Faculdade de Engenharia da Universidade do Porto

Petty Cats

[L.EIC018] Laboratório de Computadores

T02G6
Davide Teixeira up202109860
Eduardo Baltazar up202206313
Emanuel Maia up202107486
Joana Noites up202206284

Licenciatura em Engenharia Informática e Computação

Professor Regente: Prof. Pedro Souto

Professor: Prof. Nuno Cardoso

June 2, 2024

Contents

List of Figures

Chapter 1

Introduction

Our game, 'Petty Cats,' invites players to engage with various feline friends by petting them. Each cat's color indicates a unique way in which the player must move their mouse to pet them correctly. Successfully petting a cat earns the player points, rewarding their efforts. However, if the player fails to pet a cat correctly, the cat will become upset and leave. This adds a layer of challenge and excitement to the game, as players must quickly adapt to each cat's preferences to achieve high scores.

Figure 1.1: Petty Cats' Logo

Chapter 2

User instructions

This chapter is dedicated to explaining how users can interact with our program, the Petty Cats video game.

2.1 Initial Menu

When the application is opened, users are automatically redirected to the main menu. From the menu, users can select the game they want to play and access instructions on how to play. Figure 2.1 shows the visual layout of this menu.

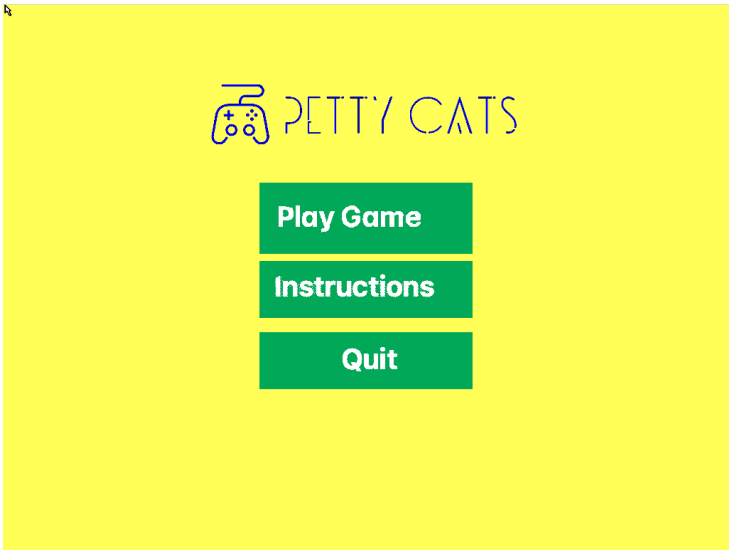

Figure 2.1: Petty Cats' Main Menu

The user can use the mouse to select an option, with one option per button. Each button changes color to blue when hovered over with the mouse (see Figure 2.2). To select an option, the user must click the left mouse button.

Figure 2.2: Petty Cats' Main Menu with Hoover

The options are:

1. **Play Game:** Initializes the Game - Figure 2.4
2. **Instructions:** Gets a page with the game Instructions - Figure 2.3
3. **Quit:** Closes the game

2.2 Instructions Menu

The instructions menu provides information about the cats and their respective state machines, which help players capture them and accumulate points. This static page, shown in Figure 2.3, aims to familiarize users with the different cats and how to interact with them. Interaction with this page is solely via the keyboard. Pressing the **ESC** key returns the user to the main menu.

