Faculty of Engineering of University of Porto

MiniPicty

Computer Laboratory
Class 13 - Group 5

Developed by:

- Félix Martins up202108837
- Francisco da Ana up202108762
- João Pereira up202108848
- Tomás Vicente up202108762

Introduction

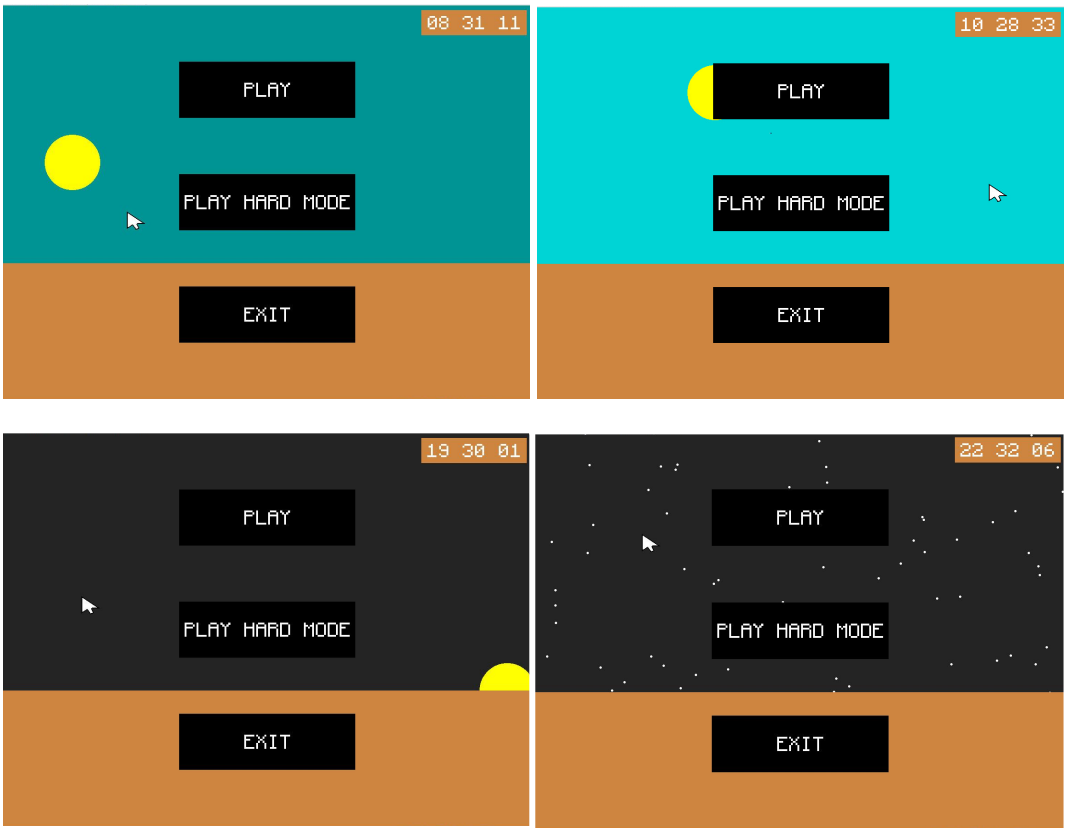
We have decided to develop the game **MiniPicty**, based on the classic Pictionary. It is an interactive game for 2 players that allows us to explore the potentials and functions of various devices that players can use in a clear and concise manner.
Each player has a different role in the game. While one player draws the image corresponding to a word generated by the game on the screen, the other player tries to guess the word based on the drawing within a certain time interval.

User instructions

Initial Menu

Once the program starts, the players are presented with a menu containing 3 buttons: **Play**, **Play Hard Mode**, and **Quit**. To select a button and execute its function, the players must use the mouse to move the cursor and click on it.

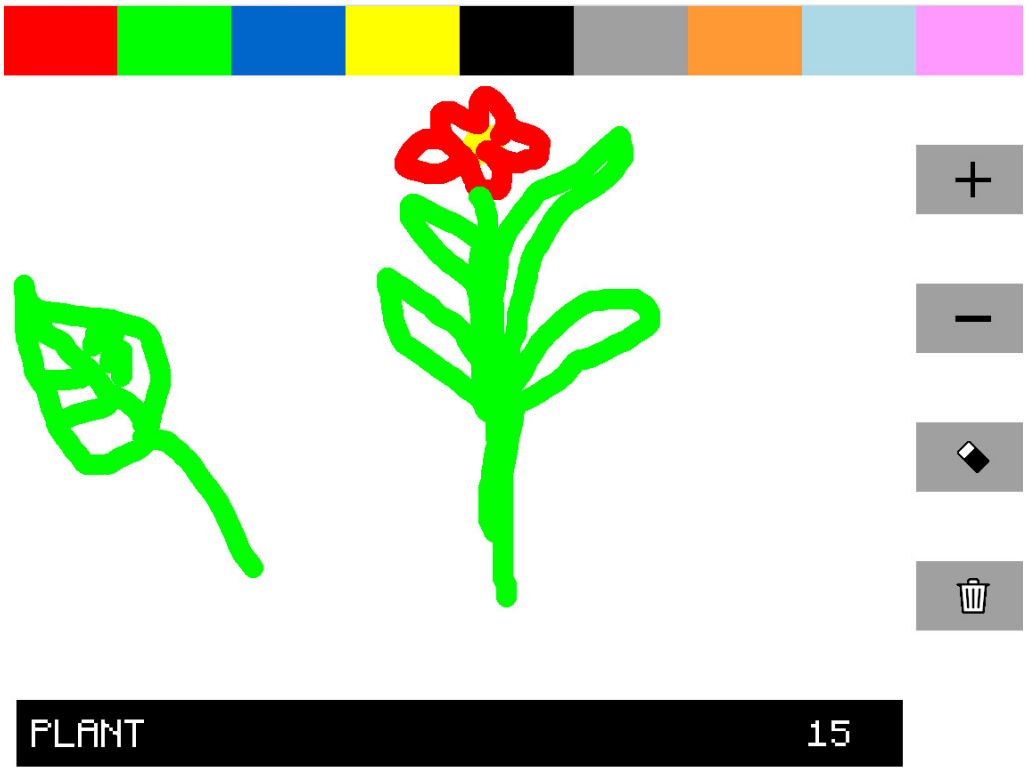
The background of the menu is an image that adjusts according to the time of day when the program is being executed. During the day, there is a sun with a blue sky that positions itself based on the time, while during the night there are stars. The colour of the sky is also changing during the day.



