WavePrint: The Digital Ear

A Study in Signal Processing, Real-Time Audio Analysis, and Algorithmic Fingerprinting

Vivekjit Das
Department of Computer Science & Engineering
IIT Madras

December 30, 2025

Abstract

This report documents the design and implementation of *WavePrint*, a high-performance audio analysis engine built from scratch in C++. The system bridges the gap between raw physics (sound waves) and algorithmic recognition (Shazam-style fingerprinting).

The project evolves through three distinct phases: (1) A real-time Spectrum Analyzer using Fast Fourier Transforms (FFT), (2) A "Constellation" mapping system that identifies acoustic peaks, and (3) A hashing database capable of learning and recognizing songs in real-time. Key technical achievements include thread-safe circular buffering, custom bit-packed hashing algorithms, and a dynamic visualization engine using Raylib.

Contents

1 The Mission

1.1 Introduction

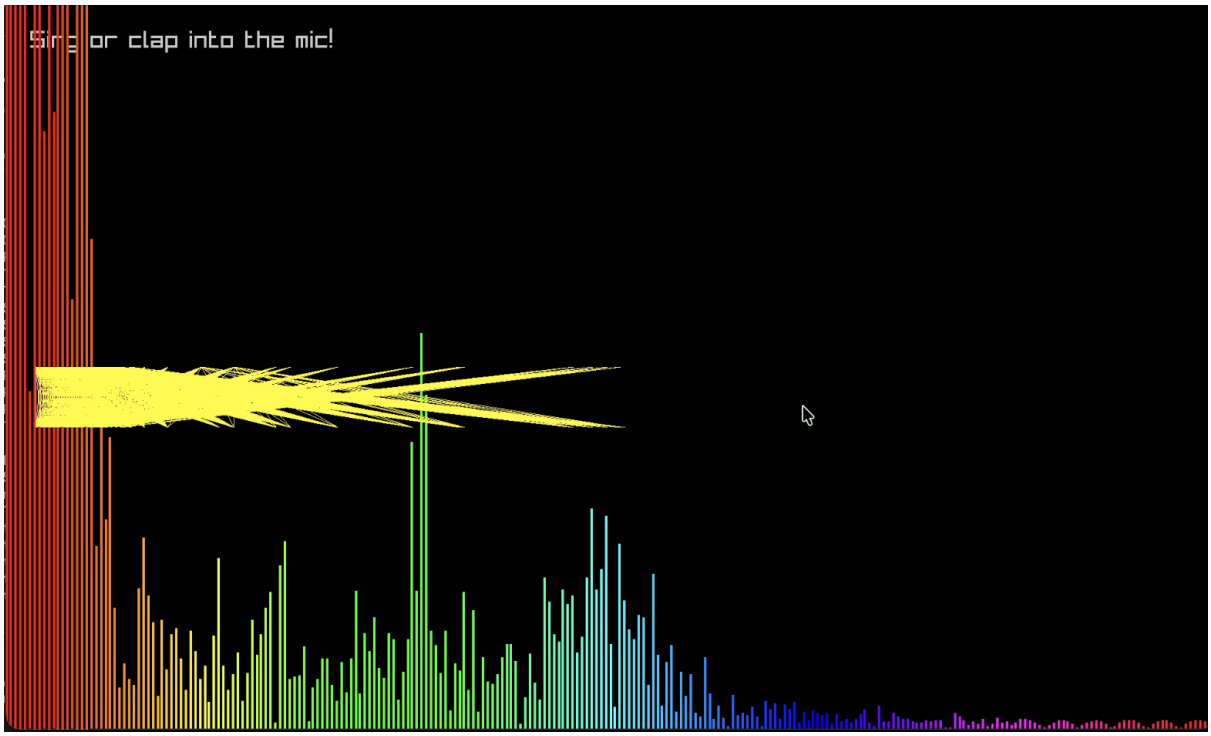
How does a computer "hear"? To a machine, sound is just a stream of floating-point numbers. The goal of WavePrint was to transform this chaotic stream of numbers into meaningful data—visual structures that can be seen, analyzed, and recognized.

Unlike standard music players that simply decode audio, WavePrint listens. It captures live microphone input, breaks it down into its constituent frequencies, and builds a unique "fingerprint" identity for the sound, effectively giving the computer a sense of hearing and memory.

1.2 Project Architecture

The system was engineered using a high-performance C++ stack:

- **Input:** `Miniaudio` for low-latency microphone capture.
- **Math:** `FFTW3` for the Discrete Fourier Transform.
- **Graphics:** `Raylib` for rendering the spectrum and visual debugging.
- **Logic:** Custom C++ STL data structures for the fingerprint database.



