ChatGPT

Tech Stack Options for an ASCII/ANSI Art Editor

Developing an ASCII/ANSI art editor with a **custom UI/UX** (targeting Linux desktop first, with potential web deployment) can be approached in multiple ways. Below we outline recommendations in **Rust**, **Python**, and **other stacks**, along with suitable frameworks for each:

Option A: Rust for a Native & Web-Capable Editor

Rust offers high performance and a single-binary deployment, which is great for a desktop app on Linux. Modern Rust GUI libraries also support cross-platform development and even web compilation (via WebAssembly):

- **Iced (Rust)** – A cross-platform GUI library for Rust focused on simplicity and type-safety (inspired by Elm) [1]. Iced can deliver a native desktop UI and is suitable for custom drawing (you could create a widget to render your text canvas). Its architecture (Elm-like update/view) helps in managing state cleanly.
- **egui + eframe (Rust)** – An easy-to-use immediate-mode GUI library in Rust. *egui* is simple, fast, and runs both natively and on the web (via WASM) [2] [3]. Using the official **eframe** framework, you can write an app once and deploy it to Linux, Windows, Mac, and even compile to WebAssembly for a browser-based version [3]. This could cover your "desktop first, but web-based explored" requirement with one tech stack. Immediate mode GUI is very flexible for custom drawing (e.g. rendering an ANSI text grid via textured triangles).
- **Terminal UI (TUI) in Rust** – If you're open to a text-mode interface for the editor, Rust has libraries like `crossterm` or higher-level frameworks like **tui-rs (ratatui)**. These allow building an interactive terminal UI with ANSI colors and mouse support. This fits the ASCII art theme and keeps it lightweight in a Linux terminal, though designing a mouse-friendly UI in a terminal is more limited than a GUI.
- **Rust + Web (Hybrid)** – You could also consider a hybrid approach using web technologies for the UI with a Rust backend. For example, **Tauri** uses a Rust backend (compiled to a native binary) and a frontend rendered in a system WebView [4] [5]. This yields a much smaller app bundle than Electron by not shipping a full browser [6]. Tauri lets you write the UI in HTML/JS (or any web framework) but keeps resource usage low (e.g. memory and bundle size are a fraction of Electron's) [6]. If you prefer pure Rust, frameworks like Yew or Dioxus let you create a web-based UI in Rust (WASM) as well.

Why Rust? It gives you speed and control – useful if you later integrate real-time features like audio visualization. You get a native Linux app with efficient use of resources, and the option to target web via WASM without rewriting your core logic. The trade-off is that Rust GUI ecosystem is younger than, say, Qt or web frameworks, so you might invest time in the GUI development. However, libraries like Iced and egui are quite developer-friendly (inspired by Elm and ImGui patterns respectively) and have active communities.

Option B: Python for Rapid Development

Python can speed up development and has multiple frameworks for building both graphical and text-based UIs. This is great for quickly prototyping your editor's features, though you may need to manage performance for very large images or real-time updates. Some Python options:

- **PyQt / PySide (Qt for Python)** – PyQt5/PyQt6 or PySide6 are Python bindings to the Qt framework. Qt is a mature C++ GUI toolkit that is cross-platform (Linux, Windows, Mac) and supports native look-and-feel [7]. Using PyQt, you can create a robust, highly functional GUI relatively easily [8]. This would let you design a custom interface (Qt's canvas and painting system could render your ANSI canvas, or even use a QTable-like grid of colored text). The benefit is a professional UI toolkit with tons of features (menus, dialogs, etc.) ready-made. The downside is the deployment – your users would need Qt libs (you can bundle them using tools like PyInstaller or cx_Freeze).
- **Kivy** – An open-source Python framework specifically for building custom UI/UX applications with graphics acceleration. Kivy is cross-platform (Linux, Windows, Mac, Android, iOS) and is designed for highly custom interfaces (it even supports multitouch) [9] [10]. For an ASCII art editor, Kivy would allow you to draw the text grid as sprites or textures on a Canvas, giving you full control over how it looks and interacts. Kivy apps don't use native widgets, so your UI can be completely bespoke (which fits "fully custom UI/UX"). It's also suitable if you think about adding animations or transitions in the editor.
- **Textual (Textualize)** – If you're open to a terminal-based UI implemented in Python, *Textual* is a modern TUI framework that brings web-like layout and styling to the terminal. It allows building sophisticated text-mode interfaces with a simple Python API [11]. Notably, Textual apps can run in the terminal *and* (in the future/currently via their platform) in a web browser as well [12]. This could be an interesting way to create an ASCII art editor that runs in a terminal (over SSH, etc.) while still supporting mouse, colors, and even potentially a web view. It's built on Rich, so it supports rich text and color rendering in terminals. This option keeps the "spirit" of ASCII art by using an actual text console for editing, but you'd have to implement a lot of editing mechanics (mouse interactions, region selection, etc.) within Textual's framework.
- **Other Python Tools** – If not using the above, more minimal routes include Python's built-in `curses` (for a simple terminal editor interface) or libraries like Pygame/Pyglet for a DIY GUI (drawing text to a window manually). These give you control but require more work. For example, using Pygame you could blit a bitmap font to simulate text mode. However, given your requirements, a higher-level framework like Qt or Kivy will get you productive faster.