Game: Drawing

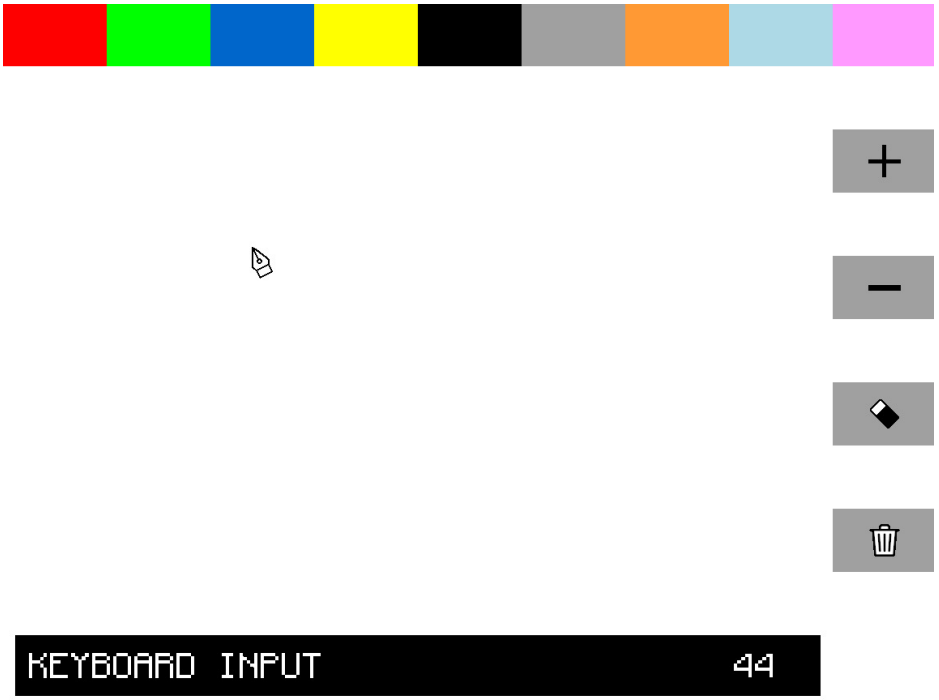
The player who is drawing has access to a palette of 9 colours and several other useful drawing tools, including the ability to increase and decrease the brush size, an eraser, and an option to clear the entire drawing.

The black rectangle at the bottom of the screen contains the word to draw on the left and a timer with the remaining time of the round, in seconds, on the right.



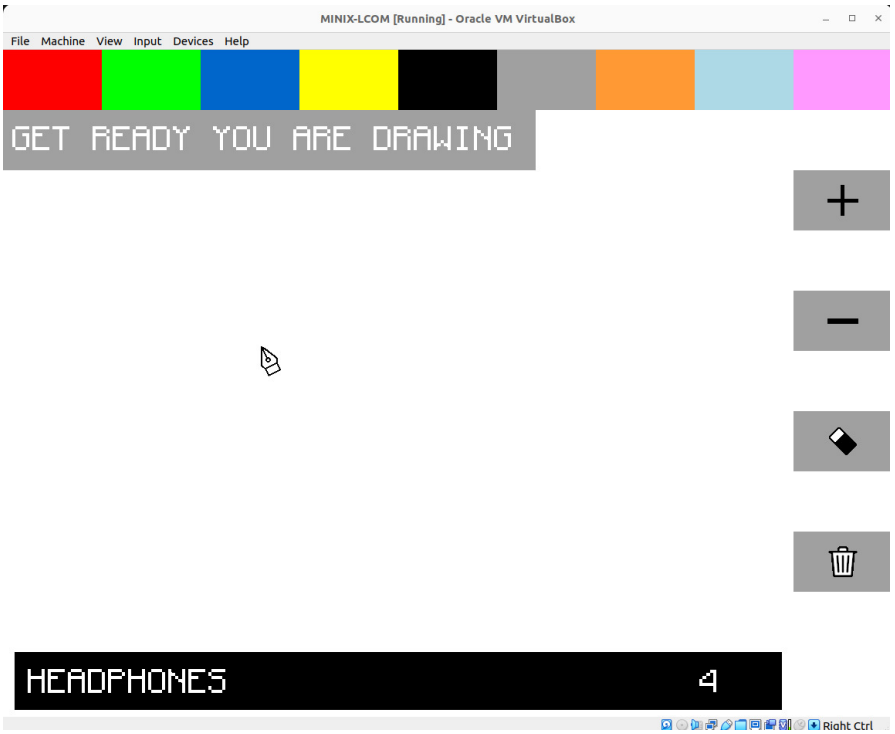
Game: Guessing

The player who is guessing only has access to the drawing being made by the other player, the timer indicating the remaining time for that round, and a text box where he should write his attempts to guess the word behind the drawing.



Game: Waiting

Before each round starts, a message is displayed to the players indicating their roles for that round. The message is shown in the upper right corner of the drawing area and remains there for 5 seconds. Once this time has passed, the round begins.

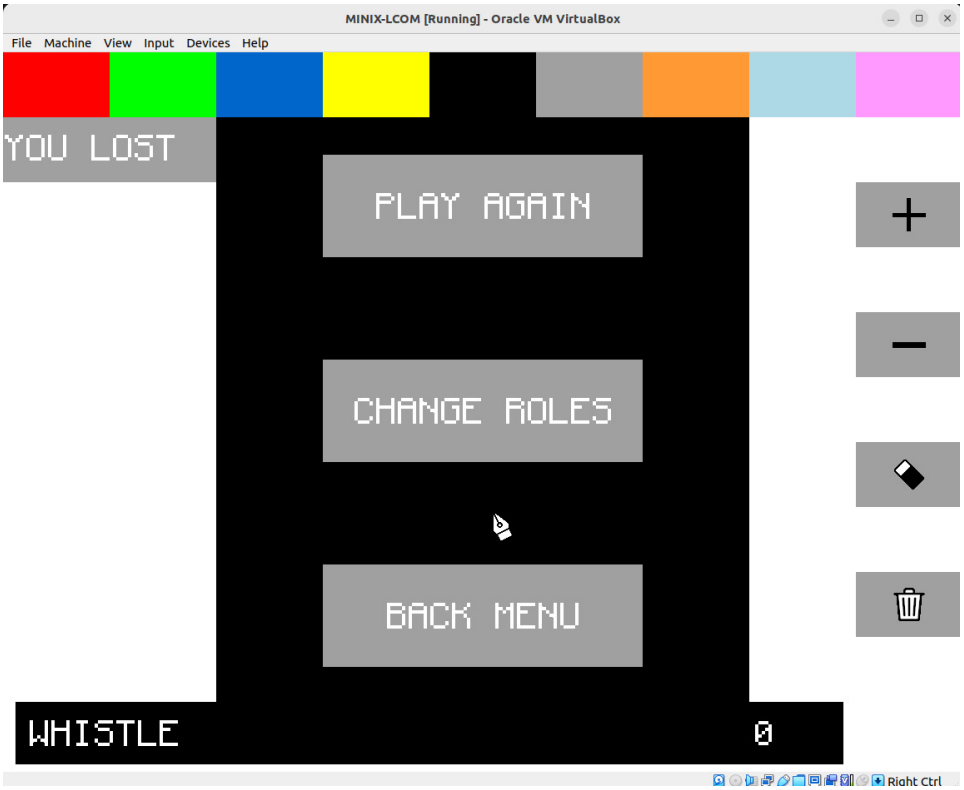


Game: Finish

After a round ends, the result is printed in the upper left corner of the drawing area. Additionally, three buttons are shown for the users to select based on what they want to do next. These are the buttons:

1. Play Again
2. Play Again with different roles
3. Quit

At this time, only the player that was previously drawing can move the mouse. Additionally, the mouse cursor is kept from the previous round. We would have liked to make this a little more intuitive with both players moving the mouse and the cursor updating as expected, but nonetheless, it works fine with these restrictions.



