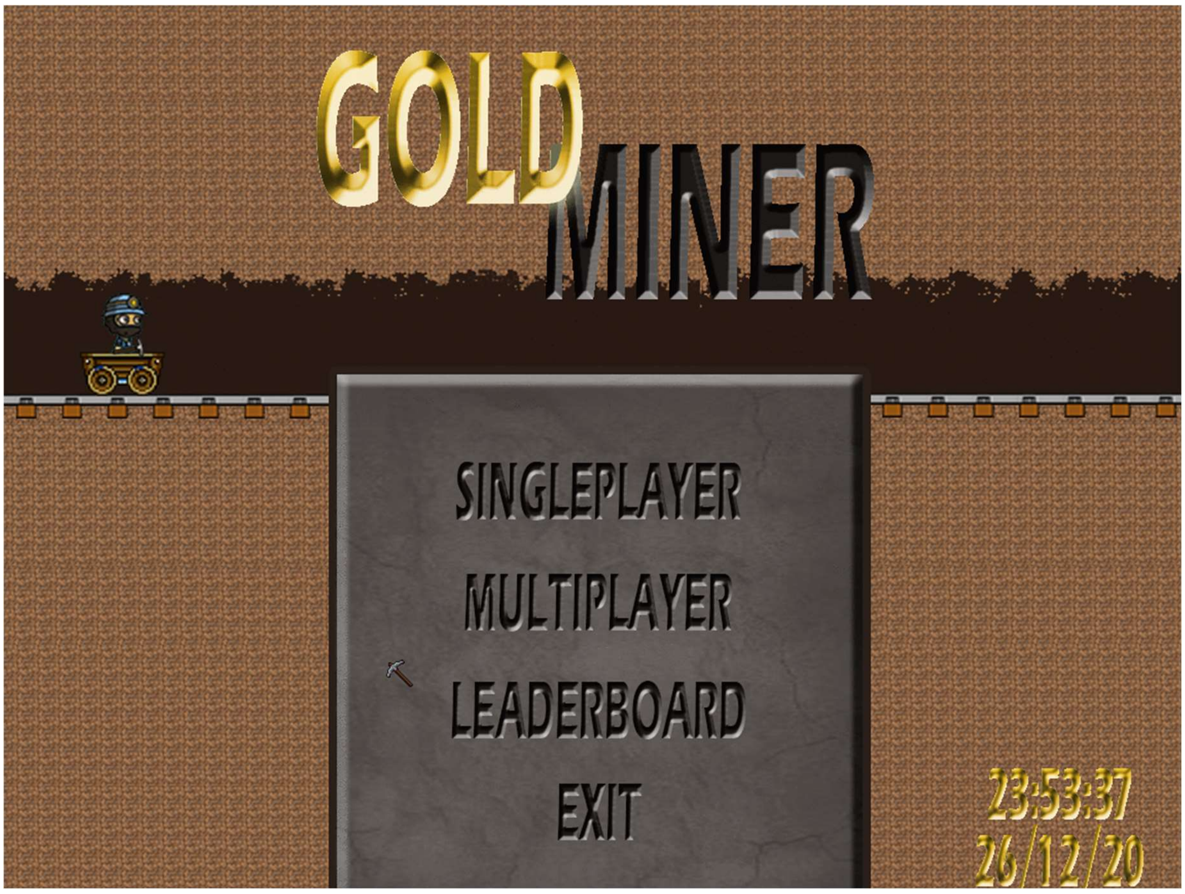
U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

GOLD MINER

Project developed for LCOM

Faculty of Engineering of University of Porto
Laboratório de Computadores 2ºYear MIEIC 2020/2021

Group members Group7-Class1:
André Diogo Bastos Pereira (up201905650)
Matilde Jacinto Oliveira (up201906954)



INDEX

1. User's Instructions

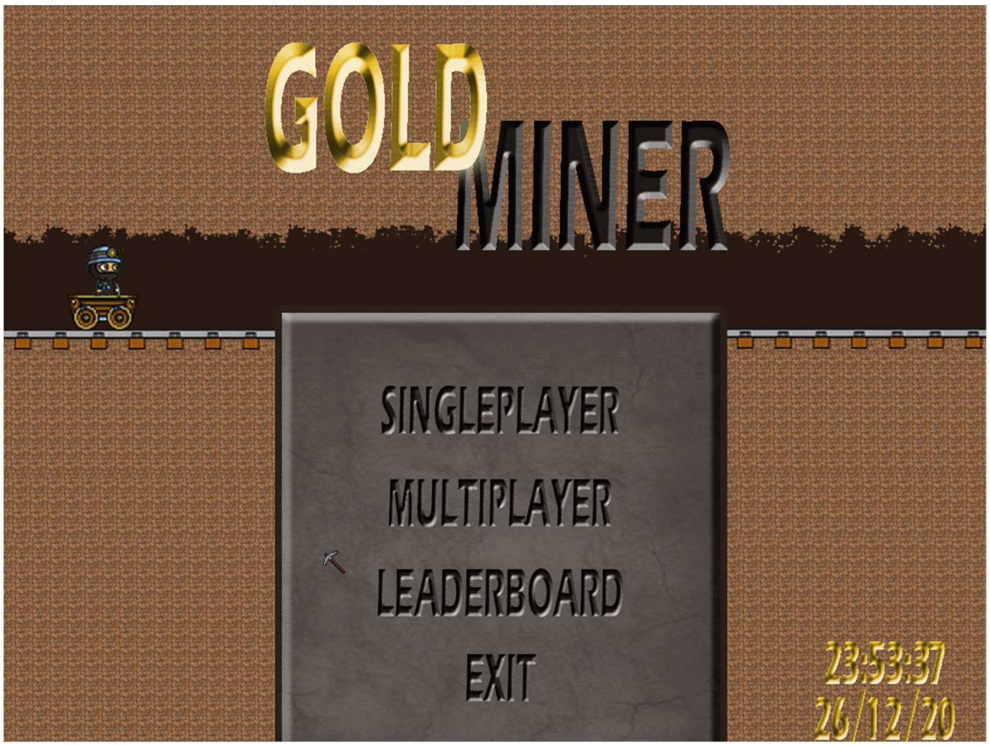
1.1 Game Description

The game objective is to catch and get as many golds as they can, in order to reach a record within a time interval: 100 seconds, for single player.

The player controls the miner's car and its claw, and the golds appear below him. With the help of the claw, controlled by the keyboard, and the pickaxe, controlled by the mouse, the player is supposed to transform the stones in golds by pressing the left button of the mouse over the stone and then catch them by moving the claw over the golds. The player should also be aware that some stones are not golds, but obsidians, and if he catches them his time decreases. In the meanwhile, during the game, there may be displayed some clocks that if catched, will give the player more time.

1.2 Initial Screen

Initially, as you start using the game, the following main menu is displayed and has available four options: SINGLE PLAYER, MULTI PLAYER, LEADER BOARD and EXIT.

