2022/23

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

TYPE RUNNER

LCOM

Grupo 6 - Turma 11

Licenciatura em Engenharia Informática e Computação

Luís Miguel Lima Tavares	up202108662@fe.up.pt
Miguel Martins Leitão	up202108851@fe.up.pt
Rodrigo Campos Rodrigues	up202108847@fe.up.pt
Rodrigo Santos Rodrigues	up202108749@fe.up.pt

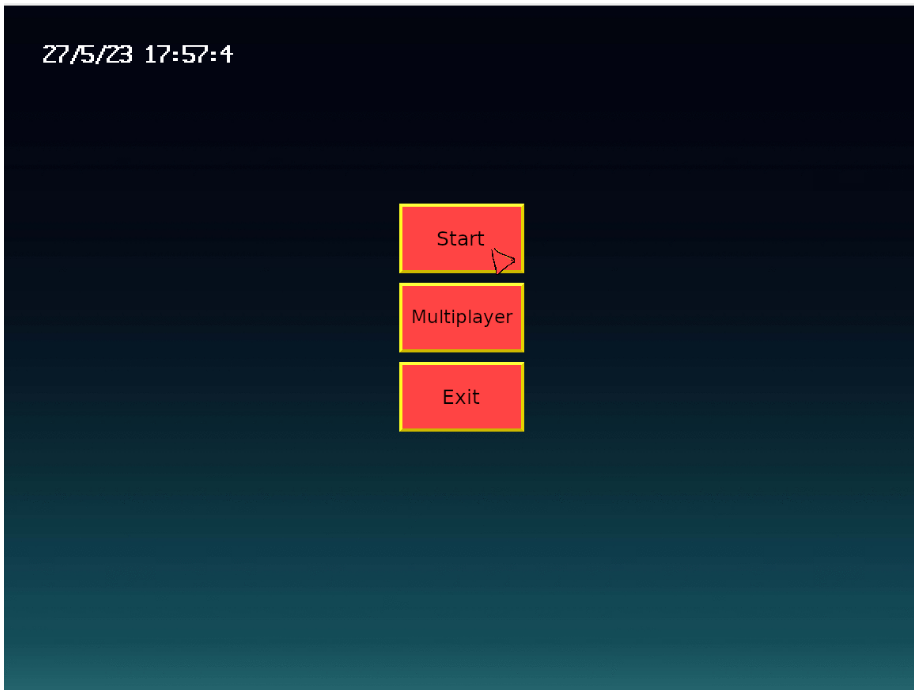
Index

Luís Miguel Lima Tavares	up202108662@fe.up.pt
Miguel Martins Leitão	up202108851@fe.up.pt
Rodrigo Campos Rodrigues	up202108847@fe.up.pt
Rodrigo Santos Rodrigues	up202108749@fe.up.pt

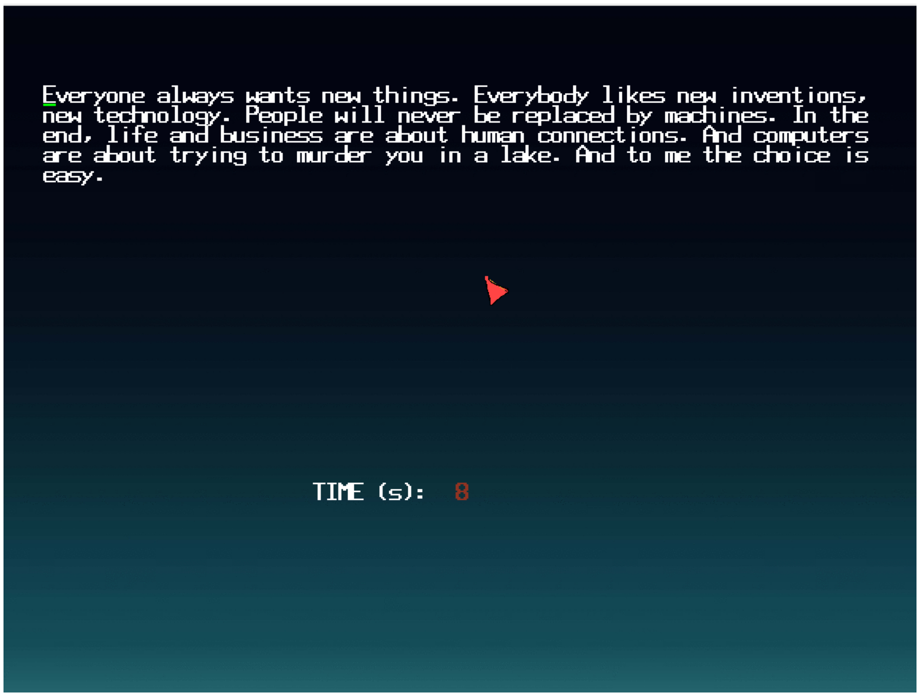
1. User instructions

1.1 - Single player

Upon pressing the start button, the user will be placed into single player racing mode.