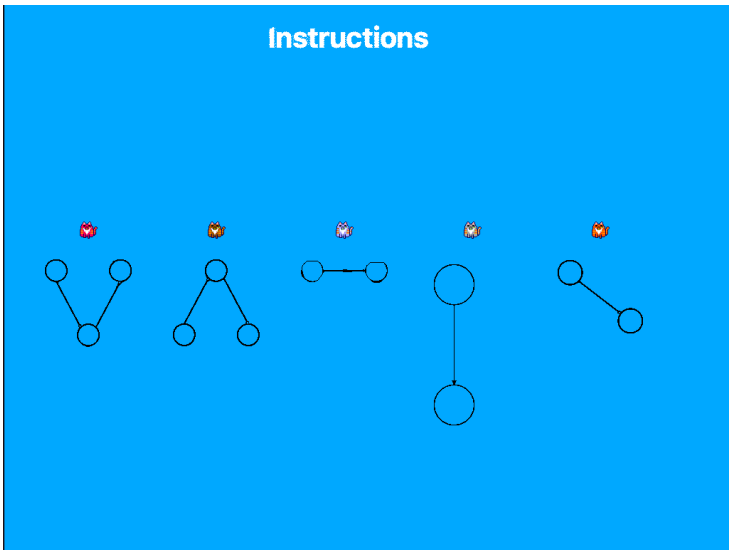

Figure 2.3: Petty Cats' Instructions Menu

2.3 Game Playing

Once the game starts, we are presented with a screen showing our character, which we can control using the **WASD** keys. Figure 2.4 has an example of a game being played.

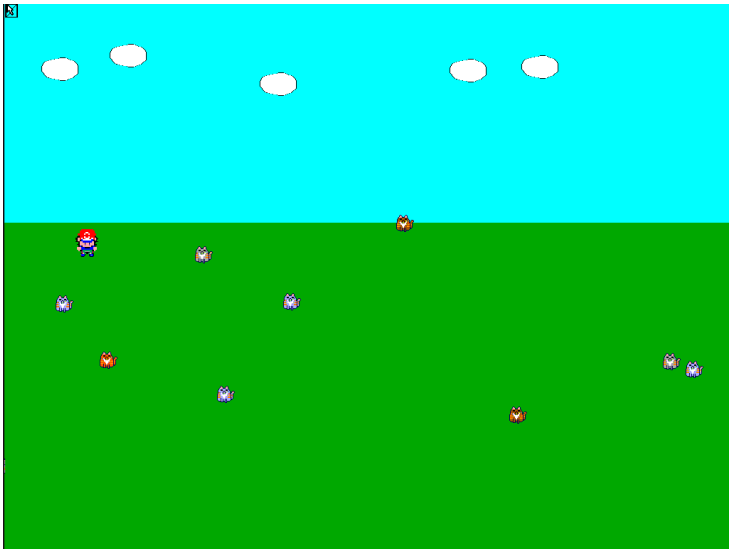

Figure 2.4: Petty Cats' Main Menu

- **W:** Move the character up
- **S:** Move the character down
- **A:** Move the character to the left
- **D:** Move the character to the right

The objective of the game is to catch as many cats as possible by performing 'destinies' on them within a set time limit. When the user approaches a cat, they must determine the correct sequence of movements required to catch it. These movements are performed using the mouse and are influenced by the physical characteristics of the cat.

If the user presses the **ESC** key, the game returns to the initial menu.

2.3.1 Cat's state machine

Each cat in the game possesses a unique state machine determined by its colors. Figures 2.5, 2.6, 2.7, 2.8 and 2.9 show the state machines for each one of the cats.