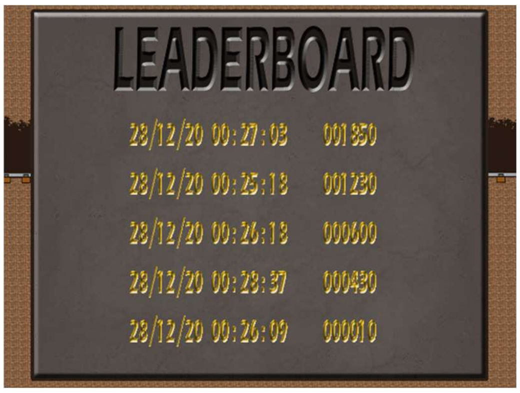


Using the movement of the mouse (represented by the pickaxe) and his left button you can choose one of the options. Additionally, the main menu presents the actual time and date.

By selecting EXIT the program stops running, as that option closes the program, frees all allocated memory and changes video card mode back to text mode.

1.3 Leader Board

If the LEADERBOARD option is pressed, the following board is displayed.

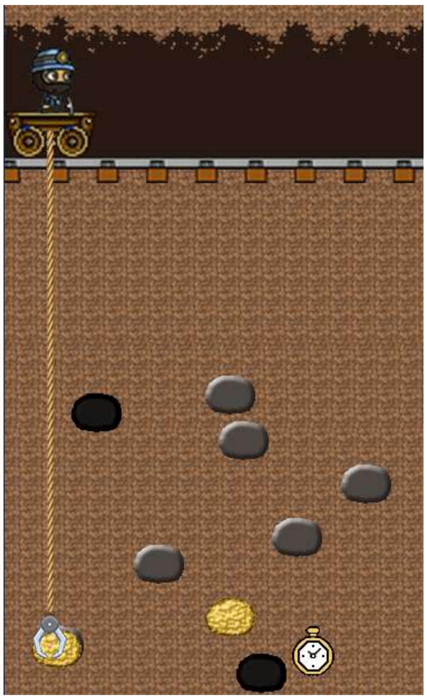


```
LEADERBOARD
28/12/20 00:27:03   001850
28/12/20 00:25:18   001230
28/12/20 00:26:18   000600
28/12/20 00:28:37   000430
28/12/20 00:26:09   000010
```

The leader board presents in descending order of scores achieved, the 5 best scores achieved in that program run with the time and date given by the RTC interrupts.

1.4 Single Player

If the SINGLEPLAYER option is pressed, the game starts and the time in the top right corner of the screen starts decreasing.



The car movement is controlled by the keyboard. By pressing ‘**A**’ the car moves left and ‘**D**’ the car moves right. To control the claw, you must use ‘**S**’ to make it go down and ‘**W**’ to make it go up.

As explained above, behind every stone that appears in the screen there´s a gold. In order to transform it to gold you must press the mouse left button with the mouse over the stone. After that it is prepared to be catched and by moving the claw over it, it is catched and must be brought to the car (you must bring the claw to its initial position right below the car). When the gold is finally catched the points are increased by 50. The player should also be aware that some stones are not golds, but obsidians, and if he catches them his time decreases by 5 seconds.

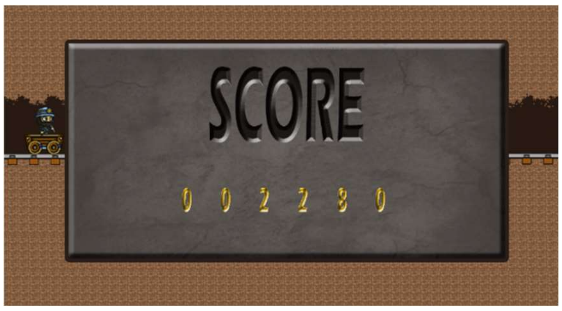
It is important to notice that the game screen is always changing and moving to the left, hence the player must be fast, because he may lose his golds once the screen catches them, they go back to stone. For the obsidians it happens the same, when the screen catches them, they go back to stones.

When a gold/stone/obsidian is catched by the user, a new stone is generated in a random position.

There is another possible catching item: the clocks. If catched they increase the time left by 10 seconds.

It is always possible to leave the game by pressing **'Esc'**.

SCORE BOARD



The score board is displayed when the player goes out of time or pressed the **'Esc'** key to quit the game. It shows the score obtained by the player in that game run. To go back to main menu the user just must press **'Esc'.**

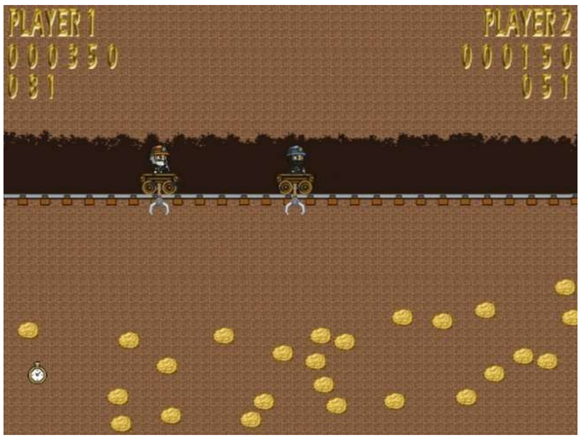
1.5 MULTIPLAYER PLAYER

If the MULTIPLAYER option is pressed, the game leads to a LOADING screen. That can lead to the two different multiplayer modes implemented: **two players** in the **same computer** or in **different computers**.

LOADING SCREEN

As we have implemented two different multiplayer game versions, after pressing the MULTIPLAYER option in the main menu, this LOADING screen is displayed. After that, if the other computer player presses MULTIPLAYER option, they are both redirected to the game screen. The other possibility is to, when in the LOADING screen, the player presses **'Enter'**, and it redirects him to a two-player mode in the same computer.

MULTIPLAYER IN THE SAME COMPUTER



If the MULTIPLAYER option is pressed, the game starts with two miner cars controlled in the same computer. Unlike in single player the competition here is to get more golds than the opponent. The multiplayer game has only golds and clocks, and the winner is the one with the higher score in the end of the game.

The players start with the same time to get has many golds as possible, during the game they are able to catch the golds or clocks.

Each gold catched gives 50 points to the correspondent player and the clocks increase the correspondent player's time by 10.

The game finishes when the two players time run out. If it happens to a player's time to run out first that player is no longer capable of catching golds or clocks, until the game finishes.

The first player car is controlled by the keyboard: by pressing '**A**' the car moves left and '**D**' the car moves right. The first player claw is controlled by the keyboard: he must use '**S**' to make it go down and '**W**' to make it go up.

The second player car is controlled with the keyboard: by pressing the **arrow to the left** the car moves left and **arrow to the right** the car moves right. The second player claw is controlled by the keyboard: he must use **arrow down** to make it go down and **arrow up** to make it go up.

The rules to catch the golds are the same as in the single player.

Multiplayer in two different computers

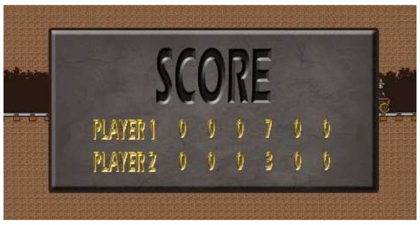
After the MULTIPLAYER option is pressed in both computers, the game starts with two miner cars controlled each by its computer. The game mode is like the multiplayer version in the same computer, having the same game objective and unwind of the game. However, the main difference stands in where each player controls its miner. In both computers, the player is the player one and the opponent is player 2, then the player 1 is always the blue hat miner and the opponent is the orange hat one. The update of scores and time is controlled in the same way as the multiplayer in the same computer version.