Game: Hard Mode

This mode works in the same way as the normal game (one player draws a picture and the other tries to guess it). The only difference is the presence of an asteroid. This asteroid prevents the player from drawing inside it and blocks vision of what is underneath. It bounces off the walls and makes it harder to paint and guess.

Project Status

Functionality Table

Features	Main devices	**Implementation state**
Menu navigation	Mouse	Fully implemented
Painting on screen	Mouse + Video Card	Fully implemented
Choosing different colours to draw	Mouse + Video Card	Fully implemented
Changing brush radius	Mouse	Fully implemented
Clearing the painting screen	Mouse + Video Card	Fully implemented
Showing the word to the drawer	Video card	Fully implemented
Text input from the guesser	Keyboard + Video Card	Fully implemented
Limiting and displaying round time	Timer + Video Card	Fully implemented
Having a “get ready” time	Timer + Video Card	Fully implemented
Hard mode with an asteroid and collision with mouse	Video card + Mouse	Fully implemented
Menu with background scenery	Video Card	Fully implemented
Showing current time on the menu	RTC + Video Card	Fully implemented
Communication between two players	Serial port	Fully implemented

Devices Table

Device	What for	Interrupts
Timer	Controlling frame rate and game countdown timer	Y
KBD	Writing guess words (text input)	Y
Mouse	Clicking menu and game buttons, drawing	Y
Video card	Display in screen, menu and game	N
RTC	Reading time and update interrupts	Y
Serial port	Communication between two players using FIFO	Y

Devices

The devices are located at "`devices/<name_of_device>`". We made all the interrupt handlers application-independent, meaning they do not handle game logic or anything of the sort. The game logic will have to access the values "retrieved" from these interrupt handlers to use them.

Timer

The timer was mainly used for controlling the frame rate. It was also used for the countdown timer in the game rounds, as seen in `game_process_timer`.

Keyboard

The keyboard was solely for receiving character inputs for the guess word (text input). Functions such as `translate_scancode` show this.

Mouse

We used the mouse packets to obtain mouse positions and to tell if the mouse had the left button pressed, to press buttons and to draw.

Function `get_drawing_position_from_packet` shows our processing of the packet.

Video card

The mode we picked for the project was a direct colour mode: `0x14C`, with 4 bytes per pixel and a resolution of `1152 x 864`.

We have moving objects: mouse and asteroid (asteroid only in hard mode). The sun in the menu also moves (but only from hour to hour). We have animated sprites on the mouse and the asteroid, but they are only animated through events and not also through timer. We would have liked to do both for the asteroid, but time was restricted.

Furthermore, we have "pixel perfect collision" with the mouse and the asteroid.

Triple buffering

We have implemented this through page flipping (`vg_buffer_flip` which is called by `game_draw` and `menu_draw`)

An optimization we developed for this was having a flag that tells us whether it is necessary to draw things again. If nothing new happens on the screen, we do not draw again and stay in the same buffer - uses `buffer_needs_updates`. More on this later in Implementation details.

Font

Since a user writes from the keyboard to the game directly while guessing a word, we decided to make an `XPM` for each letter and compose the string with those `XPM`s.

VBE functions

We used function `0x07` - Set start of display, for page flipping with triple buffering. Other functions like “set `VBE` mode” and “get mode information” are also used in our program.

Real-time clock

The `RTC` is used for reading date/time and for updating finished interrupts for one second intervals. After getting stable values from the `RTC`, we simply increment one second to our time value at each interrupt.

Serial port

Features used - `FIFO`s with interrupts, for both reading and writing (and also “polling” for writing). Further explanation is provided later on Section Implementation details - Serial port.

Communication parameters - 115200 bit rate, 8 bits per char, 1 stop bit, odd parity.

Data exchanged and exchange frequency - We are exchanging mouse positions instead of packets to facilitate synchronization (explained more in depth later on Implementation details). Additionally, we communicate guess word index, index of buttons pressed and correct guessing of the word.

Code organization/structure

Here we have a table with the description of each C file/module and the relative weight.

Modules table

Module	Path to Module	Relative Weight (%)	Description & Data Structures
Game	`modules/game/`	15	Has all the information related to the drawing game. It deals with the 3 game states - `WAITING`, `PLAYING` and `FINISHED`. The game has some buttons, defining their attributes. For the playing state, the buttons are the colours, rubber, increase/decrease brush size, clear screen. For the finished state, there are three more buttons: “Play again”, “Change roles”, and “Quit to Menu”. Additionally, it has the round timer (timer that is counting down from a certain value until it reaches 0 - means, for example, the end of a round) and the guess word of the player that is guessing. Related to this, it also holds the correct word to test whether the guess is correct. Furthermore, it has a canvas that it draws in and a boolean to identify whether it is in the hard mode. Being a state of the application, it defines some of the functions of a state, assigning them to the state in `transition_to_game`. These are `game_process_timer`, `game_process_mouse`, `game_draw`, `game_process_serial`, `game_process_keyboard` and `game_get_buttons`.
Menu	`modules/menu/`	6.5	Has all the information related with the main menu. It has the buttons of the main menu and defines their attributes, such as their `onClick` functions. Since it is a state of the application, it defines some of the functions of a state, assigning them to the state when calling `transition_to_menu` (`menu_process_mouse`, `menu_draw`, etc.). It also has the `background_scene`, which it simply copies to the VRAM buffer when drawing (`background_scene` is of type `Background*`, i.e. `uint8_t*`)
Player Menu	`modules/menu/player_menu/`	0.5	Defines a `struct` with a player. It is similar to the player but used only for the menu.