In this mode the user will be able to type out a text (race) word by word. For maximum training efficiency, upon typing a word and pressing space, the user will not be able to go back to that word (always going back to redo what you typed wrong means you are typing faster than you should and should slow down and focus on accuracy more). Upon reaching the end of the sentence, the user must press space as well, this is implemented to maintain a good practice of not just smacking the final symbol after finishing a sentence. This way the user must press space as if he were to keep on typing to maximize the training efficiency.

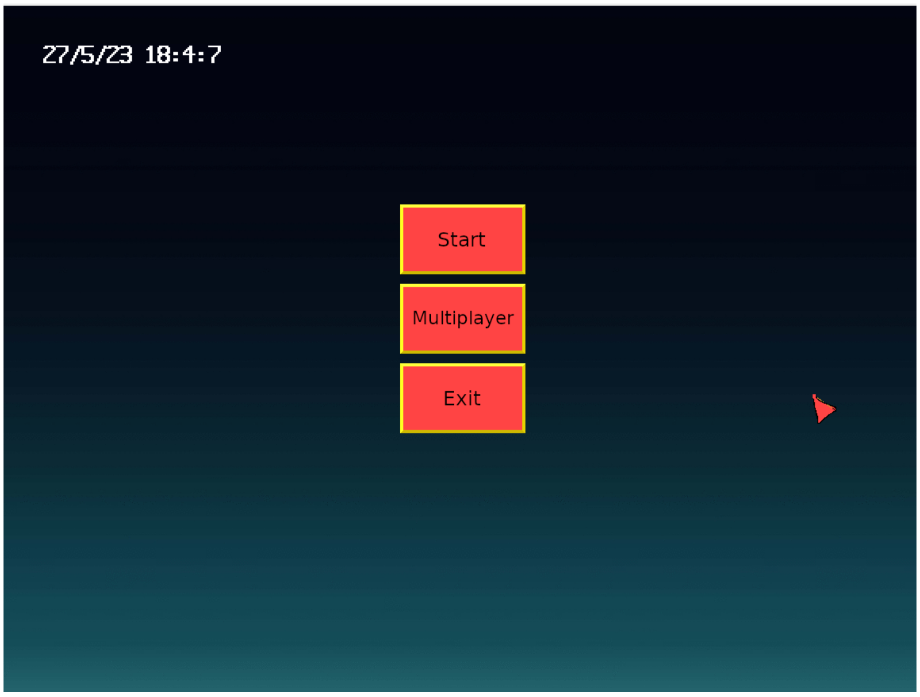


Everyone always wants new things. Everybody likes new inventions, new technology. People will never be replaced by machines. In the end, life and business are about human connections. And computers are about trying to murder you in a lake. And to me the choice is easy.

