PAINTMIX

LCOM PROJECT

Turma 03 - Grupo 09

Bernardo Manuel Esteves dos Santos – up201706534@fe.up.pt

Tito Alexandre Trindade Griné – up201706732@fe.up.pt

Index

User Instructions

1. Main Menu

When starting the program, you are presented with a menu layout with the following options:

- **Play** - Starts a new game of PaintMix
- **Draw** - Goes to draw mode
- **Exit** - Exits program

Figure 1

To select an option the user must use the **mouse** and press the **left mouse button** when the cursor is **hovering** a button. Every button has a hover animation, shown in Figure 1.

In the bottom left corner there is a **date icon**. When the cursor is hovering the icon the current date and time will pop up (Figure 2) and remain there until the cursor is no longer hovering the icon. This feature is present in **every** in-game screen (except victory screen), working the same way as described.

Figure 2

2. Setup Menu

When the user presses the **Play button**, the program will start a new PaintMix game, directing the user first to the player setup menu (Figure 3).

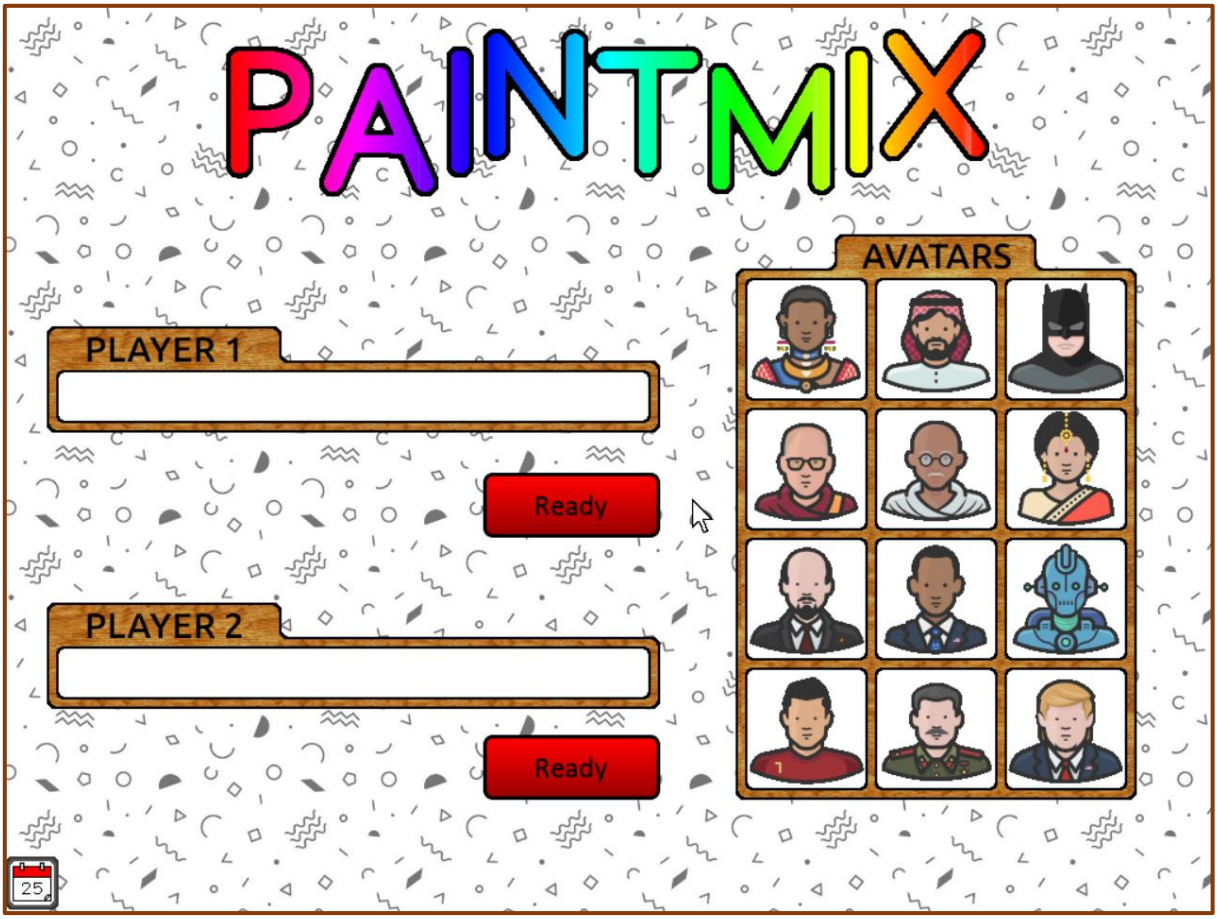


Figure 3

The first player must insert their **username** and press enter and choose an **avatar**. Only when both are chosen, will the ready button work, allowing player two to insert their name and avatar, which must be different from player one's avatar. When a name is to big the program will simply not accept more characters until at least one is deleted. When an avatar is selected, it will appear **toggled** as shown in Figure 4, with a different colour depending on which player the chosen avatar belongs to. After an avatar is chosen but the ready button has not yet been pressed it's possible to select a different avatar, the previous one will be untoggled and the new will now appear toggled. As mentioned, the **date icon** still appears in the bottom left corner and works the same as previously described.