Each player car is controlled by the keyboard in its own computer: by pressing '**A**' the correspondent car moves left and '**D**' the correspondent car moves right. The player claw is controlled by the keyboard: he must use '**S**' to make it go down and '**W**' to make it go up.

When the game ends the multiplayer score is displayed and each player is the player 1 in his own computer.

Multiplayer Score Board

The score board is displayed when the two players run out of time or someone pressed the '**Esc**' key to quit the game. It shows the score obtained by each player in that game.



SCORE

PLAYER 1 0 0 0 7 0 0

PLAYER 2 0 0 0 3 0 0

2. PROJECT STATUS

DEVICE	WHAT FOR	INTERRUPTS
Video Card	Displaying the different game screens and menus.	N/A
Keyboard	Moving the miner car and claw. Special keys to navigate between game screens.	Yes
Mouse	Choosing options in the menu. Transforming the stones in golds/ obsidians.	Yes
Timer	Controlling the frame rate. Controlling the time left to the game.	Yes
RTC	Displaying current time and date.	Yes, periodic interrupts
Serial Port	Communication between the two computers.	Yes, half-duplex with FIFOs

2.1 VIDEO CARD

Mode	Screen Resolution	Color Mode	Number of Colors
0x14C	**1152x864**	**((8:)8:8:8) 32 bits per pixel**	**16.777.216 (2^24)**

During the program **Double Buffering** was used to help the game transitions and animations flow with more ease. Since we have a moving background, we felt the need to create an **auxiliar buffer** which contains the background image, in order to keep the same frame-rates in every screens. This technique is going to be explained later (4. Implementation Details – Double Buffering).

There are several different **animations** of the game sprites used, spread throughout our code. The most visible one is our **claw** that has 2 possible positions: open and closed. However, that is not the only one. Our **miner car** has 4 possible sprites that have a period of 4 timer ticks. The **pickaxe mouse** also changes when the left button is pressed. And our **stones** change between gold, stone and obsidians. The technique used to execute these animations is going to be explained later in module sprite.c.

Most of our game consists of **collisions detections** between the claw or pickaxe mouse with the stones/golds or the mouse with the words in menu. We only have a single function that is used in every collision detection: check_collision().

Function	**Use**	**File**
void (vg_map)(…)	Maps and allocates memory for the 3 buffers	video_gr.c
void (copy_background_buffer)()	Increments pointer address and copies to the buffer	video_gr.c
void (copy_buffer)()	Copies buffer to the main buffer	video_gr.c
void handle_sprite_changes(…)	Moves and changes all the sprites according to the game changes	sprite.c
void change_sprite_animation(…)	Transition each sprite to other animation	sprite.c
int check_collision(…)	Detect collisions	sprite.c
void* vg_init(…)	Initializes the video card in the desired mode	video_gr.c

2.2 KEYBOARD

The keyboard is used mainly to **control** the **miner car** and **claw** movement and some special keys are used to navigate between game screens. By pressing **'A'** the car moves left and **'D'** the car moves right. To control the claw, you must use **'S'** to make it go down and **'W'** to make it go up. It is always possible to change screens and go back to the main menu by pressing **'Esc'**. As referred in the explanation above about the multiplayer game modes, **'Enter'** is used to choose the multiplayer game in the same computer version. In that mode the second player is controlled by the **four arrows** from the keyboard.

The main implementation was developed during Lab3 development and in keyboard.c there are all functions used in the project from that Lab.

Function	Use	File
int keyboard_handler(…)	Deals with bytes received	game.c
void kbc_interrupt_validation(…)	Checks if the key pressed matters to the game and transforms the data received in actual useful data for the program unwind	state_machine.c
int kbc_get_event(…)	Gives the event/information related to the pressed key	state_machine.c
int kbc_event_handler(…)	Moves and changes all the sprites according to the received information from the keyboard	state_machine.c

2.3 MOUSE

The mouse is used in the menu, as well as inside the game. In both cases, the mouse uses all applications: the **buttons** and the **movement/position**.

In menu it is used mainly to select one of the menu options (by moving the cursor over the option and **clicking the left button**). In the game it is used to transform the stones (by **moving the cursor** over the stone and **clicking the left button**).

The main implementation was developed during Lab4 development and in mouse.c there are all functions used in the project from that Lab.

Function	Use	File
int mouse_handler(…)	Deals with packets received	game.c
int mouse_get_event_type(…)	Gives the event/information related to the movement/button pressed	state_machine.c
int mouse_event_handler(…)	Moves and changes all the sprites according to the received information from the mouse	state_machine.c

2.4 Timer

The timer most important function is to **update the game screen**. This consists of displaying the game graphs, updating the coordinates of the sprites, collisions, animations, mouse moves and backgrounds. It is a very important feature of our game since the timer interruptions deal with all the game information.

It is also responsible to manage the **game clock** (time left to the game), by decrementing 1 second to it every second.

The main implementation was developed during Lab2 development and in timer.c there are all functions used in the project from that Lab.

Function	Use	File
int timer_handler(…)	Handles the game changes and updates the game clock.	game.c

2.5 RTC

The RTC is mainly used to **display current date and time**. **Periodic interrupts** are used to correctly get the current time and date. It was implemented a function to check whether the registers of the RTC are being updated or not, as well as a function that reads register C to make sure interruptions are thrown. To help us collect all data and time information, **struct Date_t** was created and contains all that.

Furthermore, RTC is also being used to display the last 5 best scores played during that run of the program.

Function	Use	File
int rtc_handler(…)	Handles RTC interrupts	game.c
int rtc_subscribe_int(…)	Subscribes RTC interrupts	rtc.c
int rtc_unsubscribe_int()	Unsubscribes RTC interrupts	rtc.c
int enable_periodic_interrupts()	Enables RTC periodic interrupts	rtc.c
int disable_periodic_interrupts()	Disables RTC periodic interrupts	rtc.c
int read_registerC(…)	Reads RTC register C	rtc.c
int read_date(…)	Fills all Date_t parameters with RTC information	rtc.c
bool valid_read_date()	Checks whether the registers of the RTC are being updated	rtc.c

2.6 SERIAL PORT

For the Multiplayer mode, it was implemented the serial port in order to allow communication between two players in different computers. In this project, serial port is used with interrupts and FIFOs.

The chosen communication was half-duplex. This allows us to receive/transmit data from/to both machines. Hence it is possible to move the car in both machines and the result will be seen in the opposite machine.

When a player selects the multiplayer option, he is sent to a loading menu screen and a UART_LOAD byte is sent to the other machine. It only leads to the game when the user of the opposite machine also selects that option and a UART_LOAD byte is received from them. After this, both players are ready to play and the communication is based on a 1-byte (8 bit) number that, implemented, and designed by us, has all the information about which keys were pressed. This ables the program to make the necessary changes in the unwind of the game. This feature is going to be explained later in this report (4. Implementation Details – Uart Implementation).

