Faculdade de Engenharia da Universidade do Porto
Department of Informatics Engineering

LCOM: Last Stand

Computer Laboratory Report

Authors
Gonçalo Alves
Nuno Costa

January 4, 2021

Contents

Game Concept

LCOM: Last Stand is a 2D Third-Person shooter type game. Its main purpose is to destroy the incoming enemy blasts using the character's own magic blasts and not get hit by one. The number of magic blasts the character has available is limited, so the player must use them wisely. Each enemy hit grants a point for the total in-game score. Enemies come at increasingly higher rates, called 'waves', making it more difficult for the player to stay alive and continue the game.

The game has the option for singleplayer or co-op modes.

In singleplayer mode, the game is run as stated above.

In co-op mode, the number of available charges is shared, but whenever one of the players dies, the game is over. As such, both players must be careful to not only avoid colliding with an enemy blast, but to also make sure that the partner does not get hit as well.

User Instructions

2.1 Initial screen

At the start of the game, the user is presented with the following screen:

Figure 2.1: The main menu

- The **START** option initiates the game
- The **INSTUCTIONS** option presents the user with the game's intructions
- The **CO-OP** option goes to a 2-player game
- The **EXIT** option quits the program

The player must use the mouse to choose either of the presented options, choosing one with its left button.

2.2 Playing Screen

Figure 2.2: In-game/Playing screen

The playing screen displays the player's character, the score, the available magic blasts (charges), the enemies and the current enemy wave.

The user controls the character with the A or D keys and shoots a blast with the mouse's left button click.

The game ends, as stated previously, when the character gets hit by a projectile.

2.3 Game Over

Figure 2.3: Game Over Screen

Upon getting hit, the Game Over screen is presented to the player. Any key press will return the player's screen to the main menu.

2.4 Instructions Screen

Figure 2.4: Instructions Screen

The instructions screen displays the game's instructions, and the current date and time. To exit, the user can press any key.

2.5 CO-OP

Figure 2.5: Co-op Playing screen Screen

The initial CO-OP option presents the user with a waiting screen, initially.

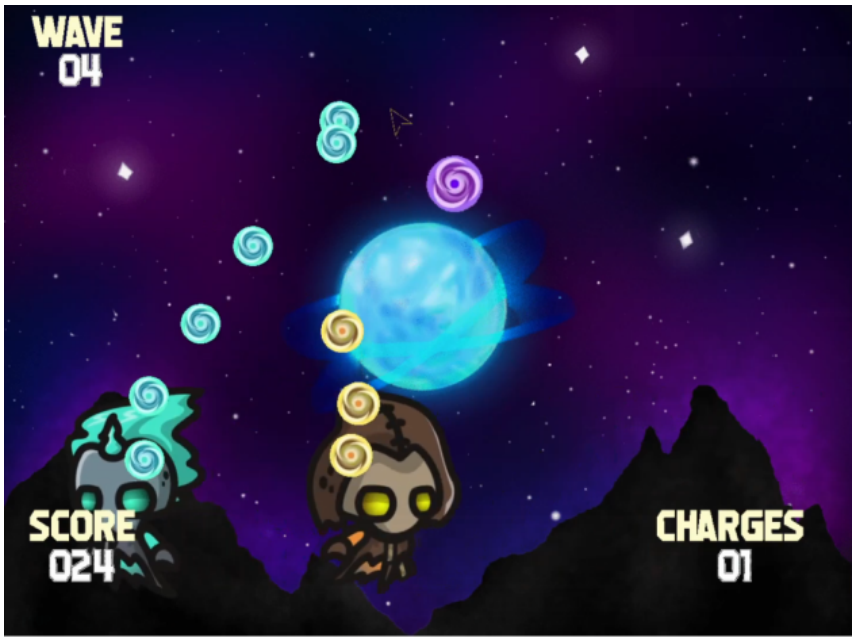

Figure 2.6: Co-op Playing screen Screen

Upon connection, the game starts. The controls are the same as for the singleplayer game, and, as stated previously, the charges and score are shared.

