U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Typing Tutor

Computer Laboratory Project Report 2023/24

Group:

Maria Rabelo - up202000130

Rui Borges - up202207475

Tiago Pires - up202208910

Vitor Moreira - up201900198

Index

User Instructions

1.1 Main Menu

The initial menu displayed when the project launches allows the user to choose between:
- **PLAY:** Redirects to the Play Menu for selecting typing test options.
- **INSTRUCTIONS:** Opens the Instructions Page explaining the controls and objectives of the game.
- **HIGHSCORES:** Displays a table showcasing the best results obtained in multiple tests.

The user can navigate all pages using the mouse or by pressing Esc to return to the Main Menu.

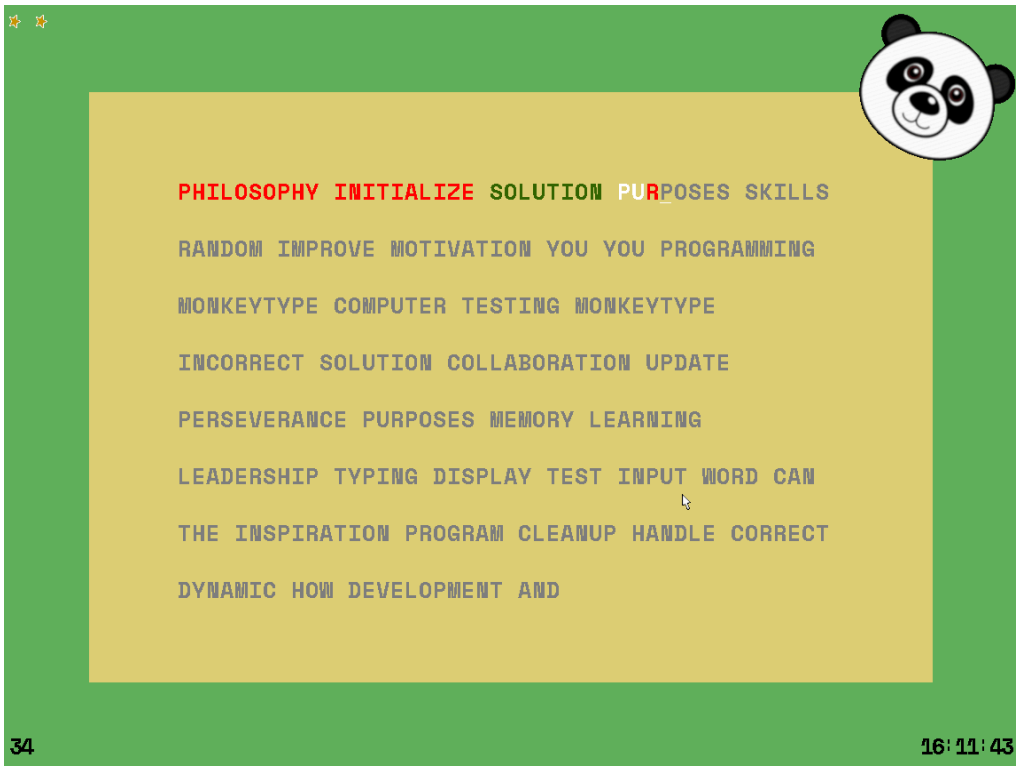
1.2 Play Menu

In this menu, the user can choose the duration of the typing test:
- 15 seconds
- 30 seconds
- 1 minute