Function	Use	File
int uart_handler(…)	Handles UART COM1 interrupts	game.c
void send_keys_bytes(…)	Sends bytes to the other computer	game.c
int uart_subscribe_int(…)	Subscribes UART COM1 interrupts	uart.c
int uart_unsubscribe_int()	Unsubscribes UART COM1 interrupts	uart.c
bool check_LSR_byte_received()	Checks if receiver buffer register is full	uart.c
bool check_LSR_THR_empty()	Checks if transmitter holding register is empty (valid to write)	uart.c
void read_RBR_byte()	Reads byte from receiver buffer register	uart.c
void send_THR_byte(…)	Writes byte to transmitter holding register	uart.c
int config_uart1(…)	Configures COM1 with all chosen predefinitions.	uart.c

3. Code Organization/Structure

I/O Devices Modules

Timer Modules

timer.c

Overall Weight: 2%
Each member contribution: André: 50% - Matilde: 50%

Description: In this module there are all functions that deal with direct manipulation of the timer interrupts (timer subscribe and unsubscribe interrupts). It was developed during lab2 implementation.

i8254.h

Overall Weight: 1%
Each member contribution: André: 50% - Matilde: 50%

Description: In this module there can be found all significant constant variables important to timer's manipulation. This module was not of our authorship, was given to us during the development of Lab2.

Keyboard Modules

keyboard.c

Overall Weight: 5%
Each member contribution: André: 50% - Matilde: 50%

Description: In this module there are all functions that deal with direct manipulation of the keyboard interrupts (keyboard handler, subscribe and unsubscribe interrupts and others). It was developed during lab3 implementation.

i8042.h

Overall Weight: 1%
Each member contribution: André: 0% - Matilde: 100%

Description: In this module there can be found all significant constant variables important to keyboard's and mouse manipulation, since both use the KBC Controller. This module was developed during the Lab3 and Lab4 implementation as well as some increments during the project development.

Mouse Modules

mouse.c

Overall Weight: 6%
Each member contribution: André: 50% - Matilde: 50%

Description: In this module there are all functions that deal with direct manipulation of the mouse interrupts (mouse handler, subscribe and unsubscribe interrupts and others). It was developed during lab4 implementation.

Video Card Modules

VIDEO_GR.C

Overall Weight: 5%
Each member contribution: André: 50% - Matilde: 50%

Description: In this module there are all functions that deal with initialization of the video card in the correct mode and painting pixels, they were developed during lab5 implementation. In addition, some functions and changes were added to this module in order to able the double buffering with auxiliar buffer technique explained in (4. Implementation Details – Double Buffering).

VIDEOCARDVARS.H

Overall Weight: 1%
Each member contribution: André: 50% - Matilde: 50%

Description: In this module there can be found all significant constant variables important to video card manipulation. This module was developed during the Lab5 implementation.

RTC Modules

RTC.C

Overall Weight: 5%
Each member contribution: André: 100% - Matilde: 0%

Description: In this module there are all functions that deal with direct manipulation of the RTC interrupts (RTC handler, subscribe and unsubscribe interrupts and others) as well as reading and interpreting all relevant information coming from their registers.

RTCVARS.H

Overall Weight: 1%
Each member contribution: André: 100% - Matilde: 0%

Description: In this module there can be found all significant constant variables important to RTC manipulation.

UART Modules

UART.C

Overall Weight: 8%
Each member contribution: André: 100% - Matilde: 0%

Description: In this module there are all functions that deal with direct manipulation of the UART COM1 interrupts (UART subscribe and unsubscribe interrupts and others) as well as reading and interpreting all relevant information coming from their registers.

UARTVARS.H

Overall Weight: 1%
Each member contribution: André: 100% - Matilde: 0%

Description: In this module there can be found all significant constant variables important to UART manipulation.

