

Terminal Renderer Program

Introduction

The terminal renderer program is a solution designed to interpret and render a terminal screen with some items printed in it based on a binary communication format. The program processes commands embedded in a binary stream and transforms them into a visually accurate terminal output. The program has support for monochrome, 256 and TrueColor terminal colour systems.

How It Works

The program is built to read a binary input stream that encodes terminal rendering instructions. The process involves:

- Binary stream parsing: The program reads and interprets commands using a predefined communication format.
- Command execution: Each command modifies the terminal screen buffer.
- Screen rendering: The program outputs the final representation of the terminal screen to the user.

Differentiation

What makes this program unique is:

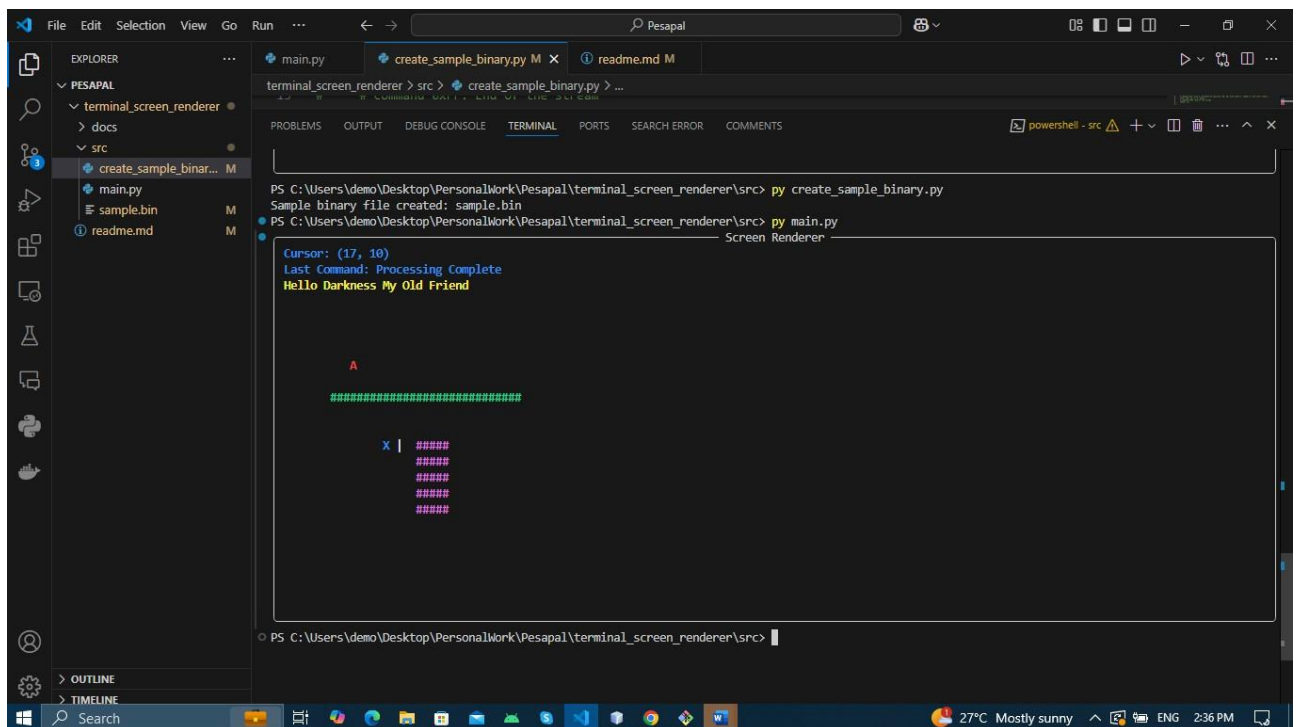
- The cursor blinker which will always show the position of the cursor while the program is running, also show the final position of the program when the program completes (Unless the screen is cleared in which case it will return to the position (0, 0)). This runs on a different thread so that it doesn't interfere with the command execution which happens on the main thread.
- The visualization which allows the user to see as the commands are being executed. (This started as a debugging feature but later decided its great for user experience. It was hard to tell if the program was working or not when the clear command would be executed in the end).
- (In progress) Updating the cursor position to move along with the text as it is being printed. Figured it would complement the visualization aspect of the program. Wouldn't be an issue if I was just plastering the result. But then if the clear command existed in the list of commands, then all the users would be seeing is an empty terminal screen.
- Added one more command (Draw Rectangle) and the test case attach showcases it.
- Added a panel at the top to show each of the commands as they are being executed and the position of the cursor.