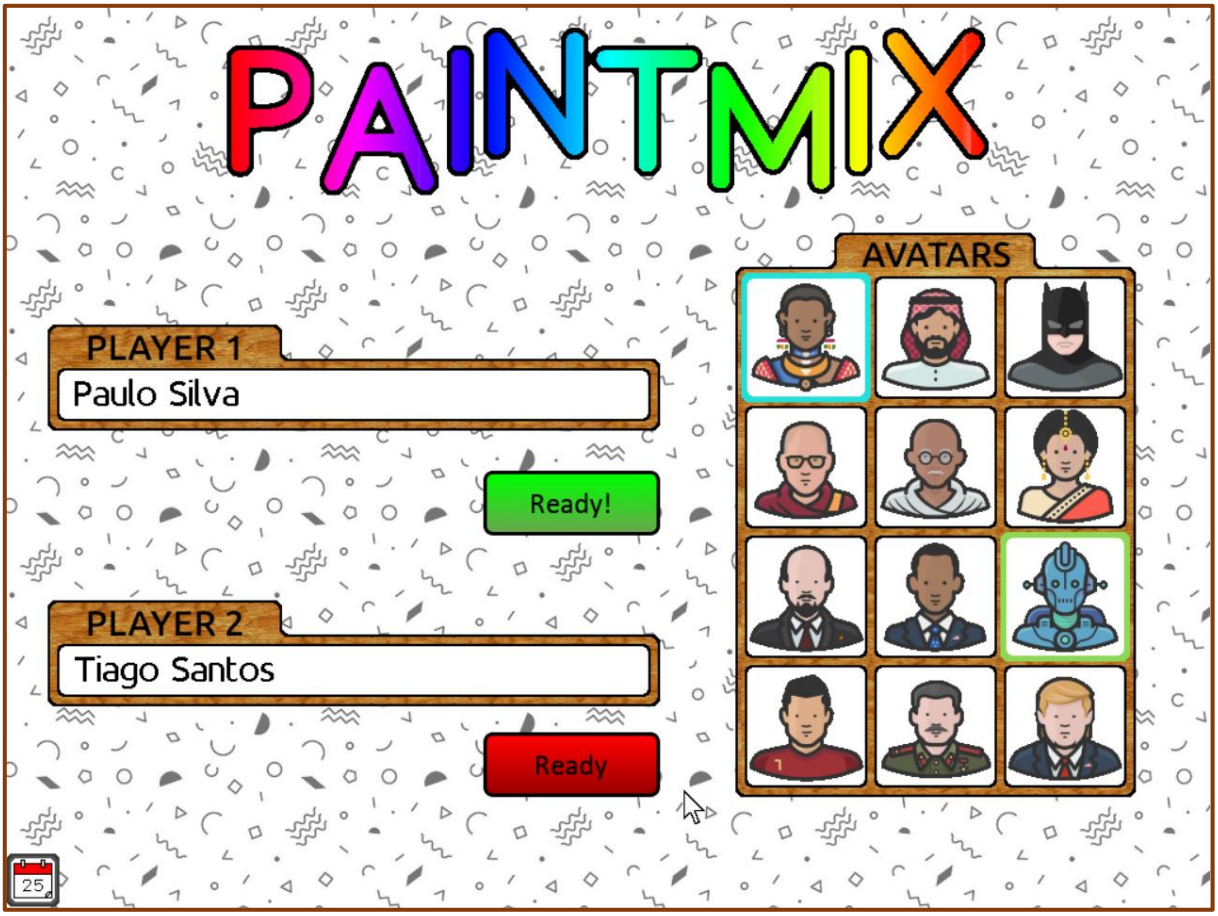


Figure 4

After **both players** have selected their avatar and inserted their names, the screen should look like showed in Figure 4. Player one's avatar appears with a green toggle and player two's avatar with a blue toggle. The top ready button is active, since player one can no longer change their information and is now ready to play. If player two pressed the **enter key**, then he can now press the ready button, and the game will begin. Player one should not be looking at the screen at this point, since the words to draw will appear in the screen after the ready button is pressed.

3. Drawing Game

At this point, **player two** will be presented with **three words** randomly chosen from a word bank, to select a word the player must click it with the **left mouse button** (Figure 5).

Figure 5

The **game** will work as following, after a word is chosen the player will have **60 seconds** to attempt to draw what the word represents, using the given paint tools (the paint tools will be better explained in the free draw explanation, in this topic we will try to focus only on the game mechanics aspect).

After the **60 seconds** are over or if the player presses the **enter key**, a guess pop-up will appear, allowing the other player to attempt to guess what word was drawn. If the player is done writing the word, he can press the **enter key** to check if it is correct, if it isn't, the word will be **erased**, and the player can give it another try. If **30 seconds** have passed or if the player has guessed the word correctly, then the scores will be calculated and appear on the screen what each player as earned. If the player didn't guess the word, neither will receive any points. The **point system** works as follows, if the player guessed the word, then both players will receive **150 points**. An additional **100 points** can be earned by the player who guessed the word depending on his performance. For each second it

took him to guess the word a point is lost from the **100 bonus points** and an additional 10 points are lost for every wrong attempt. If all 100 points are lost then the player who guessed will just receive the base 150 points, the same as the player who draw. After the earned points are displayed and added to the total score, the player that guessed will now choose a word from **three** new random words and will now be the one attempting to draw. This is repeated for **three rounds**, in each round both players draw and guess one time.