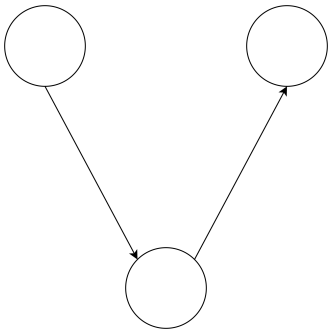

Figure 2.5: Cat 0 State Machine

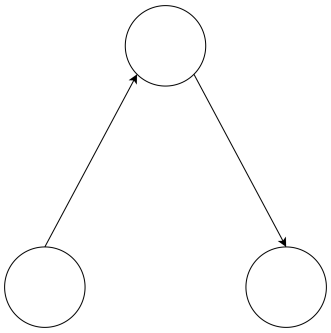

Figure 2.6: Cat 1 State Machine

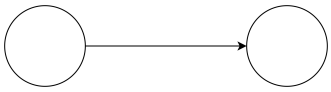

Figure 2.7: Cat 2 State Machine

Figure 2.8: Cat 3 State Machine

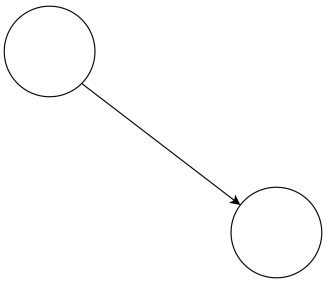

Figure 2.9: Cat 4 State Machine

For the user to interact with the cats, they must maneuver the main character into proximity with the cats (which have collision detection). Once in range, the user must perform the correct mouse movement to earn the maximum possible points. Figure 2.3 has the right mapping between cat colors and their respective state machines.

The overall state machine of the system is represented in figure 2.10:

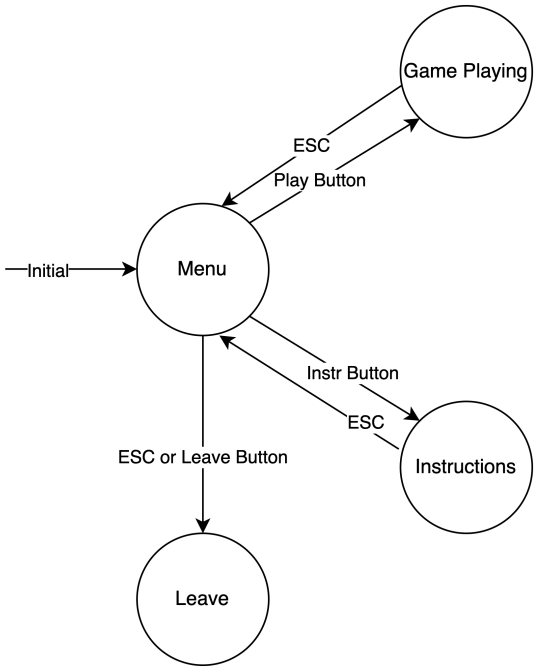

Figure 2.10: Petty Cats' Overall State Machine

Chapter 3

Project Status

This chapter provides an overview of the project's status and outlines the devices utilized to bring the project to fruition, detailing their usage through either polling or interruption mechanisms. Notable input/output (I/O) devices used within the context of this project are described in Table 3.1

Device	What for	Int.
Timer	Controlling frame rate	Y
Keyboard	User Control	Y
Mouse	Menu Selection and Cat Interaction on the Game	Y
Video Card	Application menus and screens display	N
Real Time Clock	Scenario Differentiation	N

Table 3.1: I/O devices used

3.1 Timer

In this project's context, the Timer device was utilized for frame rate control, employing interruptions for this purpose. In the code, the process begins within the main function. Initially, the timer's frequency is set to the desired value, in this case, 60 Hz, achieved through a call to the **timer_set_frequency** function.

Following the frequency setup, the program subscribes to timer interruptions to enable drawing the game at each occurrence of the timer interruption. This subscription is established using the **timer_subscribe_int** function.

Then, upon receiving a timer interruption, the game updates its user interface (UI) by invoking the **drawGame** function.

Finally, when the user closes the game, another function call is made to **timer_unsubscribe_int** to disengage the timer interruptions generated at the previously set frequency.

3.2 Keyboard

The keyboard plays a pivotal role in this project, responsible for both player movement and navigation back to the menu.

The game's process begins by subscribing to keyboard interruptions to handle user input whenever a specific key is pressed. This subscription is established using the function call **kbd_subscribe_int**.

Upon each keyboard interruption, the **kbc_ih** function is invoked to manage the interruption and return

the scan code of the pressed key via a global variable. This scan code is then passed to the keyboard's controller through **update_keys**. The controller adjusts the game state based on its current status. For instance, if the game is in the menu, only the **ESC_BREAK** key is relevant, allowing the player to exit the game. However, during gameplay, other keys may correspond to player movement, and in that case, the controller updates the player's position accordingly.

