Project Proposal: terminal-screen

1. Project Name and Tagline

Project Name: terminal-screen
Tagline: A terminal-based renderer interpreting binary commands to draw on screen.

2. Team Members

Team Members:

- **Boniface Munga**
  Role: Lead Developer and Architect
  Responsibilities: Overall system design, development of the core rendering logic, integration with terminal output.
  Why this role: Given my focus on backend systems and interest in terminal applications, leading this project will allow me to apply my knowledge of low-level system programming and terminal handling, while learning more about OCaml's functional capabilities in this context.

3. Technologies

- **Languages**: OCaml, with possible use of libraries such as `notty` for UI abstraction (or manual ANSI escape codes for fine control).
- **Terminal Output**: ANSI escape codes or **Notty** (depending on the final decision for UI handling).
- **Libraries/Tools**:
  - **Lwt**: Asynchronous I/O in OCaml for handling terminal input/output asynchronously.
  - **Notty** (if chosen): For higher-level terminal UI composition.
  - **Unix module**: For interacting with system terminals.
  - **Dune**: Build system for OCaml projects.
  - **Development Tools**: VS Code, OCaml compiler (`ocamlopt`).
- **Data Format**: Binary communication format defined in the specification.

Technology Trade-offs:

- **ANSI Escape Codes** vs. **Notty**:
  - **ANSI Escape Codes** offer maximum control and simplicity (no external dependencies), but require manual handling of screen operations and a more complex implementation for interactive features.
  - **Notty** abstracts the terminal handling, offering quicker UI development and better composability. However, it adds a dependency and could impose some limitations if ultra-fine control is required later.

4. Challenge Statement

Problem to Solve:

The project will interpret binary commands, communicate with a terminal, and render screen content according to a specified protocol. The binary stream will contain commands to configure the terminal screen, place characters, draw lines, render text, move the cursor, and clear the screen.

What it won't solve:

The project won't implement advanced terminal features like mouse events, advanced animation, or networked screen rendering. It focuses strictly on drawing ASCII characters, lines, and text, and controlling the screen through basic command handling.

Target Users:

- **Developers** who need a low-level terminal interface for rendering content based on binary data.

Locale Relevance:

This project is terminal-based and will work across systems supporting basic terminal output, such as UNIX-based environments (Linux, macOS). It is not dependent on a specific locale.

5. Risks

Technical Risks:

- **Incorrect Rendering**: Misinterpretation of the binary commands or handling edge cases (such as out-of-bounds coordinates).
    - **Safeguard**: Implement boundary checks and logging for invalid inputs. Testing with a variety of screen dimensions and edge cases.
- **Concurrency Issues** (if using Notty or async calls in OCaml): Interleaving terminal operations may cause race conditions.
    - **Safeguard**: Use `Lwt` for handling I/O operations asynchronously, ensuring that terminal rendering does not block other operations.

Non-Technical Risks:

- **Learning curve**: New to advanced OCaml techniques
- **Strategy**: Dedicate initial time to research and trial examples.

6. Infrastructure

Branching and Merging Strategy:

- **GitHub Flow**: For version control, feature branches will be used to work on specific components (screen setup, character drawing, etc.). Pull requests will be used for review and merging into the main branch after testing.

Deployment Strategy:

- The project will be run directly within a terminal environment, requiring no server-side deployment. It may be packaged for easy installation via OCaml's `opam` or simply executed directly via the command line.

Data Population:

- The binary data stream will be fed into the program, simulating input from a screen renderer. This will be parsed and processed accordingly.

Testing Strategy:

- **Unit Tests**: Testing each individual command (e.g., screen setup, character drawing, etc.).
- **Integration Tests**: Testing the rendering flow from receiving a binary stream to drawing on the terminal.
- **Manual Testing**: Verification of screen rendering in different terminal environments.

7. Existing Solutions

Similar Products:

- **Curses Library** (for C): Provides high-level terminal UI functionalities like window handling, screen drawing, etc.
    - **Similarities**: Both aim to provide terminal-based rendering.
    - **Differences**: This project is focused on interpreting and rendering binary commands specifically, whereas Curses abstracts UI components into windows and controls.
- **Notty** (for OCaml): Provides terminal UI functionalities that could be used for this project.
    - **Similarities**: Handles terminal screen rendering in a composable manner.
    - **Differences**: Notty abstracts more details, while this project focuses on manual rendering based on binary input, with low-level control over screen manipulation.