Real-Time Audio DSP Engine Documentation

Safal Subedi

January 3, 2026

Abstract

This project implements a real-time audio Digital Signal Processing (DSP) engine on a Windows platform using PortAudio. The engine provides low-pass, high-pass, and band-pass filtering, dynamic limiter control, and 3D spatial sound localization based on azimuthal angle. The system is multithreaded to achieve low-latency audio processing and supports real-time user interaction via keyboard inputs.

Contents

1 Introduction

Real-time audio processing is critical in modern audio applications, such as headphone-based spatial audio, hearing aids, and live audio effects. This project aims to provide a modular DSP engine capable of performing:

- FIR-based low-pass, high-pass, and band-pass filtering
- Limiter control with attack and release smoothing
- 3D spatial audio localization using amplitude panning and interaural time difference (ITD)
- Real-time parameter updates using keyboard controls

The engine is designed to run efficiently on standard Windows PCs without specialized DSP hardware, using multithreading to separate audio processing, control, and computation of filter coefficients.

2 Project Setup

2.1 Software Setup

- Windows OS
- Visual Studio Code with C++ toolchain
- PortAudio library for audio input/output
- Preconfigured `.vscode/tasks.json` for building the project

2.2 Hardware Setup and Device Selection

The project can be run using a standard laptop with:

- Laptop input microphone
- Headphone or speaker output

Device selection procedure:

1. Set `isdisplayActive = true` in `globals.cpp`.
2. Run the executable. The program will display all available audio input/output devices along with:
    - Device index
    - Device name
    - Host API
    - Number of input and output channels
    - Default sample rate

3. Choose the correct input and output devices by matching:
   - Number of channels (stereo = 2 channels)
   - Sample rate (e.g., 44100 Hz)
   - API mode (WASAPI, MME, DirectSound, etc.)
4. Update the device indices in `main.cpp`:
   - `int inputMic = {deviceIndex};`
   - `int outputSpeaker = {deviceIndex};`
5. Set `isdisplayActive = false` to start normal audio processing.

3 Control Interface (`controlT.cpp`)

The `controlT.cpp` file allows real-time user interaction via keyboard. Each key press adjusts a specific audio parameter:

Key	Parameter	Description
q/a	GL	Increase/Decrease headphone amplification (Left channel)
w/s	GR	Increase/Decrease headphone amplification (Right channel)
e/d	LPF cutoff	Increase/Decrease low-pass filter cutoff frequency
r/f	HPF cutoff	Increase/Decrease high-pass filter cutoff frequency
t/g	BPF lower cutoff	Increase/Decrease band-pass lower cutoff
y/h	BPF upper cutoff	Increase/Decrease band-pass upper cutoff
3	LPF ON	Enable low-pass filter
4	HPF ON	Enable high-pass filter
5	BPF ON	Enable band-pass filter, disable others
c	LPF OFF	Disable low-pass filter
v	HPF OFF	Disable high-pass filter
b	BPF OFF	Disable band-pass filter
l	Limiter toggle	Enable/Disable limiter
k/;	Compression ratio	Increase/Decrease limiter compression ratio
,	Spatial toggle	Enable/Disable spatial localization
¡ / ¿	Azimuthal angle	Rotate sound source left/right within [-90°, +90°]
.	Auto-rotate	Enable/Disable continuous rotation of azimuthal angle

Table 1: Keyboard control mapping in `controlT.cpp`

Implementation Notes:

- All parameters are stored using atomic variables to ensure thread-safe updates.
- Frequency changes trigger recomputation of FIR coefficients.
- Spatial audio rotation is limited to azimuthal angles between -90° (left) and +90° (right).

4 Audio Processing and Delay Buffers

The audio callback in `main.cpp` uses delay buffers for:

- FIR filtering: Each channel has a circular delay buffer of length equal to the filter length. Incoming samples are pushed to the buffer, and the output is computed as the dot product of the filter coefficients with the delay buffer.
- Spatial audio: Left and right channels maintain an additional delay buffer of size `ITDmaxDelay`. The delay depends on the azimuthal angle and head size. The far ear gain is applied to simulate interaural intensity differences.

4.1 FIR Filter Delay Buffer

- `delayL[]` and `delayR[]` store past samples for convolution.
- For each audio frame, the buffer is shifted, and the current sample is added at index 0.
- Output is computed as $y[n] = \sum_{k=0}^{N-1} h[k] \cdot delay[k]$.

4.2 Spatial Localization Delay Buffer

- `SdelayL[]` and `SdelayR[]` store delayed audio samples for ITD-based localization.
- Delay in samples is computed using:

$$\text{delay} = \frac{\text{SAMPLE\_RATE} \times \text{head\_size} \times \sin(\theta)}{343}$$

- `FarEarGain` is applied to simulate interaural intensity difference.

5 Source Files Description

This section describes the purpose of each '.cpp' file in the `src/` folder:

- **main.cpp** - Initializes PortAudio, sets up audio streams, spawns threads, and implements the audio callback where filtering, limiter, and spatial localization are applied.
- **controlT.cpp** - Handles real-time keyboard input to control amplification, filter parameters, limiter, and spatial localization.
- **lpfResponse.cpp** - Computes low-pass filter coefficients using Hamming window and provides FIR impulse response for filtering.
- **hpfResponse.cpp** - Computes high-pass filter coefficients using Hamming window and provides FIR impulse response for filtering.
- **bandpass.cpp** - Computes band-pass filter coefficients and updates FIR impulse response.

- **limiter.cpp** - Implements dynamic limiter calculations including target gain computation, attack/release smoothing, and peak tracking.
- **spatialeffects.cpp** - Computes interaural time difference (ITD) and amplitude panning for 3D azimuthal localization.
- **displayAvailable.cpp** - Prints all available audio input/output devices using PortAudio.
- **globals.cpp** - Stores global variables and static arrays for filter coefficients and limiter thresholds.

6 Project Limitations

- Only azimuthal angle control (left-right) is implemented.
- Elevation or front-back separation is not supported. Implementing this would require HRTF (Head-Related Transfer Function) processing, which is not compatible with the current architecture.
- Limiter works per-channel but cannot handle extreme peaks exceeding configured thresholds in highly dynamic signals without adjusting sample rate or buffer size.
- CPU usage can increase for very low buffer sizes due to multithreading overhead.

7 Conclusion

The real-time DSP engine provides a modular framework for:

- FIR filtering (LPF, HPF, BPF)
- Limiter control with attack/release
- Azimuthal spatial audio with ITD and amplitude panning
- Real-time parameter adjustments via keyboard

This engine demonstrates a foundation for headphone-based spatial audio effects. Future work can include:

- HRTF-based front-back/elevation localization
- GUI for real-time parameter adjustments
- Additional audio effects (reverb, equalizer)