Beam Audio: Proprietary Engine Architecture (No-Framework) v2.0

1. Executive Summary

Beam Audio is a proprietary, standalone Digital Audio Workstation. It is built on a bespoke C++ engine ("BeamEngine") that handles all OS abstraction, audio processing, and rendering directly, without reliance on commercial frameworks like JUCE or Qt.

The Stack (Permissive License / Custom)

Layer	Technology	License	Responsibility
Platform Abstraction	**SDL3**	Zlib	Window creation, Input events, Threading primitives.
Audio I/O	**miniaudio**	MIT-0	Low-level hardware access (WASAPI/CoreAudio/ALSA).
File Formats	**dr_libs**	MIT-0	Decoding/Encoding WAV, MP3, FLAC.
Graphics API	**OpenGL 3.3+**	n/a	Raw GPU commands (loaded via GLAD).
Application Logic	**C++20**	Proprietary	The Beam Audio codebase.

2. Core Subsystems

2.1 The Host Shell ("BeamHost")

Instead of a JUCE `MainComponent`, we implement our own `main()` entry point and Event Loop.

- **Startup Sequence**:
  1. Initialize SDL Video & Audio subsystems.
  2. Create an `SDL_Window` (The "Studio Window").
  3. Create an OpenGL Context attached to that window.
  4. Initialize `miniaudio` device contexts.
- **The Main Loop (Game Loop Pattern)**: Unlike event-driven apps, we run a continuous high-performance loop:

```
while (isRunning) {
    handleSDLEvents(); // Keyboard/Mouse
    updateLogic();     // Animations / Physics
    renderFrame();     // GPU Draw
    swapBuffers();     // VSync
    }
```

2.2 The Graphics Engine ("BeamGraphix")

We write a raw OpenGL renderer optimized for 2D.

- **Render Architecture**:
  - **The Quad Batcher**: A custom C++ class that aggregates sprites into a single `std::vector<Vertex>` .
  - **Shader Pipeline**:
    - `UI_Shader` : Standard texture shader for faceplates.
    - `Cable_Shader` : SDF-based procedural shader for infinite-resolution cables.
  - **Text Rendering**: Use **FreeType** to generate a font atlas texture for labels (e.g., "COMPRESSOR", "RATIO").
- **Coordinate System**:
  - We implement our own `Camera2D` struct containing `Zoom` , `PanX` , `PanY` .
  - We implement a `WorldToScreen()` function to map mouse clicks to UI controls.

3. The Audio Engine ("BeamDSP")

This is the heart of the proprietary technology. It runs on a dedicated high-priority thread spawned by `miniaudio` .

3.1 The Audio Callback

`miniaudio` requests samples via a C-style callback. We bridge this to our C++ engine.

```
// This runs on the Real-Time OS Thread
void data_callback(ma_device* pDevice, void* pOutput, const void* pInput, ma_uint
    BeamEngine* engine = (BeamEngine*)pDevice->pUserData;

    // 1. Process Inputs (Microphone) -> Write to "Tape Machine" Input Ring Buffe
    engine->captureInputs((float*)pInput, frameCount);

    // 2. Traverse Graph -> Generate Output
    engine->processGraph((float*)pOutput, frameCount);
}
```

3.2 "The Machine" (Tape Logic & File I/O)

Since we cannot use JUCE's file readers, we build a custom **Disk Streaming System**.

- **File Format Handling**:
  - Use `dr_wav.h` to read WAV headers and generic PCM data.
  - **Streaming**: Do NOT load full songs into RAM.

- **Double-Buffering**:
  - The **Disk Thread** reads 4096 samples from disk -> fills `Buffer A`.
  - The **Audio Thread** reads from `Buffer A`.
  - When `Buffer A` is empty, Audio Thread swaps to `Buffer B` and signals Disk Thread to refill A.

3.3 The Graph (Signal Flow)

- **Node Structure**: A pure C++ class `AudioNode` with virtual `process()`.
- **Latency Compensation**: We must manually calculate delay compensation for the graph if we introduce look-ahead limiters.

4. Input Handling & Interaction

Without a UI framework, we must implement **"Hit Testing"** from scratch.

4.1 The Event System

1. **SDL Event**: `SDL_MOUSEBUTTONDOWN` at (800, 600).
2. **Screen-to-World**: Convert (800, 600) -> World Coordinates (e.g., 520.5, 100.0) based on current Zoom/Pan.
3. **Raycast**: Iterate through all `Modules` in reverse draw order (top-most first).
   - `if (module.rect.contains(worldPos))` -> Focus Module.
   - Check sub-components (Knobs, Switches).
4. **Interaction**:
   - If `Knob` hit: Lock mouse, hide cursor, enter "Drag Mode".
   - Calculate delta Y movement -> update `AtomicFloat` parameter.

5. Threading & Synchronization

We use C++ standard threading (`std::thread`, `std::atomic`) coupled with lock-free queues.

- **The Command Queue (Lock-Free)**:
  - **UI Thread** pushes `struct { NodeID, ParamID, Value }` into a Ring Buffer.
  - **Audio Thread** pops these commands at the start of every buffer block.
  - *Why?* Mutexes in the audio thread cause dropouts (glitches).
- **The Metering Queue**:
  - **Audio Thread** calculates RMS.
  - **Audio Thread** performs an `atomic_store`.
  - **Render Thread** performs an `atomic_load` to draw the VU meter needle.

6. Development Roadmap (From Scratch)

Phase 1: The Foundation (Month 1-2)

- Set up CMake project with SDL3 and Miniaudio.
- Get a window to open.
- Get white noise to play out of the speakers.
- Get a microphone input to loop back to the speakers.

Phase 2: The Renderer (Month 3-4)

- Implement `Shader` class (compile GLSL).
- Implement `BatchRenderer` (draw 10,000 quads).
- Implement `TextureLoader` (using `stb_image`).
- Draw the "Infinite Floor" grid.

Phase 3: The Engine Logic (Month 5-7)

- Implement the `AudioGraph` (Nodes & Cables).
- Implement `dr_wav` integration for loading audio files.
- Implement the Resampler (Windowed Sinc) for pitch-shifting audio regions.

Phase 4: The UI Interaction (Month 8+)

- Implement Mouse Drag logic for knobs.
- Implement Bezier Curve math for cables.
- Implement the "Splicing Deck" visual editor.