 bits for each color component Red, Green and Blue, yielding a total of approximately 16.8 million colors.

Simple buffering was used to eliminate flickering. All graphics are first drawn to the scree buffer, and only after all graphical operations (that is, at the end of the processing of the timer interrupt) is the buffer information copied to the VRAM.

The modules `rectangle_t` and `menu_t` were developed for displaying simple shapes and menus. The modules `basic_sprite_t` and `sprite_t` were developed for displaying moving sprites, allowing for rotation around the "center" of the image, as well as scaling and movement.

The modules `font_t` and `text_t` were developed to allow for dynamic rendering of text, using as default (and, for now, sole) font Consolas. The only type of supported fonts are bitmap fonts.

Configuration of the UART was made using the functions `graph_init` and `graph_cleanup` developed during the project. Drawing to the buffer is made primarily through the functions `graph_get_XRes`, `graph_get_YRes`, `graph_set_pixel`, `graph_clear_screen` and `graph_draw`.

Basic sprites are constructed from small XPM files, or loaded at runtime from XPM2 files. A XPM2 file is a XPM file stripped from all the C syntax, leaving only the strings. This format is easier to load at runtime than XPM. A XPM file can be easily converted to a XPM2 file using the function `xpm_save_as_xpm2`. All XPM arrays of strings are loaded to bitmaps using the function `xpm_load` provided by the LCF.

2.5 Real-Time Clock (RTC)

The RTC was configured to issue interrupts and if the interrupt source is an Update Event, in which the bit 4 of Register C will be set, then the values of time will be updated by reading their registers.

The date isn't updated unless asked, this is, on the interrupt notification the date isn't updated, and the reading process is different from the time. If it's asked to read the date then the date registers will be read two consecutive times, and will repeat the process if the date values of consecutive readings isn't equal, this ensures the values read are correct in case any update occurs. These updates aren't as frequent as the time values so there's no need to update it every second.

To manage RTC interrupt subscription, function `subscribe_rtc_interrupt` was implemented. In order to enable the Update Event Interrupts, the function `rtc_set_updates_int` that enables or disables the Update Events interrupts by writing to bit 4 of register B. To manage mouse interrupts, function `rtc_ih` was implemented, that verifies if the source if from an Update Event by reading register C and verifying bit 4 and if so updates the time values (seconds, minutes and hours). To ease the process of reading and writing values to the RTC, functions `rtc_read_register` and `rtc_write_register` were implemented as general functions, while having specific functions for reading the date and time values, such as `rtc_read_min` among others, that use the general functions cited above.

The RTC is used to obtain date and time for the scoreboards on the game-modes.

2.6 Serial port

The UART was configured to issue interrupts for Receiver Ready and Transmitter Empty. Communication is processed with the same parameters at both ends, at a bit-rate of 9600bps, 8 bits per char, 2 stop bits. UART FIFOs are used, with trigger level of 4 bytes per interrupt. The protocols that were developed will be discussed in section 4.

In multiplayer mode, data is transferred from host to remote at a frequency of 60Hz, with each message having at least 24 bytes. Data transference from remote to host is made whenever needed, with each message having on average 9 bytes.

3 Code organization/structure

3.1 `libs`

Collection of useful classes and functions. This module was developed with the goal of being as general and independent from `proj` as possible, although there are rare occasions where sub-modules of `libs` include sub-modules of `proj`.

3.1.1 `classes`

Provides classes `list_t` and `queue_t`. These classes achieve generality by storing pointers to `void`, which have as major disadvantage requiring the use of `free`.

3.1.2 `graph`

Provides basic elements for screen drawing, like drawing pixels, rectangles, text. Manages screen buffering and VRAM.

3.1.3 `uart`

Provides basic functions and the NCTP protocol for communication between computers.

3.1.4 `utils`

Common functions, as well as functions for handling XPMs and XPM2 file format.

3.1.5 others

- `kbc`: Manage mouse and keyboard
- `rtc`: Manage RTC, get current time
- `timer`: Manage timer 0 interrupts and subscriptions

3.2 `proj`

3.2.1 `campaign`

Campaign mode module.

3.2.2 `chat`

Exchange messages with connected computer.

3.2.3 `ent`

One of the most important modules. Implements the most important entities, and controls their interactions.

3.2.4 `hltp`

Provides function to interpret and send information for the serial port.