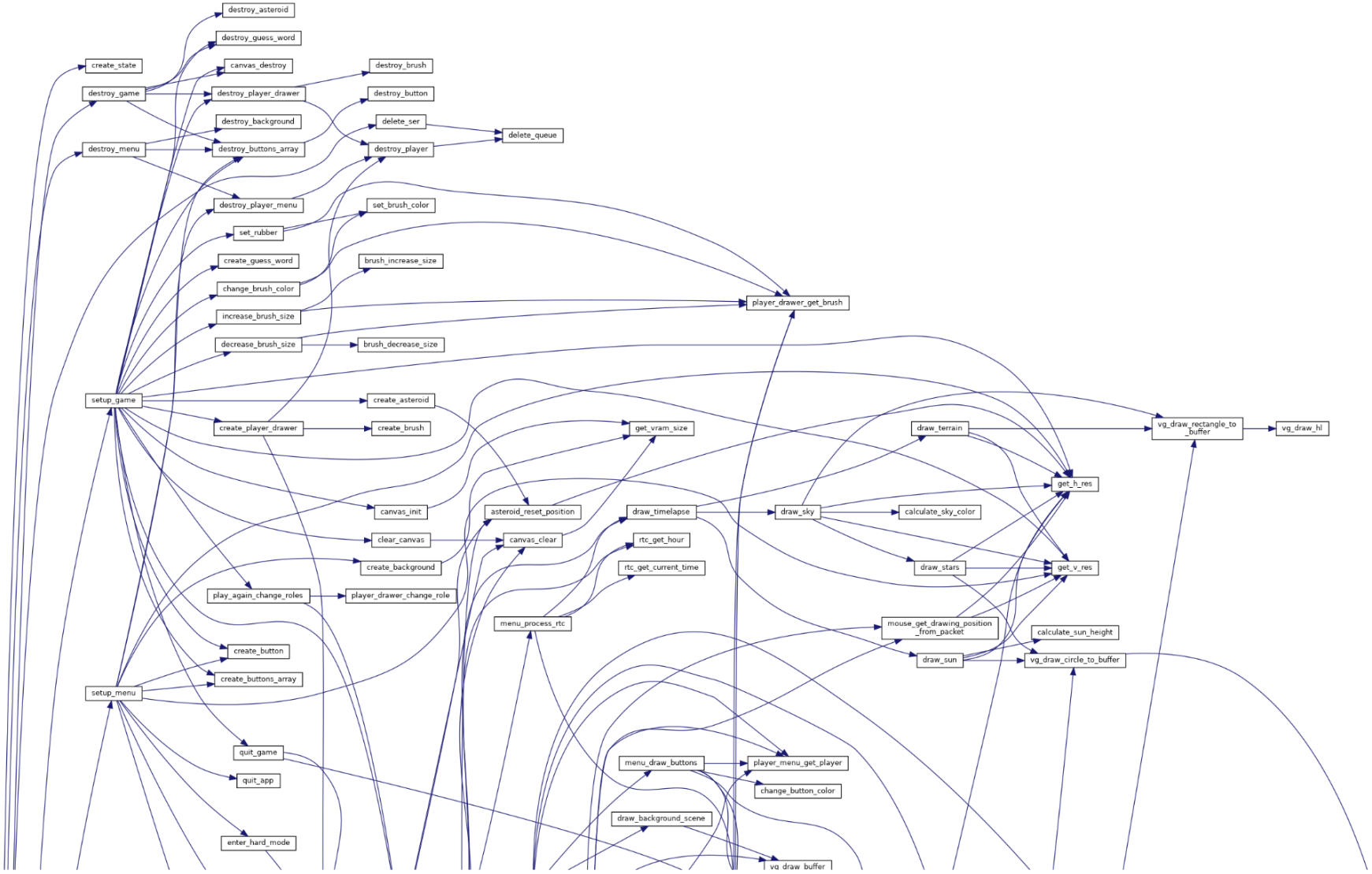
Module	Path to Module	Relative Weight (%)	Description & Data Structures
Brush	`model/brush/`	1	Defines a `struct brush_t` with a colour and a radius. Example functions are `brush_increase_size` to increase the radius and `set_brush_color` to set the brush's colour.
Player	`model/player/`	3	Defines a `struct player_t` with the last mouse position and a queue of positions. Additionally, there is an `enum player_type_t` that identifies whether a player is drawing or not. The most important functions in this module are `player_add_next_position` that adds a position to the queue, `player_get_next_position` that pops and returns a position from the queue and `player_get_current_position` that returns the position at the back of the queue.
Player Drawer	`modules/game/player_drawer/`	2	Defines a `struct player_drawer_t` with a player, a brush, a `player_state` (`player_type_t`) and a cursor (`cursor_type_t`). The `player_state` identifies if it is the current user drawing or the other user. The cursor identifies the cursor `XPM` to draw. Examples of methods are `player_drawer_change_role` that changes the `player_state` (from `SELF_PLAYER` to `OTHER_PLAYER` and vice versa) and `player_drawer_set_cursor` that sets the cursor's `enum`.
Guess	`model/guess/`	3	This module defines a `struct guess_word_t` that holds the current guess's characters. It has methods to add a character to the guess word (`write_character`), delete a character (`delete_character - backspace`), to check if the guess word is the same as the correct word (`validate_guess_word - enter`) and to remove all characters from the guess word (`reset_character - enter`).
Queue	`model/queue/`	3	Defines our `queue` data structure. It is a limited size generic circular queue. Has methods to pop the queue, access the back element (last inserted element) and to push elements to the queue.

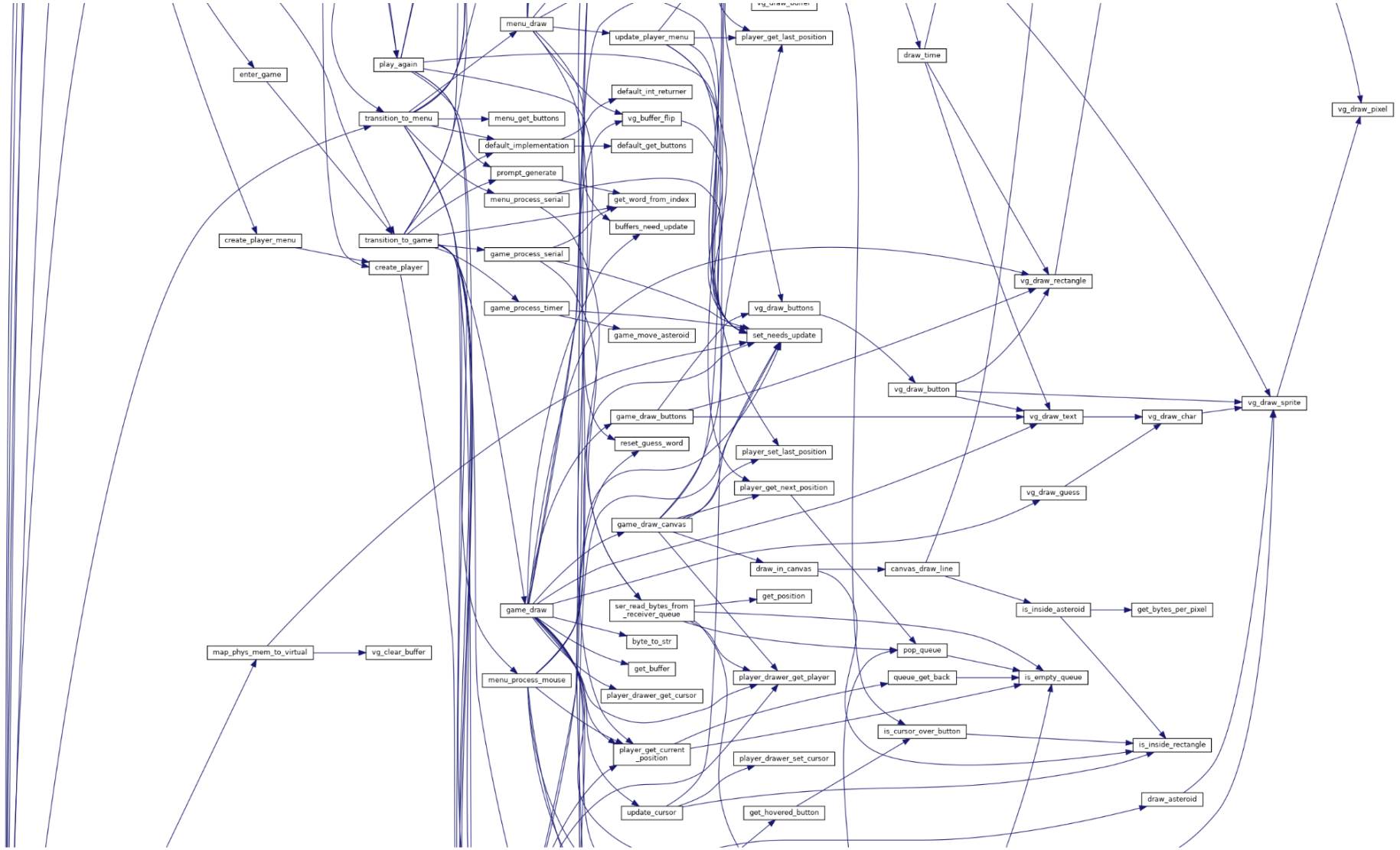
Module	Path to Module	Relative Weight (%)	Description & Data Structures
Sprite	`model/sprite/`	2	Defines a struct `Sprite`, with an `XPM`. This module is only responsible for loading `XPM`s and freeing them.
State	`model/state/`	2	Defines our abstract state, with default implementations for the state functions. Has functions like `get_buttons`, `process_mouse`, `process_keyboard` ..., a boolean to tell if the app should keep running and the current correct word index. This module is used for the state pattern, and our concrete states are `Menu` and `Game`.
Canvas	`model/game/canvas/`	4	Defines a `struct canvas_t` where the drawings are painted. It has the methods for drawing lines between 2 positions, taking into account the asteroid in the case that it is present.
Interrupts	`modules/interrupts/`	0.5	Defines a function to subscribe all interrupts and another to unsubscribe them.
Background	`modules/menu/background/`	4	Defines a `Background struct` that holds the menu's background. This is for fast copying in the menu (instead of drawing, just uses `memcpy` once)
Position	`model/position/`	2	Defines a `struct position` with x and y coordinates. It is also responsible for some useful general functions that handle positions. Also defines a `struct drawing_position` with a position and a boolean determining if the mouse was pressed for that position.
Serial Port	`devices/serial_port/`	14	This handles everything serial port related. It has functions that initialize serial port parameters, subscribe interrupts and create queues for receiving and transmitting Furthermore, it handles our program's communication protocol, by sending and interpreting the bytes read from the serial port as per the protocol.