Before concluding the process and returning, the game unsubscribes from keyboard interruptions, similar to the timer, using the **kbd_unsubscribe_int** function.

3.3 Mouse

In this project, the mouse stands out as the most crucial I/O device for enhancing user experience, responsible for menu interactions and gameplay. As mentioned in the previous chapter, the game utilizes various mouse state machines to interact with the different cats present.

The initial function call for this I/O device is the **issue_cmd_to_mouse** function, which enables the mouse's data reporting. Subsequently, following the standard procedure for interrupt-based devices, a routine subscribes to the mouse's interruptions using the **mouse_subscribe_int** function.

Upon each mouse interruption, the **mouse_ih** routine is invoked to retrieve the packet containing information about the mouse's interrupt. This information is then passed to the mouse controller to update its state, achieved through the **moveMouse** function. Initially positioned at (0,0) on the screen, the mouse's position is altered during interruptions based on the displacement provided in the mouse's packet. This positional data is essential for defining the respective state machines.

Furthermore, the mouse is utilized for interacting with menu buttons. The menu checks for overlap and left-button presses. If detected, the program's state changes accordingly.

Finally, as the main function concludes, mouse interruptions are unsubscribed using **mouse_unsubscribe_int**, and data reporting is disabled via **kbc_restore_mouse**.

3.4 Video Card

The video card serves as the I/O device responsible for displaying the game on the screen.

In the main function, the setup of the video card begins. This involves entering the video mode with the **enter_video_mode** function, which is provided with the code corresponding to the desired screen resolution (in this case, the indexed mode 0x105).

Subsequently, the **create_vram_buffer** function is called to generate the VRAM buffer and the double buffer for optimization purposes.

Within our game logic, for each timer interruption, the double buffering (referred to as **back_buffer**) is copied to the actual VRAM buffer (referred to as **front_buffer**) for display. The **back_buffer** is only updated when the game's state changes. This double buffering strategy ensures that the VRAM frame buffer is not reset with every timer interruption, enhancing program efficiency, particularly at a 60 Hz frequency.

For drawing the game, an API was developed containing key functions. There are two abstraction layers for drawing the game. The first layer is a low-level abstraction layer (the GPU lab), comprising several low-complexity functions applicable in various scenarios. These functions include **draw_pixel**, **draw_line**, **draw_rectangle**, **set_background_color**, **draw_xpm**, and **draw_text** (utilizing different fonts). They directly interact with the VRAM buffer (and the double buffer), providing a generic API intended for buffer manipulation to facilitate display.

The second layer (the high-level API) adapts the GPU API to our project's context. It features functions

such as **drawGame**, **drawMenu**, **drawClouds**, and **drawCat**, tailored specifically for our project's needs.

3.5 Real Time Clock

In this project, the Real-Time Clock (RTC) is utilized to alter the game's scenario based on the time of day. For example, the game environment changes to reflect a dark sky during nighttime.

The RTC is integrated into the project via polling. The concept involves checking the current date and time at a frame rate of once per minute and adjusting the scenario accordingly.

Using timer management, the program invokes the **updateGameTime** function every minute to examine if the current date and time justify a scenario change. If a change is warranted, the game's state transitions to the night mode.

Chapter 4

Code Structure and Organization

This chapter is focused on the organization of code documentation, highlighting how each I/O device interacts with the game state, as well as the game view and user interface (UI).

This chapter shows how input devices such as the keyboard, mouse, and RTC influence the game state, affecting gameplay mechanics and scenario changes. In addition, it shows how output devices like the video card facilitate the presentation of the game view and UI elements on the screen.

As expected, some functions are derived from code shown in LCOM's slides, authored by prof. Pedro Souto.

