Ambilight TV Project with Raspberry Pi

Rafael Soares Ribeiro
Masters in Informatics
and Computing Engineering
University of Porto
Porto, Portugal
up201806330@edu.fe.up.pt

Maria Aurora Bota
Bachelor in Systems Engineering
and Applied Informatics
Technical University of Cluj-Napoca
Cluj-Napoca, Romania
up202111378@edu.fe.up.pt

Diogo Miguel Ferreira Rodrigues
Masters in Informatics
and Computing Engineering
University of Porto
Porto, Portugal
up201806429@edu.fe.up.pt

Xavier Ruivo Pisco
Masters in Informatics
and Computing Engineering
University of Porto
Porto, Portugal
up201806134@edu.fe.up.pt

***Abstract*—This paper presents the steps taken to develop an Ambilight TV project clone, where an LED strip installed on a monitor is synced with the image's border colors, extending the picture beyond the screen. The platform used as both controller and video player was a Raspberry Pi. The program is divided into three processes that execute concurrently and share information via shared memory.**

I. Requirements

Since our project is heavily inspired by existing products in the market of the same nature, we modeled our requirements to be similar to these.

The system is aimed at enhancing the overall viewing experience of media consumption, so the effect of the LEDs is secondary to the playing of the video itself on the display. Ensuring that the video is played in high quality and maintains high frame rates constitutes the primary requirement.

The process of syncing the LED strip with the video content can be described as a soft deadline system, since the loss of quality is only noticeable if the deadline is missed by some small period of time, up to which the loss is unperceivable by the user. *A priori*, this was estimated to be about $100\,\mathrm{ms}$.

II. Hardware

To develop this system, we used the following hardware:

- A 24' monitor with maximum resolution 1920x1080 (in pixels) and refresh rate $60\,\mathrm{Hz}$.
- An addressable $5\,\mathrm{V}$ LED strip with 116 LEDs, model WS281x, digitally controlled through a single control pin.
- A Raspberry Pi 4 model B, with a $1.5\,\mathrm{GHz}$ CPU and $2\,\mathrm{MB}$ of RAM, running Raspberry Pi OS.
- A keyboard and mouse, to interact with the Raspberry Pi and with the intensity controller.

We laid out the LED strip around the monitor so that the sides have 20 LEDs each, and the top and bottom have 32 LEDs each. See Fig. 3 for a more detailed explanation of the LED layout.

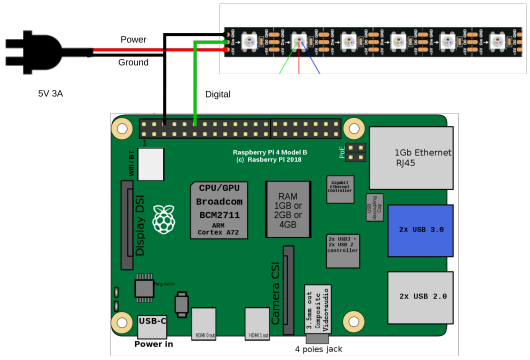

Fig. 1. Raspberry Pi - LED connection diagram

The connection between the controller and the LED strip is described in Fig. 1.

III. Software architecture

This project is composed of three programs, Fig. 2, to be run in different processes in parallel:

- *Screenreader*: reads parts of the screen and calculates the color of the LEDs;
- *Intensity Controller*: handles keyboard strokes and changes the value of the intensity of the lights according to the user input;
- *Led Controller*: receives information from the other two processes and sends the information to the LED strip.

Inter-process communication is performed using a shared memory block, protected by a named semaphore with count 1 to provide exclusive access to the shared memory, so as to avoid inconsistencies.