1.3 Instructions Menu

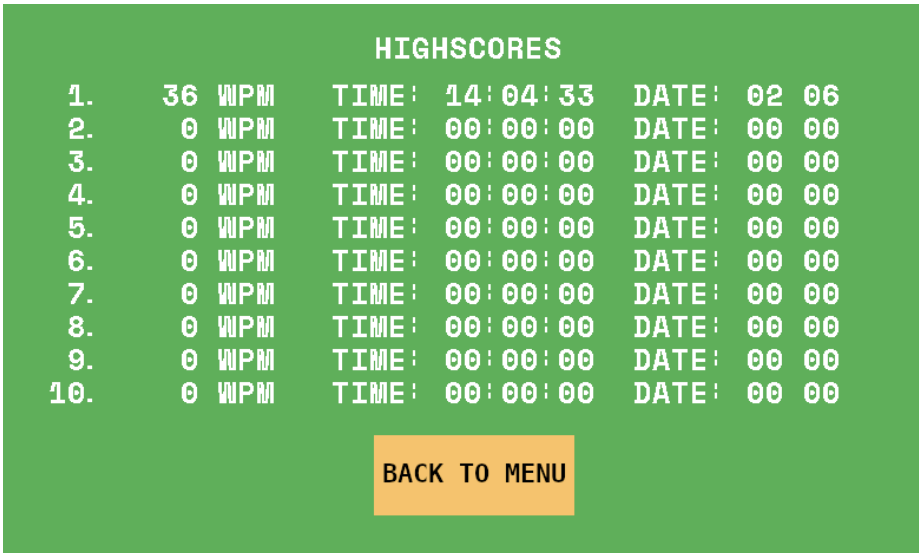
A simple menu explaining the test mechanics and controls.

1.4 Play Screen



The game screen dynamically shows the word to be typed. The correct ones get green when completed, and the wrong ones red. At the top, there are three stars-bonuses which the player can use to finish difficult words quickly. On the top right corner, there is a panda animation. The remaining time in the game is shown at the bottom left, and the current day time is presented at bottom right.

1.5 Highscores Menu



A menu showing the ten highest scores with their WPM (Words Per Minute) and the registration time.

DEMO VIDEO: https://www.youtube.com/watch?v=E1PDbRIIAgY

2. Project Status

Device Overview

Device	Purpose	Interrupts
Timer	Defines the frame-rate, Defines the time of the game, Controls animation transition	Yes
Keyboard	Menu traversal and typing test	Yes
Mouse	Menu traversal and gestures	Yes
Video Card	Screen display and menus	No
RTC	Current Date and Time , Background Color, HighScores date and time	Yes

Timer

- Functionality: The timer is used to control the frame-rate of the game, this being that the content of the frame-buffers will be copied between them in each X number of interruptions of the timer, it is also responsible for the progress of the animations. It also is responsible for limiting the time that each user has to type letters depending on the game-mode.
- Code Reference: drivers/mouse/mouse.c

Keyboard

- Functionality: The keyboard is used for navigation between menus, if you press ESC you will go to the MENU state, unless you are in the MENU state in which case you will quit the game. It is mainly used to verify the input of the users in the game itself, triggering the input verification that will possibly trigger an animation (If word is complete) and will change the colors of the letters depending if they were typed correctly or not.
- Code Reference: drivers/keyboard/keyboard.c

Mouse

- Functionality: Its position is tracked in the coordinates of its sprite, allowing the user to press buttons to navigate between states. In game it allows for the user to use a capability of completing a word by replicating a specific gesture.
- Code Reference: drivers/mouse/mouse.c

Video Card

- Functionality: Sets the initial video mode and manages all the drawings that are printed to the user screen. We load all images in XPM (Most converted to that using GIMP) format to sprites (the implementation of sprites is heavily inspired in the API provided by professor João Cardoso in https://web.fe.up.pt/~pfs/aulas/lcom2011/proj/sprite.html) and then print the sprites, the sprites have a X and Y attribute that is useful in the case of buttons for example. We use three frame_buffers, the main one, a secondary where we save the static elements (secondary_frame_buffer_no_mouse) and a third to dynamic elements (secondary_frame_buffer), in each interruption of the timer we copy the content of the secondary static one to the main, and then the contents of the one with static elements to the dynamic one. With the necessity of drawing many different words we opted by the usage of a font we got using fontForge which allows to extract each letter to XPM format (with some slight errors compared to the XPM version used by MINIX, but easily fixable).
- Code Reference: drivers/graphics/video.c

Real-Time Clock

- Functionality: Reads date/time to display during the game and to save data in the Highscores Menu. When drawing the background the time is also fetched to determine the tone of green to be used (lighter to darker) depending if it is earlier or later in the day.
- Code Reference: drivers/rtc/rtc.c

Code Organization/Structure

Module Descriptions

proj.c

Overview: This module contains the main entry point for the program and manages the overall setup, execution, and cleanup of the application. It coordinates the initialization and interaction between various components such as the keyboard, timer, mouse, RTC, and graphics.