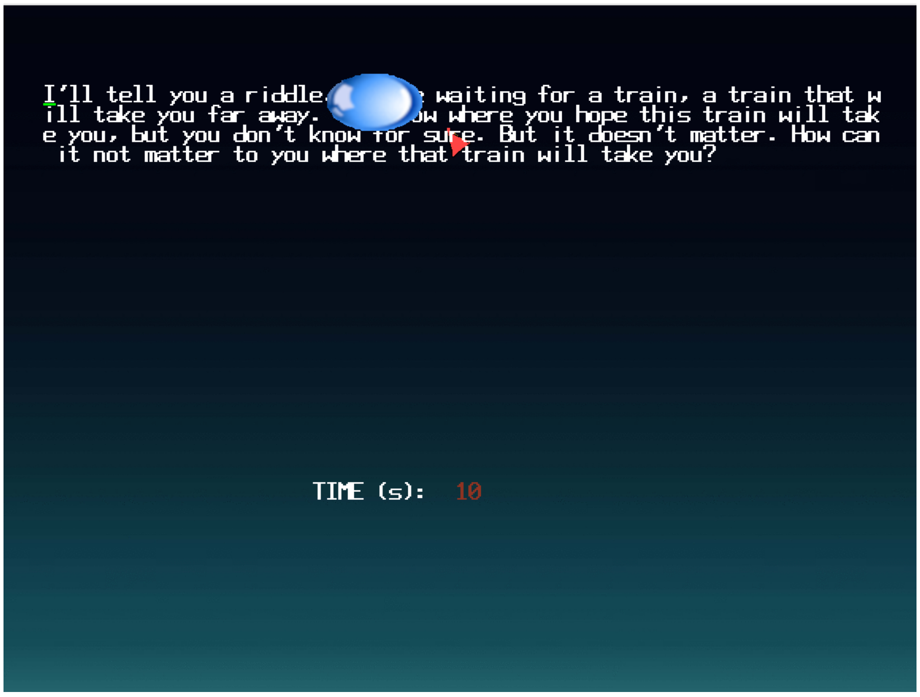
TIME (s): 8

1.2 - Multiplayer

Upon pressing the multi button, the user will be taken to the multiplayer mode screen.



The keyboard user will try to type out the sentence as fast as possible without getting hindered by the player with the mouse, the player with the mouse is able to cover words ahead to make it harder to type the sentence out. The cover will be removed every 3 seconds. This is a good way to practice reading ahead while typing which will help with increasing typing speed. Upon reaching the end of the sentence, the user must press space as well, this is implemented to maintain a good practice of not just smacking the final symbol after finishing a sentence. This way the user must press space as if he were to keep on typing to maximize the training efficiency.



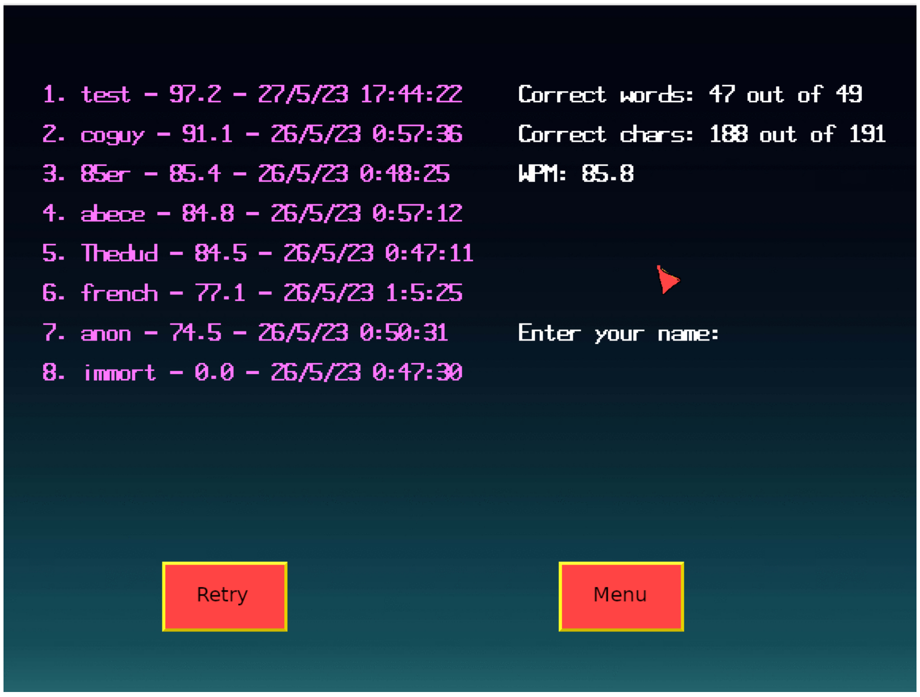
```
I'll tell you a riddle.[illegible] waiting for a train, a train that w
ill take you far away. [illegible]ow where you hope this train will tak
e you, but you don't know for sure. But it doesn't matter. How can
 it not matter to you where that train will take you?

TIME (s):  10
```

1.3 - Game Over:

When the user finishes typing the full sentence, be it either in singleplayer or multiplayer mode, the game over screen will pop up and show the amount of words that were correctly typed, as well as the amount of characters, this is calculated word by word instead of a full sentence comparison to allow for more sophisticated wpm calculation and a more forgiving typing experience where you don't need to hit every character to be rewarded.