Test Case

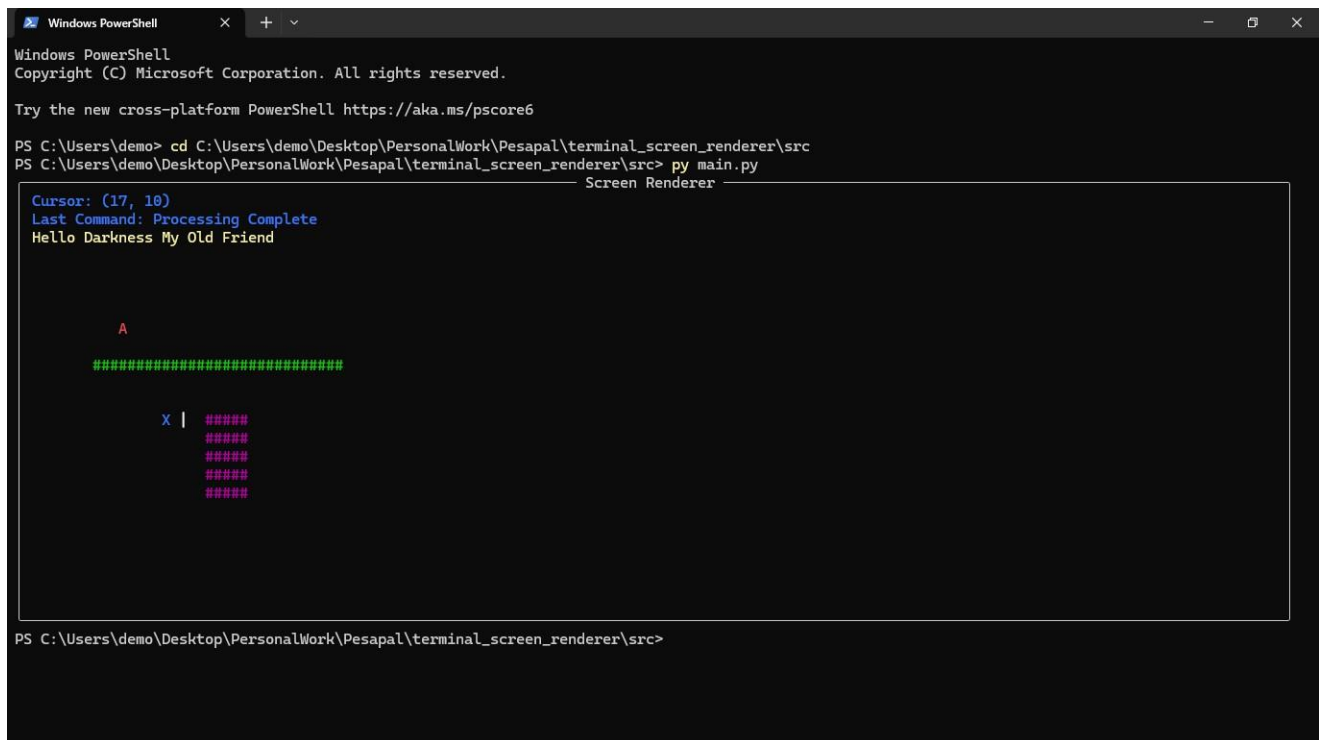
```
commands = [  
    (0x1, [40, 20, 2]), # Screen setup: 40x20, 256 colors  
    (0x2, [10, 5, 1, ord('A')]), # Draw char at (10, 5) with color 1, 'A'  
    (0x3, [7, 7, 35, 7, 2, ord('#')]), # Draw horizontal line with '#'  
    (0x4, [0, 0, 3] + list(b"Hello Darkness My Old Friend")), # Draw text  
    "Hello" at (0, 0) with color 3  
    (0x5, [15, 10]), # Move cursor to (15, 10)  
    (0x6, [ord('X'), 4]), # Draw 'X' at cursor with color 4  
    (0x5, [17, 10]), # Move cursor to (17, 10)  
    (0x7, [20, 10, 5, 5, 5]), # Draw rectangle at (20, 10) with width=5,  
    height=5, color=5  
    (0xFF, []), # End of stream  
]
```

Output

VSCode



Windows Terminal



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\demo> cd C:\Users\demo\Desktop\PersonalWork\Pesapal\terminal_screen_renderer\src
PS C:\Users\demo\Desktop\PersonalWork\Pesapal\terminal_screen_renderer\src> py main.py
Screen Renderer

Cursor: (17, 10)
Last Command: Processing Complete
Hello Darkness My Old Friend

A
#####

X | #####
   #####
   #####
   #####
   #####
```

Challenges

There were challenges encountered during the development process, and I believe I should share them.

- First, I was using the curses module from python but after trying to run I was getting library clashes. Upon some research, I found out that curses module depends on `_curses` which isn't natively supported on Windows, hence the switch to rich. Rich is excellent for rendering colours and styled texts, has support for ANSI colour codes which is what is being used in the program, and this makes it easier to switch between monochrome and coloured modes.
- When testing out the function of visualizing the commands, the program run but it was creating a new terminal screen for each command, which wasn't the desired case. Upon research, I found out about the Live submodule in rich which creates a terminal that can be updated real-time.
- Upon switching to live, I had to modify the solution because the Live submodule expects Str, Text or Object to render instead of the normal strings. This was especially troublesome to identify because the program was initially working.
- Upon testing monochrome option in Gitbash, the program now won't show colors on Gitbash and won't show the visualization, it only prints the result. I am still looking into this.

Lessons

This problem and coming up with the solution have enabled me to gain a deeper understanding of how terminals work. The thread management that may be involved when it comes to various actions such as cursor management, the colour schemes of various terminals and the limitations of environments (Windows in this case).

Diving back into python after such a long time has refreshed my mind and reminded me of how simpler things were in python with not too much boiler plate code.

I have also been able to learn about processing binary streams, specifically how computer interpret each command. I'm sure it runs far deeper than what was in the problem, but this was a great start.