Relative Weight: 5%

Data Structures:

GameState: A structure that holds the current state of the game, including the current menu and score.

real_time_info: A structure that contains real-time clock information.

TypingTest: A structure representing the typing test state and details.

View

view.c

Overview: This module handles rendering visual elements of the application, including sprites, backgrounds, text, and other graphics based on the current game state. It manages frame buffer setup and maintenance to ensure smooth graphics rendering.

Relative Weight: 20%

Data Structures:

Sprite: Represents graphical objects, used for characters, backgrounds, and UI elements.

Caret: Represents the text input cursor with its position and dimensions.

GameState: Enum defining different states of the game (e.g., MENU, GAME).

TypingTest: Holds typing test details including words, current indices, and input status.

real_time_info: Contains real-time clock information for time-based game features.

Statistics: Tracks typing test statistics such as typed words, accuracy, and speed.

Animation: Represents animation sequences, including frames and current frame status.

messages.c

Overview: This module contains the text messages displayed in various parts of the application, such as game instructions, timer selection prompts, and player statistics. These messages provide necessary guidance and feedback to the user during the game.

Relative Weight: 2,5%

Data Structures: None

Model

model.c

Overview: This module manages the core game logic and data models. It handles the initialization and management of game states, sprites, key mappings, typing tests, and statistical tracking. The module also processes user inputs and updates the game state accordingly.

Relative Weight: 25%

Data Structures:

Sprite: Represents graphical objects such as characters and UI elements.

Caret: Represents the text input cursor with its position and dimensions.

GameState: Enum defining different states of the game (e.g., MENU, GAME).

TypingTest: Holds typing test details including words, current indices, and input status.

Statistics: Tracks typing test statistics such as typed words, accuracy, and speed.

HighScore: Stores high score data including words per minute (WPM) and the time achieved.

sprite.c

Overview: This module provides functions for creating and managing sprites. It handles the loading of sprite images, memory allocation for sprite data, and cleanup of sprite resources.

Relative Weight: 5%

Data Structures:

Sprite: Represents graphical objects, including properties such as width, height, position, and pixel data.

Animation: Represents animation sequences, including frames and current frame status.

wordPool.c

Overview: This module defines a pool of words used in the typing test. It provides an array of words that the application can randomly select from to generate typing challenges for the user.

Relative Weight: 2,5%

Data Structures: None

Drivers

video.c

Overview: This module handles video-related functionalities such as setting the graphic mode, managing the frame buffer, and drawing basic shapes like pixels, lines, and rectangles on the screen. It is essential for rendering graphics in the application.

Relative Weight: 10%

Data Structures: None

keyboard.c

Overview: This module handles keyboard input, including subscribing and unsubscribing to keyboard interrupts, and reading scancode data from the keyboard. It ensures accurate detection and processing of key presses and releases.

Relative Weight: 10%

Data Structures: None

mouse.c

Overview: This module processes mouse input, including subscribing and unsubscribing to mouse interrupts, reading mouse data packets, and handling mouse events such as movements and button clicks. It also manages the state transitions for complex mouse gestures.

Relative Weight: 5%

Data Structures:

packet: Represents the structure for storing mouse packet data, including button states, movement deltas, and overflow indicators.

rtc.c

Overview: This module handles interactions with the real-time clock (RTC), including subscribing and unsubscribing to RTC interrupts, reading the current date and time, and converting BCD values to binary. It is essential for time-based features in the application.

Relative Weight: 5%

Data Structures:

real_time_info: Contains fields for storing current time information such as seconds, minutes, hours, day, month, and year.

timer.c

Overview: This module manages the timer functionality, including setting the timer frequency, subscribing and unsubscribing to timer interrupts, handling timer interrupts, and retrieving and displaying timer configuration. It is essential for controlling the game's timing and synchronization.