The left side of the screen also shows top 10 fastest users stored in the highscores.txt file, along with the date/time in which the record was set for that user. If you are the same user and have set a new record, the record will be updated and the new positions will be shown in real time.



```
1. test - 97.2 - 27/5/23 17:44:22        Correct words: 47 out of 49
2. coguy - 91.1 - 26/5/23 0:57:36        Correct chars: 188 out of 191
3. 85er - 85.4 - 26/5/23 0:48:25         WPM: 85.8
4. abece - 84.8 - 26/5/23 0:57:12
5. Thedud - 84.5 - 26/5/23 0:47:11
6. french - 77.1 - 26/5/23 1:5:25
7. anon - 74.5 - 26/5/23 0:50:31         Enter your name:
8. immort - 0.0 - 26/5/23 0:47:30
```

1.4 - Exit:

In the Main Menu, when the exit button is pressed or [ESC] key, the game will reset everything back to normal, such as interrupt subscriptions, deallocate all dynamic memory used and exit into Minix text mode.

NOTA - Races

Races can be added to the folder located in the races folder in the src directory and will be dynamically read. In the game.c file, the macro responsible for the race location can be edited to point to whichever file the user wants to load races from. A race file should only contain the text to be typed out and nothing more, special characters and punctuation are allowed, as the fontset used allows for them.

2. Device use/function in the application

2.1 - Devices used

- The **timer** was used to maintain a constant frame-rate in the game (60hz), to update the game event and clear the bubble in multiplayer mode. We used interrupts.

- The **graphics card** was used to draw sprites (buttons, backgrounds, animated bubble and cursor) and to draw fonts (alphabet.c). We did not use interrupts.

- The **keyboard** was used to read inputs from the user in single/multi mode such as the key being typed or modifier keys for special characters. There is also a quick restart functionality implemented that lets the user press [Tab] and quickly load another race for quick practice. We used interrupts.

- The **mouse** was used to navigate the menus and in multiplayer mode for the second player to control the place where the bubble will be drawn. We used interrupts.

- The **RTC** was used to display the date/time in the main menu for quick access to the time without leaving the game, and in the high score functionality of the game to save the time in which the current score was achieved by a certain player. This is used to keep track of progress. We did not use interrupts since we only needed to get the date/time in certain moments of the game.

- The **serial port** was not used in the project.

2.1.1 - Timer

The timer is used to keep a steady framerate of 60fps, Every interrupt of the timer, there is a check for a variable called redraw. If this variable is true, there is a full redraw of the app. The timer is also responsible for updating the screen when a second has passed in single/multiplayer mode to show the total time elapsed (get_gamecounter() % 60).

2.1.2 - Graphics Card

The graphics card is set up in 0x14C video mode (1152x864 direct color) because of the increased resolution and a bigger range of colors to choose from.

Double buffering via memcpy() was the chosen technique for rendering because of its simplicity and the nature of the game.

A big part of the game and its construction was finding a way to draw multiple characters to the screen without having to build multiple XPMs, so we found a StackOverflow thread that had a solution that consisted in storing the pixels to be colored in each character in stacks of bytes, where the 1 in the byte in each row would be colored with the desired font color (https://stackoverflow.com/questions/2156572/c-header-file-with-bitmapped-fonts/). In the replies a user replied with a full ascii font implementation of this idea, so we stuck with it. Later on we were informed by a professor that we could do our own fonts without using multiple XPMs, but we already had implemented this, so we stuck with it. The function responsible for drawing a font by ascii code is the vg_draw_char, which also makes each character bigger by extrapolation, since the character set we had was too small for the resolution of the screen.

Animations were implemented in the bubble in multiplayer mode, we used a struct animated_sprite to hold multiple sprites and circulate them depending on the animation speed (number of frames per sprite).

This approach was introduced in the theoretical classes so we stuck to that implementation and made it as simple as possible.

There are collisions between the cursor and the buttons using a set of pixels and lengths and calculating intersections. As we don't have any characters to check collisions with each other, we consider that the cursor collision with the edges of the screen can be considered as a collision check also, since the logic would be the same for both purposes.

2.1.3 - Keyboard

The keyboard is used both to control the menu of the app in case of quick resetting the game with [Tab] and to input the characters to type out the sentence in game mode.

We implemented an abstraction layer around the keyboard with the function kbc_get_event() which returns an event that has an action and a char associated with it. This is achieved by having a mask for the modifier keys currently pressed and comparing the scancode of the current pressed key to a set of scancodes provided by the LCOM lab documentation (https://www.win.tue.nl/~aeb/linux/kbd/table.h).