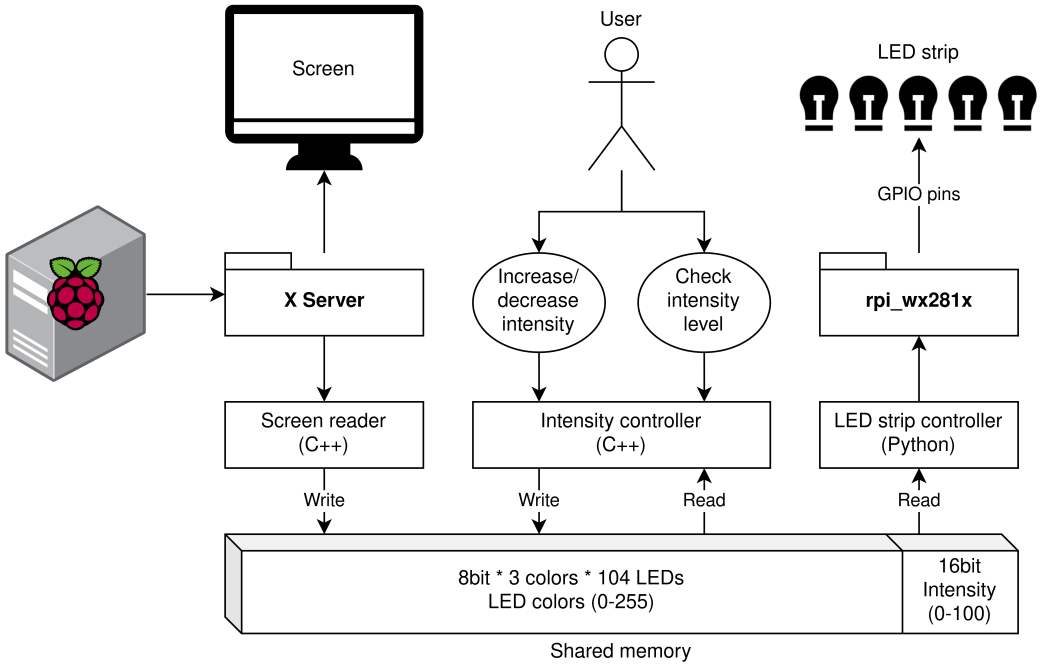

Fig. 2. System architecture. The desktop icon represents the rest of the system and all processes that use the X Server to display to the screen.

The three processes are not synchronized by any mechanism, since this isn't included in the real time requirements of our application.

A. Screenreader

The *Screenreader* is the program responsible for reading the current screen pixels and calculating the color that the LEDs should present.

The screen pixels are first read using the X Server. To be more formal, the screen is an area into which graphics may be drawn, and the contents of a screen may then be displayed on a monitor. Therefore, we are reading pixels from the screen generated by the X Server, so the X Server does not allow us to read the pixels from the monitor but it allows us to read from the X Server "screen", which is the virtual representation of what is shown in the monitor.

Each LED is assigned an area of the screen from which it will get its color. The area assigned to each LED is a rectangle of fixed width W_{LED} and height H_{LED}. We have assigned areas to LEDs in such a way that contiguous LEDs have contiguous areas, so in a 1920x1080 screen each LED is assigned an area with width $W_{\text{LED}} = 1920\,\text{px}/32 = 60\,\text{px}$ and height $H_{\text{LED}} = 1080\,\text{px}/20 = 54\,\text{px}$. The areas assigned to LEDs near the corners are completely overlapping, Fig. 3.

To speed up the reading process, we don't read the whole screen but only the pixels that are relevant for calculating LED colors, which are the border pixels. This means we are only reading 15.63% of the pixels of the screen.

With the information of all the needed pixels colors, we then calculate the average of each of those pixel squares, and from the resulting colors, we create an array with the colors that each of the LEDs should have.

