A 3D music visualisation in OpenGL using the Discrete Fourier Transform

Matt Young
s4697249
m.young2@uqconnect.edu.au

March 2024

Abstract

Constructing computer graphics from music has important implications in the field of live entertainment. The Discrete Fourier Transform (DFT), often computed via the Fast Fourier Transform (FFT), is the typical method to convert time domain audio signals to a frequency domain spectrum. With this spectral data comes an almost unlimited number of ways to interpret it and construct a visualisation. In this paper, I investigate applying the DFT to construct a semi real-time audio visualisation using OpenGL. The visualisation consists of offline spectral data that is rendered in real-time in the form of "bars" with emissive lighting, and a set of pre-programmed camera moves computed via spherical linear interpolation.

Contents

1 Introduction

Since the first music video in (YEAR) (CITE), there has been broad interest in constructing "music visualisers": the process of building an often real-time visual representation of the audio.

- creative and technical - music visualisers used by live musicians - many different ways to interpret audio signals - demoscene inspiration

1.1 Visual inspiration

The particular music visualiser I aim to construct is an improved version of the "spectrum of bars" once used by Canadian record label Monstercat, shown in Figure 1. The label has since transitioned away from this computational music visualiser, and instead use custom music videos.

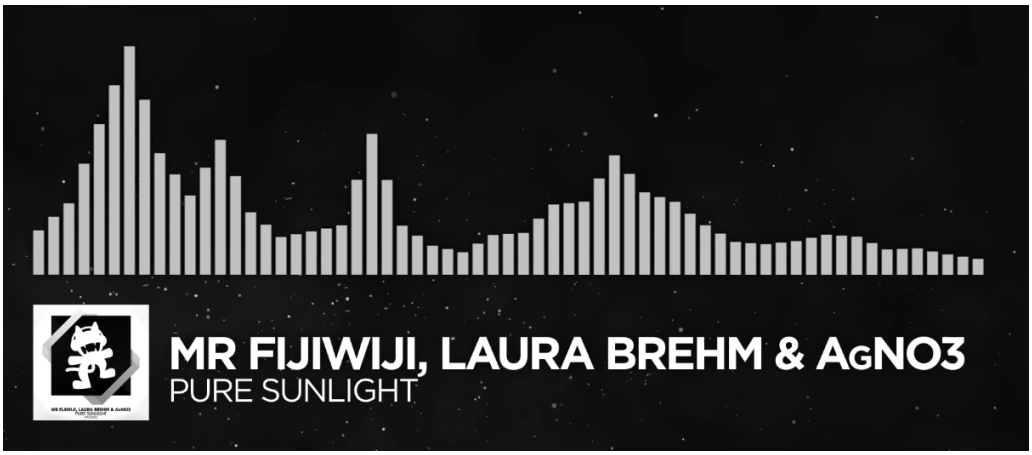

Figure 1: Visualisation of the song "Pure Sunlight", CITATION.

- demoscene inspiration

2 Architecture overview

2.1 Overall description of sub-applications

The visualiser itself is developed and tested in a Linux environment, and is split into two sub-applications: the visualiser itself (written in C++20), and the analysis script (written in Python). Signal processing is a very complex subject, and real-time ("online") signal processing is even more so. Python generally has simpler tools to address signal processing problems, such as NumPy and SciPy, so the signal processing part was moved offline into a Python script.

(DIAGRAM)

The analysis script is first run that reads a FLAC (CITATION) audio file, and produces a Cap'n Proto (CITATION) encoded binary file containing the necessary data to render a spectrum for the chosen song. Each song consists of a directory containing one audio file, `audio.flac`, and one spectral data file, `spectrum.bin`. These are stored in the `data` directory. For example:
`data/songs/LauraBrehm_PureSunlight/{audio.flac},{spectrum.bin}`

(DIRECTORY TREE)

Free Lossless Audio Codec (FLAC) is a lossless audio compression and container format. It was chosen because (REASONS).

Cap'n Proto is a binary serialisation format that is regarded as a faster successor to Protocol Buffers. Compared to Protocol Buffers, which are also a widely used binary serialisation format, Cap'n Proto has the added benefit of requiring zero decode time and supporting very fast in-memory access. This is perfect for the spectral data, which needs to be written once by the Python code, and re-read continuously by the C++ rendering code. Some of this reading takes place in the audio callback, which could be regarded as a soft real-time function, and hence requiring very minimal overhead. [1]

[1] Additionally, Cap'n P integrates nicely with my editor, Neovim, which was one of the bigger reasons for its selection.

The FLAC file for a typical song weighs in at around 10 MB, and the spectral data is only around 200 KiB thanks to Cap'n Proto's packed stream writing feature. [2]

2.2 Visualiser

The visualiser is what we see on the screen, the real-time rendering system that displays the bars, once the spectral data has been computed. It is written in C++20 using OpenGL, and is built using industry standard tools CMake, Ninja and the Clang compiler. It uses a number of open-source libraries:

- SDL2: Platform window management, keyboard/mouse inputs, OpenGL context creation
- glad: For OpenGL function loading and feature queries
- glm: The OpenGL maths library, used for computing transforms and its matrix/vector types
- Cap'n Proto: An extremely fast data serialisation format, used to transport data between Python and C++.

2.3 Analysis script

The analysis script is written in Python, and computes the spectral data itself. It takes a FLAC file, and computes the spectral data necessary to render the bars. It also uses a number of open source libraries:

- NumPy
- SciPy
- spectrum.py
- Cap'n Proto

3 Signal processing

The overarching goal of the signal processing is to turn a time-domain audio signal into a frequency domain spectrogram, similar to the Monstercat visualiser shown in Figure 1. This turns out to actually be quite an involved process, so this section will essentially a whirlwind tour of audio signal processing!

3.1 Fourier transforms and the DFT

3.2 Decoding and chunking audio

In order for the Fourier transform to be possible, a *chunk* of audio needs to be processed. In other words, we can't just process individual samples since we're doing a time-domain to frequency-domain transform. After decoding the FLAC audio using the `audiofile` library, the audio samples are then split into chunks/blocks of 1024 samples using NumPy. Additionally, the stereo signal is transformed into a mono signal by averaging the left/right channels, since multi-channel FFTs are much more complicated.

This is all just a few lines of Python:

```
# load audio
signal, sampling_rate = audiofile.read(song_path, always_2d=True)
# assume a stereo signal, let's mix it down to mono
mono = np.mean(signal, axis=0)
```

[2] Due to its speed, Cap'n P has to make some trade-offs in regards to message size vs. speed, usually preferring larger messages. Since my files are checked into Git, I used packed message writing to prefer smaller messages with an ever so slightly longer decode time.

```
    # split signal into chunks of BLOCK_SIZE
    blocks = np.split(mono, range(BLOCK_SIZE, len(mono), BLOCK_SIZE))
```

3.3 Power spectrum and periodogram

3.4 Conversion to dBFS

3.5 Sampling and binning spectral data

4 Computer graphics

5 Results

6 Conclusion

7 References