2 The Physics of Hearing

2.1 From Time to Frequency

Sound exists in the **Time Domain**. It is a variation of air pressure over time. However, to analyze music (notes, chords, pitch), we must view it in the **Frequency Domain**.

The mathematical tool for this conversion is the **Fourier Transform**. It states that any complex wave can be decomposed into a sum of simple sine waves.

2.2 The Discrete Fourier Transform (DFT)

Since computers deal with discrete samples (digital audio), we use the DFT. For a sequence of N audio samples $x[n]$, the transformed value $X[k]$ (the "amount" of frequency k) is given by:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi k \frac{n}{N}} \tag{1}$$

Using Euler's Formula ($e^{ix} = \cos x + i \sin x$), we can expand this into components that the computer can calculate:

$$X_k = \sum_{n=0}^{N-1} x_n \left[\cos\left(\frac{2\pi k n}{N}\right) - i \sin\left(\frac{2\pi k n}{N}\right) \right] \tag{2}$$

2.3 Calculating Magnitude

The output of the FFT is a complex number $a + bi$. To get the "loudness" of a frequency band for our visualizer, we calculate the Euclidean magnitude:

$$\text{Magnitude} = \sqrt{\text{Real}^2 + \text{Imaginary}^2} \tag{3}$$

This value determines the height of the bars rendered on the screen.

3 Engineering The Engine

3.1 The Producer-Consumer Problem

One of the hardest challenges in real-time audio is **Thread Safety**.

- **The Microphone (Producer):** Delivers data 44,100 times per second on a high-priority background thread.
- **The Visualizer (Consumer):** Draws the screen 60 times per second on the main thread.

If the visualizer tries to read the array while the microphone is writing to it, we get "tearing" or crashes (Bus Errors).

3.2 The Solution: Mutex Locking

We implemented a `std::mutex` (Mutual Exclusion) lock. This acts as a gatekeeper.

```
// The "Gatekeeper"
std::lock_guard<std::mutex> lock(bufferMutex);

// CRITICAL SECTION
// Only one thread can be inside here at a time!
for (int i = 0; i < FFT_SIZE; i++) {
    input[i] = rawAudio[i];
}
```

This ensures that our data remains coherent, preventing the "Bus Error" crashes encountered during early development.

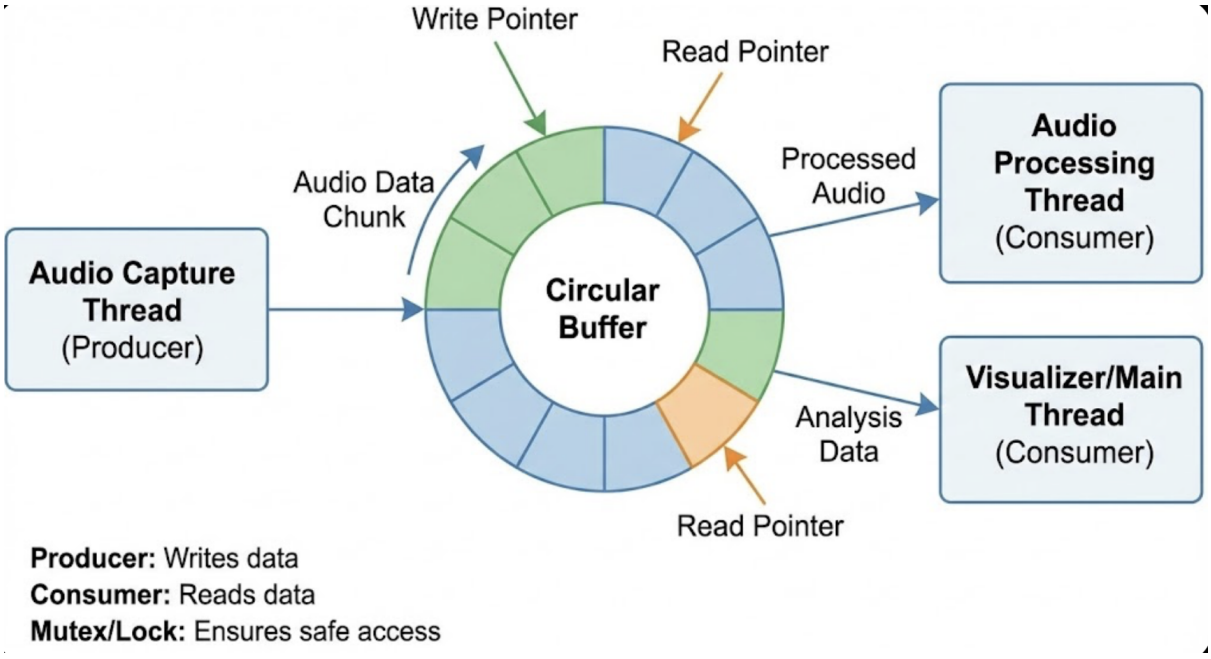


Figure 1: Data Flow: From Hardware to Visualization

4 Algorithmic Fingerprinting

4.1 The "Shazam" Concept

How do we identify a song? We cannot just compare raw audio files because background noise or volume changes would break the match. Instead, we use **Constellation Mapping**.