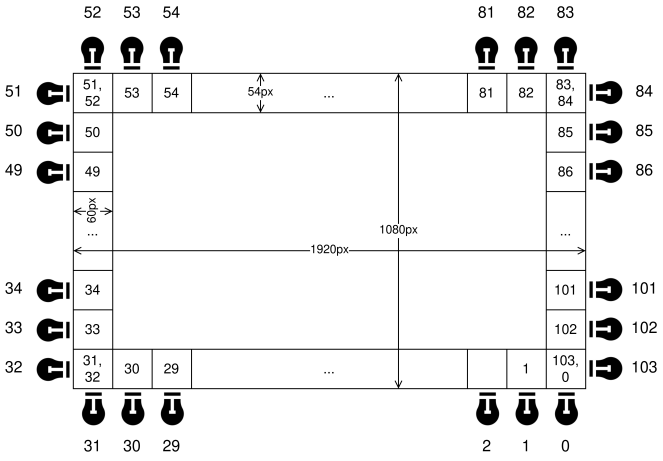

Fig. 3. Layout of the LEDs and areas that they calculate their colors from.

Finally, the LED colors are written to shared memory. Each triple of consecutive bytes corresponds to the RGB components of the color of one LED.

B. Intensity controller

The intensity controller is a short program that captures key presses of Arrow Right and Arrow Left when the terminal running that program is focused, and increases/decreases the LED intensity accordingly. This allows the user to control the LED light intensity.

Intensity is an integer between 0 (no light) and 100 (maximum intensity). The initial value is 100, and the user can change the intensity in increments/decrements of 10.

Intensity is stored in shared memory, in position `shm+104*3`, as a 16bit unsigned number (the end of the block, after all LED RGB values). Every time the user increases or decreases intensity, the intensity controller locks the semaphore, modifies the intensity, and unlocks the semaphore.

This program may be useful to the users because each person is different and some prefer more light than others, especially depending on the light in the room.

C. LED controller

The *LED controller* reads from shared memory, processes, and sends that information to the LED strip.

The color of each LED from the strip is already prepared by the screenreader process, so this program updates LED color using an exponential decay formula, and stores the colors in a list to be used when setting LED colors.

1) Exponential decay: The exponential decay helps at smoothing out rapid changes in screen colors, so the LEDs are less likely to react intensely to rapid alternating colors. This makes the user experience more pleasant because the LEDs project light into a large area (presumably a light-colored wall behind the monitor), so rapid and intense flashes over a large surface could negatively affect the experience.

This functionality also helps at making the lag between monitor update and LED update less noticeable to the human eye. This allows us to increase the time between executions of the screenreader with a minor impact on user experience since it looks like there are more updates than the actual rate at which we are collecting screen data due to the smoothing of color transitions.

The exponential decay is implemented using equations

$$\Delta_i = t_i - t_{i-1} \tag{1}$$

$$w_i = 1 - e^{\Delta_i/\beta} \tag{2}$$

$$C_i = (1 - w_i) \cdot C_{i-1} + w_i \cdot c_i \tag{3}$$

where c_i is the average color of pixels at time instant t_i, C_i is the color of the LED at time instant t_i, and w is the weight of the new color read from the screen. This method of exponential decay means that occasional delays do not affect the rate at which the LED colors are updated, because the formula of w was devised so it only uses the difference Δ_i between time instants. Therefore, the larger Δ_i is, the closer to one w is, and thus the larger the update is.

This method of updating LED colors requires precise time measurements. To meet that requirement, we used the Python `time` module, which allows us to query the current time with as much precision as provided by the system by using function `time.time()`.

IV. Performance

In order to check the performance of our program, we used the `high_resolution_clock` with the precision set to 1 ns for the C++ programs and the time library in python.

The tests consisted of measuring the average execution time of each process over 10 runs. These were run on the Raspberry PI with two different system resolutions (1920x1080, 720x400), as well as using different system priorities. A Chromium tab playing a Youtube video was used as the video player, for simplicity's sake. To measure the video player's performance, we used the Chrome DevTools FPS meter.

Despite having precise measurements of each of the programs, we had no rigorous way of measuring missed deadlines besides visually comparing the LEDs' colors and the content on the screen. The consequence of a process missing a deadline is a very brief drop in FPS. To improve performance, we decided to not constantly check if a process took more time than expected.