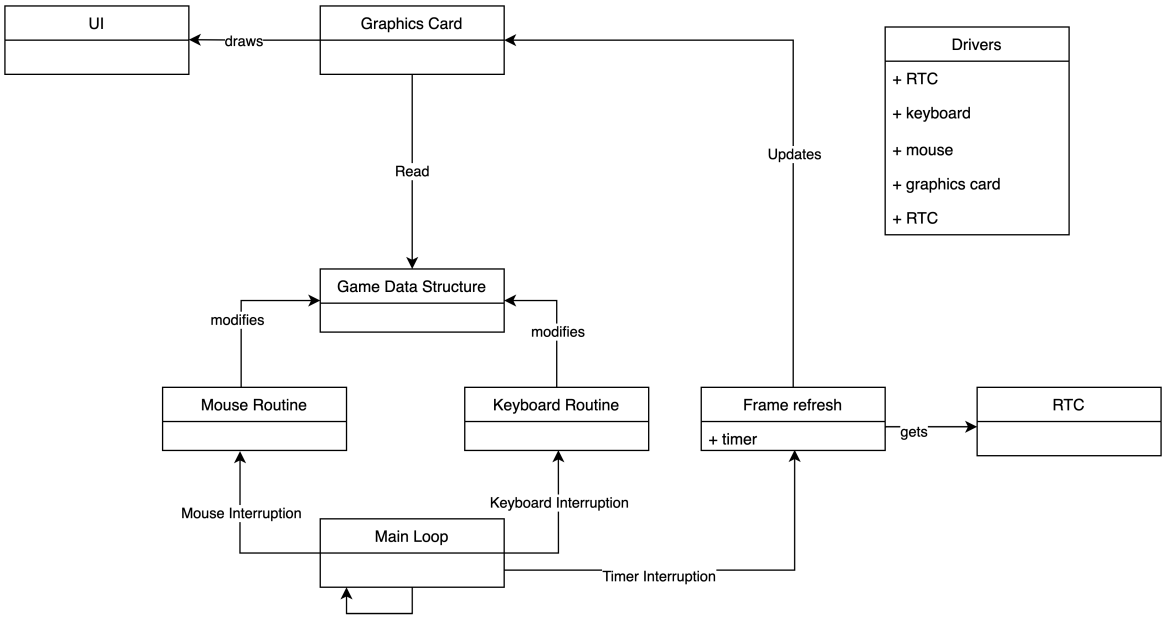

Figure 4.1: System's general Architecture

4.1 Drivers

To isolate the abstraction layer responsible for direct communication with the I/O devices, a directory named **drivers** is implemented. In this directory, all the logic necessary to establish communication with these devices is encapsulated. This design decision ensures that the core game logic remains decoupled from the specifics of the underlying hardware.

4.1.1 gpu.c

This file can be seen as the main API for the application to interface with the GPU. It uses primitive data structures enabling client functions to effectively interact with the PC's Video Card. These primitives include the creation of the VRAM buffer, functions to enter graphics mode, and other utilities like drawing XPM files, single pixels, or rectangles.

Derived from Davide's lab, with modifications to expand its functionality, this code has a range of functions to operate outside of this file, including features like setting background colors. Drawing inspiration from Lab 5[1]'s recommended API and code structure, it maintains compatibility and coherence with established practices while extending functionality to suit the specific requirements of the project.

Weight: 5%

4.1.2 keyboard.c

Imported from Davide's Lab 3[2] without doing any modifications, this code maintains its original structure and functionality. Its primary role is to interact with the keyboard device and return the scan code corresponding to any key pressed.

Developed following the guidelines provided in Lab 3.

Weight: 5%

4.1.3 mouse.c

Imported from Joana's and Eduardo's Lab 4[3], it did not suffer modifications from the lab. This file's main responsibility is to communicate with the mouse upon an interrupt and parse the mouse bytes in order to create the mouse packets. Developed following the guidelines provided in Lab 4

Weight: 14%

4.1.4 rtclock.c

This file contains Emanuel's implementation of an interface for Minix's Real Time Clock, based on the needs of this project, which essentially means fetching the current date-time, and storing it in a specific data structure. It development mostly follows the guidelines provided in Lab 5 and the lecture slides for the course.