Project Status

3.1 Used I/O devices

Device	Usage	Interrupts?
Timer	Sets the frame-rate and prints the display	Yes
Keyboard	Character movement and Exiting Game Over state	Yes
Mouse	Menu option handling and Blasts throw direction upon button press	Yes
Graphics Card	Handle screen display	N/A
Real Time Clock	Date & Time, Wave Generation and Enemy Blast Throw	Yes
Serial Port	Multiplayer information sending and receiving	Yes

3.2 Timer

The timer is used to set a fixed frame-rate on the display update. The frame-rate that best fit this project was 60 frames per second. However, it must be noted that simply setting the timer's frequency to 60 Hz and updating the screen every frame made the mouse interrupts slower, for reasons we could not detect. As such, the fix was to set the frequency to 120 Hz (timer_set_frequency()), but only update the display screen in half of the frequency (every two ticks).

The display update encapsulates different aspects of the game. As such, the timer ends up being responsible for calling several functions, that do various things such as the collision detection, sprite animation updates, printing of the background and every other sprite displayed, the game reset upon return to the main menu and the change of menu display according to the current game state.

The only difference between the flow of the timer in singleplayer and co-op modes is the sending of an *'end'* signaling byte, upon detected character collision on either side.

3.3 Keyboard

The keyboard is used primarily for character movement (handle_button_presses()). Upon the receiving of a given makecode, if of interest in-game (in this case, any of the 'A' or 'D' keys), a movement in the given direction ('A' for left and 'D' for right) will be made on the character's player controlled sprite, which will now move with the 'Walking' animated sprite set. When the respective break-code is triggered, the movement is stopped and the player's character will return to the 'Idle' state.

The keyboard was also used when exiting the Game Over screen. In this case, any received scancode will trigger the aforementioned action.

3.4 Mouse

At every given mouse interrupt, the in-game mouse cursor is updated with its relative motion, in order to select or point to a given position at any given time.

The mouse is mainly used for launching projectiles. Both the position and the left button click are used to execute this action. When the mouse's left button is pressed, the cursor's position is checked, and a magic blast, with fixed speed, will be thrown in that direction (according to the player's character current position). The 'Attack' animation state of the character is activated. When finished, it returns to the 'Idle' state.

We also use the mouse's cursor for selecting either of the options of the initial screen with a left button click, as well as for the hover animation on the main menu with its movement.

All that was referred previously is globally handled by the function handle_mouse_packet().

3.5 Graphics Card

For the graphics card' configurations, the mode 0x14C was used, which has a resolution of 1152x864, using a 32 bits per pixel format (8:8:8:8), which corresponds to, approximately, 17 million colors (due to Virtual Box not supporting the alpha values, even though this mode does - it would support around 4294 million different colors).

It was set a form of triple buffering with page flipping (the Vertical Retracing didn't appear to be a problem), using the get and set BIOS calls for the current display buffer (swap_buffer()).

A collision detection system was also made, using pixel overlap (checking_collision(), enemy_collision() and wall_collision()).

For the character, several animated sprites were used, itself being an array of animated sprites, with 'Idle', 'Walking' and 'Attacking' states. Every single state had its own sprites, orderly fashioned, in order to allude to a character movement animation.

It was also used a function to draw the HUD and the Date in the 'Instructions' menu, digit by digit, according to the game state/current date and time.

3.6 Real Time Clock

The RTC is mainly used for enemy wave management in game. Using the alarm interrupt, every 15 seconds a new wave is launched. Whenever a new wave is formed, the periodic interrupts, responsible for the launch of an enemy (using a delayer/counter variable to make the frequency reasonable) are shortened, or, in other words, their frequency is increased, leading to a higher frequency of enemy spawn and throw (handle_rtc_ingame_changes()).

The RTC date and time registers are displayed in a user friendly manner in the 'Instructions' menu using the HUD's draw_digit(), updating the project variables which hold these values every second through the update interrupt(rtc_updater()).