Figure 6

At the end, the player with the **highest score** will appear on the victory screen (Figure 7), showing his **avatar** and **player name**. Balloons move around on the screen **colliding** between themselves and the Minix window borders for a few seconds. Because this screen is the only one where the mouse input is ignored, since we didn't see a use for it, the date icon doesn't appear, and the current date and time can't be seen. After this the program returns to the **main menu**, with the same options as when the program begins.

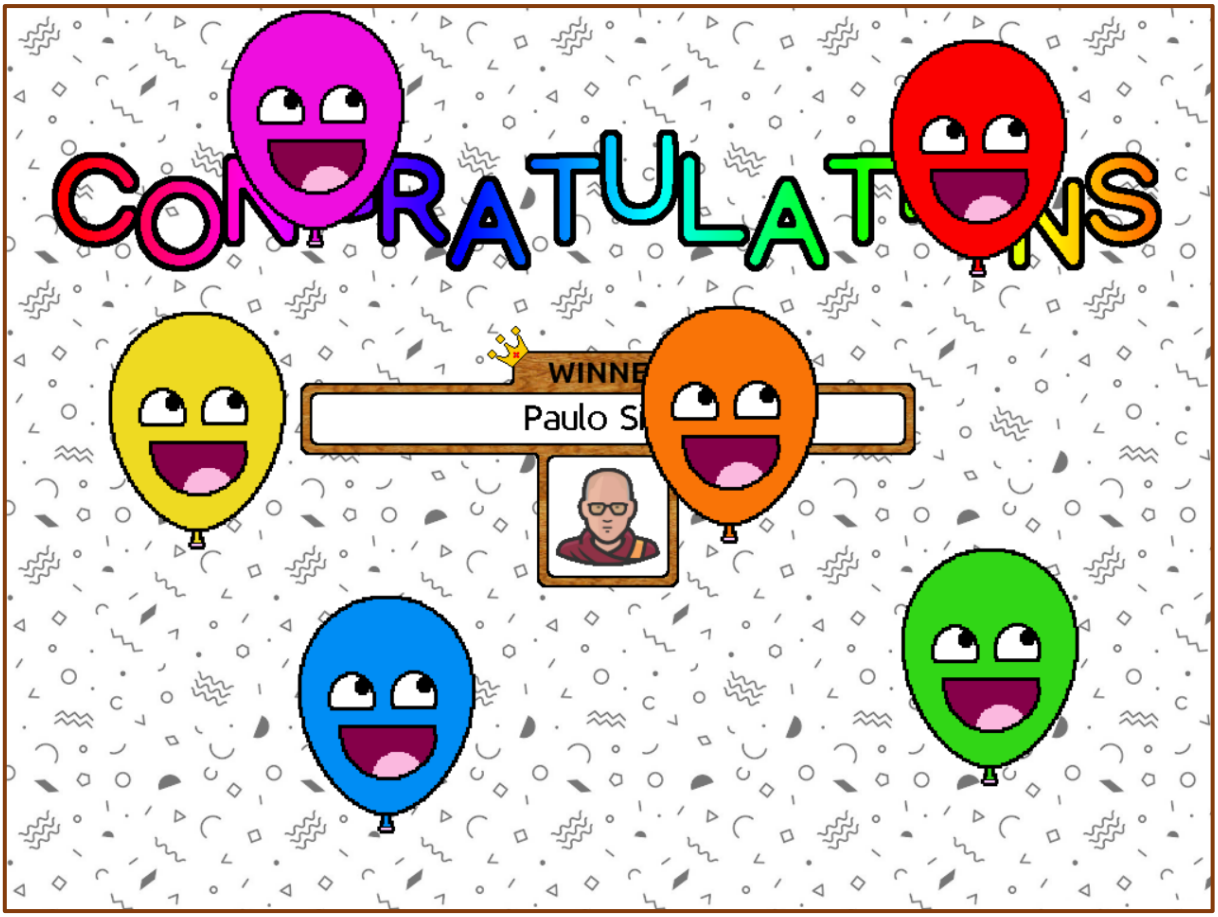


Figure 7

4. Free Draw

When the Draw button is pressed in the main menu, a paint screen, similar to the one during the draw game screen, is shown (Figure 8). Since there doesn't need to be any room for the stats display, the canvas in this mode is a bit larger, providing the user with more space to draw in. In this mode the user is free to draw whatever he wishes, using the **paint tools** provided by the buttons on the right side of the canvas. The **default** is the **pencil tool** (top button) that will draw a rounded line mimicking a pencil or a dot if there is no mouse displacement while the **left mouse button** is pressed. The tool immediately bellow is the **pen tool**, when used it draws a sharper line resembling a fountain pen. Next is the **spray tool**, simulating a paint spray can, it draws a circle pattern with random pixels painted within that circle giving the mentioned effect. Following this, there is the **bucket tool**. When selected and the left button is clicked within the canvas limits, it paints the area where the cursor position is confined. The following tool is the **eraser tool**, and as the name suggests it paints a white rounded rectangle mimicking an eraser. Lastly, there is the **trash tool**, which is just a button, that simply clears the canvas, no matter what was painted there, painting it white. Except for the trash

tool, every other tool is used by pressing the left mouse button and dragging the mouse pointer around to draw.