Weight: 5%

4.1.5 timer.c

This file represents a consolidation of Lab 2 contributions from every student in the group. Each function included in this file was selected based on the best implementation from the respective Lab 2, and chosen for integration into the project.

No modifications have been made to this file since its creation. It uses the structure recommended in Lab 2[4], as shown in the corresponding lab materials, ensuring consistency and compatibility across all contributions from the group members.

[1] https://web.fe.up.pt/~pfs/aulas/lcom2223/labs/lab5/lab5.html
[2] https://web.fe.up.pt/~pfs/aulas/lcom2324/labs/lab3/lab3.html
[3] https://web.fe.up.pt/~pfs/aulas/lcom2324/labs/lab4/lab4.html
[4] https://web.fe.up.pt/~pfs/aulas/lcom2324/labs/lab2/lab2.html

Weight: 5%

4.1.6 utils_drivers.c

This file serves as an adaptation of the Lab 2's recommended file, incorporating modifications to address needs throughout the project's development. Adjustments were made as necessary to accommodate additional functionalities, such as the implementation of functions to generate random numbers.

Weight: 3%

4.2 Controller

This directory has the logic responsible for manipulating the game state in response to I/O device interruptions. Changes to the game state are dependent upon the current state of the game itself.

4.2.1 controllerKeyboard.c

This file contains the logic for manipulating the game state based on both the current state of the game and keyboard interruptions. The actions triggered by a key press, such as the **W** key, vary depending on whether the game is in progress or if the user is navigating the menu.

Weight: 5%

4.2.2 controllerMouse.c

This file contains the functions for the 5 state machines, a function to limit the mouse's position (**move_mouse()**) and an enum containing all the possible states: INIT, DUP, DDOWN, DSIDE, FAIL, SUCCESS. **move_mouse()** is called every interrupt to impose bounds on the mouse and control overflows and the state machines are called only in one state of the game.

Weight: 10%

4.3 Model

This directory has the data structures that comprise the game, encapsulating the entirety of the game's model within these structures.

4.3.1 CatInfo.c

This file contains the function that maps a cat to its associated information.

Weight: 2%

4.3.2 Entity.c

In this file, it has a versatile Entity structure designed to represent different elements within the game world, whether it's a cat, a player character, or any other object. This Entity structure encapsulates key attributes and functionalities shared among different entities, providing a flexible and reusable template for modeling various game components.

Weight: 5%

4.3.3 Game.c

This file has the **Game** data structure, which is a container for aggregating all of the game's information data structures. This structure has representations of the game's state, room data structures, and other essential components necessary for managing and saving gameplay. The **Game** structure acts as the central hub through which various parts of the game interact and communicate, providing an API for organizing and controlling the game's behavior.

Weight: 5%

4.3.4 Room.c

This file has the data structures responsible for storing all entities involved in a game while it is in progress. This structure has definitions for various entities such as the player character, cats, clouds, and stage elements. By encapsulating these entities within a single data structure, the game's runtime environment becomes more manageable and efficient, facilitating interactions and updates between different game components.

Weight: 5%

4.4 Utils

This directory defines several auxiliary data structures and functions to support various aspects of the game's functionality.

4.4.1 Hurtbox.c

Define the Hurtbox to attribute to each of the cats. Each hurtbox is defined by a δ_x and a δ_y value that is used to check the collision.

Weight: 5%

4.4.2 Position.c

This file has an auxiliary data structure that associates a specific entity with its x and y positions within the game world.

Weight: 4%

4.4.3 utils.c

This file has some useful functions designed to help in manipulating the game's model. Included are functions for converting between radians and degrees, etc.

Weight: 2%

4.5 Viewer

This folder encompasses all the logic required to render and display the game on the screen.

