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

List of Figures

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.

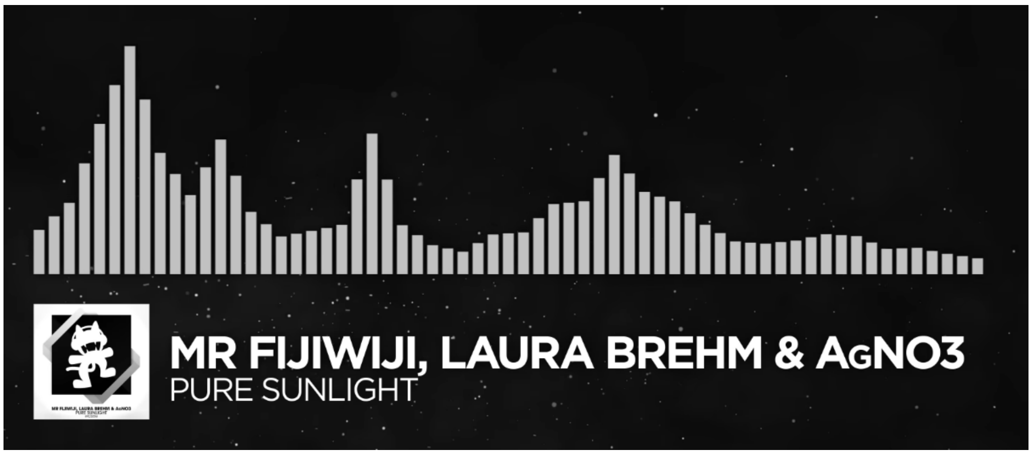

Figure 1: Visualisation of the song "Pure Sunlight" [1]

- 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]

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. 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 I'm checking the data directory into the Git version control system, I used packed message writing to prefer smaller messages with an ever so slightly longer decode time.

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, uses the OpenGL graphics API, and is built using industry standard tools CMake, Ninja and the Clang compiler. This code runs "online", meaning it runs in real-time and we would ideally like to spend no longer than 16 ms per frame (for 60 FPS). 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++.
- dr_flac: A single-file FLAC decoder.
- Tweeny: A library for tweening (interpolation).

2.3 Analysis script

The analysis script is written in Python, and computes the spectral data used by the visualiser to display the bars. It computes this data "offline", meaning it does not run in real-time, unlike the C++ code. It also uses a relatively standard setup of Python libraries, with some additions due to the audio work involved:

- NumPy
- SciPy
- spectrum.py: Used to compute the periodogram
- Cap'n Proto
- audiofile.py: For loading the FLAC file
- Matplotlib: For graphing the waveform, debugging

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

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 Background: Computer audio

Firstly, to clarify, a raw audio signal consists of a number of digital *samples* that are sent through an audio system at a particular *sampling frequency*, typically 44.1 KHz. [2] Most audio is stereo, and hence there are typically two channels (although cinemas, for example, may use many more channels). Eventually making its way out of userspace and into the kernel, these samples are converted into an analogue signal using a Digital-to-Analogue (DAC) converter, which is typically located on most PC motherboards or as a separate peripheral. The samples cause the speaker *driver* (the membrane, usually driven by magnets) to vibrate, which creates the sensation of sound. A typical digital audio signal is shown in Figure 2.

In order to build the visualiser, we will need to convert this time-domain collection of samples into a frequency-domain spectrogram.

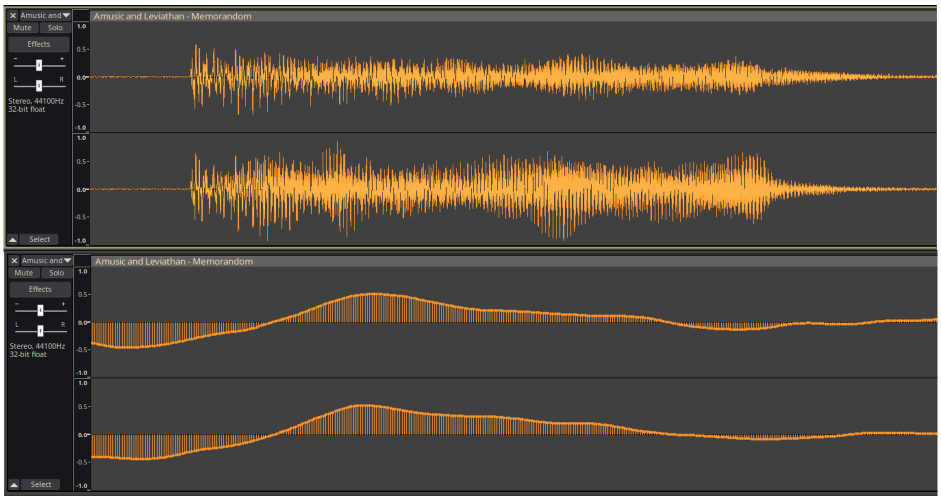

Figure 2: An audio signal loaded in Audacity. Top is zoomed out, bottom is zoomed in showing samples.