Figure 8

All paint tools are shown in Figure 8. The user can also change the **colour** used to paint in pencil, pen, spray and fill functionalities. This is done by moving the mouse pointer to either the **coloured** or **grey palette** so that the tip of the cursor is hovering the desired colour and then clicking with the **left mouse button**. No matter the current colour, the eraser and trash tool will always paint **white**. Also, the mouse pointer icon for each of the tools, while inside the canvas, is different depending on the tool and the colour (except eraser tool), see Figure 9 (**1** - pencil, **2** - pen, **3** - spray, **4** - fill, **5** - eraser). If the **"R" key** is pressed while the user is drawing, the colours will change, gradually looping through the colour palette every interruption, resembling a **rainbow** (*easter egg*).

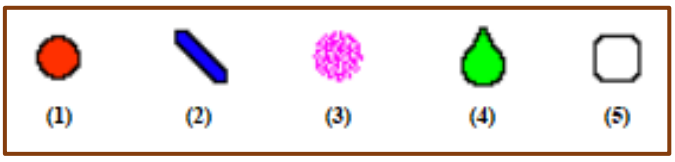


Figure 9

There is also a **return button** bellow the grey palette that when clicked returns to main menu. To avoid accidentally clicking this button while selecting other tools or colours, in order for this button to activate the **right mouse button** must be clicked instead of the left mouse button.

5. Exit Program

When the **exit button** is pressed with the **middle mouse button**, the program will terminate clearing used memory and resetting all peripherals devices to their default mode. It only works with the middle mouse button to avoid accidental clicks. While in the program, no matter where, pressing the **ESC key** will exit the program.

Project Status

Peripheral	Use Case	Interruptions
Timer	Updating current program state, counters and animations. Used to update the RTC display when active.	Yes
Keyboard	Inputting usernames and word guesses. Used certain keys for other actions, such as exiting the program.	Yes
Mouse	Used to navigate menus, interact with buttons and drawing.	Yes
Graphics Card	Program graphics.	Doesn't apply
Real Time Clock	Used to read current date and time. Set off alarms.	Yes

1. Timer

This peripheral is used to update the status of the program, by calling the function *update_state()*, that, based on the current program state and other peripheral events, makes the necessary changes. Every **60 interruptions**, a boolean is set to true indicating a **second** has passed, this is relevant information for updating the RTC and global counters used for program events. **Cursor, drawings and other animations** are also dependent on timer interruptions as well as **collision detecting** in the victory screen. It's the backbone of the program, managing how/when all information in the program is handled.

2. Keyboard

When there is an interruption by the keyboard, after handling it, the keyboard events are updated using the function *update_keyboard_event()*. This updates the internal structure used in the program in order to, in the next timer interruption, deal with the relevant events and executing the necessary functions. This is mostly used for **word input**, since, in the program, the user can be asked to write a username, or to guess a word. It's also used to check for the **ESC key** to exit the program, the **enter key** to confirm actions and the **R key** for drawing *easter egg*.

3. Mouse

Probably the peripheral **most used** by the user, in every interruption, after it's handled, the function *update_mouse_events()* is called and, in the same way as the keyboard, it updates the internal structure used by the program, so that the program can act accordingly depending on **mouse button events** (pressed or released) and **mouse displacement**. Since mouse interruptions are slower then timer interruptions, mouse events are also dealt by the program itself (in events.c), to prevent, for example, multiple reads of the state **RELEASED** in the time interval between the mouse interruptions. Since our project relies significantly on the user painting and drawing, this peripheral is heavily used and crucial for a good user interface with the program.

4. Graphics Card

In this program we use mode **0x144**, that has a **resolution of 1024x768** with **32 bits per pixel**, the first byte is used for **transparency** and the remaining three for the **RGB components**.

For a better user experience, we implemented **double buffering**, preventing flickering of the screen. For this there is a copy of the **videocard** in an array of the same size and all changes are first made in this array and when the function *double_buffer()* is called, it copies the values from this array into memory using *memcpy()*.

There are quite a number of sprites used in this program and, even though most where initially taken from the internet, all of them where modified at least to some extent using GIMP, and some where made from scratch using this software.

Given the fact that the program allows the player to draw, what is currently on the screen can't be predicted, therefore, the sprites can´t simply be redrawn every interruption, for example, if the user moves the cursor, since information would be lost. To circumvent this problem, every change made to the screen is also changed in an array with the same size as the videocard (different from the array used for double buffering) but excluding all changes made by the cursor. For this there are special functions to **draw** and **erase** the current cursor, different from the ones used for the other sprites. This way, a copy of the screen without the cursor interference, is always kept and, when the cursor needs to be moved, the pixel values **on this copy** are inserted in memory where the cursor previously was, and the cursor is redrawn in the new position.

The user is allowed to write, using the keyboard, on the screen, and so there are fonts and special character sprites to allow this feature.

Collision detecting was also implemented in the victory screen with balloons moving around the screen (similar to an old school windows screen saver). The balloons can collide with the window's

edges or between themselves. Depending on the type of collision (horizontal or vertical) the correct speed components are updated and the screen is redrawn.