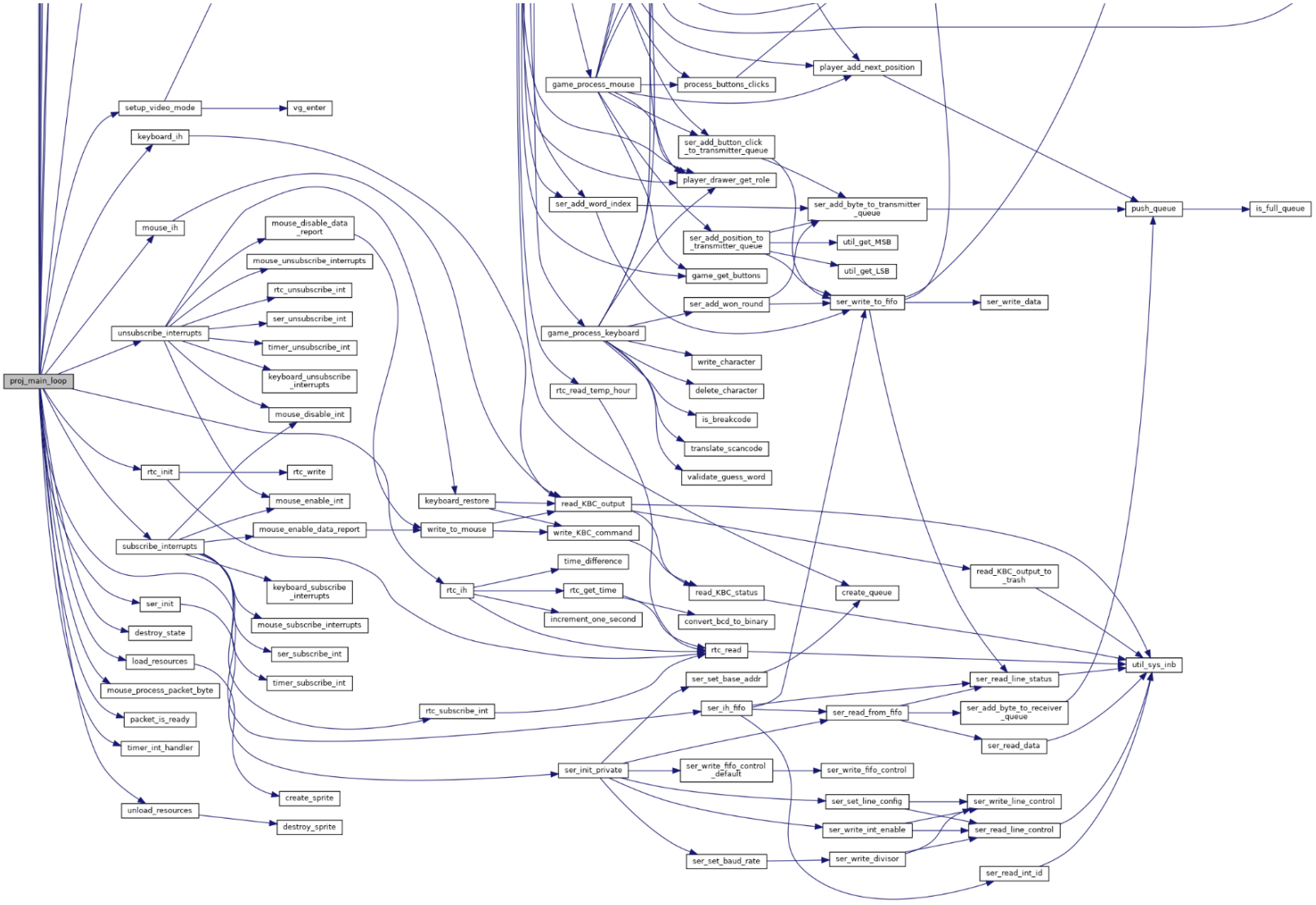
Module	Path to Module	Relative Weight (%)	Description & Data Structures
Real-time clock	devices/rtc/	3	Defines functions that interact with the RTC. Examples are (un)subscribing interrupts and the interrupt handler (rtc_ih). Some other important functions are rtc_get_time which is a private function to get the hours, minutes and seconds and rtc_get_current_time, a public function to get the time in the format "hh mm ss", which is used in displaying the time in the menu.
KBC	devices/controllers/kbc/	2	This defines our program's interface into the KBC. Defines functions to read the output from the KBC, and write commands to it, as well as reading the KBC status register.
Mouse	devices/mouse/	6.5	Defines functions that interact with the mouse. Examples are (un)subscribing and attending interrupts, and processing packets by returning positions.
Keyboard	devices/keyboard/	4	Defines functions that interact with the keyboard. Examples include (un)subscribing and attending interrupts, and processing a scan code by returning the corresponding character.
Timer	devices/timer/	1	The timer simply has functions to subscribe, unsubscribe interrupts and to attend them (timer_int_handler).
Video Card	devices/video_card/	7	Implementation of triple buffering with vg_buffer_flip. Drawing of general objects: circles, rectangles, Sprite. The most important functions are setup_video_mode, vg_draw_text, vg_draw_rectangle, vg_draw_buttons, vg_buffer_flip and vg_draw_pixel.
Button	model/button/	5	Defines a struct button_t with a certain position, width, height, background colour, text, icon and onClick function. Also defines a struct buttons_array_t for an array of buttons. The most important functions in this module are change_button_color, get_hovered_button and process_buttons_clicks (which is basically a "state machine" to determine when to process the onClick function of a button).