3.7 Serial Port

The serial port is used for the communication between VM's, when playing in the co-op mode.

It was important to send information about two key actions of a character: horizontal movement (scancode - send_scancode()) and throw (direction - send_mouse_info()). As such, a communication protocol was defined in order to properly send and receive information with a specific meaning (handle_received_info()).

A sync sequence was also built in order to properly connect the game at its beginning (handle_coop_start()).

Two of the UART interrupts were used: Received Data and Transmitter Holding Register Empty. The UART was also configured with a bit rate of 115200, with 8 bits per char and odd parity.

A queue was also implemented in order to avoid data loss, as well as to assure that all received data is done sequentially (that, for example, all 4 bytes related to the mouse information are received sequentially, without any other information in between).

Code Organization/Structure

In this section, the project's modules will be discussed and for each one there will be a call graph of one important function, when applicable.

4.1 character_movement

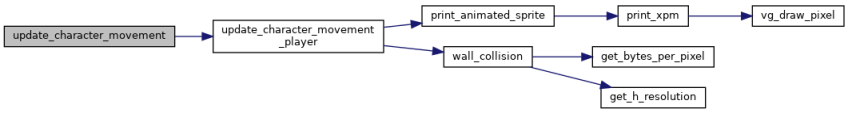

Figure 4.1: update_character_movement() function call graph

This module mainly deals with the game's character movement, handles the keyboard and/or mouse inputs and the wall collisions. It also handles the animation of the player's animated sprites, moving it to the next sprite after a certain number of function calls (defined in the instantiation of the character animated sprite). These character sprites were obtained here and are free to use.

Weight: 15%

Contributors: Nuno Costa, Gonçalo Alves

4.2 enemies

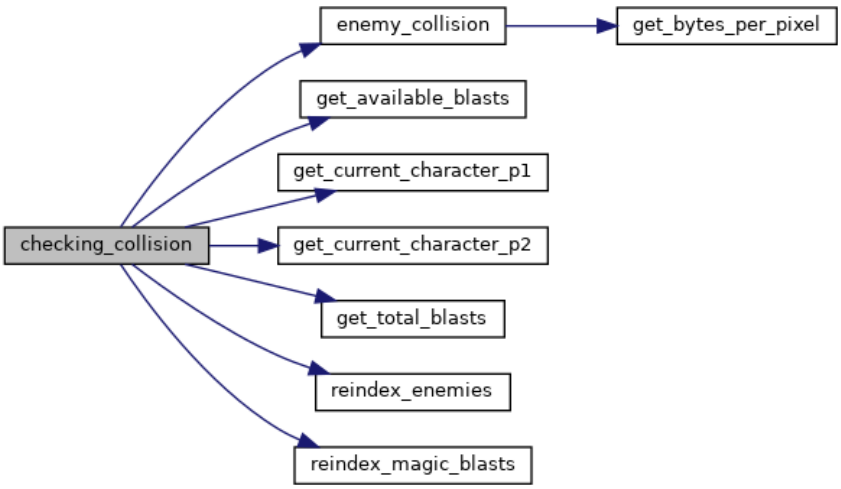

Figure 4.2: checking_collision() function call graph

This module deals with the in-game enemy spawning and handles the collisions between the enemies and either the player's magic blasts or the player itself. It also holds and updates the current score variable.

Weight: 15%

Contributors: Nuno Costa, Gonçalo Alves

4.3 game_state

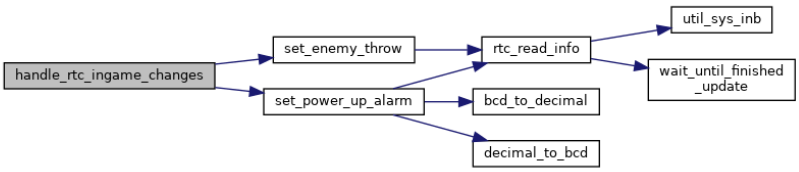

Figure 4.3: handle_rtc_ingame_changes() function call graph

This module deals with the generation of enemy waves, upon the reception of an RTC interruption.