5. Real Time Clock

This peripheral is used to read the **current date and time**. Even though it has separate functions to read the date and the time, the program only needs to call one function, *get_time_string()*, that receives an array of char's of necessary size, reads the current date and time, and formats it into a string (dd/mm/yyyy hh:mm:ss). Using the function sprintf(), no matter the current time, the string will always have the same length and fills missing characters with 0's, for example, when the month is 8, it converts it to 08. It's also used to generate alarm interruptions in order set an accurate time interval for showing the victory screen. The alarms can only be set for time intervals less than 60 seconds, since that was enough for what we used them in our program. Only alarm interruptions are used because they're the ones that seemed more adequate for what we were looking for.

Code Organization/Structure

1. Mouse Module

This module contains functions that directly manipulate the **mouse peripheral** (i.e. the mouse's interrupt handler, the mouse's interruptions subscribe and unsubscribe functions and a function that issues commands to the mouse registers). These functions were developed during the lab classes for **lab4**, and little was added to them in the project development. However, the mouse's interrupt handler function now does tasks that previously (in lab4) were done in the interruption loop, in order to simplify the code in the mentioned loop.

2. Player Module

This module contains an implementation of a **player class** and the functions used to manipulate said class. We utilize this class exclusively in the **PLAY mode** of our game in order to store information about each player, this includes their **name**, **avatar** and the **points** earned during the game. Therefore, we developed functions that construct an empty player (with no information), set the name of a player, set the player's avatar and to update the player's score.

3. Queue Module

In this module we have an implementation of a **queue** that is based on Professor João Cardoso's (jcard@fe.up.pt) implementation, that we had access to via the slides made available in the following link's PowerPoint (https://web.fe.up.pt/~pfs/aulas/lcom2018/at/17oo.pdf). However, a small modification was made for the queue to store a **pair**, a type defined by us that store the **coordinates**, of both the x and the y axis. This module is used in the paint fill functionality, in order to store the coordinates of the pixels, that need to be analyzed, in the selected confined area.

4. Timer Module

The timer module contains the functions that were developed for **lab2** during the lab classes (i.e the timer's interrupt handler and the necessary functions to subscribe and unsubscribe the timer's interruptions). Even though this is a very simple module it has a considerable weight in our project, since the timer peripheral is crucial for the program's state machine updates and actions.

5. Words Module

This module contains functions that are very important for the **interaction** between the player and the game, more specifically, this module adds to our game the capability of displaying, on the screen, the player's **keyboard inputs** when needed, including the possibility to use **backspace** if he makes a mistake (this functionality is explained in further detail in the chapter "Implementation Details"). Other than this, it also contains the **word bank** for our game, which is where the words that can be given to a player to choose from, during the **PLAY mode**, are stored. The functions responsible for randomly selecting words from the word bank are also implemented here.

6. MACROS Module

This module was developed during the **lab classes** and the **project development**. It consists of a set of **macros** used all through out the other modules in order to improve code readability. It has some general-purpose macros and a then specific ones, for each of the used peripherals, apart from the timer.

7. i8254 Module

This code was imported from **lab2**, it was made available to us during this lab by the **professors**, and it has the set of macros specific for the **timer peripheral**. There were no significant changes made to this file and therefore we should not be credited for it's development.

8. Keyboard Module

Most of the functions of this module were developed during the lab classes for **lab3** (i.e the keyboard's interrupt handler, the subscribe and unsubscribe functions for the keyboard's interruptions, and the function that issues commands to the keyboard controller). However, a functionality of our program was implemented on this module during the development process. The function *key_handler()* is used to **map** a **keyboard key scancode** into the corresponding **character**, based on a keymap made by us (this functionality is explained in further detail in the chapter "Implementation Details").

9. RTC Module

This module contains functions capable of reading the **real time clock**'s current status, allowing the program to access the current **date and time**. The information is parsed and stored in a **struct** (TIME), that can easily be accessed. This module has a key feature of having a function that, given an array of correct size, it stores in it the current date and time read from the RTC in a user-friendly way, ready to be displayed and taking into account values less than 10, by filling the position on the left with a **0**. It has the necessary functions to subscribe/unsubscribe from the **RTC interrupts**, enabling/disabling **RTC alarm interruptions** and setting an alarm interruption.

10. Sprite Module

This module is one of the most important in our project, since **sprites** are used throughout the whole project, and are fundamental in order to allow the user to draw. Our implementation of the sprites is based on an implementation by Professor João Cardoso (jcard@fe.up.pt) and is available in the following link (https://web.fe.up.pt/~pfs/aulas/lcom2018/at/14sprites.pdf). However, many of the functions were developed by us, and others were significantly altered, in order to work with our implementation of the **cursor** and **paint functionalities**. The ones that were made little to no changes to are *create_sprite()* and *destroy_sprite()*, and therefore we shouldn't be credited for the implementation of these functions.

11. Graphics Module

This module is responsible for manipulating the **pixels** individually in order to display the images and it's heavily linked to the **sprite module**. Besides changing the value of a pixel, it also allows to **retrieve** the colour of a pixel that is being displayed. Furthermore, it is here where the functions that allow for **double buffering** are implemented.

12. Images Module

This module contains **header files** that have all the **xpms** used in our program, such has **background images**, **buttons**, **avatars**, different **mouse pointers**, **letters** and special **characters**. This collection of images kept growing throughout the whole project development and, although at first, we attempted to group related xpms in their own **header files**, it became increasingly harder to do so due to the variety of xpms used. It's important to note that even though many images were taken

from the internet, all have been modified to some extent using the image manipulation software GIMP, which took up a considerable amount of time from the development process.