4.5.1 gameView.c

This is the sole viewer file within the folder. It contains all the application interfaces with specific functions dedicated to rendering various elements of the game, such as buttons, cats, the player character, the sky, or clouds. Additionally, functions for rendering the menu are also included in this file.

Weight: 15%

4.6 main.c

This main file houses the **proj_main_loop**, which is responsible for receiving interruption notifications, invoking the interruption handlers, and calling either the controllers or the viewer to modify the game state or render the game, respectively.

Weight: 5%

4.7 Function call graph

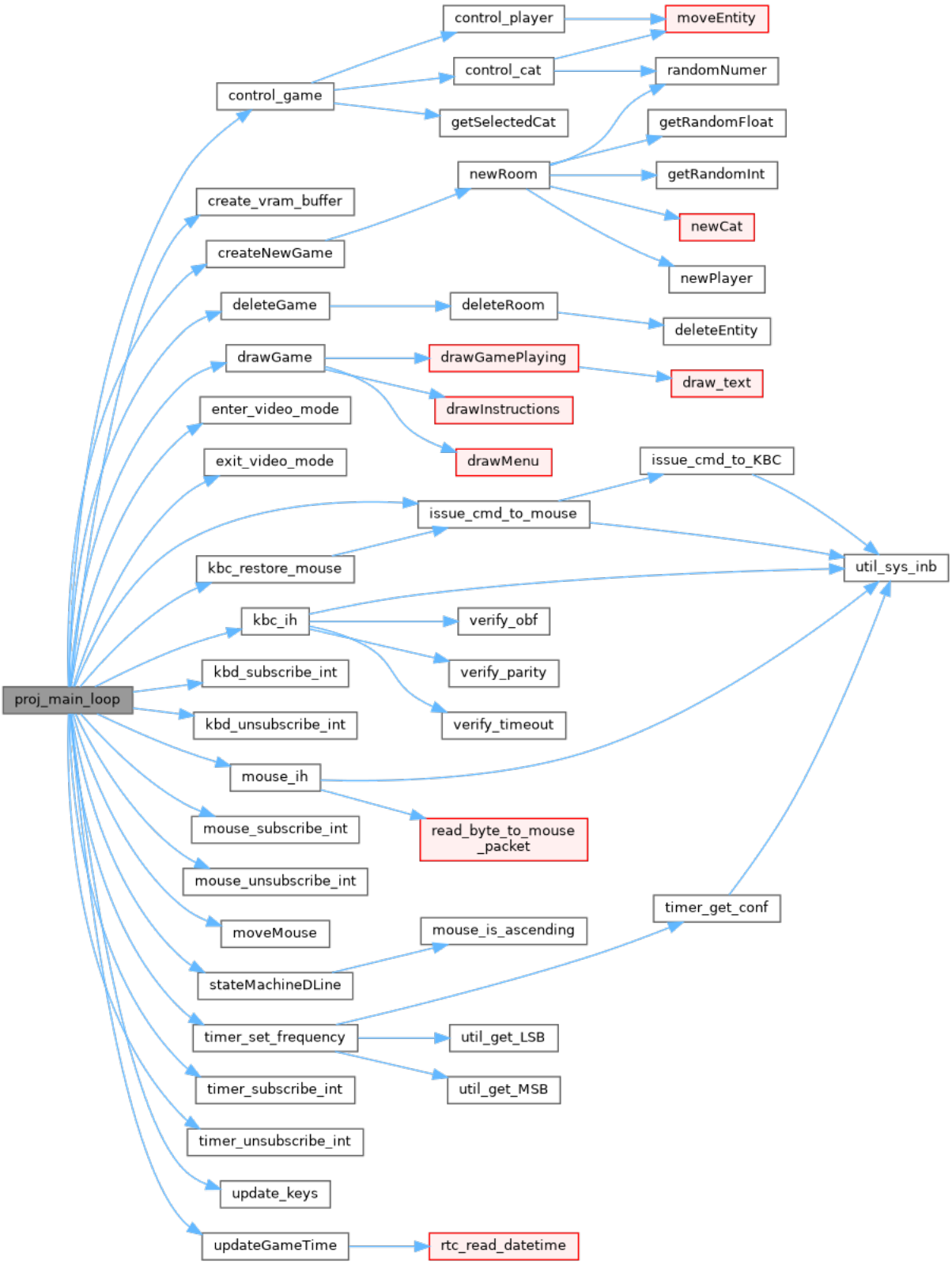

Figure 4.2: Petty Cats' Call Graph

Chapter 5

Implementation Details

5.1 State Machines

Mentioned in the lectures and implemented in the **mouse_test_gesture()** function in lab 4, mouse state machines are essential to our project. In **controllerMouse.c** 5 of them were developed with the objective of detecting if the mouse completes:

1. A 'V' shaped movement (**stateMachineV()**)
2. An inverted 'V' shaped movement (**stateMachinenInvertedV()**)
3. A vertical line (**stateMachinenVLine()**)
4. A horizontal line (**stateMachinenHLine()**)
5. A diagonal line (with a slope of (approximately) 1) (**stateMachinenDLine()**)

The state machines detecting both the 'V' and inverted 'V' movement work similarly to the one developed in lab4, except only the left button of the mouse must be pressed along the movement. When arriving to the vertex, the user must release the button before continuing to the second part of the 'V'. The rest of the state machines also require only the pressing of the left mouse button.

Another crucial difference when compared to the state machines in the lectures is that these have 2 additional (not related to motion) states: SUCCESS and FAIL, that let the game know if there was a mistake in the movement, so we can determine whether the cat should be satisfied or not.

5.2 Collisions

Although collision detection and correction between entities is not essential to the game's core mechanics, it helps with immersion.

Collisions are detected through hurt box intersections. Hurt boxes define a width and a height that, relative to the corresponding entity's position, will be compared against other hurt boxes and checked for overlapping. This work is done in the **Entity.c** file in the **CollisionType checkCollision(Position cPosA, Entity* entityA, Entity* entityB)** function, which returns a "CollisionType" enum (defined in **Entity.h**) that determines the collision axis if "entityA" were to move to "cPosA".