Module	Path to Module	Relative Weight (%)	Description & Data Structures
Asteroid	`model/asteroid/`	4	Defines a struct `asteroid_t` that holds the asteroid's position, all four `XPM`s corresponding to the different directions, the x and y speed, and the current index for the `XPM` (which is of type `enum asteroid_type_t`). In this file, we have the algorithm for collision with the asteroid in is_inside
Resources	`modules/resources/`	4	Here, we define a `Resources struct` that has all the `XPM`s used in our program. This module loads those resources into that `struct`, and unloads those resources. The submodules define `XPM` and related enums. The percentage showcases the work of getting these `XPM`s and creating these files, besides getting a way to only have to load them once.
Utils	`utils/`	1	Has functions used by several other modules. The most important functions are from lab2: `util_sys_inb`, `util_get_LSB` and `util_get_MSB`.

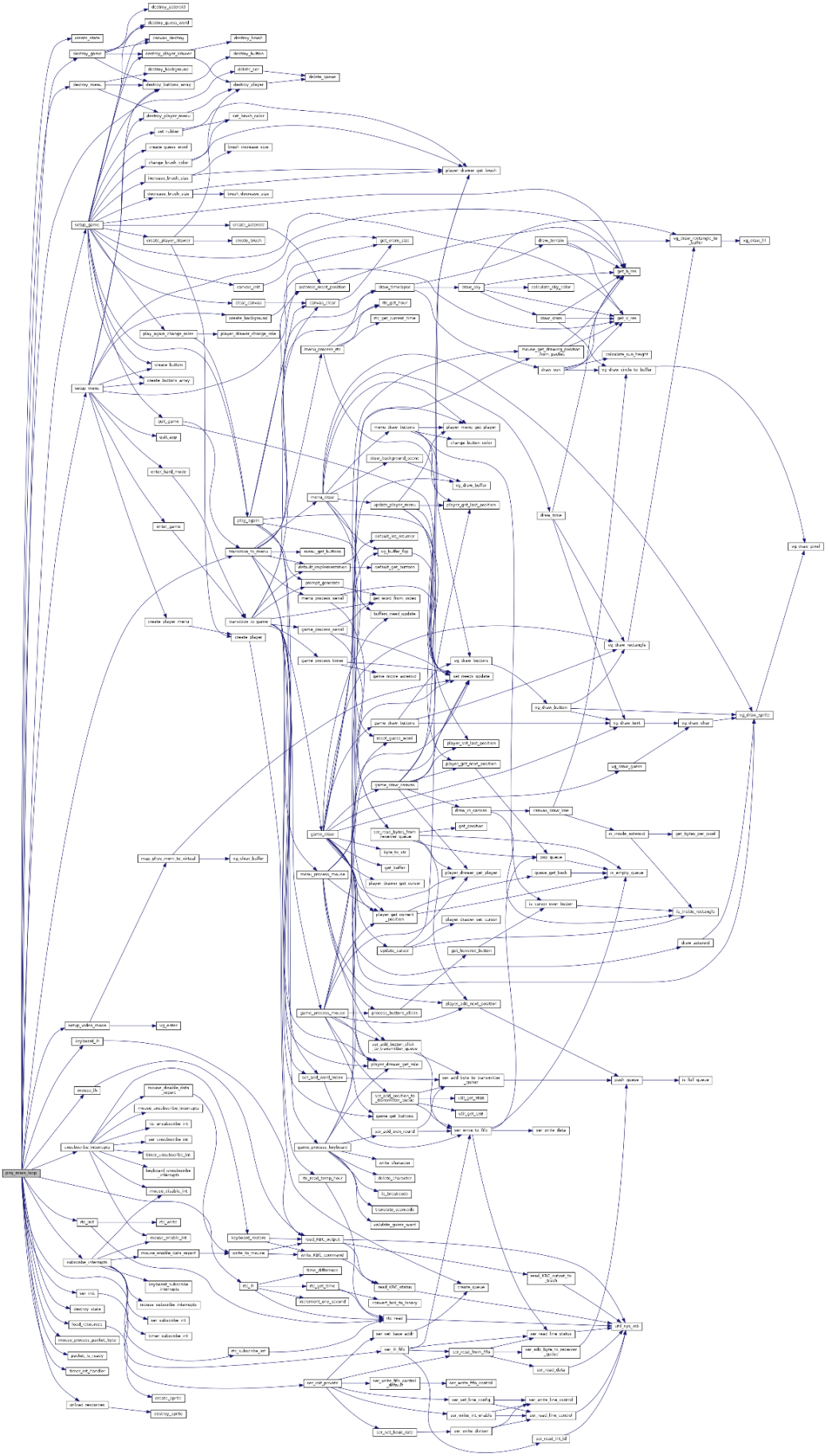
Function call graph




menu_draw
update_player_menu
player_get_last_position
draw_time
play_again
enter_game
default_int_returner
vg_draw_pixel
transition_to_menu
menu_get_buttons
vg_buffer_flip
default_implementation
default_get_buttons
prompt_generate
menu_process_serial
get_word_from_index
buffers_need_update
create_player_menu
transition_to_game
game_process_serial
create_player
vg_draw_rectangle
game_process_timer
vg_draw_buttons
game_move_asteroid
set_needs_update
vg_draw_button
vg_draw_sprite
game_draw_buttons
vg_draw_text
vg_draw_char
reset_guess_word
player_set_last_position
player_get_next_position
vg_draw_guess
game_draw_canvas
draw_in_canvas
canvas_draw_line
get_position
game_draw
ser_read_bytes_from_receiver_queue
is_inside_asteroid
get_bytes_per_pixel
map_phys_mem_to_virtual
vg_clear_buffer
byte_to_str
pop_queue
get_buffer
menu_process_mouse
player_drawer_get_player
queue_get_back
is_empty_queue
player_drawer_get_cursor
is_cursor_over_button
player_get_current_position
is_inside_rectangle
player_drawer_set_cursor
draw_asteroid
update_cursor
get_hovered_button