This module is made up of several header files (**alphabet.h**, **avatars.h**, **background.h**, **buttons.h**, **Images.h**, **main_menu.h**, **menu_buttons.h**, **paint_canvas.h**, **paint_menu.h**, **pallets.h**, **setup_menu.h** and **victory_menu.h**) each containing only xpms. However, in order for a file to access these xpms, it's not necessary to include them all, it must only include the **Images.h** header file. This is because this file already includes all the other ones, making it easier for a file to ensure that it can use all of this module's images.

13. Events Module

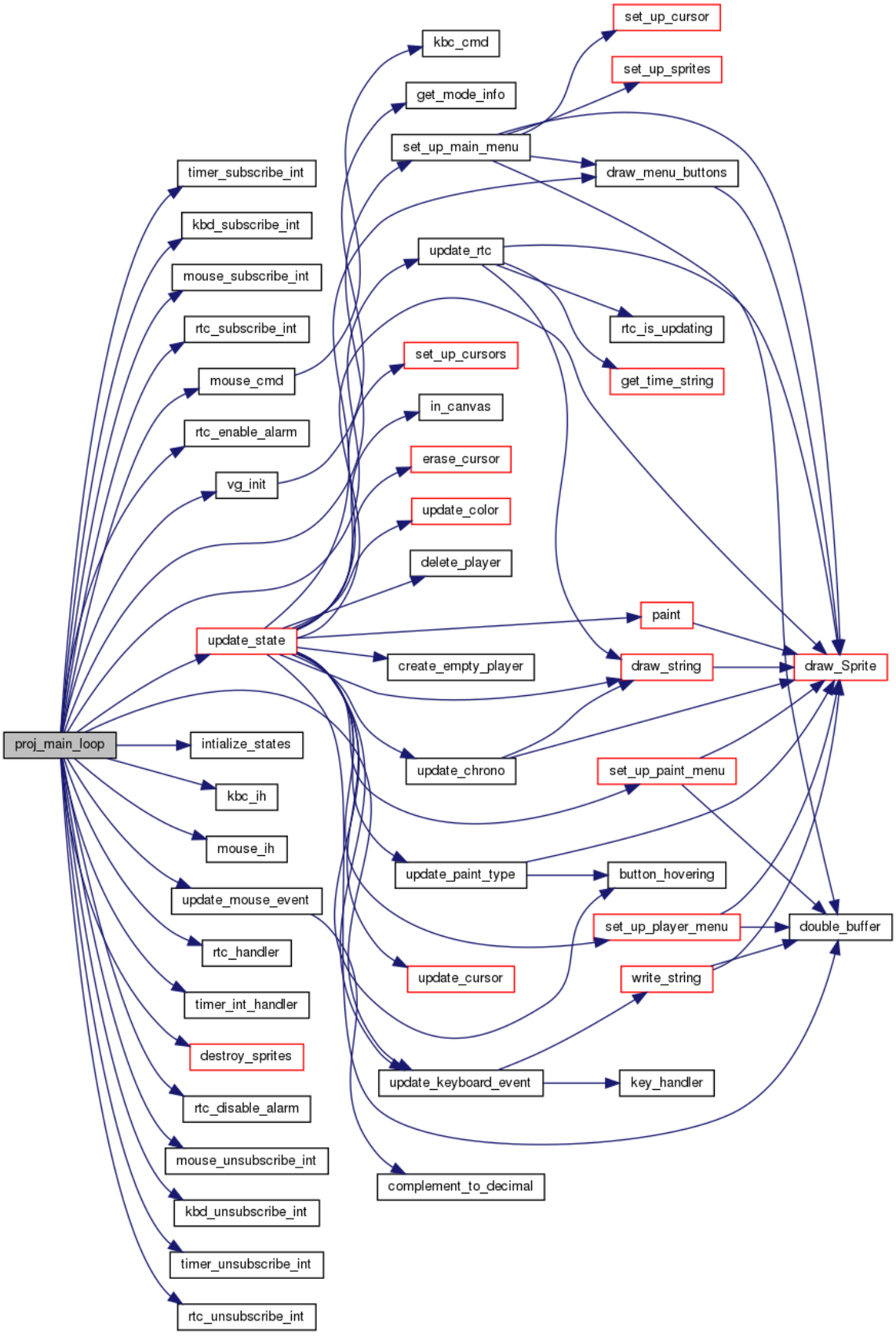
This module is the backbone of our project since it's here where the **state machine** is implemented, as well as any function responsible for updating and handling the program's status. Therefore, this is where the program and peripherals states are stored. Further details on the operations of the state machine are explained in "Implementation Details".

Module's relative weight and member contribution

Modules	Relative Weight (%)	Contribution (%)	
		Bernardo Santos	Tito Griné
1. Mouse Module	10	50	50
2. Player Module	6	100	0
3. Queue Module	3	100	0
4. Timer Module	8	60	40
5. Words Module	8	100	0
6. MACROS Module	2	50	50
7. i8254 Module	1	0	0
8. Keyboard Module	8	70	30
9. RTC Module	4	0	100
10. Sprite Module	12	0	100
11. Graphics Module	10	30	70
12. Images Module	8	30	70
13. Events Module	20	50	50
TOTAL	100	49.3*	50.7*

* These percentages don't take into account the 1% of the project (i8254 Module) that we consider having little to no contribution to.