No test cases were designed using lower priority for the video player than for the remaining processes, since the consequences of this go directly against our requirements.

Three different tests were run for each resolution, using the default, high and low priorities for all processes.

A. Default Priorities

This test case used default priorities for all processes (80).
Resolution at 1920x1080:

Program	Average	Worst Case
Screenreader	94.20 ms	139.84 ms
Intensity controller	0.02 ms	75.48 ms
LED controller	23.60 ms	34.01 ms
Video player	34.2 fps	23.4 fps

Resolution at 720x400:

Program	Average	Worst Case
Screenreader	6.51 ms	8.64 ms
Intensity controller	0.04 ms	0.12 ms
LED controller	0.95 ms	2.5 ms
Video player	58.2 fps	40.2 fps

B. High priority

In this test, all processes were executed with a priority of 40 and the video player with a priority of 80 and then 35. This test was ran in order to ascertain if setting the priority of our processes higher than most system programs would benefit the experience.

1) Video player default priority (80):

Resolution at 1920x1080:

Program	Average	Worst Case
Screenreader	64.27 ms	83.44 ms
Intensity controller	0.03 ms	25.70 ms
LED controller	5.75 ms	28.00 ms
Video player	20.1 fps	17.7 fps

Resolution at 720x400:

Program	Average	Worst Case
Screenreader	6.60 ms	8.55 ms
Intensity controller	0.04 ms	0.08 ms
LED controller	0.59 ms	0.70 ms
Video player	46.5 fps	31.5 fps

2) Video player high priority (35):

Resolution at 1920x1080:

Program	Average	Worst Case
Screenreader	70.88 ms	91.04 ms
Intensity controller	0.04 ms	12.65 ms
LED controller	9.23 ms	24.50 ms
Video player	35.1 fps	28.4 fps

Resolution at 720x400:

Screenreader	5.99 ms	12.52 ms
Intensity controller	0.04 ms	0.16 ms
LED controller	0.41 ms	1.45 ms
Video player	36.2 fps	54.0 fps

C. Low priority

In this test, all processes were executed with a priority of 100 and the video player with the default priority (80). This way we could see if the delay forced by other programs would influence the experience in our system.

Resolution at 1920x1080:

Program	Average	Worst Case
Screenreader	94.49 ms	153.44 ms
Intensity controller	0.08 ms	24.09 ms
LED controller	9.04 ms	21.91 ms
Video player	41.2 fps	35.1 fps

Resolution at 720x400:

Program	Average	Worst Case
Screenreader	7.92 ms	10.47 ms
Intensity controller	0.07 ms	3.11 ms
LED controller	0.64 ms	0.85 ms
Video player	44.5 fps	30.2 fps

As predicted, process priority slightly influenced average and worst case execution time of the processes. Higher priority of the three programs developed resulted in better execution times and higher priority of the video player resulted in higher frame rates. Lower priorities did not affect performance comparatively to the default ones. The impact of this factor was, however, negligible to the experience, since the best results came from lower resolution test runs.

Despite the great gap in performance between low and high resolution test runs, we still believe that at 1920x1080 resolution, the effect of the LEDs is enjoyable, and as such downgrading resolution is not a needed compromise.

V. Conclusion

Our initial requirements definition was not strict and can be summarized as building a system with negligible delay, such that the human eye would not be able to notice missed deadlines in the LED's response, all the while not sacrificing video quality. We feel that this was achieved, as is evident in the video demonstration in the last section of the paper.

After performance tests, we decided to schedule the screen-reader program for running at 10FPS and the led controller at 20FPS in order to avoid missing deadlines from both of the programs. Even though a refresh rate of 10FPS may cause sudden color changes to be more noticeable, the effect created by the exponential delay implemented on the led controller counteracts this and keeps these cases from impacting the experience.

VI. Demonstration

Available here: https://youtu.be/_RozNdIdeds