Game Modules

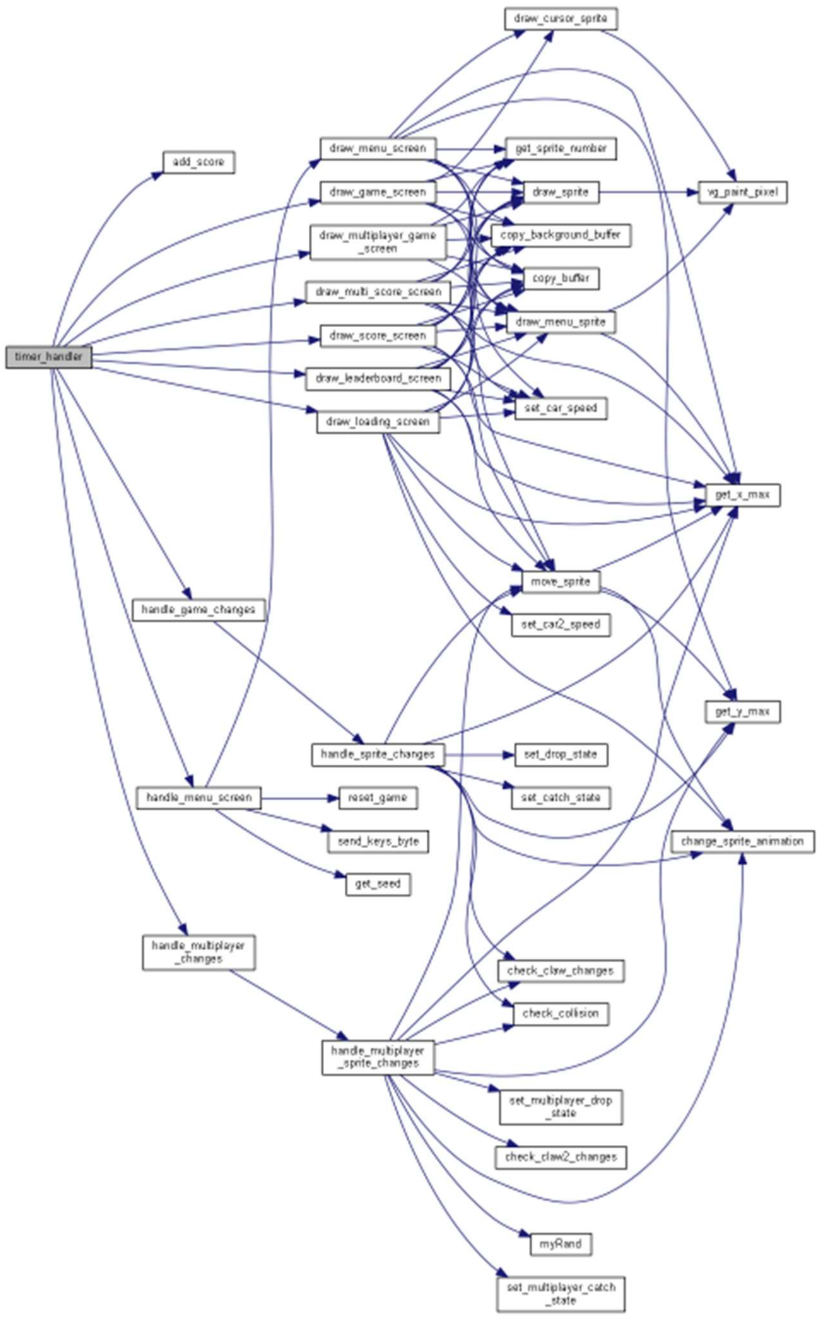
Game Changing Modules

GAME.C

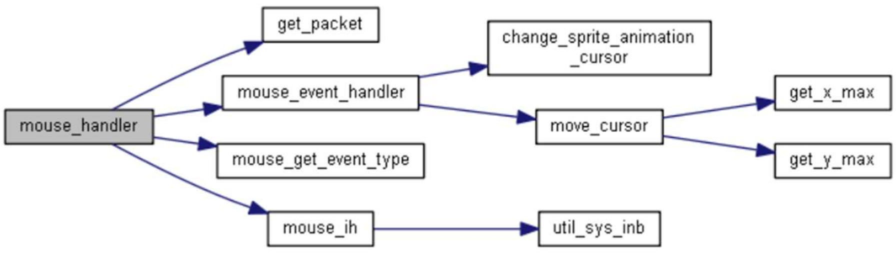
Overall Weight: 22%
Each member contribution: André: 60% - Matilde: 40%

Description: In this module there can be found the main functions that deal with receiving and handling consequent interruptions from the different devices that able the game to unwind. All the game structure is implemented based on a state machine, and almost every device has a smaller state machine behind its functionality. This feature is going to be explained later in this report (4. Implementation Details – State Machine). However, it must be mentioned here to a better understanding of the following graphs.

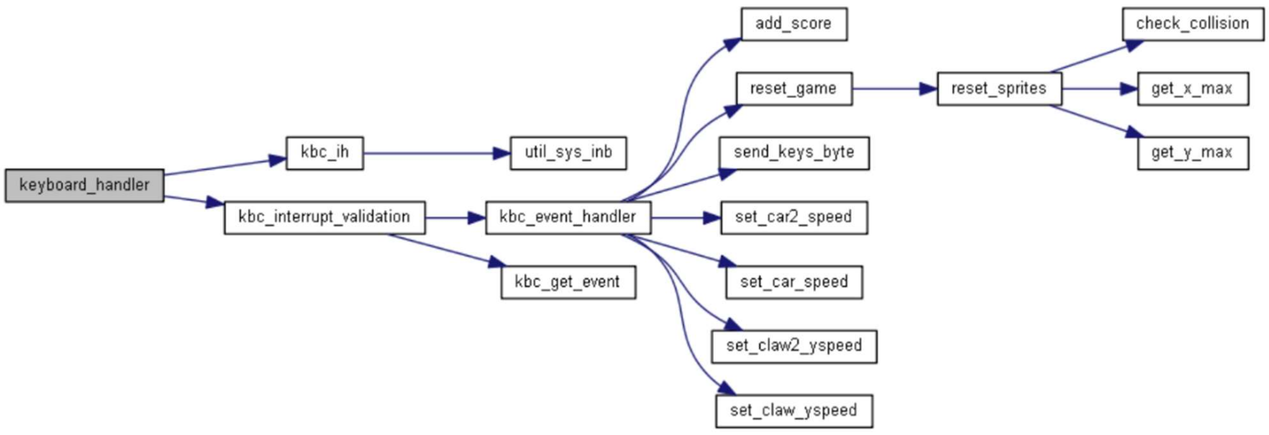
- **Timer**: for each interrupt, timer is set to control his interrupts and the game logic, first handling the game changes according to the screen state (controlled by a state machine), then draws the correct updated screen.



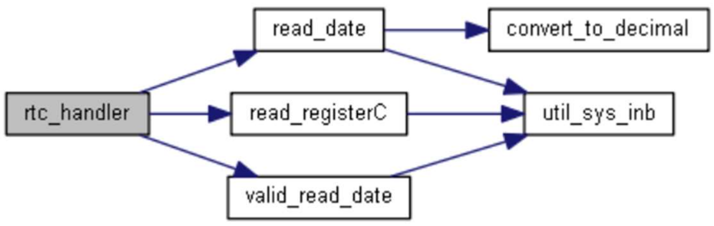
- **Mouse**: is used in a different way depending on the game screen. When a mouse interruption is received the game updates all related state machines, mainly the screen state, as well as some game animations such as its mouse cursor as well as some game updates (stones transformed in golds in single player mode).

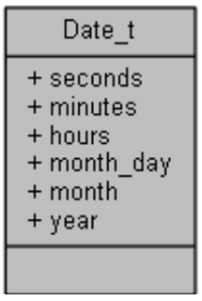


- **Keyboard:** only acts in a main role when the game is played. Every other interruption excepting the 8 moving key, 'Esc' and 'Enter' are ignored in this program. This device controls the act of catching golds, so leads to a state machine whose importance is to update whether the player catched or not a gold, please read: 4. Implementation Details – State Machine.

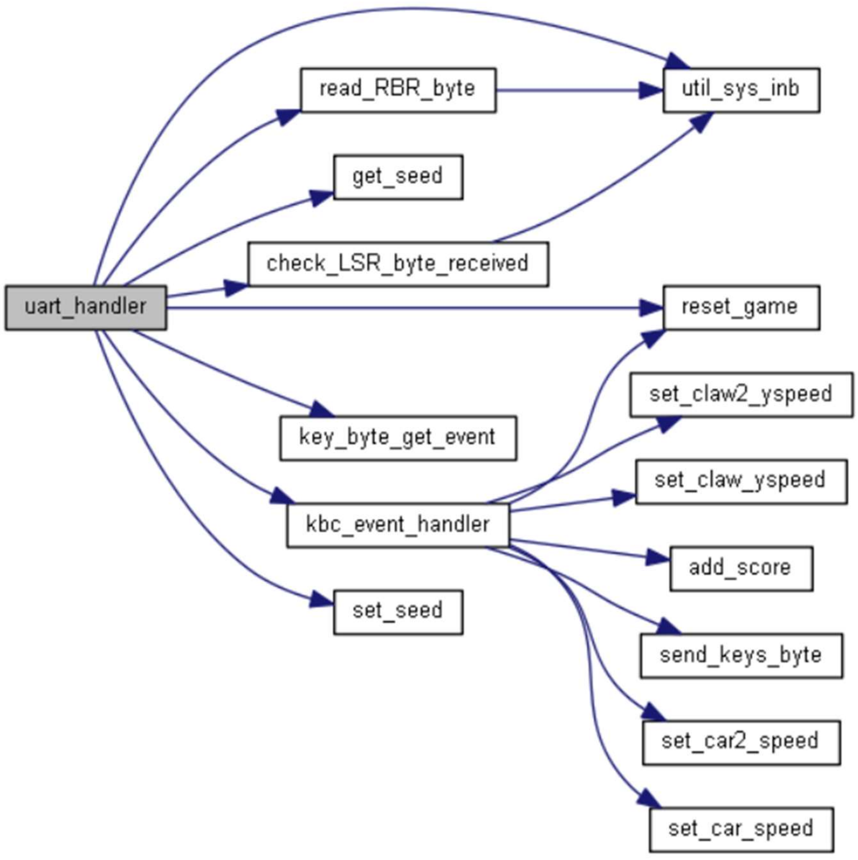


- **RTC**: handles RTC interruptions by updating the **struct Date_t** if the date registers are not being updated at that moment.





