Beam Audio Flux

High-Performance Audio Engine & DAW Prototype

Technical Development Report

Beam Development Team

January 29, 2026

Contents

Chapter 1

Introduction

1.1 Project Overview

Beam Audio Flux is a high-performance Digital Audio Workstation (DAW) developed in C++20. Unlike commercial DAWs that rely on heavy frameworks like JUCE or Qt, Beam Audio Flux is built on a "No-Framework" philosophy, utilizing only minimal dependencies: **SDL3** for hardware abstraction (Windowing, Audio I/O) and **OpenGL 3.3** for hardware-accelerated rendering.

The primary goal of this project is to create a lightweight, modular, and highly optimized audio environment that bridges the gap between creative "Flux" (node-based modular synthesis) and linear "Splicing" (timeline-based editing).

1.2 Core Philosophies

- **Performance First**: Every subsystem, from the UI rendering to the DSP graph, is designed for cache coherence and minimal allocation.
- **Modern C++**: Leveraging C++20 concepts, smart pointers, and standard algorithms to ensure memory safety without sacrificing speed.
- **Separation of Concerns**: Strict decoupling between the UI (Main Thread) and the Audio Engine (Audio Thread).

Chapter 2

System Architecture

2.1 High-Level Design

The application is structured into three distinct layers, orchestrated by the `BeamHost`.

1. **Application Layer (`BeamHost`)**: Manages the OS window, OpenGL context, and the main event loop.
2. **Interface Layer (UI)**: A custom component-based UI system that handles user interaction and visualization.
3. **Engine Layer (DSP)**: A headless audio processing graph that runs on a high-priority thread.

2.2 The Application Loop

The entry point is standard SDL3. The `BeamHost::run()` method executes a classic game loop:

```
while (isRunning) {
    handleEvents(); // Poll SDL events, pass to InputHandler
    update();       // Update UI logic (animations, state)
    render();       // Batch geometry and submit to GPU
}
```

This loop runs independently of the audio callback, ensuring that the UI remains responsive even under heavy DSP load.

Chapter 3

The Rendering Engine

To avoid the overhead of immediate-mode GUIs or heavy retained-mode frameworks, Beam Audio Flux implements a custom **Quad Batcher**.

3.1 The `QuadBatcher`

In a typical DAW, thousands of small rectangles (knobs, faders, timeline clips) need to be drawn every frame. Issuing a draw call for each would bottleneck the CPU-GPU bus.

The `QuadBatcher` solves this by:

1. **Accumulation**: UI components submit their geometry (vertices) to a CPU-side buffer.
2. **Batching**: The batcher waits until the buffer is full or the frame ends.
3. **Single Upload**: It uploads all vertices to the GPU in one go and issues a single `glDrawElements` call.

3.2 UI Components

The UI is built as a hierarchy of `Component` objects.

- **Base Class**: `Component` handles bounds, visibility, and parent-child relationships.
- **Input Handling**: The `InputHandler` iterates the component tree in Z-order (top-most first) to determine which element consumes a mouse click.
- **Reactive State**: Components like `Knob` or `Slider` do not store business logic; they observe parameters and invoke callbacks when changed.

Chapter 4

The Audio Engine (DSP)

The heart of Beam Audio Flux is the `FluxGraph`, a dynamic audio processing graph.

4.1 Graph Architecture

Audio processing is modeled as a Directed Acyclic Graph (DAG).

- **Nodes (`FluxNode`)**: Represents a processor (e.g., Filter, Gain, Reverb).
- **Ports**: Each node has N input ports and M output ports.
- **Connections**: Defines the flow of audio buffers between ports.

4.2 Topological Sorting

To process the graph, we must ensure that a node is only processed *after* all its dependencies have produced output. We use **Kahn's Algorithm** to linearize the DAG into a flat "Execution Schedule".

```
// Simplified Logic
void FluxGraph::rebuildSchedule() {
    schedule.clear();
    // 1. Calculate in-degrees for all nodes
    // 2. Queue nodes with 0 in-degree
    // 3. Process queue, appending to schedule and decreasing neighbor degrees
}
```

4.3 The Process Cycle

The `AudioEngine` calls the graph's `process()` method inside the high-priority SDL audio callback.

1. **Prepare**: Clear all node output buffers.
2. **Sum Inputs**: For each node in the schedule, sum the audio from connected upstream ports.

3. **Process**: Call `node->process(context)`.

4. **Output**: The result is now available for downstream nodes.

Chapter 5

Data & Persistence

The project state is managed by `FluxProject` and serialized using `nlohmann::json`.

5.1 Project Structure

A project file (`.flux`) contains:

- **Graph Topology**: A list of all nodes, their types, and their unique IDs.
- **Connections**: A list of ‘(SourceID, Port) -¿ (DestID, Port)‘ pairs.
- **Parameters**: The current value of every knob and slider.
- **UI State**: Window positions and layout preferences.

Chapter 6

Conclusion & Future Work

Beam Audio Flux has successfully demonstrated that a high-performance DAW can be built with modern C++20 and minimal dependencies. The custom rendering engine provides a fluid 60FPS experience, while the graph-based DSP engine offers the flexibility required for professional audio design.

6.1 Roadmap

1. **Plugin Support**: Wrapper for VST3/CLAP plugins to be hosted as 'FluxNode's.
2. **Waveform Splicing**: Advanced time-stretching and slicing tools in the Timeline view.
3. **Multi-Threading**: Parallelizing independent graph branches to utilize multi-core CPUs.