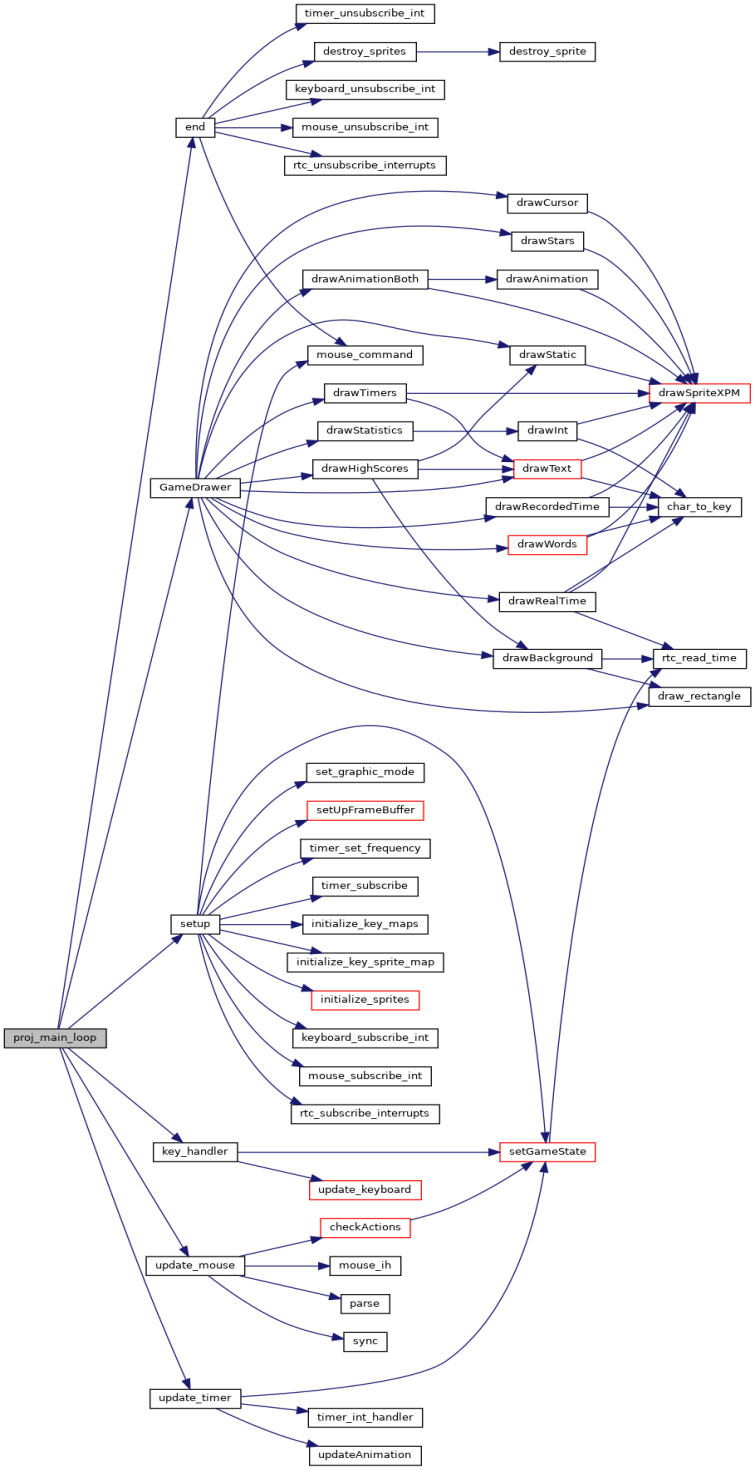
Relative Weight: 5%

utils.c

Overview: This module provides utility functions for handling basic operations such as extracting the least significant byte (LSB) and most significant byte (MSB) from a 16-bit value and reading a byte from an I/O port. These helper functions are used throughout the application to facilitate low-level data manipulation.

Relative Weight: 5%

Function Call Graph (https://git.fe.up.pt/lcom/2324/t14/g3 /-/blob/main/proj/doc/Doxygen/html/proj_8c.html)



Implementation Details

Animation

The animation were made in a different way compared to professor João Cardoso implementation, we opted for a struct that has the following attributes:

- Sprite** **frames**: Array of the frames corresponding to the animation.
- int **frameCount**: Number of frames in the animation.
- int **currentFrame**: Index of the current frame.
- bool **isActive**: Indicates if the animation is happening or not.
- int **frameDuration**: Number of frames that a sprite should be shown for, before switch.
- int **frameCounter**: Number of frames passed since last switch.