Function Call Graph



Implementation Details

For this project, as suggested, we implemented a **state machine** in order to organize the code and help with its **modularity**. With every peripheral interruption, a **struct**, and some **global variables** are updated accordingly, keeping track of the meaningful events that occurred. In every timer interruption the state machine checks the existing information available and, depending on it's current program state, **parses** what is useful information and performs the necessary changes. Perhaps this is better explained with some examples.

When the program starts, the current state is the **INITIAL_MENU** by default. When in this state, the program is looking mostly for mouse events. If there is a **displacement**, it updates de cursor position as necessary always checking the window's limits. It also checks if any of the available buttons are being **hovered** and **animating** them if so. It checks for the mouse buttons state to see if any were pressed while hovering a button, which may result in changing the internal state of the program. Keyboard events are also important since if at any point the user presses the **ESC key**, the program will terminate immediately. Lastly if the mouse pointer is hovering the **date icon**, in the bottom left corner of the screen (see Figure 2), then the program will read the date and time from the **RTC** and display it on the screen for as long as the mouse button hovers the icon.

Depending on the current state, to move from a state to another, the program may check for specific **button clicks** (screen buttons), **key presses**, **alarm interruptions** or **counters** that reached 0. The program has 9 different states, 7 of them are related to the drawing game.

Some states require a more detailed verification of peripheral events, for example, when in the **DRAW** or **PLAY** state, the player can draw on a canvas and it's where the program has more to deal with. It must check if the player is hovering the canvas or not, updating the cursor sprite accordingly, if there were any button presses that change the paint tool or the colour the user wants to paint with. Much like the main menu, it must also check to see if the user wants to check the date at any point. One of the best examples of the way the program has to deal with multiple events it's when using **rainbow mode**. If the mouse pointer is on the canvas, the **left mouse button** is pressed and the **R key** as well, then the program will iterate through the rainbow colours as it paints, giving an effect as shown in Figure 10. At the same time, it may have to update a counter in the case of **PLAY**. We had to find ways to simplify as much as possible the way the program deals with all these events so as to maintain a smooth and pleasant user experience.