Weight: 5%

Contributors: Nuno Costa, Gonçalo Alves

4.4 hud

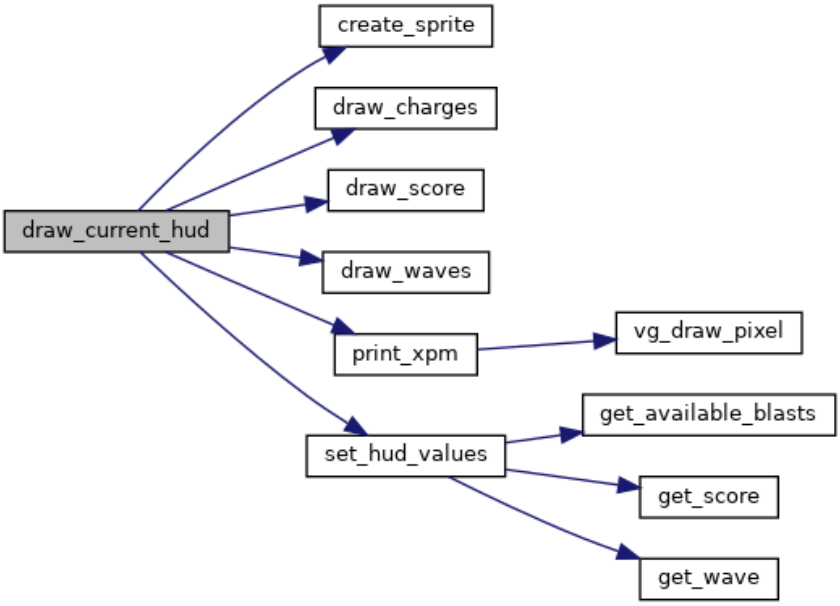

Figure 4.4: draw_current_hud() function call graph

This module deals with the display of the HUD in the game and the display of the RTC date/time in the 'Instructions' menu.

Weight: 5%

Contributors: Nuno Costa, Gonçalo Alves

4.5 magic_blast

Figure 4.5: throw_magic_blast() function call graph

This module deals with the generation of magic_blasts after a player's mouse input, on the direction given by the player and cursor.

Weight: 12%

Contributors: Nuno Costa, Gonçalo Alves

4.6 menu

This module deals with the selecting process in the main menu.

Weight: 1%

Contributors: Nuno Costa, Gonçalo Alves

4.7 ds12887

Constants for programming the ds12887 Real Time Clock Controller

Weight: 1%

Contributors: Nuno Costa, Gonçalo Alves

4.8 i8042

Constants for programming the i8042 Keyboard Controller (and useful i8254 Timer constants)