The manager.c file will then use a state machine to handle which events should do what depending on the state of the game and will send out the events to the corresponding handlers such as the game handler [game_handle_event(event_t e)] or the gameover handler [game_over_handle_event(event_t e)].

2.1.4 - Mouse

The mouse was implemented alongside the keyboard (keyboard.c) since they use mostly the same registers.

Since the logic for moving the mouse was fairly simple to implement with the help of the labs made in the classes during the semester, we decided to simply make a handler in each menu page that checks collisions with the buttons and whenever the LMB is being pressed the action associated with that button is performed (mostly changing game states). The functions responsible for this are which are handle_click_main_menu(uint16_t x, uint16_t y) and handle_click_game_over(uint16_t x, uint16_t y) in the files menu.c and gameover.c respectively.

The movement of the mouse is handled in manager.c directly in the mouse_int() function.

2.1.5 - RTC

The rtc was implemented to keep track of the date/time in the main menu and to save the dates of the highscores of different users.

In the gameover.c file, the function game_over_handle_event(event_t e) will handle an [ENTER] as the action to save the high score of the current race with the user name inputted before. Inside this handler, the current time will be asked from a wrapper function in the rtc.c file get_full_date_time(char* date_time) which will update the date/time with the update_time() function and convert the struct with the information into a string and place it in the char array passed in the argument. The score is then saved using the save_highscore(char *name, float wpm, char *date) function.

3. Code structure/organization

3.1 - Game

Contains the necessary functions to handle the game itself.

In other words, it is responsible for generating a race, as well as performing actions such as advancing a word, comparing a word written by the player with the same word written in the text file, and finally ending the game, generating the score and resetting all the variables.

3.2 - Gameover

Responsible for the mechanisms that happen after the player completes the race.

This is where the score (in this case words per minute) is calculated, comparing the typed sentence with the one in the text file, and also taking into account the total time. It also contains the necessary functions to record highscores, as well as display the game's top10 high scores (ordered by wpm).

3.3 - Gamestate

Contains a gamestate_t enumerated type that can have multiple values (such as MAIN_MENU, SINGLE, etc...).

3.4 - Menu

Responsible for the main menu event handling and click handling.

Contains the necessary function to switch to the appropriate gamestate depending on where the user clicked the mouse. It also contains, in an array, the positions of the 3 buttons.

3.5 - Alphabet

Contains a 2D array representing a bitmap font.

Each element of the array corresponds to a character in the ASCII range from 32 to 126, and each character is represented by a 13-byte row, where each byte represents a pixel.

3.6 - i8042

Contains necessary macros for working with the keyboard and the mouse.

This was developed in lab3 (practical classes).

3.6 - i8254

Contains necessary macros for working with the timer.

This was developed in lab2 (practical classes).

3.7 - Keyboard

Responsible for handling both the Keyboard and the mouse. Contains the necessary functions to subscribe and unsubscribe from interrupts, to read and interpret scancodes, and to handle interrupts, all from both devices.

3.8 - Manager

Responsible for defining the sprites and variables used in the game.

Calls the subscribe interrupts functions for all devices. Contains the main game loop function, which handles all the interrupts from all devices, and calls the appropriate functions to draw on the screen. It is also responsible for destroying all the sprites when it’s appropriate to do so.

3.9 - RTC

Responsible for handling the RTC (real-time clock).

Contains the functions to initialize the RTC, subscribe and unsubscribe from interrupts, as well as its interrupt handler. It also contains a function that updates the time from the RTC and formats it into a string, which is used to show the date of when a highscore was set.

3.10 - Timer

Responsible for handling the Timer, more specifically the Timer 0.

Contains the functions that subscribe and unsubscribe to Timer interrupts, handle Timer 0 interrupts, and retrieve and display timer configurations. It's also possible to change the timer's frequency.

This was implemented in lab2 (practical classes).

3.11 - Utils

Contains utility functions.

Contains functions to retrieve both the MSB (most significant byte) and LSB (least significant byte) from a 16 bit value. It also contains util_sys_inb() that typecasts a 32-bit value read from sys_inb() to a 8 bit value before assigning it to a variable.