proj_main_loop
setup_video_mode
vg_enter
keyboard_ih
mouse_ih
unsubscribe_interrupts
mouse_disable_data_report
mouse_unsubscribe_interrupts
rtc_unsubscribe_int
ser_unsubscribe_int
timer_unsubscribe_int
keyboard_unsubscribe_interrupts
mouse_disable_int
rtc_init
rtc_write
mouse_enable_int
subscribe_interrupts
mouse_enable_data_report
keyboard_subscribe_interrupts
mouse_subscribe_interrupts
ser_subscribe_int
timer_subscribe_int
ser_init
destroy_state
load_resources
mouse_process_packet_byte
packet_is_ready
timer_int_handler
unload_resources
create_sprite
destroy_sprite
keyboard_restore
write_to_mouse
rtc_ih
rtc_subscribe_int
read_KBC_output
write_KBC_command
time_difference
rtc_get_time
increment_one_second
rtc_read
read_KBC_status
convert_bcd_to_binary
ser_set_base_addr
ser_ih_fifo
ser_init_private
ser_write_fifo_control_default
ser_write_fifo_control
ser_set_line_config
ser_write_int_enable
ser_set_baud_rate
ser_write_divisor
ser_write_line_control
ser_read_line_control
ser_read_int_id
ser_read_line_status
ser_read_from_fifo
ser_add_byte_to_receiver_queue
ser_read_data
read_KBC_output_to_trash
create_queue
util_sys_inb
game_process_mouse
process_buttons_clicks
player_add_next_position
ser_add_button_click_to_transmitter_queue
player_drawer_get_role
ser_add_word_index
ser_add_byte_to_transmitter_queue
push_queue
is_full_queue
ser_add_position_to_transmitter_queue
util_get_MSB
util_get_LSB
game_get_buttons
ser_add_won_round
ser_write_to_fifo
ser_write_data
game_process_keyboard
write_character
delete_character
is_breakcode
translate_scancode
validate_guess_word
rtc_read_temp_hour

Complete call graph



Main functions

`Proj_main_loop`, located in `proj.c`, is our only function that calls `driver_receive`. It calls the interrupt handlers and the functions to process interrupts from the current `app_state`. Aside from that, it also loads the resources, sets up the game and menu, and subscribes and unsubscribes interrupts (all through functions located at other files).

Implementation details

Serial port

Lower level use - `FIFO`

Transmitter and receiver queues to obtain an application-independent handler. When the game wants to send some bytes, it simply pushes to the transmitter queue (`transmitter_queue`). In the interrupt handler, we simply process the bytes from that queue. Reading wise, in the interrupt handler, we just push to the receiver queue the bytes received. Later, our program can call a function `ser_read_bytes_from_receiver_queue` and process those bytes.

We chose to use `FIFO` with interrupts and polling. We use polling to write bytes for a simple reason: the serial port may raise a transmitter empty interrupt (`THRE`) while there are no bytes to send. Since the interrupt corresponds to the serial port becoming empty, without sending anything, the serial port would remain empty, causing the interrupt to not be raised again. This way, the communication stops until something is written by polling. Therefore, both interrupts and polling (with `FIFO`) are used for transmitting, while only `FIFO` interrupts for reading.

Communication protocol

A very important factor for the drawing to be consistent between the two players was sending the mouse positions and not the packets to avoid losing synchronization of the mouse position. Another thing we've done is reserve a byte in the protocol that means the end of the mouse position, for example. We used this byte several times to our advantage. However, the bytes from positions have arbitrary values, from `0x00` to `0xFF`. For this reason, we decided to not allow communication of the byte `0xFF` and transformed it to `0xFE`. This loses only a pixel of precision, while choosing something like `0x00` to always be `0x01` could be problematic since it can be the most significant byte.

Additional to the mouse positions, we transmit button clicks through indices of the current state's buttons, new word generation also through word indices and, finally, a simple byte that means the guesser identified the word. All of this is accompanied by our `0xFF` special byte, when it is necessary and/or useful.

Triple buffering and page flipping

In order to reduce flickering and visual artefacts, thus increasing smoothness, we have mapped three buffers into the video ram.

To save the canvas painting between the buffers, we created a new local buffer (`canvas_t.buffer`) where the paintings are drawn. To display it, we perform a single `memcpy` system call to place the canvas content into the `VRAM`.

We do the same thing for the menu for performance purposes. In this state, there is also a background scene buffer (`Background`). This is also copied into one of the three `VRAM` buffers when drawing, by using a single `memcpy` to improve performance.

Animating sprites

We are using an enum with indices that point to the semantically logical `XPM` in the array of `XPM`s. This is used with the cursors and the asteroid for handling change of states.

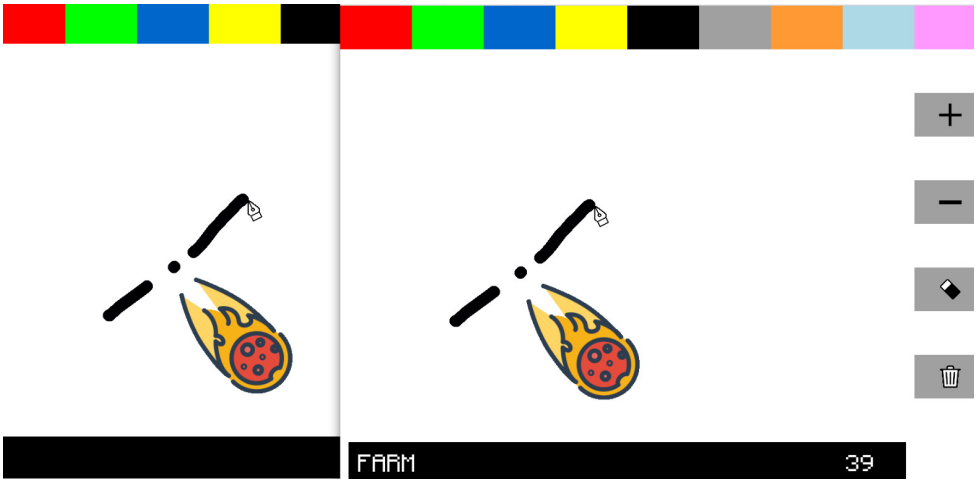
Collision detection and drawing

To check if a position is colliding with a `Sprite` (the `asteroid`), we first check for its boundaries and if the position is outside the limits, it returns false. By doing that we don't need to run the following code as we already know it is not colliding.

Then we calculate the index of the mouse position inside the `Sprite`. The pixel colour of that index gives us information about the collision. If it's transparent, there is no collision, otherwise there is.

This collision algorithm is run every time we are drawing a new circle. To draw the lines, we use Bresenham's line algorithm. We draw a line with a given radius across two following mouse positions by looping through the positions in between them, drawing circles.

In our game, the asteroid moves indefinitely and collides with the `canvas_t` boundaries. Because of that, we also have collision detection between the `asteroid` and the `canvas_t`. In our `game_move_asteroid` method, we test for collisions with the canvas, changing the direction of the movement and animating the asteroid sprite.



Real-time clock

We enabled update-ended rtc interrupts in our program to get one-second difference interrupts and for better read of the time values in the RTC registers (since it gives us 999 ms to read the values).

However, there is still the problem of synchronization with updates of the time values. To ensure our values are correct, we decided to read the time (`hh mm ss`) several times until two following values have an exactly one-second difference. This makes sure that the values readed are not wrong because of asynchronous updates in the RTC time registers. After reading two following correct values in the RTC (with a one-second difference), we simply increment one second at each RTC interrupt inside `rtc_ih`.