Weight: 1%

Contributors: Nuno Costa, Gonçalo Alves

4.9 i8254

Constants for programming the i8254 Timer

Weight: 1%

Contributors: Nuno Costa, Gonçalo Alves

4.10 interrupt_handler

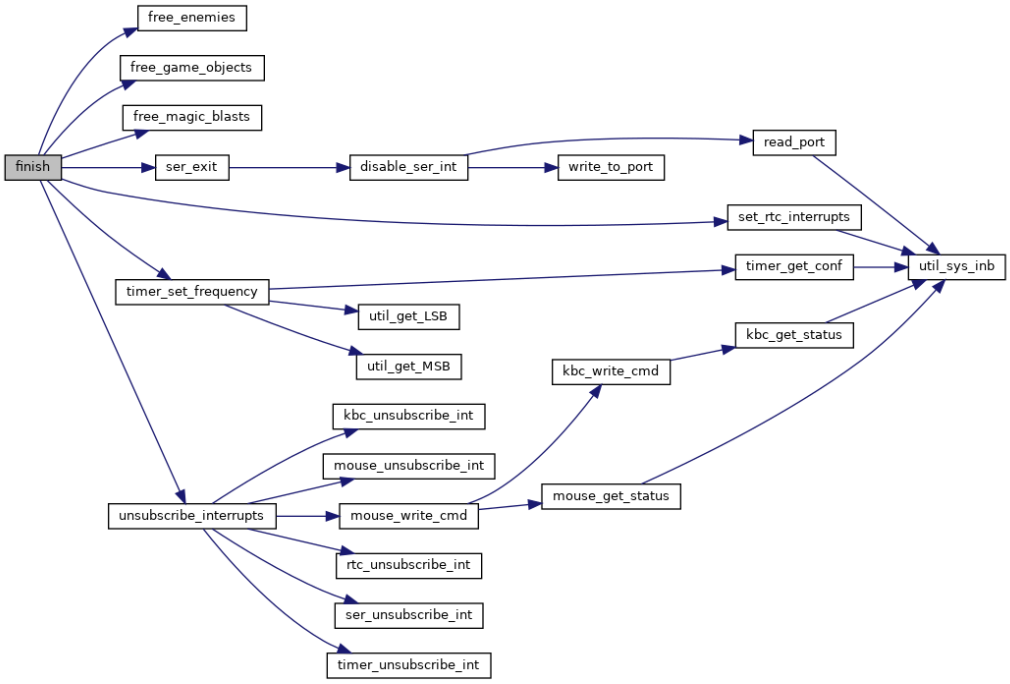

Figure 4.6: finish() function call graph

This module deals with the program setup (subscribing interrupts, creating game objects) and handling the different interruptions. It is also responsible for setting up the termination of the program.

Weight: 15%

Contributors: Nuno Costa, Gonçalo Alves

4.11 kbc

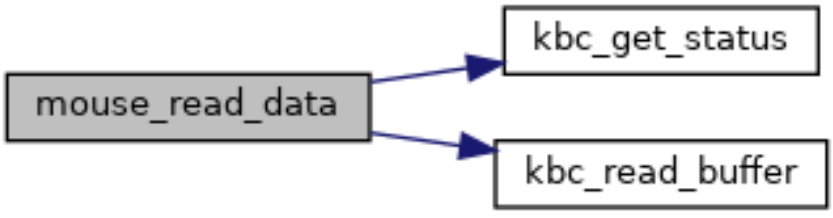

Figure 4.7: mouse_read_data() function call graph

This module deals with the KBC Controller, which encapsulates both the Keyboard and the Mouse.

Weight: 5%

Contributors: Nuno Costa, Gonçalo Alves

4.12 queue

This module implements a queue in C. Its implementation was inspired in this tutorial.

Weight: 5%

Contributors: Nuno Costa, Gonçalo Alves

4.13 rtc

Figure 4.8: rtc_read_info() function call graph

This module deals with the RTC interrupts and date/time update.

Weight: 2%

Contributors: Nuno Costa, Gonçalo Alves

4.14 serial_port

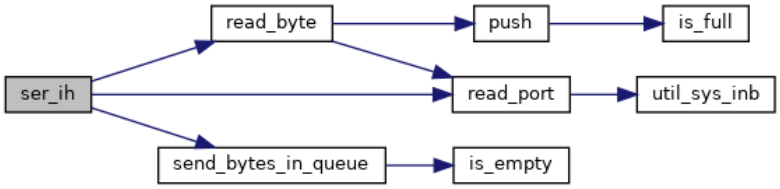

Figure 4.9: ser_ih() function call graph

This module deals with the UART interrupts, as well as all functions related to the specific communication protocol.