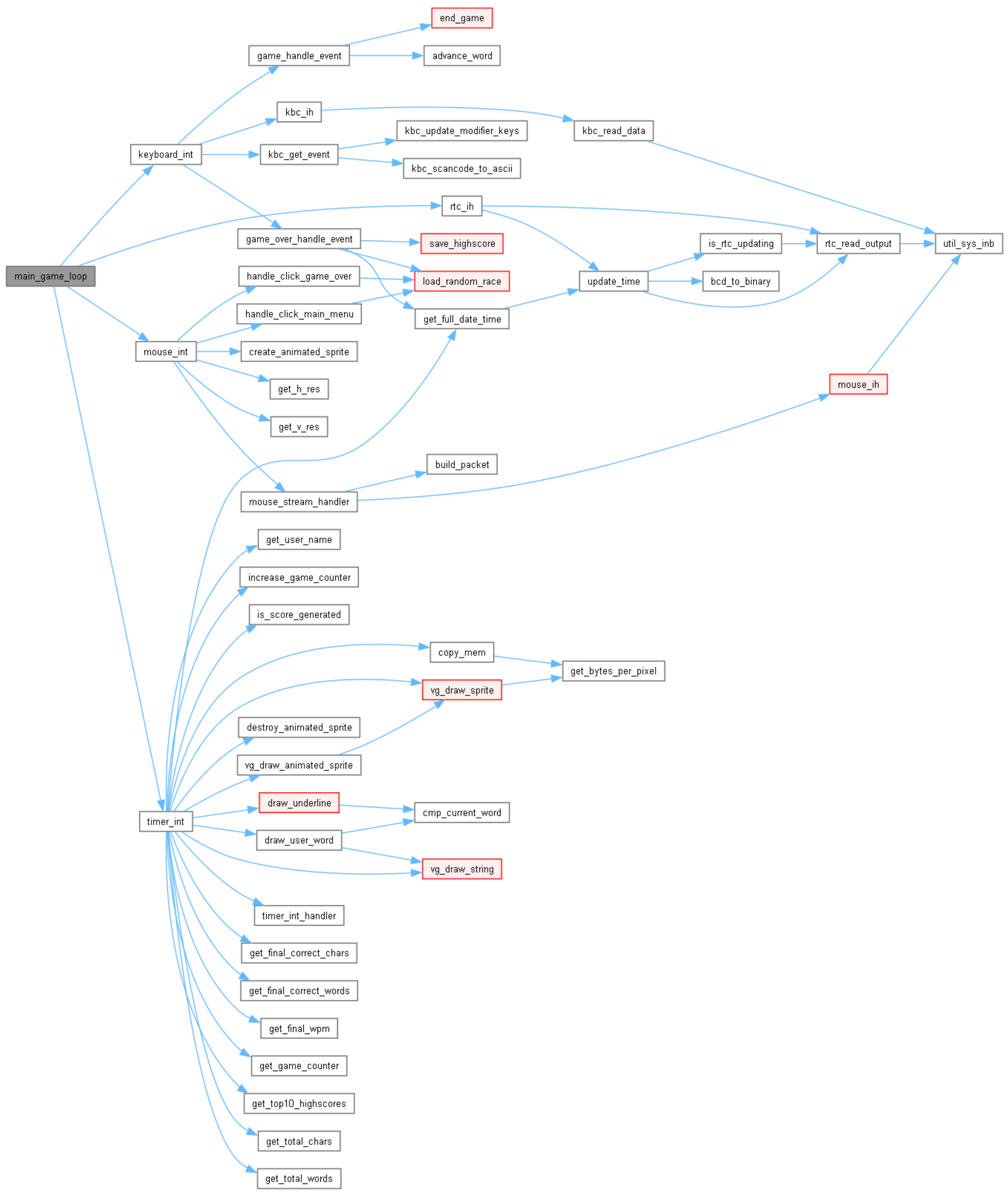
3.12 - Video_gr

Responsible for handling video card functionality.

Contains functions responsible for initializing the video in the desired mode. It also contains functions that draw and destroy sprites and animated sprites, as well as individual pixels and characters alike.

This code was implemented in lab5 (practical classes).

4. Functional call graph



5. Conclusion

We achieved most objectives in the making of the game, however we know there could have been some improvements made in the overall look of the game and maybe with some more time multiplayer mode could have been implemented using the serial port to allow for races between players.

Page flipping could also have been implemented, however the game is not very graphically demanding and we don’t do any complex calculations so the simple memcpy() double-buffering alternative sufficed in this case and we decided to focus on the rest of the program instead.

Overall, we are satisfied with the way the project turned out.