- **UART:** handles changes upon the information being received from the serial port.



GAMEVAR.H

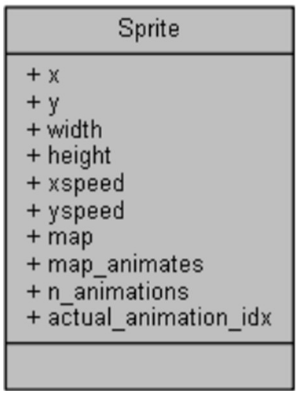
Overall Weight: 1%
Each member contribution: André: 50% - Matilde: 50%

Description: In this module there can be found all significant constant variables important to game unwind such as important positions, speeds, and animations.

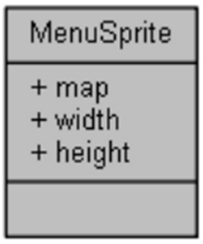
SPRITE.C

Overall Weight: 24%
Each member contribution: André: 55% - Matilde: 45%

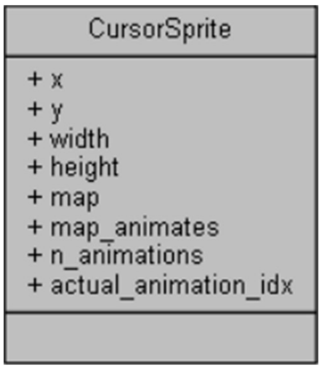
Description: This module contains everything that deals with xpms, animations, collisions, printing xpms in the right positions. As well as some important functions that are constantly being called to handle every sprite on the screen: handle_sprite_changes() and when in multiplayer mode handle_multiplayer_sprite_changes(). To deal with sprites we created three different structs that are more appropriated for a specific type of sprite on the screen. This implementation was based on the sprites referred in classes, during Lab5, with some needed changes.



SPRITE: This first one, is mainly used for active sprites on the screen, this consists of for example the car, claw and golds, who need to have speed to constantly change their positions. In addition, it was implemented an array of pixmaps that have different possible images, to create animations to these sprites. When an animation is supposed to happen the parameter map changes to a chosen pixmap from the array and updates de index of the actual pixmap being shown.



MENUSPRITE: This one, as the name refers to, is used mainly for menus display. As they only need a static position, it was not necessary to have more parameters to this struct.



CURSORSPRITE: This is a specific one because as the name tells, represents the mouse cursor and has its needed parameters (the same one as in sprite, without speed since the positions are given by the mouse interrupts). Since it was implemented a small animation, the 3 last parameters were considered necessary to have the same style of animation implemented for the Sprites.

STATE_MACHINE.C

Overall Weight: 14%
Each member contribution: André: 50% - Matilde: 50%

Description: As mentioned above in this report, state machines were used throughout the program to deal with different issues of the game.

1. There is a state machine for the whole game, controlling the options showed in the menu.
2. There is also other to control the game itself, containing information about having or not catched the golds.
3. There is also a state machine controlling the keyboard events (keys pressed).
4. And other to control mouse events (keys pressed and moves).

All this events and state machines are implemented in this module and have is appealed enumerations declared in state_machine.h as well as their documentation. This type of implementation is going to be explored in more detail later (4. Implementation Details – State Machine).

PROJ.C

Overall Weight: 2%
Each member contribution: André: 0% - Matilde: 100%

Description This module is the first to be called and, thus calls for the initialization and configuration of the game, starts and runs it.

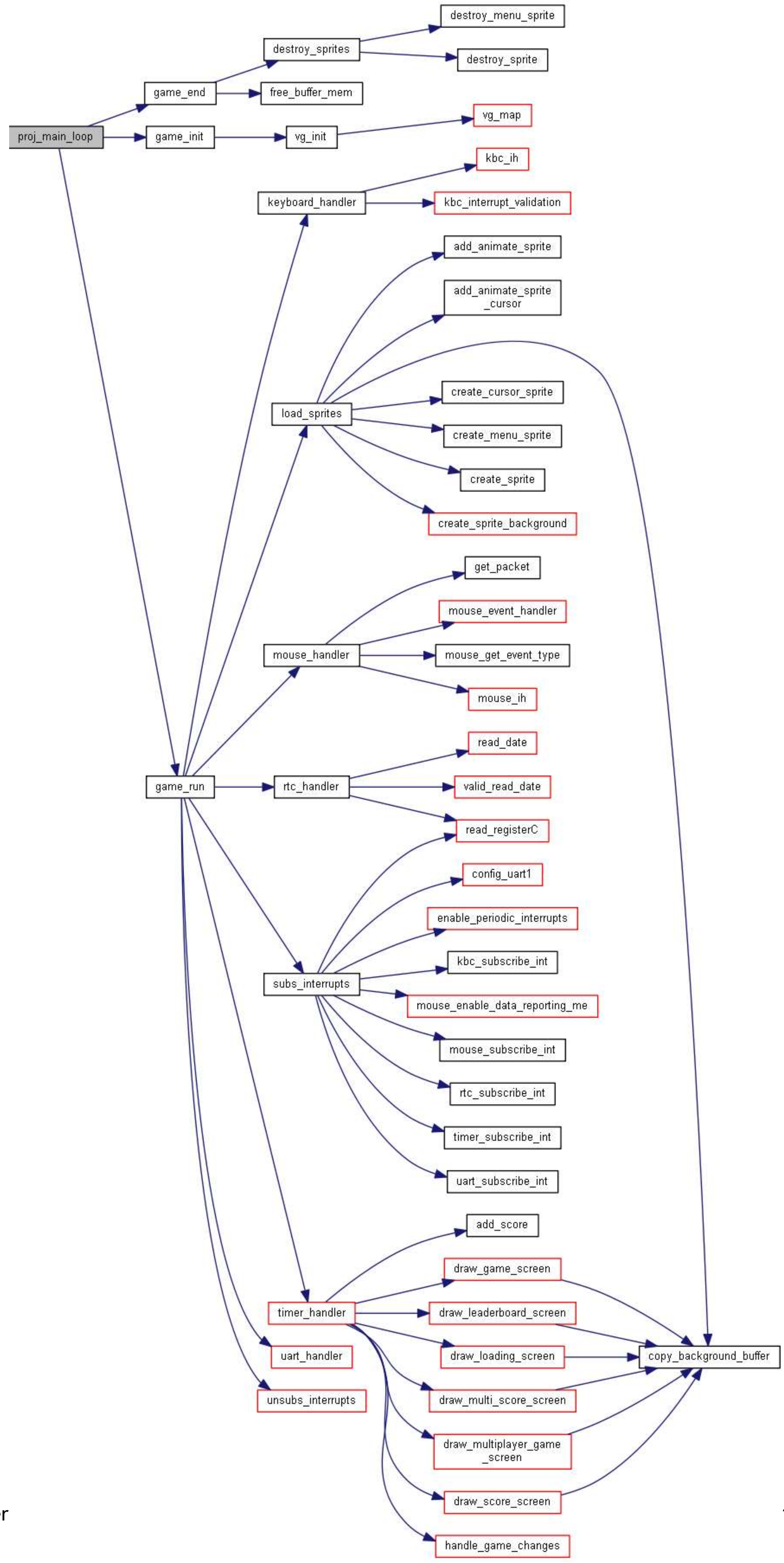
OTHER MODULES

UTILITIES MODULES

UTILS.C

Overall Weight: 1%
Each member contribution: André: 0% - Matilde: 100%

Description: In this module there are all functions that help with registers manipulation. It was developed during lab2 implementation.


