Music Transcription

Technical Documentation

Jerzy Baran
13307086

Supervisor:

Alistair Sutherland

2017 – 05 – 20

This project uses Fourier Transform to analyse and process binary audio files to help musicians transcribe music from popular audio formats like MP3s to sheet music or other form of music notation. It is capable of analysing the frequencies of audio in real-time as well as slowing the music down up to 4 times to help the user with fast pieces of music.

Table of Contents

1. Motivation

1.1. Needs

The idea for the project came from my frustration with lack of sheet music on the internet for the songs I'd like to play on piano. I am a self-taught musician and have had no formal lessons in ear training. This made it difficult for me to play songs I liked from ear. Complex melodies coupled with high tempo were especially difficult to nail down.

1.2. Interests

I have been especially interested in multimedia programming since my 2nd year in DCU. It started with learning OpenGL for the graphic part of the games I developed in my spare time. Learning the internals of OpenGL made me curious about how do the audio files work exactly. I wrote my first WAV loader soon afterwards.

The prospect of joining those two subjects was very interesting when considering ideas for my 4th year project. Audio analysis and visualisation in OpenGL seemed like a perfect fundament for my project. My interests, and my needs, led me to develop an application utilising both audio DSP and OpenGL rendering, capable of extracting the frequencies out of non-human-readable audio format and displaying them in a friendly manner on the screen.

2. Research

2.1. Similar Applications

When reading this section, please note that I focus solely on applications and libraries available on Linux. I have not researched extensively any Windows-only applications as I do not see my project catering to that sector.

The most similar application and my project's direct inspiration is "Transcribe!"[1]. It does everything my project does and more, but is unfortunately a paid application. I took inspiration from its user interface, as it made the most sense to me as a user.

I am aware of the existence of certain plugins available for commercial DAWs(music making software) on Windows and Mac, capable of slowing down the sound, but I did not have a chance to try any of them out.

1 https://www.seventhstring.com/

On the free/FOSS front, there are a few libraries/terminal commands capable of taking an input file and outputting a slowed down version of it. Most notable are RubberBand[2] and SoundTouch/SoundStretch[3]. The problem with such applications is that they add an unnecessary step to the process by requiring the user to first convert the file and then perform analysis on it. Also, all of them work only on WAV files, which are uncompressed and therefore big. This adds yet another step to the process, requiring the user to first convert their music from MP3 to WAV.

2.2. Frequency Analysis

For frequency analysis, I decided to use Fourier Transform(FT). It is capable of converting the audio data from time domain to frequency domain. A derivation of this algorithm, Short Fast Fourier Transform(SFFT) is fast enough to perform in real-time.

The problem with FT, and subsequently with SFFT, is that they need at least 10 cycles of a wave at specific frequency in order to accurately measure its intensity. This is fine at frequencies of about 440Hz, but the lowest frequency a piano is capable of is 27Hz. Lower frequency means more data needs to be collected, which in turn means higher computational latency and loss of accuracy at higher frequencies(too many signals get gathered in the window and get fused together into useless data).

A solution to that problem would be wavelets. But they also use SFFT for their operation, so the above problems can not be solved by using them. Rather, they go around it. Wavelets are a series of windows, one for each frequency range. Each window performs separately from the rest, such that windows at lower frequencies are larger and take longer to fill and compute, and windows at high frequencies are small and fast. I did not feel that wavelets would add any value to my application, not to mention the cost of multiple SFFT passes for each window, so I decided against implementing them.