Why Python? It excels in rapid development and has a gentle learning curve for UI work (especially with Qt designer or Kivy's KV language). If you anticipate a lot of iteration on the editor's features or are more comfortable in Python, this route gets you results faster. Just plan for distribution (bundling a Python app can be trickier than a Rust binary) and be mindful of performance if the editor manages very large text buffers or needs real-time responsiveness (you might need to use numpy or C extensions if heavy processing is involved).

Option C: Other Tech Stacks (C++ or Web Technologies)

Considering you are open to C++ and JavaScript as well, it's worth noting a few other paths:

- **C++ with Qt or similar** – This is the traditional route for cross-platform GUI applications. Using C++ with the Qt framework would give you maximum performance and native UI on Linux/Windows/macOS. Qt is a comprehensive toolkit (everything from GUI to audio, networking, etc.) and produces highly optimized native code [13]. You could leverage Qt's text rendering to implement the ANSI art canvas and use Qt's model-view features for things like color palettes, layers, etc. The learning curve is steeper (and development slower) compared to Python, but you get full control. Another C++ option is **Dear ImGui** (immediate-mode GUI library) which can be useful if you embed your editor in a rendering loop (commonly used in game tools). There are also lighter GUIs like FLTK or SDL if you prefer simpler frameworks. Overall, C++ is a solid choice if you want to follow the footsteps of older editors (many DOS-era editors were in C/C++, and the modern cross-platform PabloDraw uses a C# (Mono) approach similar to C++). For instance, *PabloDraw* adopted the Eto.Forms .NET framework to achieve native UIs on all OSes [14] – showing that performance-critical parts can be handled in C/C++ (or .NET) while getting a cross-platform UI.

- **Web App (HTML5/JS)** – Building the editor as a web application can be attractive for a rich, easily-accessible UI. You'd create the interface using web technologies (HTML/CSS for layout, Canvas or WebGL for drawing the text grid, JavaScript/TypeScript for logic). This makes it simple to run on any platform (just open in a browser), and you could later wrap it in an Electron shell or similar for desktop distribution. In fact, the **Moebius** ANSI editor is a recent project that uses web tech – it runs as a desktop app but is built with an Electron/Web stack (providing cross-platform support on Linux, Windows, Mac) [15]. The advantage is you can leverage existing libraries (for example, any JS library for color pickers, etc.) and web development tools. However, pure Electron apps tend to be heavy on resources (since you're essentially running a Chromium instance). The alternative **Tauri**, as mentioned, can mitigate some of this by using system webviews and Rust for backend – but either way, a web UI will use more memory than a native UI. If you go this route, you could also consider a progressive web app approach (for the browser) and only use Electron/Tauri if a desktop installation is needed. Given that audio visualization might be a future component, note that the Web platform has WebAudio and Canvas which could handle that as well, albeit not as performance-efficient as native code.

- **Hybrid or Other** – There are other stacks like **Flutter** (Dart) which can build desktop and web apps with one codebase, or **Java** (with JavaFX or Swing) for desktop, but these might be less ideal given your openness to C++/JS and the mentioned options. If you want a *single* framework that can target desktop and web, you might look at something like **Slint** – a declarative UI framework that supports Rust, C++, JavaScript, and even Python for the logic layer [16]. Slint allows designing the UI in a markup language and integrating with logic in the language of your choice, compiling to native code. This is a more niche solution, but it shows how multiple language support could be possible in one project.

Summary & Recommendation

Each approach has its merits. **Rust** would give you a lean, high-performance native app (with an easier path to WebAssembly if you want a browser version of your editor). It's a great choice if you value efficiency and

plan to possibly integrate lower-level audio/graphics processing. If you do choose Rust, libraries like *Iced* [1] or *egui* [2] are strong candidates for building a custom UI from scratch. On the other hand, **Python** lets you get something working quickly. With frameworks like PyQt (leveraging Qt's power) [7] [17] or Kivy [9], you can craft a fully custom interface and iterate faster; just be prepared to optimize or move parts to native code if performance becomes an issue. And if neither of those feels right, a **C++/Qt solution** remains a reliable (if more labor-intensive) route for a polished cross-platform app, while a **Web-based solution** (JS/TS) could shine if you want maximum portability and a rich UI, at the cost of higher resource usage (Electron) unless using a hybrid like Tauri [6].