destroy_menu_sprite
destroy_sprites
destroy_sprite
game_end
free_buffer_mem
vg_map
proj_main_loop
game_init
vg_init
kbc_ih
keyboard_handler
kbc_interrupt_validation
add_animate_sprite
add_animate_sprite _cursor
create_cursor_sprite
load_sprites
create_menu_sprite
create_sprite
create_sprite_background
get_packet
mouse_event_handler
mouse_handler
mouse_get_event_type
mouse_ih
read_date
game_run
rtc_handler
valid_read_date
read_registerC
config_uart1
enable_periodic_interrupts
kbc_subscribe_int
subs_interrupts
mouse_enable_data_reporting_me
mouse_subscribe_int
rtc_subscribe_int
timer_subscribe_int
uart_subscribe_int
add_score
draw_game_screen
timer_handler
draw_leaderboard_screen
draw_loading_screen
copy_background_buffer
uart_handler
unsubs_interrupts
draw_multi_score_screen
draw_multiplayer_game _screen
draw_score_screen
handle_game_changes

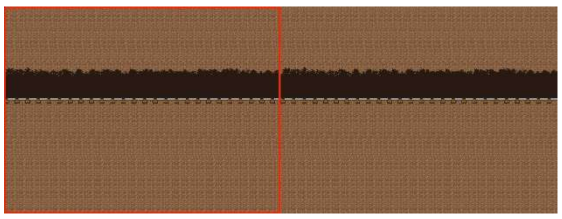
4. Implementation Details

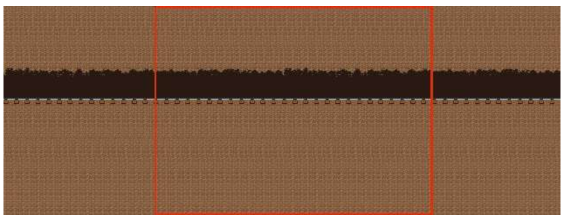
Double Buffering With Auxiliar Buffer

To implement **Double Buffering** we used the implementation explained in classes notes. That consists of writing everything to the second buffer (named *"buffer"*) and then copy it to the main buffer (named *"video_mem"*).

Since this implementation did not fully fulfilled our needs, we implemented another buffer that helped with the moving sensation of the background the *"background_buffer"*.

This technique consists of having a background buffer with a duplicated background image and the address which the **background auxiliar buffer** points to, changes with the time (technique explained with the images below). Basically, the *background_buffer* points to the top left corner of the red rectangle that is the actual printed screen.





We studied the possibility of implementing **page flipping**, but we came to the conclusion that, since we had a moving background that obligates us to constantly copy all the pixels, it wouldn´t be a viable solution. Thus, double buffering with an auxiliar buffer appeared to be the best one.

Object-Oriented Programming

This was mainly implemented with the use of sprites to deal with its pixmaps, animations, moves and collisions. As explained above in the module sprite.c we implemented three different structs and respective functions to use and module this sprites changes.

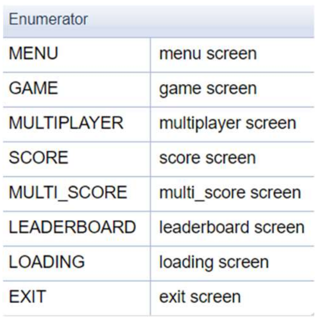
Multiple Different Game Runs

The generation of the position of our stones/golds are based in the rand() function. With this being said, the user can play the game multiple times and will never find the same set up of objects distribution. It is important to notice that, at the beginning of every new singleplayer or multiplayer game, the function reset_game() is called with the objective of resetting all the positions and animations.

State Machine

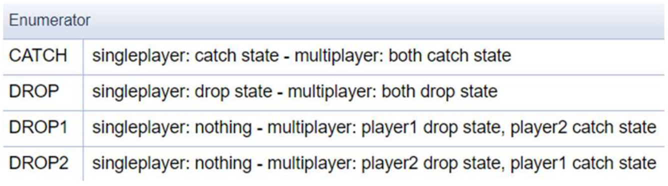
The game implementation is a **State Machine** and has others state machines to control different aspects of the program. All state machines are explained below with the respective enumeration of states.

Game Screen, controlled by the variable “screen_state”, which tells which screen is being displayed.



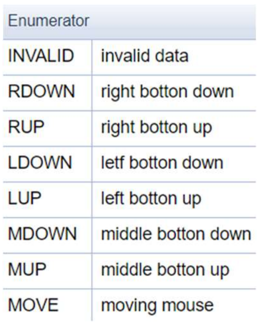
Enumerator	
MENU	menu screen
GAME	game screen
MULTIPLAYER	multiplayer screen
SCORE	score screen
MULTI_SCORE	multi_score screen
LEADERBOARD	leaderboard screen
LOADING	loading screen
EXIT	exit screen

Game State, controlled by the variable “game_state”, which tells which game event is happening.



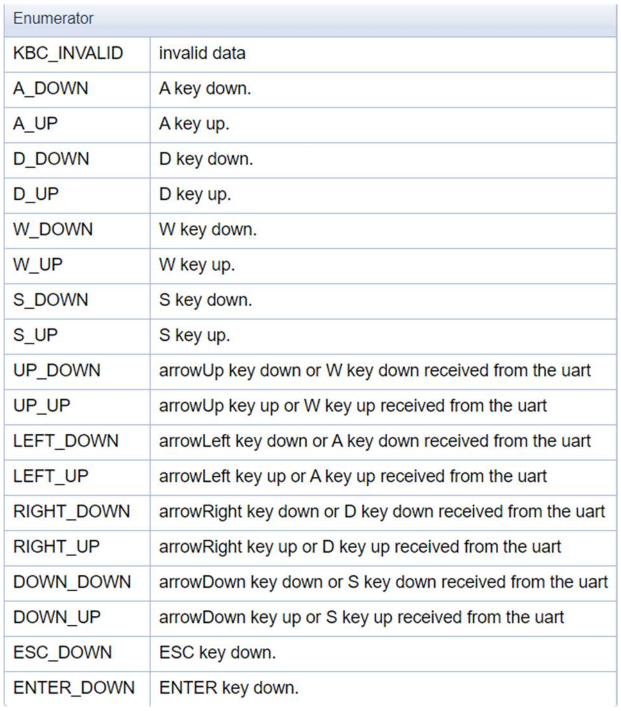
Enumerator	
CATCH	singleplayer: catch state - multiplayer: both catch state
DROP	singleplayer: drop state - multiplayer: both drop state
DROP1	singleplayer: nothing - multiplayer: player1 drop state, player2 catch state
DROP2	singleplayer: nothing - multiplayer: player2 drop state, player1 catch state

Mouse State, controlled by the variable “event” in the mouse handler, which tells which mouse key was pressed or if it is moving.