2.3. Time stretching

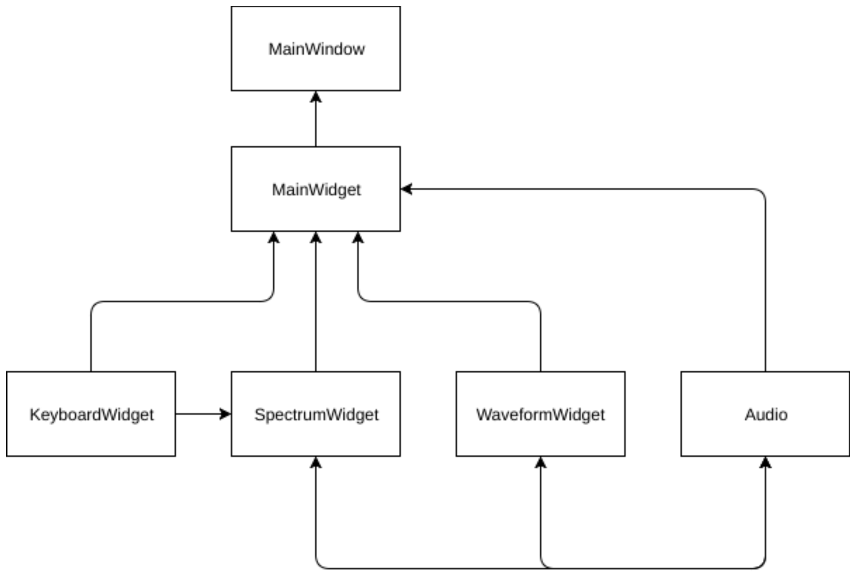
Time stretching can also be performed by using FT. In its simplest form, time stretching involves converting audio from time domain to frequency domain, interpolating the data in-between, and then converting it back to time domain, all ready to be played. Unfortunately, this alone will not make the music sound well. There are many complicated operations that need to be involved in order to receive an acceptable result, such as overlapping the windows, aligning the phases of waves, etc.

This is why I decided to delegate that operation to the library RubberBand. RubberBand allows me to easily stretch the sound and transpose it’s pitch.

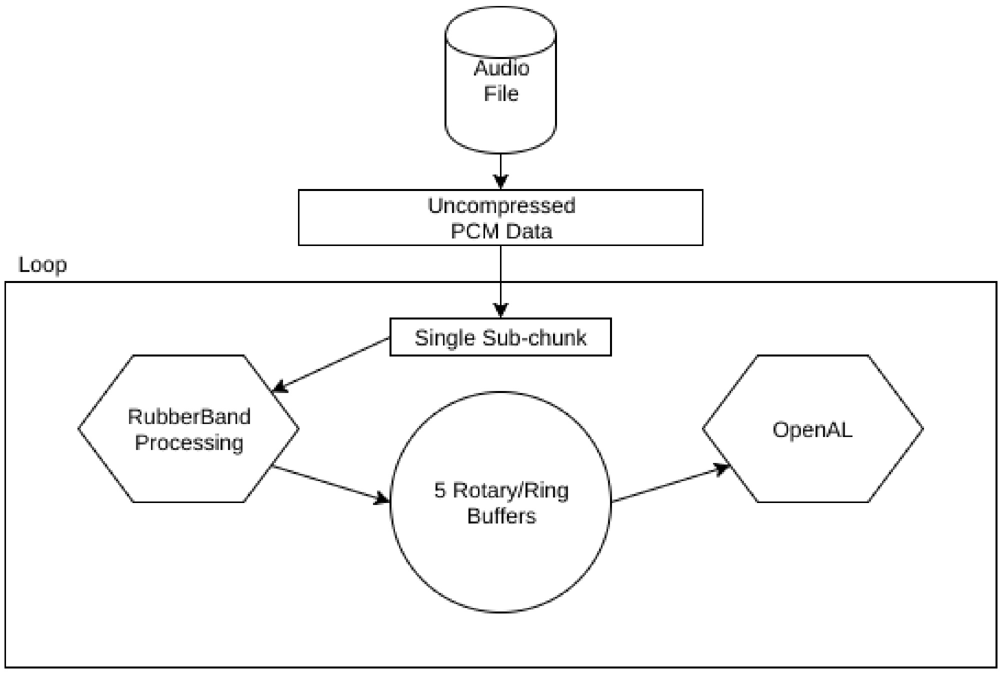
2 http://www.breakfastquay.com/rubberband/
3 http://www.surina.net/soundtouch/

3. Design

3.1. Class Diagram



3.2. Data Flow



4. Implementation

I made sure to choose cross-platform libraries for my project as I wanted it to be theoretically possible to also compile on Windows.

4.1. Language

I chose to use C++ for this project because it gives me a greater control over memory allocation. FT is a complex algorithm and even though it can be run in real-time, it's very expensive. That's why I wanted to make sure I could make up for the time lost on FT in other parts of the program by manually optimising the code. C++ makes memory management easy and I have a better view of the internals of the program.