In practice, many modern ANSI art editors have embraced cross-platform stacks: for example, *PabloDraw* used .NET with Eto.Forms for native UIs [14], and *Moebius* uses an Electron web app approach [15]. Given your criteria, **my suggestion** would be to prototype quickly in Python (to flesh out the features and UI design), but consider moving to Rust (or C++) for the implementation if you need the performance and a long-term lighter-weight solution. If you're comfortable with web development, you could also prototype the UI in a browser (easy testing and iteration) and then later wrap it with a Rust backend via Tauri for a more efficient desktop app.

Ultimately, the "best" tech stack depends on your familiarity and the importance of factors like development speed vs. runtime performance. Any of the above can achieve a Linux-first ASCII/ANSI editor with a custom UI – it's about what trade-offs you're most willing to make. With modern frameworks, both Rust and Python ecosystems can deliver what you need:

- *For fastest results:* Python with Qt/Kivy will get a working editor up quickly.
- *For best performance and longevity:* Rust or C++ with a native GUI toolkit will yield a snappier, leaner app (and still cross-platform).
- *For broad accessibility:* Web tech (JS/HTML5) ensures anyone can run it in a browser, and you can still package it for desktop later.

Consider trying a small proof-of-concept in one or two of these stacks to see which aligns with your development style and requirements. Good luck with your ASCII/ANSI art editor project!

Sources:

- Rust Iced GUI library – *"iced is a cross-platform GUI library for Rust focused on simplicity and type-safety inspired by Elm."* [1]
- Rust egui library – *"egui… is a simple, fast, and highly portable immediate mode GUI library for Rust. egui runs on the web, natively, and in your favorite game engine."* [2]
- egui's eframe (multi-platform support) – *"eframe is the official egui framework, which supports writing apps for Web, Linux, Mac, Windows, and Android."* [3]
- Textual (Python TUI) framework – *"Build sophisticated user interfaces with a simple Python API. Run your apps in the terminal or a web browser!"* [11]
- PyQt (Qt for Python) overview – *"PyQt is a set of Python bindings for the Qt toolkit… It supports Windows, Mac OS and Linux… allows Python programmers to create programs with a robust, highly functional GUI, simply and easily."* [7] [17]
- Kivy framework homepage – *"Build and distribute beautiful Python cross-platform GUI apps with ease. Kivy runs on Android, iOS, Linux, macOS and Windows… built to be easy to use, cross-platform and fast."* [9] [10]

- PabloDraw cross-platform approach – *"PabloDraw… uses the awesome Eto.Forms cross-platform framework to provide native UI for Windows, OS X, and Linux."* [14]
- Moebius ANSI Editor info – *"Moebius is a new ANSI and ASCII Editor for MacOS, Windows, and Linux… (half-block brush feature, etc.)"* [15]
- Tauri vs Electron resource usage – *Comparison: Tauri (Rust + WebView) vs Electron (Chrome) – e.g., bundle size ~8.6 MiB vs 244 MiB, memory 172 MB vs 409 MB in a sample app* [6]. This highlights the efficiency of a Rust+WebView approach.

---

[1] GitHub - iced-rs/awesome-iced: A curated list of custom widgets, example projects, integrations, and resources made with/for iced
https://github.com/iced-rs/awesome-iced

[2] [3] GitHub - emilk/egui: egui: an easy-to-use immediate mode GUI in Rust that runs on both web and native
https://github.com/emilk/egui

[4] [5] [6] Tauri vs. Electron: performance, bundle size, and the real trade-offs
https://www.gethopp.app/blog/tauri-vs-electron

[7] [8] [17] PyQt: cross-platform GUI programming - DEV Community
https://dev.to/jones268/pyqt-cross-platform-gui-programming-ba9

[9] [10] Kivy: Cross-platform Python Framework for GUI apps Development
https://kivy.org/

[11] [12] Textual
https://textual.textualize.io/

[13] Qt | Development Framework for Cross-platform Applications
https://www.qt.io/product/framework

[14] PabloDraw Reviews & Pricing 2025
https://www.goodfirms.co/software/pablodraw

[15] Moebius ANSI Art Editor
https://blocktronics.github.io/moebius/

[16] Slint | Declarative GUI for Rust, C++, JavaScript & Python
https://slint.dev/