Mouse clicking buttons

A user would not expect to click a button when clicking outside the button, moving the mouse to top the button and releasing it. Therefore, a button click is defined by both pressing and releasing inside the button.

This action is processed by a state machine that, upon reaching the final state, executes the `onClick` function of that button. This function is specified when defining the button, according to the functionality we want the button to perform when pressed. As a consequence, we don't need to think about which function should be executed when the final state is reached, as it is already defined and assigned to the button itself.

This state machine of the buttons is implemented by the `process_buttons_clicks function`.

Note that in order to know if a button is pressed, we need to iterate through all the buttons and update the state. Another thing important to consider is whether the previous and current position had the left mouse button pressed or not. All of this related to the state machine is the role of the `process_buttons_clicks` function.

Object-Oriented Programming

We used an implementation of the state pattern in C, as seen in `model/state`, `modules/game` and `modules/menu`. This is done mainly by using functions inside structs and reassigning them when transitioning between states. This is better specified in the following section (State Pattern).

Another more basic example of OOP is in `model/queue`. Here, we have an implementation of a limited size circular queue, used for the serial port and for the player's mouse positions. We implemented this by abstracting the `struct variables` from outside the module, making the fields private.

State Pattern

Following an object-oriented approach, we needed to sum up the general app state, so we could treat the menu and game the same way. Switches were a poor choice for interrupt driven development. For each device, each interrupt would need a new `switch`. This increased the code complexity.

A `struct state_t` was created holding a number of functions, like `draw`, `process_mouse`, `process_keyboard`, etc. Then, we simply store a variable of type

`state_t*` to hold the current state. When we switch to a new state, the state functions are updated with references to new ones.

Now, instead of calling a `switch`, we only need to execute the current function presented in the state. For example, `app_state->process_keyboard(app_state)` for processing the keyboard in the current state.

However, a state might not have a `process_keyboard` event. For this state pattern to work, a default implementation of the state functions was defined to ensure that all the references point at least to a default function.

More reading on this implementation can be found at the “Patterns in C” book by Adam Tornhill where we get our inspiration, referenced in the end of the report.

Optimized draw horizontal lines function

Initially, we decided to implement a draw rectangle function that calls a draw line function that calls draw pixel. The draw pixel function is basically a `memcpy` system call for a single colour (4 pixels in our mode).

For big rectangles, this was an issue due to the excessive number of system calls that was significantly reducing the performance. Because of that, we needed to find some way to optimize the function.

We focus on the draw line function. Why? If we think about it, a line is basically somewhere a continuous sequence of bytes in the virtual memory where the new colour needs to be copied.

So, we get rid of the many system calls for a single line by calculating the start and end position of the sequence in the virtual memory where we need to copy the colour and sequentially setting the new colour.

By optimizing these functions, we saw a big increase in the performance of our game.

Resources load

We have a set of regular `XPM` resources. Instead of loading an `XPM` every time we want to draw it, we decided to load all app resources at the beginning (setup) with the appropriate allocated space and unload them at the end.

Conclusions

Problems and delays that led to faulty/not implemented functionality

- We started by using indexed graphics mode. However, we then decided to change to a direct mode to make colours more flexible and to make colour transparency easier. For example, adjusting the blue sky colour based on time was not possible in the indexed mode.
- Another problem we encountered was one of our buttons was getting arbitrary field values, such as text colour, width, etc. It turns out we were creating the buttons in the stack and assigning them to an array, but they were getting discarded (we believe it was a lack of space since 18 buttons worked but not 19 or 20). We fixed it through

saving them in the heap with `malloc`, but this was somewhat time-consuming to figure out.

- Related to the serial port, we had a problem where we were losing more bytes than we wished because of our communication protocol. To make the protocol more robust, we chose not to allow the byte `0xFF` to be communicated normally, and have a special meaning (as described in Implementation details).
- Another problem we encountered was with the `KBC` having a byte from the mouse when we wanted a keyboard byte, and vice versa (if we inverted the order of the function calls in the `unsubscribe_interrupts` function). Since it was only at the end of the program, we decided to discard any non-wanted bytes and, after that, access the KBC normally.

Things to add in the future

Some things that we would like to add in the future would be:

- Displaying in the pen cursor the colour the user is drawing with.
- Add more action buttons in the right and make all buttons have key binds (e.g. R enables Rubber). Related to this, we could make the key binds runtime modifiable, saving them persistently, so the user does not need to change them every time.
- Add a display of the current brush radius when drawing.
- Expand menu options with a leaderboard and perhaps a possibility to save paintings when in the finished game menu.
- Make a moon appear on the menu when it is night/dark.
- Add the possibility to hold the shift key for drawing straight lines.
- Add more XPMs to the asteroid to make it animated not only when hitting walls, but also as time passes. Furthermore, we could add several asteroids initially randomly placed within the canvas to make the game substantially harder.
- Another thing we could add is the possibility for the user to use control-z to remove his last stroke, and control-y to redo that action.
- Add a tip system to help if the guesser gets stuck (In fixed interval of time, reveal letters of the drawing prompt)
- Finally, we could allow both users to move their mouse in the game finished menu and change/reset the cursor as the user would expect.

Main achievements

Some of the parts of this project we're producer of are:

- Serial communication protocol.
- We have worked with all the available devices.
- The input of a string in run-time, editable and independent of other function calls.
- The approach to allow buttons to be clicked by defining callbacks inside the button.

Lessons learned

During the implementation of our game's serial port protocol, we learned how to develop a protocol, to test it and improve it, taking into consideration tradeoffs and general problems of a protocol.

Also, by programming a full game in C, we learned more about memory management and how the computer reacts to runtime allocation of memory. We experienced some interesting things in this field, and now better understand the difference between Heap and Stack memory.

By not using heap memory allocation, we risk that the content is not as we want. We have a story where we created a button without using `malloc`/heap and the text was completely random. We freak out, obviously. Then, we changed it to use the `malloc` system call, meaning Heap. No related problems have been found since.

Appendix: installation instructions

To run our project, there should be two open virtual machines, and they should be configured correctly with the COM1 serial port. Furthermore, to run the code, please run "`lcom_run proj remote`" in one of them and, only after that, run "`lcom_run proj host`" on the other one. After that, the game should be up and running. Check this reference for the tutorial we used to set up both virtual machines: Computer Labs Project: Serial Communication Setup, Development and Testing.

References:

- Souto, Pedro F. (2018). Computer Labs: OO-programming with C. Retrieved from https://web.fe.up.pt/~pfs/aulas/lcom2018/at/17oo.pdf
    -
- Tornhill, A. (2012). Patterns in C Part 2, STATE. Retrieved from https://www.adamtornhill.com/Patterns%20in%20C%202,%20STATE.pdf
    -
- Bresenham, J. E. (1965). Algorithm for computer control of a digital plotter. IBM Systems Journal, 4(1), 25-30. Retrieved from https://www.cse.iitb.ac.in/~paragc/files/bresenham_line.pdf