Weight: 5%

Contributors: Nuno Costa, Gonçalo Alves

4.15 serial_port_controller

Constants for programming the Serial Port

Weight: 1%

Contributors: Nuno Costa, Gonçalo Alves

4.16 timer

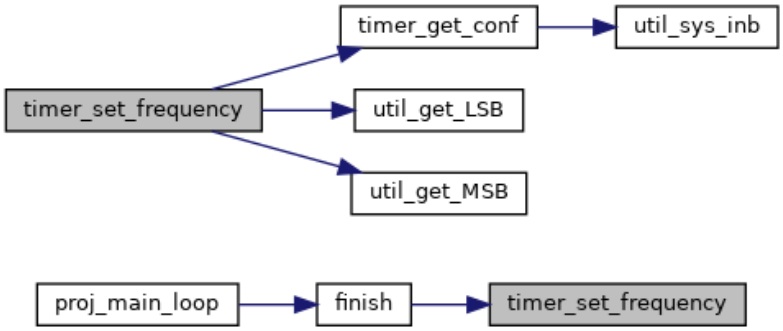

Figure 4.10: timer_set_frequency() function call and caller graph

Constants for programming the Serial Port

Weight: 1%

Contributors: Nuno Costa, Gonçalo Alves

4.17 video_gr

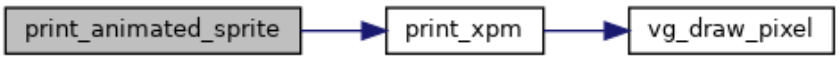

Figure 4.11: print_animated_sprite() function call graph

This module deals with the Graphics Card, either dealing with page flipping or drawing sprites on the secondary buffer.

Weight: 10%

Contributors: Nuno Costa, Gonçalo Alves

4.18 Driver Receive function Call Graph

The driver receive loop, included in the function interrupt_call_receiver(), calls, according to the interrupt it received, one of the 5 handlers that exist for each of the 5 possible interrupts. The following images show the caller graph of the aforementioned function, as well as the call graphs of each of the handlers (**Note:** the keyboard and mouse handlers do not show the in-application handling functions as they are not in the same directory).