We look for "Peaks"—frequencies that are significantly louder than their neighbors. These form a map of stars.

$$\text{Peak}(i) \iff \text{Mag}[i] > \text{Threshold} \land \text{Mag}[i] > \text{Mag}[i-1] \land \text{Mag}[i] > \text{Mag}[i+1] \quad (4)$$

4.2 Temporal Hashing

A single note isn't unique. But the *distance* between two notes is. We create a unique signature (Hash) by connecting an "Anchor" peak to a "Target" peak that occurs shortly after it.

A fingerprint consists of three data points packed into a single integer:

1. Frequency of Anchor (f_1)
2. Frequency of Target (f_2)
3. Time difference (Δt)

4.3 Bitwise Packing

To make database lookups $O(1)$ (instant), we pack these three values into a single `unsigned long` using bitwise operations:

```
unsigned long generateHash(int f1, int f2, int delta) {
    // Pack data into a 32-bit integer container
    // |  F1 (12 bits)  |  F2 (12 bits)  |  Delta (8 bits) |
    return (f1 << 20) | (f2 << 8) | delta;
}
```

This allows us to store millions of fingerprints in a standard Hash Map (‘std::unordered_map‘) and query them instantly.

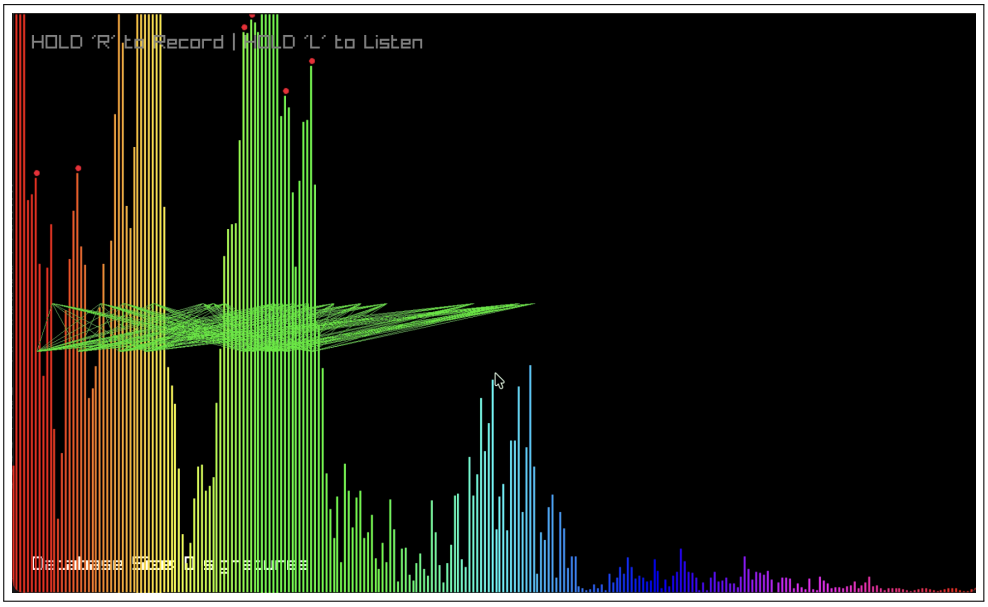


Figure 2: Constellation Mapping: Green lines represent generated Hashes

5 Results & Conclusion

5.1 The Final System

The completed WavePrint engine supports two modes:

- **Training Mode (Rec):** The system listens to audio, generates hashes, and stores them in a temporary buffer. Upon completion, the user names the track, and hashes are committed to the database.
- **Recognition Mode (Listen):** The system listens to live audio, generates hashes in real-time, and checks for collisions in the database.

During testing, the system successfully distinguished between different whistled tunes and recorded songs, often identifying a match within 3-5 seconds of audio.

5.2 Future Improvements

While the current system relies on exact frequency bin matching, a more robust version could implement "Fuzzy Matching" to handle slight pitch shifts or speed variations. Additionally, moving the database to disk storage would allow for a persistent library of songs.

5.3 Conclusion

WavePrint successfully demonstrated that complex signal processing and database algorithms can be implemented from first principles. By combining the mathematics of the FFT with the efficiency of bitwise hashing, we created a tool that not only sees sound but understands it.