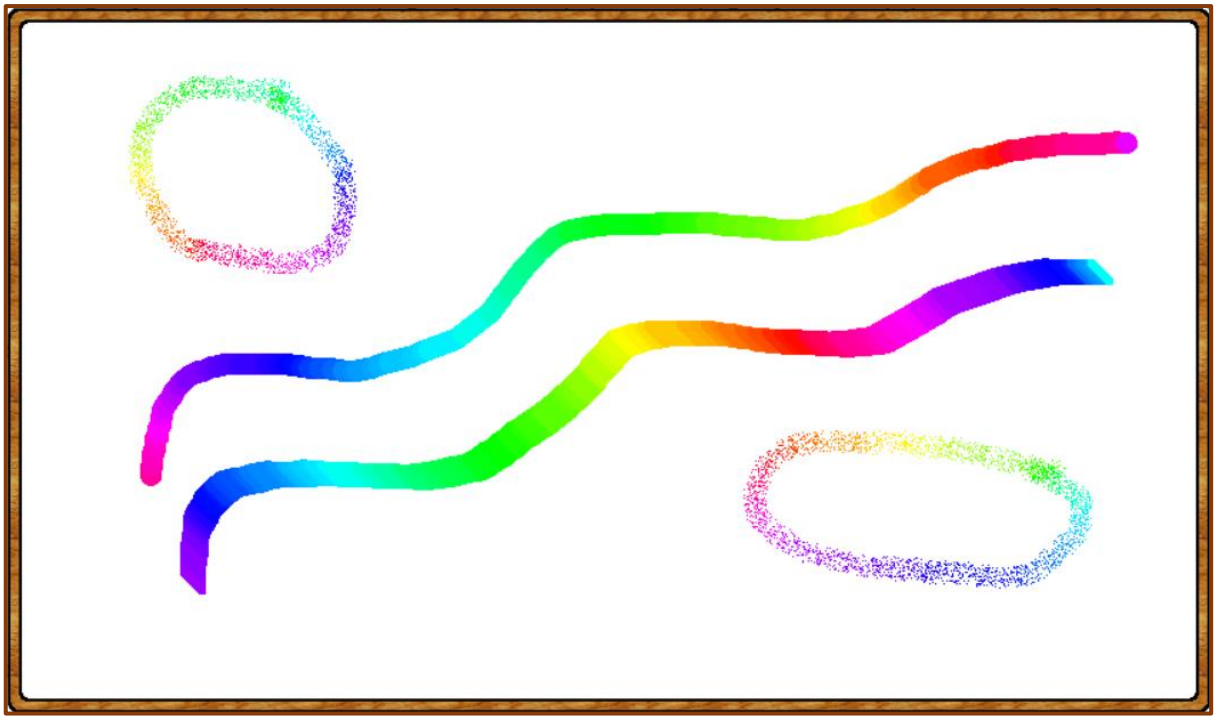


Figure 10

However, not all states require this level of event handling, being the case of the **END** state, which is the state after a game of PaintMix is played, where there is a victory screen, with the winner's avatar and username and balloons move around the screen. In this specific case we did not consider relevant for the user to be able to move the mouse around since there is nothing to interact with. So, in this particular state, the program is only concerned with **keyboard key presses**, namely the **ESC key**, for exiting the program, or the **Enter key** to go straight to the main menu, and RTC alarm interruptions. This victory screen is also where **collision detecting** is most noticeable. Although through out the whole program, "collisions" of the mouse cursor with the screen borders and buttons are accounted for, this might not be considered actual collision detecting. However, in this screen, where the balloon sprites move around, see Figure 7, the program checks every timer interruption if the balloons have collided between themselves and/or the screen borders. This was done by creating two functions, *detect_collision()*, that given a sprite, checks if it has collided with the window limits, changing the sprite's x and y speed accordingly, and *detect_collision_pair()*, that receives two sprites and checks if there were any collisions between the two, by seeing if their horizontal positions and vertical positions have coincided at some point. If so, it checks whether the **collision** was **horizontal** or **vertical**, and changes both sprites **vertical** and **horizontal speeds** if necessary. It's also in here were we take advantage of the **RTC alarm interruptions**. When the game ends and the state machine goes to the **END** state, an alarm interruption is set to go off 10 seconds after the current time read from the RTC.

If the player does not press the Enter or Escape key before, the alarm interruption is read, and the program returns to the main menu.

As mentioned before, the program allows the player to use the keyboard to **input characters** that are displayed on the screen in real time. This functionality was implemented, primarily, to make the game more customizable, letting the players choose their own usernames, and to make the guessing part of the game possible. The player can then guess multiple times and if needed can delete characters by pressing the **backspace** key. The development of this functionality was interesting for the most part, but rather tedious when creating the xmps for the entire alphabet and special characters.

There are three main steps the program takes for writing a character on screen. The first part's role is to get a **scancode** every time a key is pressed, this was developed during the lab classes and is of the keyboard interrupt handler's responsibility. The second part converts a given scancode into a char. This is done by interpreting the scancode as an index to a **keymap** (array of keys) and returning the key on that position, which will then be turned into a char, taking into account the state of **caps lock** and of both the **left** and **right shift** keys. The last section displays each character on to the screen after a key press and, in case of backspace, it will erase the previously displayed character. For each character to be displayed, the corresponding **pixmap** is obtained by using its **ASCII** code as an index to an array of xpms, it will then create a sprite, draw it, and destroy it as to not waste unnecessary memory.

Another challenge we faced was discovering a way for the mouse pointer to not erase what was on the screen as it moved from one location to another. Because we use the mouse cursor in practically every program state, and since we allow the user to **paint**, we quickly realised we couldn't just redraw every sprite on the screen every time the cursor moved, for we would loose changes the user could have done. Therefore, to resolve this, we created **specific functions** for drawing and erasing a **cursor sprite**, different from the ones used for the rest of the sprites. The main difference is that, for normal sprites, they are drawn in an array with the same size as the video card memory, but cursor sprites are not. This way, whatever changes happened in the screen, there is always a copy of the screen that has no **mouse pointer interference** and as such can be used to know what was previously on the screen until the mouse cursor "covered" it. It's in this array where the function to erase the mouse cursor gets the pixel values to replace it.

To prevent screen flickering, which we considered to be very damaging for a paint/drawing program, **double buffering** was implemented by simply making the changes first in an array, of the same size as the video memory, and only at the end are **all** the values of this array copied into memory using the *memcpy()* function.

As we already mentioned, the current date and time aren't displayed at all times and will take this opportunity to explain why. This decision is purely for aesthetic reasons. We just didn't see the utility in having the current date and time being displayed at all times and found it didn't look particularly well. Therefore, we decided to only show it when the user so chooses, which is done by simply hovering the date icon that is present in most screens (except victory screen).

When the program terminates, all peripherals used are **unsubscribed** and **disabled**, in the case of the **mouse** and the **RTC alarm**. All **memory** occupied by the **sprites** used is also freed, not matter if a sprite was or wasn't drawn at any point during the program.

We consider important to point out that, in order for the program to look the way it does, we had to learn how to use the **GIMP software**. This was a trial and error process to perfect most of the program sprites down to the **individual pixel**, spending more time then maybe we should have on this task. However, with this, a good skill was learnt, that may come in handy in future projects.

Conclusions

Looking back on this semester and more specifically on this unit, we are surprised on how much was learnt in these few months. It's a course unit that requires a lot of work by the students, especially when compared to other units of the same semester and the amount of credits attributed. In the first few weeks it's very easy to feel overwhelmed, and it's difficult even to ask teachers for help since there are times where we don't even know what exactly is it that we are not understanding or need to know. After this initial hurdle is passed however, it becomes more enjoyable to work and it becomes very apparent how available the teachers are to help us. Besides the fact that sometimes the time constrains are a little to harsh, our only suggestion would be to try and make sure every student is able to surpass this initial stage as soon as possible, otherwise it becomes increasingly harder to do well on this unit as time goes on,

Although this project was a challenge, much like each lab assignment, it's the first time where we actually feel like we had to work on something of our own, having to come up with solutions to our specific situation and design decisions, for how the program was going to look like and what would it allow the user to do. For this both members had to work many hours in order to achieve all goals set in the begging and making sure the final product was as good as we could make it, given the time constrains. While stressful at times, it was a fun project that made us feel more confidant on our programming abilities. Our only regret is to not have had enough time to implement the serial port in our program as we hoped would be done. Overall, we come out of this project with a positive outlook on this whole experience.