Figure 4.12: interrupt_call_receiver() function caller graph

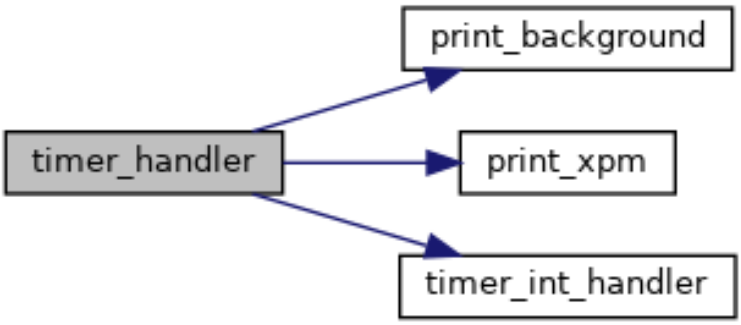

Figure 4.13: timer_handler() function call graph

Figure 4.14: kbd_handler() function call graph

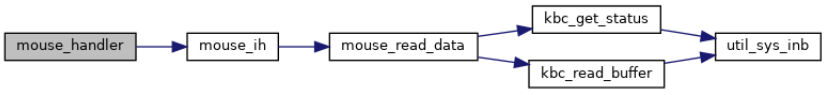

Figure 4.15: mouse_handler() function call graph

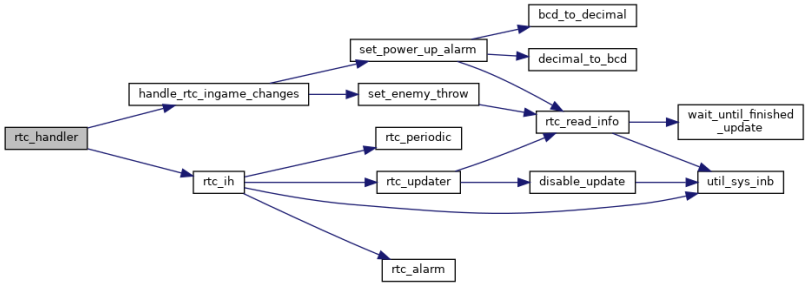

Figure 4.16: rtc_handler() function call graph

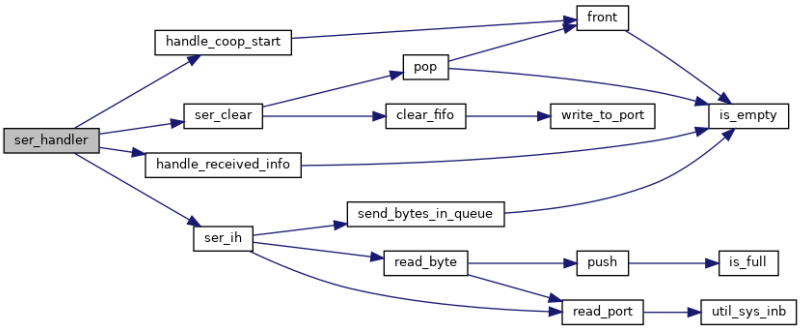

Figure 4.17: ser_handler() function call graph

Implementation Details

In this project, we tried our hardest to separate the functions that dealt with the peripherals from the functions that dealt with the game logic itself, that is why we have two separate directories. This layering allowed us to write better and more modular code.

For event driven code, all of our program was based around user events and/or interrupts: key presses, mouse presses and movement, rtc updates and alarms, timer and rtc periodic interrupts and even serial port interrupts (all handled by different handlers in interrupt_call_receiver()). Any of this events produce a different response by the program, making the program reactive and, therefore, event driven.

In regards to state machines, we built the game around the notion of game states (Menu, Playing, Game Over, etc, in the enum gameState), which alter the behavior of several of our functions. This way, the program effectively works as a state machine.

There was the need to implement a good collision detection algorithm, so some research was made to find a way to precisely detect collisions, by pixel overlap, instead of a more simple square collision, by image borders (enemy_collision()).

For that, the pixel transparency color of each of the xpm was used, using the auxiliary function xpm_transparency_color().

For the Serial Port and Real Time Clock only interrupts were used.

In the Serial Port, it was clear that sending two bytes in quick succession would greatly increase the probability of a byte loss, most likely due to overwrite. Even using the FIFO's from UART, the problem was still way too critic, as the synchronization process was practically inexecutable. As such, the need to assure that the bytes were sent without major losses was secured by implementing a queue. This, with the help of the interrupts, made the process of sending data via UART much more reliable. It still must be noted that when trying to run the co-op mode without any connected virtual machine via serial port, sometimes the program would receive its own data. Why that happens is still unclear, and the fact that it happened sporadically made it even more difficult to understand , but the most important part of this module - the connection between two different virtual machines - was done successfully (tested locally and via Remote Port Forwarding).

In the RTC, all three interrupts were used, as explained previously in 3.6. However, the display of the date on screen produced lag on the mouse movement. That is probably due to the fact that the letters are printed to the screen every tick, but even then the solution seemed to be pretty complex.

The mouse lag was, in fact, an issue in the implementation of this project. The lag due the frequency was already mentioned (3.2), as well as the lag with the date/time in the 'Instructions' menu, but a lot of it was fixed by having a background buffer, as the background had fixed area and position. Simply filling a buffer with the corresponding xpm and using memcpy to the display buffer every frame was found to be the best solution, and fixed this problem for the menus and game (except, as referenced, the 'Instructions' menu).

Conclusions

This project, along with the previous work in this course unit, provided us with a more detailed insight into the peripherals of a computer, be it how they work or how they are managed.

Since the project built is a game, we had to build our program around the notion of game states, and as suggested in the theoretical classes, a state machine was the way to go. In this regard, we had to develop a new way of critical thinking and of programming, that was far different from what we were accustomed to.

Along with this, the whole course unit brought us personal development since we had very little help through out it and therefore had to do our own research and rely upon ourselves.

There were some hardships along the project, the major one being the UART, but overall the final product is a project to be proud of.