3.2.5 `proj_func`

Functions related to the game dynamics, such as updating the game state, updating movement, keys presses, game timer and building the information for serial port.

3.2.6 others

- `interrupts_func`: group some interrupt handling functions together
- `makecode_map`: map makecodes to chars
- `proj_macros`: macros used throughout the project
- `proj_structures`: mostly structures for transmission via HLTP
- `singplayer`: simple menu

		Weight	Resp.	Contribution DR	Contribution TB
libs	classes	13%	DR	Everything	-
	graph	10%	DR	Most part	Some contributions
	kbc	2%	TB	In labs	In labs, adapted to project
	rtc	4%	TB	-	Everything
	timer	2%	TB	In labs	In labs, adapted to project
	uart	16%	DR	Everything	-
	utils	3%	DR	In labs, most part	In labs, some contributions
proj	campaign	3%	DR	Most part	Some contributions
	chat	7%	DR	Everything	-
	ent	10%	DR	Everything except TB	`bullet_t`
	hltp	4%	DR	Most part	Some contributions
	interrupts_func	3%	TB	Small contribution	Most part
	makecode_map	1%	TB	-	Everything
	proj_func	9%	TB	1/2	1/2
	proj_macros	1%	TB	-	Everything
	proj_structures	1%	TB	-	Everything
	scoreboards	2%	TB	-	Everything
	singleplayer	1%	DR	Everything	-
	zombies	8%	DR	Everything	-

4 Implementation details

4.1 Object-oriented programming

Object-oriented programming was implemented to its greatest extent possible. Classes were declared using the `typedef struct` expression, and their public methods declared in the corresponding header file, always requiring as first argument a pointer/const pointer to that class. Classes were defined in the corresponding source file, and private member functions were defined as `static`.

The majority of entities were encapsulated in classes.

4.2 XPM and XPM2

The extensive use of large XPMs by simply including them with an `#include` directive gives rise to large executable files, besides making it harder to change the used XPMs without recompiling the project.

The XPM2 file format is similar to XPM, except it is stripped from all the C syntax, making it a plain text file. This file format has the main advantage of being easy to load on runtime, unlike XPM that would require extensive parsing.

To use the XPM2 file format, two functions `xpm_save_to_xpm2` and `xpm_load_xpm2` were implemented; the first one to convert XPM files to XPM2 files, and the second one was used in the project to load the XPM data (as an array of C-strings) from the XPM2 file format.

4.3 Communication protocols

4.3.1 Non-Critical Transmission Protocol

NCTP was designed to encapsulate the basic UART functions that were developed, by providing a set of middle-level functions that allow communication between computers.

NCTP allows to send a message to another computer through the function `nctp_send`, which sends the contents of a set of memory blocks as a single message. The first argument is the number of memory blocks, followed by an array of pointers to void denoting the beginning of each memory block, and an array of sizes, where `sz[i]` is the number of bytes of the block starting at `ptr[i]` that should be sent. On receiving, NCTP calls a user-provided function to interpret the message.

NCTP uses queues on transmission and receiving. When a Tx interrupt occurs, NCTP tries to send as many chars of the transmission queue as possible, and on a Rx interrupts NCTP tries to extract from the receiving register as many chars as possible, putting them in the receiving queue.

When a complete message is detected, the chars of the message are popped out of the receiving queue, and passed to the user-provided function for interpretation.

A message is composed of a header, a body and trailling filler chars. The header is a pair of bytes denoting the size of the body. Then comes the body, and finally the trailling filler chars, whose purpose is to make sure the total size of the message is a multiple of 4, to prevent problems like a number of final message chars smaller than the trigger level causing a timeout.

4.3.2 High-Level Transmission Protocol

HLTP fills the gap left by NCTP, by providing a way to interpret messages. For that, HLTP keeps an enumeration of all the data types it knows, and for each deta type a pair of functions for coding and decoding a message containing that data type.

HLTP inserts at the beginning of the body a byte indicating the data type, so the message can then be decoded on the other computer.

4.4 Path-finding

The implementation of the following behaviour of zombies was made using a modified version of Dijkstra's path-finding algorithm. Because running a regular Dijkstra every frame causes tremendous lag, a modification was made; for each pixel, it is known the zombie should go to another pixel. At the beginning, a complete Dijkstra is ran, and each time the position of the player changes Dijkstra is only ran in the vicinity of the player.

This solution does not always provide the shortest path, but is significantly more efficient and returns a working path to the player.