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

Matt Young

46972495

m.young2@uqconnect.edu.au

May 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 3D bars. A multitude of graphics techniques are used, including: quaternion camera animation, camera shake using Simplex noise under fractal Brownian motion, a skybox, and a post-processing stage that implements chromatic aberration. The application is written in a mix of C++ and Python.

Contents

List of Figures

1. Introduction

A *music visualiser* is a computer program that takes as input an audio signal, typically music, and produces as output a graphical visualisation. Technically, this can be an entirely offline process, but most often music visualisers run in real-time. Music visualisers have an important role in the live entertainment scene [6], particularly for Electronic Dance Music (EDM). With an almost infinite number of ways to interpret an audio signal, music visualisation is an eclectic mix of art and engineering.

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]

The visualisation in Figure 1 is a common type of visualisation that visualises the audio spectrum - it shows the magnitudes of the varying frequencies that make up the song. This spectral data can be interpreted in a number of ways, for example, the MilkDrop [7] software includes visualisations that encode this in a number of complex ways. However, for our use case, bars are visually appealing and an intuitive approach to visualising the music.

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 in C++ 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. This is shown in Figure 2:

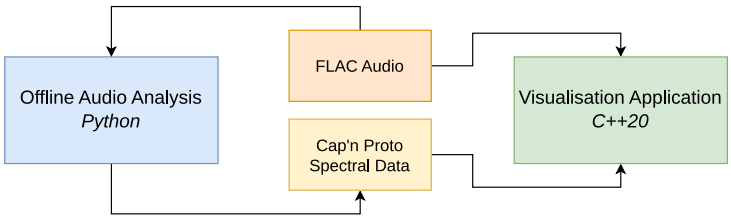

Figure 2: Block diagram of music visualisation application

The analysis script is first run that reads a FLAC [8] audio file, and produces a Cap'n Proto [9] 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}`.

Free Lossless Audio Codec (FLAC) [8] is a lossless audio compression and container format. It was mainly selected due to its ability to be easy loaded in C++ through open-source libraries like `dr_flac`, the fact that it's entirely royalty and patent free, as well as its lossless nature. Although codecs such as MPEG-3 and Vorbis are transparent at around 320 Kbps (meaning there are no perceptible differences between the compressed codec and uncompressed audio), there may still be subtle artefacts introduced in the spectrum that could be picked up by the visualiser. In any case, storing the audio in an uncompressed format introduces very little overhead and can always be transcoded later down the track if marginally smaller file size is desired.

Cap'n Proto [9] 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.

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. The Clang-Tidy and Clang-Format tools are used to ensure high quality code. 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 fast single-file FLAC decoder.
- **stb_image**: An image file decoder for many file formats.

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**: Numerical computing library.
- **SciPy**: Scientific computing library.
- **spectrum.py**: Used to compute the periodogram.
- **Cap'n Proto**: Data inter-change format, see above.
- **audiofile.py**: For loading the FLAC file.

- **Matplotlib**: For graphing the waveform, debugging

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

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.[1] 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 3.