However, collision correction does not utilize the previously mentioned returned value to its fullest, as the **Position futurePos(Entity* entity, Room* room)** function limits itself to returning a **Position** in the opposite direction of "entity" in case of an imminent collision, holding now different value based on the

type of collision (this would allow for entities to slide along each other's hurt boxes without coming to a full stop). This is a result of a compromise due to an unfixed (in time) bug which would allow the hurt boxes to sometimes be trespassed.

5.3 Object Orientation

During the development of this project, we tried to adopt principles of the object-oriented paradigm, despite the limitations of the C language in directly supporting this approach. By using the capabilities of structs in C, we were able to modularize our game's information into several data structures. This strategy not only facilitated the application of best practices but also significantly reduced redundancy within the project.

By treating structs as objects, we encapsulated data and provided function interfaces to interact with this data. This approach helped us maintain clear boundaries between different parts of the program, increasing readability and maintainability.

Chapter 6

Conclusions

After concluding the development of 'Petty Cats', it can't go without saying that LCOM is a very challenging subject, if not one of the most challenging so far. Unfortunately, the serial port feature was not implemented due to a lack of time. Balancing the demands of this project with other subjects proved to be quite difficult. Each team member had multiple assignments and exams to prepare for in other modules, which left limited time to dedicate to this aspect of the project.

While the completed functionalities are a source of pride, the reality of managing other deadlines and commitments meant that certain elements of the project had to be prioritized over others. Despite these challenges, the experience and skills gained through this project are really valuable as they allowed for the better understanding of low level programming, even if done on an outdated virtual machine.

Appendix A

Installation Instructions

A.1 Compilation Command

To compile the project, use the following command:

```
make clean && make
```

A.2 Run the Project

To run the project, use the following command:

```
lcom_run proj
```