With that being said, we have a function to update the auxiliary fields (void updateAnimation(Animation *animation)), and a drawing function to show the respective frame based on that (void drawAnimation(Animation *animation, int x, int y)). In the specific case that we use the animations in our game requires us to have another auxiliary function (void drawAnimationBoth()), because the animations are done in the same place, so this last function choose what animation should be shown depending on their states or if just the default panda image should be shown.

TypingTest

The TypingTest is the structure responsible for storing the logic relative to the typing test itself, it stores the words that should be typed (chosen randomly from a wordPool), the current word index, the index of the current letter in the currentWord, and whether everything is wrong or right.
On each interrupt of the keyboard the game will check if the pressed key was a letter and if so if it was correct or not, if the key pressed was in other case the space it will save the status of the word and advance to the next one.
Another thing worth mentioning is the TypingTest is always the same size and will not increase is size depending if the user types more words than the shown, it rather moves the ones that were in the penultimate line to the first, and the ones in the last line to second, and then chooses randomly more words to fill the rest of the test. With this detail we have to note that the final statistics cannot be generated based on this structure, this is why we created a separate Statistics structure that at key process moments also stores if it was correct or not, number of words, etc…

MVC

The game is based on the Model, View, Controller architecture, meaning that all the drivers that allow the game to function are in the Controller (driver folder), the logic of the game itself is in the model, and the visual part of the game in the view. These are kept separate, giving for example the TypingTest structure as example, this structure that is responsible for the verification of the correct typing, storing of the words is handled by the model and the view draws these changes independently without ever having to handle its logic.

State Handling

In our game we make use of a state machine to handle the states in our game, we have a GameState that can be either of the modes, MENU, TIMERS, INSTRUCTIONS, HIGHSCORES, STATISTICS, we then using a state machine switch between these causing the app to have different behaviors.

Highscore

After each typing test, we evaluate some metrics like words per minute and accuracy that we then compare to previous tests to determine and showcase the top 10 results. Each entry on the Highscores consists of WPM, real-world time and the date when the test happened. In case of a session restart the top 10 results are saved into a file (highscores.txt) that is then loaded to display the results in the current session , making the top 10 the best results since the start.

Conclusions

Our project, PandaType, is aimed to develop a dynamic typing game inspired by Monkeytype. We aimed to implement all features from the initial proposal and incorporate additional functionalities, such as Real Time Clock and serial port integration, to extend beyond basic requirements.

We could successfully implement the vast majority of the features we planned. We integrated all the devices covered in labs class in our project, being the timer, keyboard, mouse and video card. Additionally, we integrated the Real Time Clock in the game. With that, we had a perfectly playable single player version of PandaType. We also advanced with the implementation of the serial port, as can be seen in the project drivers folder, but, despite our best efforts, given its complexity in sharing information between two machines, it could not be integrated with the game itself. Thus, the only goal we could not achieve was to have a functional multiplayer version of the game.

Balancing the project with academic commitments and other projects created some resource constraints. Our team managed these challenges through effective prioritization and allocation of tasks, ensuring that each project component received due attention within our limited resources. However, some team members faced hardware and software compatibility issues that complicated our efforts, particularly with the serial port, underscoring the technical programming difficulties we found.

During the development process, we also gained a better understanding of C programming. We acquired practical experience with C typical data structures, as structs and enums, which were crucial for organizing our game logic and managing the state machine. Working with C also gave us the opportunity to better understand lower level programming, specifically relative interrupts handling, pointers use and memory management - allocating, managing and freeing memory dynamically.

Looking forward, we found enhancements that could improve the program in future iterations: completing the serial port functionality for a multiplayer version, enhancing the user interface with interactive elements and adding new game modes would better the game's robustness and user experience.

Despite the challenges, we achieved several key milestones. We successfully implemented all core features, including basic game mechanics, mouse and keyboard input, and even animations for the graphics card. This project was a valuable learning experience. We met our main goals and considered that we achieved all the course objectives.