3.2 Fourier transforms, DFT and FFT

The history of the Fourier transform dates back to 1822 when mathematician Joseph Fourier understood that any function could be represented as a sum of sines [2]. From that theory, the Fourier transform was developed.

The core equation for the Fourier transform is given by [3]:

$$\widehat{f}(\xi) = \int_{-\infty}^{\infty} f(x)\ e^{-i2\pi\xi x}\,dx.$$

In simplistic terms, the Fourier transform converts a time-domain signal, where the x axis is time and the y axis is magnitude, into the frequency-domain, where the x axis is frequency and the y axis is the magnitude of that particular frequency. A diagram of this transform is shown in Figure 3.

[2]The exact reasons why are out of scope for this paper, but it has to do with the Nyquist-Shannon theorem and history with CDs.

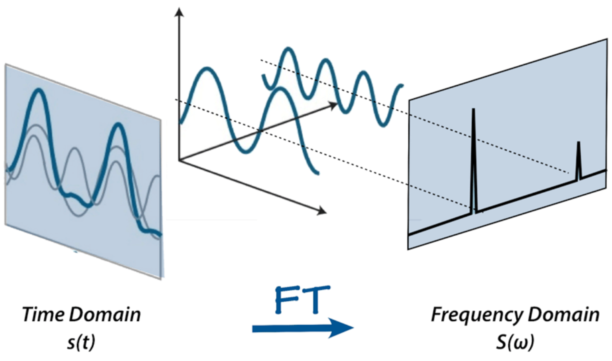

Figure 3: Diagram showing the time-domain to frequency-domain conversion of the Fourier transform

From the Fourier transform comes the concept of the Discrete Fourier Transform (DFT). The equation for the DFT is similar, but slightly different to, the equation for the regular continuous Fourier transform:

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

The DFT itself has many applications outside of audio analysis. A close cousin of the DFT, the Discrete Cosine Transform (DCT), is used in many image compression algorithms, for example.

Computing the DFT naively has a time complexity of $O(n^2)$ [4] , which is typically considered unfavourable in CS. This led to what has been described as "the most important numerical algorithm of our lifetime" [5]: the Fast Fourier Transform (FFT). The modern FFT was invented by James Cooley and John Tukey in 1965 [6], although Carl Freidrich Gauss had earlier work of a similar nature from 1805 [7]. This algorithm reduces the complexity to $O(n \log n)$, which is much more palatable.

It's also important to note here that evaluating the general DFT operates on complex numbers, and its result is also complex. However, since we are using a real-valued audio signal, we can instead make use of specialised variants of the FFT that use purely real signals [8]. The complex output is then just converted to its magnitude, e.g. given $z = x + yi$ we take $r = \sqrt{x^2 + y^2}$.

3.3 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:

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

3.4 Power spectrum and periodogram

3.5 Conversion to dBFS

3.6 Sampling and binning spectral data

4 Computer graphics

4.1 Audio and spectrum decoding

4.2 Transforming and rendering bars

5 Results

6 Conclusion

7 References

[1] Monstercat Music. *[Electronic] - Mr FijiWiji, Laura Brehm & AgNO3 - Pure Sunlight [Monstercat Release] — youtube.com*. https://www.youtube.com/watch?v=F_QrKYAv1NE. [Accessed 30-03-2024]. 2014.

[2] Eric Weisstein. *Fourier Series – from Wolfram MathWorld*. https://mathworld.wolfram.com/FourierSeries.html. [Accessed 30-03-2024]. 2024.

[3] Eric Weisstein. *Fourier Transform – from Wolfram MathWorld*. https://mathworld.wolfram.com/FourierTransform.html. [Accessed 30-03-2024]. 2024.

[4] Elias Rajaby and Sayed Masoud Sayedi. "A structured review of sparse fast Fourier transform algorithms". In: *Digital Signal Processing* 123 (2022), p. 103403. ISSN: 1051-2004. DOI: https://doi.org/10.1016/j.dsp.2022.103403. URL: https://www.sciencedirect.com/science/article/pii/S1051200422000203.

[5] Gilbert Strang. "Wavelets". In: *American Scientist* 82.3 (1994), pp. 250–255. ISSN: 00030996. URL: http://www.jstor.org/stable/29775194 (visited on 03/31/2024).

[6] James W. Cooley and John W. Tukey. "An algorithm for the machine calculation of complex Fourier series". In: *Mathematics of Computation* 19 (1965), pp. 297–301.

[7] Michael T. Heideman, Don H. Johnson, and C. Sidney Burrus. "Gauss and the history of the fast Fourier transform". In: *Archive for History of Exact Sciences* 34.3 (1985), pp. 265–277. ISSN: 1432-0657. DOI: 10.1007/bf00348431. URL: http://dx.doi.org/10.1007/BF00348431.

[8] H. Sorensen et al. "Real-valued fast Fourier transform algorithms". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 35.6 (1987), pp. 849–863. DOI: 10.1109/TASSP.1987.1165220.