4.2. User Interface

For the user interface, I chose to use the framework Qt. It is tried and tested, widely used in the Linux ecosystem as well as having a support for Windows. It makes it very easy to embed OpenGL canvas in the window, and has a plethora of useful utility functions.

4.3. Audio

Audio is supported by OpenAL Soft, an open source alternative to OpenAL. It's quite bare, but gives me enough control to do what I need. I'm also using OGG Vorbis and Mpg123 codecs for loading OGG and MP3 files respectively.

4.4. Graphics

Graphics is powered by Core OpenGL 3.3. That means the project is not compatible with graphics cards older than 2010.

4.5. Testing, Verification & Validation

The project required a lot of human testing. This is because the application doesn't output any data that could be easily tested by automated tests.

4.5.1 Playback & Time stretching

I have prepared four short, sample pieces of music for testing if the feature works with various formats of audio. Those are 8 bit mono, 8 bit stereo, and 16 bit equivalents of those. After implementing a feature or changing the code in any significant way, I played those files through my project to see if everything was still in order. If the project outputted noise or heavily distorted sound, I knew there was a bug somewhere, and the nature of the file (its bit rate and/or number of channels) often helped me narrow down the offending piece of code fast.

4.5.2 Tone Transposition

For testing pitch shifting, I prepared a short sine wave at 440Hz, an equivalent of the key A4 on piano. First I would play the file normally, and compare it to the sound of A4 on piano, to make sure the piano is tuned correctly and that both sounds give the same tonality. Then I would shift the sine wave up by one tone in my project, and press the next key on the piano keyboard, which would be A#4 (one tone up). If the tone transposition worked correctly, those two sounds would also match in tone.

5. Sample Code

This piece of code takes an array of complex numbers and performs FFT on it.

```
void SpectrumWidget::fft(vector<complex<double> >& x)
{
        const size_t N = x.size();
        if (N <= 1) return;

        // divide
        vector<complex<double> > even;
        even.resize(N / 2);

        vector<complex<double> > odd;
        odd.resize(N / 2);

        for(unsigned i = 0; i < N / 2; i++)
        {
                even[i] = x[i * 2];
                odd[i]  = x[i * 2 + 1];
        }

        // conquer
        fft(even);
        fft(odd);

        // combine
        for (size_t k = 0; k < N/2; ++k)
        {
                complex<double> t = std::polar(1.0, -2 * M_PI * k / N) * odd[k];
                x[k]     = even[k] + t;
                x[k+N/2] = even[k] - t;
        }
}
```

6. Problems Solved

6.1. Poorly Written Examples

In order to get the RubberBand library to work, I had to work through its example code. It took me a couple of days because the code was not commented at all, had an inconcise variable naming scheme, and contained a lot of things that were of no value to get the library working.

I worked through it by isolating the bits of code I need, and then going through them line by line, making sure I understood fully what they were doing. After that I wrote a small prototype which was very similar to the example, but without all the cruft. Then I started slowly transforming the prototype to work more like my project. This involved putting the time stretching code into a function, writing converters(library works on array of floats whereas my project uses vectors of bytes), converting the function to work on stream chunks instead of on a whole file at once, and black-boxing the function in general to make it as seamless as possible to integrate into my project.

7. Results

7.1 The Application

I managed to implement the main, most important features that I wanted. The application is capable of stretching the sound up to 4 times, and transposing the pitch up to 12 tones in both directions. Unfortunately due to the nature of FT, it's not very accurate in the lower octaves and on music containing too many instruments (eg: rock).

7.2 What's Missing

There are many things I haven't had time to implement. Most notable are:

- Some sort of settings/preferences so the user could be able to change the size of the FT window for example.
- Custom markers on the waveform for adding notes or marking the structure of the song.
- Equaliser, giving the user the capability to block out certain ranges of frequencies.

8. Future Work

I plan to implement the features I designated in the functional specifications, clean up the code, and then open source the project. I feel like what I made is worth sharing, and it might even help other unfortunate souls trying to get RubberBand working. Another thing I'd like to do in the future is to package the application for easy deployment. Right now there is no way to install the application easily.