Enumerator	
INVALID	invalid data
RDOWN	right botton down
RUP	right botton up
LDOWN	letf botton down
LUP	left botton up
MDOWN	middle botton down
MUP	middle botton up
MOVE	moving mouse

Keyboard State, controlled by the variable “event” in the keyboard handler, which tells which keyboard key was pressed.



Enumerator	
KBC_INVALID	invalid data
A_DOWN	A key down.
A_UP	A key up.
D_DOWN	D key down.
D_UP	D key up.
W_DOWN	W key down.
W_UP	W key up.
S_DOWN	S key down.
S_UP	S key up.
UP_DOWN	arrowUp key down or W key down received from the uart
UP_UP	arrowUp key up or W key up received from the uart
LEFT_DOWN	arrowLeft key down or A key down received from the uart
LEFT_UP	arrowLeft key up or A key up received from the uart
RIGHT_DOWN	arrowRight key down or D key down received from the uart
RIGHT_UP	arrowRight key up or D key up received from the uart
DOWN_DOWN	arrowDown key down or S key down received from the uart
DOWN_UP	arrowDown key up or S key up received from the uart
ESC_DOWN	ESC key down.
ENTER_DOWN	ENTER key down.

UART IMPLEMENTATION & RAND FUNCTION

In other to our multiplayer program to work well, both computers should receive the same information, to keep all the visual data coherent. To do that we need to send all the data related to the keys pressed (break and make codes) and keep all the golds in the same positions. These were the main problems related to our multiplayer game version. Thus, to make this possible we felt the need to simplify the information to be sent.

With this being said, in order to resolve the first mentioned problem, to send all the break and make codes, we came up with a simplified way to deal with it: we created a byte that we called the **"keys_byte"** that contains all the info relate to the 'A', 'W', 'S', 'D' keys pressed/released (as showed below).

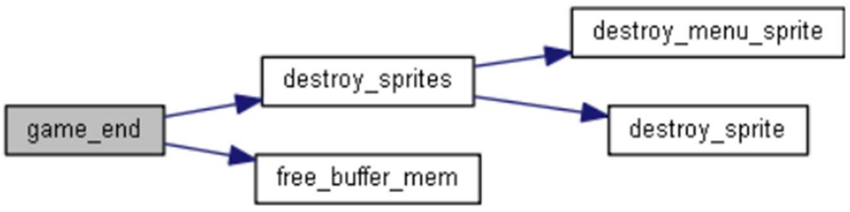
BYTES	7	6	5	4	3	2	1	0
INFO	A MAKE	A BREAK	D MAKE	D BREAK	W MAKE	W BREAK	S MAKE	S BREAK

The begin state of the keys_byte is 01010101, every key starts in its break code state (none of them is being pressed). If for example the key 'A' is pressed the byte changes to 10010101 and the **keys_byte** is sent via serial port to inform the other side that some change happened. After receiving it, the receiver side handles it and makes the necessary changes in their screen, in this case starts moving the player2 car to the left.

The other problem would be making all the golds appear in the same position in both computers. This was solved by sending a seed via serial port. To send the seed we first send a byte (UART_SEED) that tells the receiver side that the following byte will be the seed. This seed is generated randomly with the rand() function and after that sent. We found some problems while developing this solution cause the srand() (declares the seed that will be used for rand() function) of the random library was not giving the same numbers to both computers. Professor Mário Cordeiro gave the idea to create our own rand() function. In order to make that possible we used the code from stack overflow: https://stackoverflow.com/questions/7953612/using-a-random-number-to-generate-a-series-of-repeatable-numbers-in-c. After receiving the seed, when in the multiplayer mode for 2 different computers, we start using the function myRand() to generate the golds positions and they appear in the desired places.

FREE ALLOCATED SPACE

In our program run we have allocated a lot of space for various reasons. The main ones were to save the sprites coloured bits, and to have space for our buffers. Hence, it would be bad practise if we did not free all that allocated space. Therefore, our game_end() function frees all the allocated memory.



5. Conclusion

To conclude our report on this exciting and a little addictive project, we first wanted to mention the high relevance of this unit and its importance for our growing skills. It was the first time we had contact with low level language programming more specifically I/O devices, and it was an experience we both will remember for its challenges and different opportunities. In fact, this was the unit that took most of our time this semester due to its captivating and exacting subjects.

Despite the important and interesting learning topics, LCOM has multiple points to improve we would like to mention.

First, we were aware that this unit was going to take a lot of our time. However, being an interested student that participates actively in both theoretical and practical classes, is not enough and many weekends, apart from what we imagined, were fundamental to catch up with the class matters and labs implementation. We believe that this problem is related with this being the first time we had contact with I/O devices programming. Now, after having realized all the proposed labs, we can tell that we have all information needed. Although, in the beginning, it was rather confusing not having the information displayed the easier way. Therefore, we strongly agree that we should be more instructed about these topics before, as well as having better instructed ways of following the subjects. In addition, and to conclude this topic, 6 credits appear to be less than what the workload of this course unit reserves. Having as comparison term AEDA with 7.5 credits, the time spent with both is completely the opposite of the credit's distribution.

The second point, has, for both of us, a great impact on the feeling of commitment with the unit. The lcom library is a great implemented library, but, since the robustness of it, its necessary to take some time to understand specific techniques and how to solve errors. Labs were a great experience on the path to a better understanding of it, since we spend significant time experiencing this. On the contrary, in the two performed tests this semester, we were not able to fulfil that objective. Consequently, these tests may be decisive of our performance in the unit and not portray our understanding since the library can be hard to deal with in case of errors. After the tests, due to our performance in them, our motivation to work hard on the project decreased. However, we are pleased to see that it did not stop us from going this further with the project, especially implementing the multiplayer game version, that was really a challenge and we are proud of it. We must take this opportunity also to show our dissatisfaction with not having our results published yet, as it is good to monitor our progress during the semester.

We must not finish this conclusion without thanking all monitors and professors for all the support given throughout the project and all unit.

Our video demonstration is available in the following link, please check it out: https://www.youtube.com/watch?v=t039Zrj8C0s.

Appendix - Installation Instructions

It is mandatory to enter the proj directory and then the src directory with the command “cd src”, in order to finally run the “make” command.

Funny Mode Instructions

Little Miner Party Mode: this is apart from the evaluation and is something that we discovered during the project implementation. Try to go to video_gr.c, in function: copy_background_buffer() change line 2 of the body of function to: increment = (increment + BACKGROUND_SPEED*3) % (h_res*8);. And enjoy the game with some party music :).

Despite the funny side of this, we are aware that this is caused by bad programming and it is an error. We would like you to know the cause of it. The “increment” variable is used to move the pointer of the background buffer. Thus, this variable must be a multiple of 4 since the video card mode used represents each pixel with 32 bits (4 bytes). Adding a multiple of 3 leads to a bad colour reading, since it assumes, for example, that the bits reserved to the green portion of the colour assume the blue portion of it and that mixes all the colours up.

Check this out: https://www.youtube.com/watch?v=rhO7Rp3MV8o.