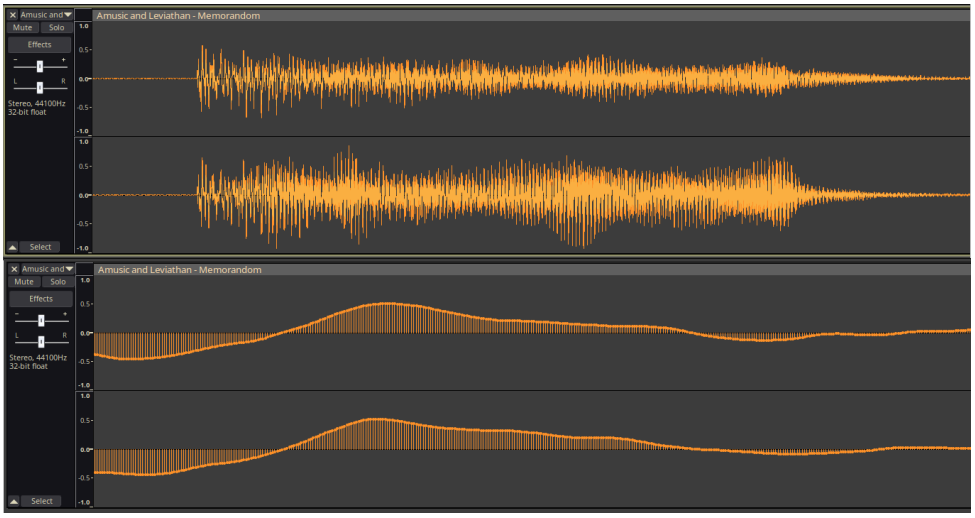

Figure 3: An audio signal loaded in Audacity. Top is zoomed out, bottom is zoomed in showing samples.

3.2. Fourier transforms, DFT and FFT

In order to build the visualiser, we will need to convert this time-domain collection of samples into a frequency-domain spectrogram. We can achieve this through the Fourier transform technique.

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 [10]. From that theory, the Fourier transform was developed.

The core equation for the Fourier transform is given by [11]:

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x)\, e^{-i2\pi\xi x}\, dx. \tag{1}$$

[1] 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.

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 4:

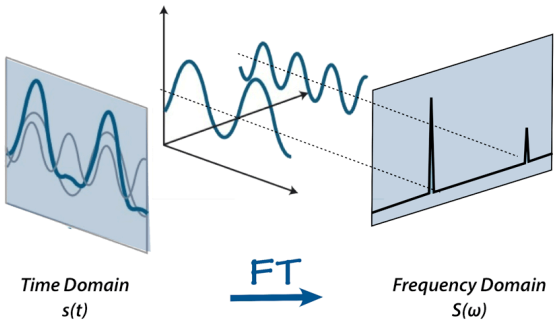

Figure 4: Diagram showing the time-domain to frequency-domain conversion of the Fourier transform [2]

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:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i 2\pi t \frac{k}{N} n} \tag{2}$$

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 $O(n^2)$ [12] , which is typically considered unfavourable in CS. This led to what has been described as "the most important numerical algorithm of our lifetime" [13]: the Fast Fourier Transform (FFT). The modern FFT was invented by James Cooley and John Tukey in 1965 [14], although Carl Freidrich Gauss had earlier work of a similar nature from 1805 [15]. 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 [16]. 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:

```
# 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

From the DFT, a power spectrum - otherwise known as "power spectral density" (PSD) - can be computed. The power spectrum "describes how a signal's power is distributed in frequency" [17]. Consider a signal $U_{T(t)}$, whose Fourier transform is given by $|U_{T(V)}|^2$. The power spectrum can be computed as follows, using the definition in [17]:

$$\mathrm{PS}\ (V) = \frac{|\ U_T(V)|^2}{N^2} \tag{3}$$

Where N is the total number of sample points.

Then, the power spectral density can then be simply computed as:

$$\mathrm{PSD}(V) = \frac{\mathrm{PS}(V)}{\Delta v} \tag{4}$$

Where Δv is the space between data points in frequency space (so, the sampling rate).

Critically, the result of this equation is in the units of amplitude squared per frequency unit - which is unwieldy and not particularly useful for our type of audio analysis. Instead, it would be better if we converted it into dBFS ("Decibels relative to full scale"). This is a unit where 0.0 dBFS is the loudest possible sound a system can emit, and it decreases negatively from there. For example, −10 dBFS is 10 dB quieter than the maximum possible sound a system can emit.

The conversion from PSD to dBFS can be achieved as follows, given our signal $U_{T(t)}$:

$$\mathrm{dbFS}_{T(t)} = \sum_t \log_{10} \frac{\mathrm{PSD}(t)}{\max \mathrm{PSD}} \tag{5}$$

Implementing the above in Python, this produces the plot in Figure 5:

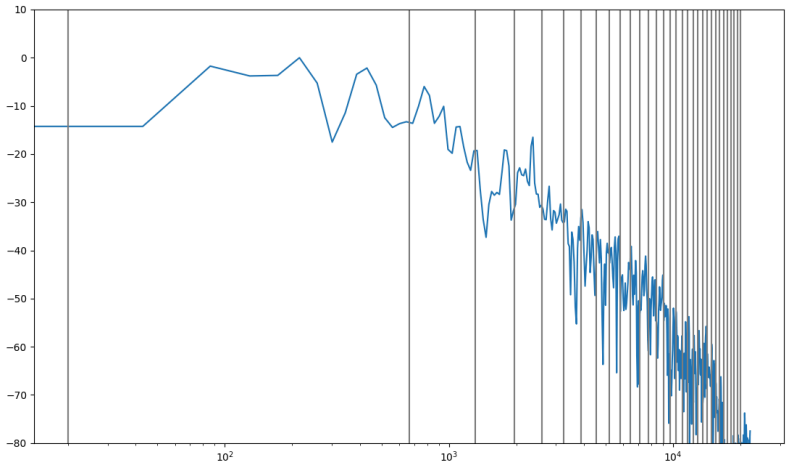

Figure 5: Power spectral density (PSD) in dbFS of a block of audio

Also to note here is that we compute a metric known as *spectral energy*, which is a measure of the "excitement" of the audio. It's computed as the magnitude squared of the FFT:

$$S_E = \sum_{i=0}^{k} \mathrm{PSD}(i)^2 \tag{6}$$

3.5. Binning spectral data

Now that we have spectral data for each block of audio, we need to process it into "bins", or blocks, and use that to construct the bars. Given we know the number of bars we want (by default, 32), we can simply sample this using NumPy:

```
samples = np.linspace(FREQ_MIN, FREQ_MAX, NUM_BARS)
```

Where `FREQ_MIN` and `FREQ_MAX` refer to the human hearing range, 20 Hz to 20 KHz, respectively.

Then, we simply walk through the linear space considering our current and previous value, and compute the mean amplitude inside this block. There is some additional code that maps this value, in dbFS, to a `uint8_t` bar height from 0 to 255, for Cap'n Proto serialisation.

Once this is computed, the data is serialised and can be loaded in C++ - and the signal processing is complete.

4. Computer graphics

4.1. Render pipeline and states overview

The visualiser uses a fairly standard, simple, forward rendering pipeline based on OpenGL 3.0. The exact rendering pipeline depends on which one of the two states the application may be in.

The application is divided in two two states: "intro" state, and "running" state. In the intro state, a fullscreen textured quad displays three introduction slides which display the project name, my name, and the song name[2].

The image in Figure 6 shows the construction of the intro slides, which are static images designed in Inkscape and exported to PNGs.

Figure 6: Construction of intro slide images in Inkscape

Once the intro is completed, the application transitions into "running" mode, which performs the actual visualisation. This mode first binds a framebuffer. Then, it draws the bars using a GLSL shader, then the skybox as a cubemap, and finally draws the framebuffer using a special fragment shader to provide some post-processing effects.

The diagram in Figure 7 shows the rendering pipeline:

[2] At this time, the song name is hardcoded and has to be manually changed when the song is changed

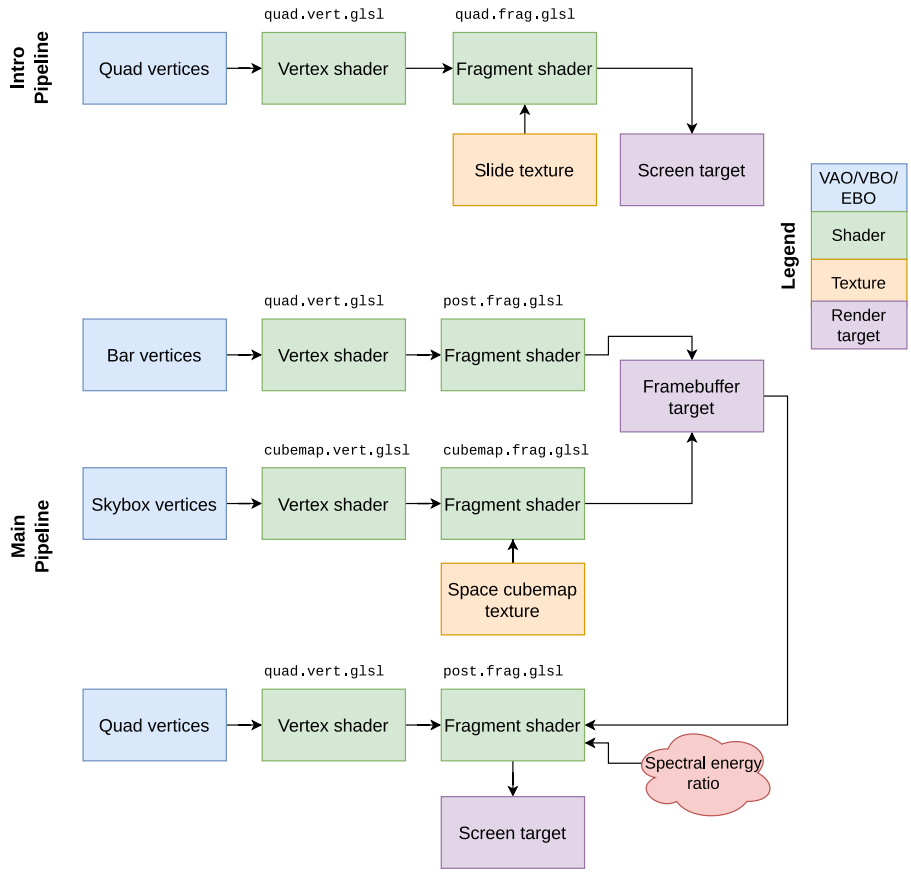

Figure 7: Diagram of graphics render pipeline

4.2. Initialisation

SDL and glad are used to initialise the system platform. SDL is used to create the window, manage the audio context, and load the GL driver. glad is the OpenGL loader which provides the function and constant definitions from the OpenGL specification.

At startup, the application is passed the path to the data directory and the name of the song to display. Once platform initialisation is complete, the FLAC audio is decoded using the dr_flac library, and the Cap'n Proto spectrum data is decoded using the C++ Cap'n Proto library. This is stored in the `cosc::SongData` class, which also handles mixing the audio into the SDL audio stream using `cosc::SongData::mixAudio`. The `mixAudio` routine also handles resampling using the SDL `AudioStream` API.

One particularly important note to make regarding platform initialisation is the audio stream. For a music visualisation application, it's important that the on-screen graphics are precisely synchronised to the audio being played. This means that a low-latency audio stream is required, or in other words, an audio stream where the number of samples submitted to the audio driver per callback is less than or equal to the spectral data block size. I achieve this by using SDL's new audio APIs, which allow resampling based on the platform audio driver's selection of what it believes is the best format. In the code, we request a particular low sampling rate using `SDL_AudioSpec`, but allow SDL to change the actual underlying datatype if it desires. For example, while dr_flac uses signed 32-bit ints, SDL may prefer to resample to floats on some audio drivers.

4.3. Transforming, rendering and lighting bars

Each frame, we query the `cosc::SongData` to figure out which spectrum block we are currently playing, and use that to query the heights of all the bars from the decoded Cap'n Proto document. The

heights are a `uint8_t`, so range from 0 to 255 inclusive. These are remapped to a float based on a configurable minimum and maximum height.

The spectrum itself simply consists of N unit cubes. The cubes are transformed by means of a transformation matrix, exactly as was done in the previous Graphics Minor Project.[3] The height of the bars is simply determined by their scale on the Y axis.

This is summarised in the following code snippet:

```
size_t barIdx = 0;
for (auto &bar : barModels) {
    // first, get bar height from 0-255 directly from the Cap'n Proto
    auto barHeight = block[barIdx];
    // map that 0 to 255 to BAR_MIN_HEIGHT to BAR_MAX_HEIGHT
    auto scale = cosc::util::mapRange(0., 255., BAR_MIN_HEIGHT, BAR_MAX_HEIGHT,
barHeight);
    // apply scale, also applying our baseline BAR_SCALING factor!
    bar.scale.y = scale * BAR_SCALING;
    barIdx++;

    // now update transforms, and off to the GPU we go!
    bar.applyTransform();
    bar.draw(barShader);
}
```

The bar shader is almost identical to the shader used in the Graphics Minor Project. This shader has a lot in common with a simple Phong [18] specular model, and was was based on the Phong specular calculations from LearnOpenGL [19]. The main work is this small block of GLSL:

```
// compute angle of this fragment's normal against the camera
vec3 norm = normalize(Normal);
vec3 viewDir = normalize(viewPos - FragPos);
float angle = max(dot(norm, viewDir), 0.0);
```

For the vertex that this fragment is attached to, we normalise its normal vector (i.e., we compute the unit vector from the given normal vector). The `viewPos` uniform refers to the camera's current xyz position, and the `FragPos` uniform refers to the world-space transformed fragment coordinates that we computed in the vertex shader - all based on the model transform matrix. Finally, we're able to compute the angle between these two vectors by taking the dot product, and clamping it to be above 0.0 using `max`.

Now that we have a single float describing the angle between the camera and the model, we need to transform that into an RGB value for coloured shading. This can be achieved using a colour map, in this case, "Inferno" from Matplotlib. I also tried the rainbow colourmap "Turbo", but Inferno works the best for this application because it shows the angle in a intuitive and visually appealing manner. I used a polynomial approximation from [20], instead of the usual 256-element lookup table, which works better for fragment shaders.

The final composite is very simple: the fragment colour is just set to the RGB value that Inferno returned:

```
FragColor = vec4(inferno(angle), 1.0f);
```

The images in Figure 8 and Figure 9 show the Phong shading. Notice the "hotspot" visible on the central bar in Figure 9, which is a classic artefact of Phong shading methods.

[3] The only difference is that transformations are computed using glm, rather than manually.

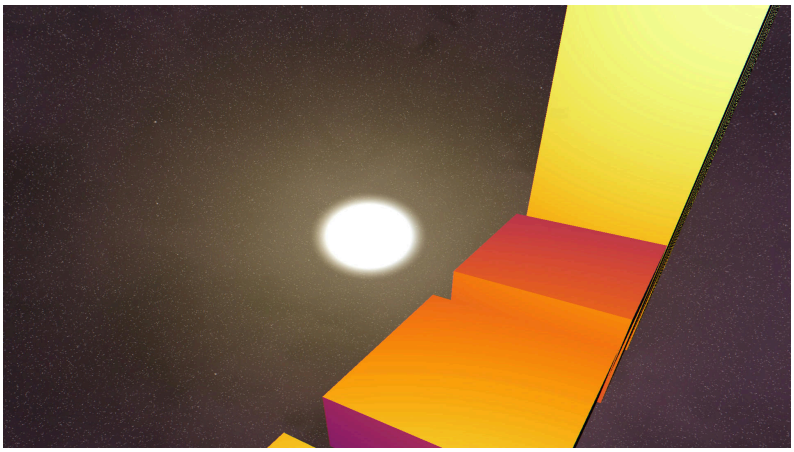

Figure 8: First image demonstrating Phong shading

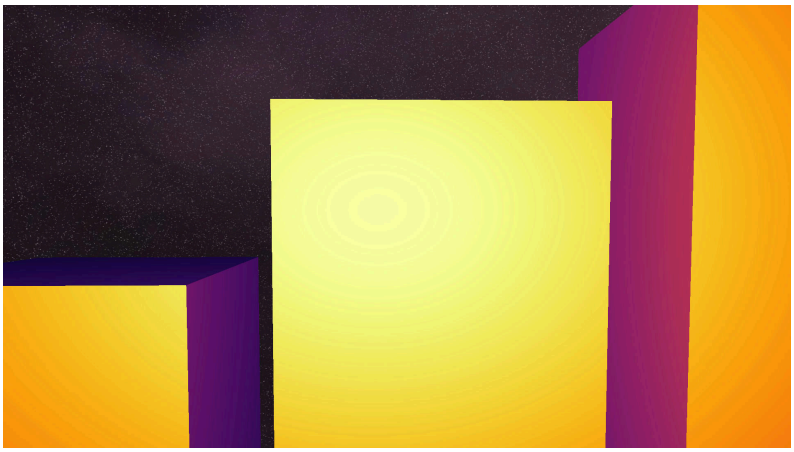

Figure 9: Second image demonstrating Phong shading

4.4. Computing camera animations

One of the goals I had in mind for the visualiser was automated and smooth camera animations, that would also be quick to describe in code. However, in order to achieve this, the camera class from the Graphics Minor Project, which was based on code from LearnOpenGL, would need to be rewritten. There was also the need for a bug-free "freecam" mode,[4] which can be moved around using the normal WASD and mouse controls. This is necessary for me to find the begin and end points for each animation and for debugging.

The new camera is based on a slightly modified version of the perspective camera from the Cinder project [21], which is a C++ creative coding framework. The main difference compared to the previous LearnOpenGL camera, is that the new camera represents its pose as just a 3D vector and quaternion, rather than a set of vectors. This means that the pose of the camera can be described much more easily, and can also be interpolated to create animations. Other than that, it still outputs the same perspective and view matrices, has an FOV and uses a perspective projection. Modifications to the upstream Cinder project camera include a small refactor, removal of unnecessary code for this project such as pick ray projection, and the addition of the freecam controller.

To implement camera moves, I chose to represent any given move as a begin pose, end pose, and a duration. This is encoded in the `cosc::CameraAnimation` class. Then, in turn, a `cosc::CameraAnimationManager` is attached to the `cosc::Camera`, which enables it to animate the camera every frame.

[4] The LearnOpenGL camera had certain bugs regarding mouse rotation in freecam mode.

```
/// The pose of a camera, with its position and orientation.
class CameraPose {
public:
    glm::vec3 pos;
    glm::quat orientation;
    ...
};

/// A camera animation
class CameraAnimation {
public:
    /// Beginning camera pose
    CameraPose begin;
    /// Ending camera pose
    CameraPose end;
    // Duration of the animation in seconds
    float duration;
    ...
};

/// Manages a camera with a set of animations.
class CameraAnimationManager {
public:
    explicit CameraAnimationManager(Camera &camera) : camera(camera) {};

    void update(float delta, float spectralEnergyRatio);

    void addAnimations(const std::vector<CameraAnimation> &animation) {
        ...
    }
    ...
};
```

Every frame, the `cosc::CameraAnimationManager` takes the delta time of the last frame and a spectral energy ratio, to compute the camera animations. The spectral energy ratio is simply the current spectral energy divided by the max spectral energy across the whole song (see Equation 6). This lets us animate effects to the *intensity* of the song, which is covered in the next section.

The CameraAnimationManager stores which of the animations it's currently in, and handles the logic to transition to the next animation. When the end of the animation list is reached, it's simply wrapped around using modulo. The animation progress is tracked by keeping a sum of the delta values provided to the `cosc::CameraAnimationManager::update()` function, forming an `elapsed` value.

Using the aforementioned spectral energy ratio, the rate of change of the song is also increased during "intense" moments, and decreased during "relaxed moments". This is simply achieved by computing a multiplier to the delta time sent to the `cosc::CameraAnimationManager`, which currently ranges between 0.5x and 20x the normal speed. This particular feature was added after further observation of the original Monstercat visualiser, where this effect appears to be subtly applied to the background stars.

Animating the 3D position of the camera between the begin and end pose turns out to be very easy, using simple linear interpolation, as follows:

```
auto pos = glm::mix(anim.begin.pos, anim.end.pos, progress); // performs linear
interpolation
camera.setEyePoint(pos);
```

The `progress` value ranges from 0.0 to 1.0 and is computed by `elapsed / anim.duration`. I also experimented using more complex interpolation (usually called "tweening" in the games industry), such as quadratic interpolation. Surprisingly, linear interpolation continued to look much better and was retained as the final interpolation method.

Animating the orientation of the camera is much more difficult. There are a number of ways of expressing 3D orientations, the two main paradigms being Euler angles and quaternions. Euler angles represent rotations as a 3D vector of angles around axes: pitch, roll and yaw (Figure 10).

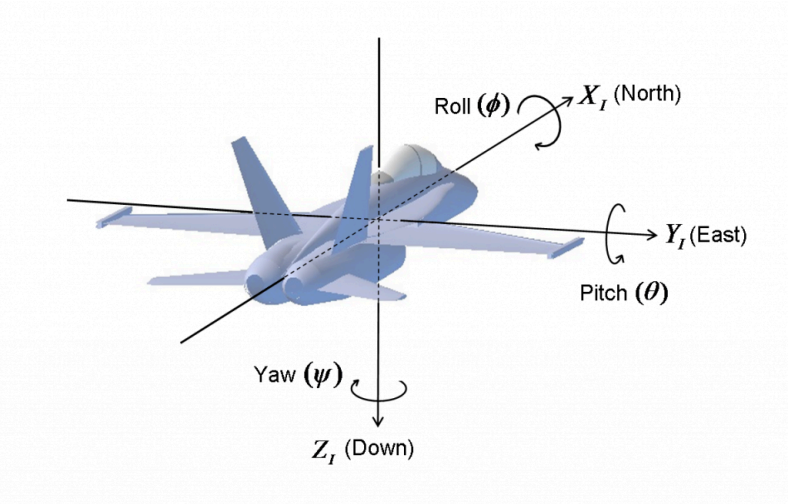

Figure 10: Demonstration of Euler angle rotation system [3]

Unfortunately, Euler angles suffer from well-described problems such as ambiguity and gimbal lock [22], which make their use unsuitable for freeform 3D rotations and animations as is required in this project.

Instead, the system of quaternions are used. A quaternion is a 4D complex number system that can be used to encode 3D rotations, and has a direct correlation to rotation matrices. Quaternion multiplication is the process used to rotate them. A quaternion is represented in the form:

$$q = a + bi + cj + dk \tag{7}$$

Where $a, b, c, d \in \mathbb{R}$ and $1, i, j, k$ are basis vectors. In particular, it is worth noting that quaternions are the most widely used system to reliably encode rotations in video games and other domains such as mobile robotics.

Quaternions can be both converted to and from Euler angles. In freecam mode, to ensure consistency and stability, the camera class stores the pitch and yaw directly, meaning that the pitch and yaw are stored directly as the ground truth, rather than the quaternion. When the mouse is pushed upwards/downwards (Y axis in SDL), the pitch is changed. When the mouse is moved left/right (X axis in SDL), the yaw is changed. Roll is extremely undesirable for this application and is kept to a constant 0. In order to prevent rotation issues at the poles, pitch is clamped between −89 and +89 degrees.[5] If the pitch angle were to reach 90 degrees, it was found to become extremely unstable.

In the main animation mode, one of the most useful aspects of quaternions come in handy: spherical linear interpolation. Remember, our goal is to smoothly animate the transition between the start and end camera pose. It's not possible to directly apply simple linear interpolation to the quaternion $wxyz$ values directly, as was done with the xyz camera position vector. Instead, the system of spherical linear

[5]Note that this is common behaviour in most first-person video games as well.

interpolation ("slerp") can be used to achieve a visually similar result. This system was first described by Ken Shoemake in 1985 [23], and has since been used in almost every 3D video game since that time.

Given two quaternions q_1 and q_2, and an interpolation parameter u, $\{u \in \mathbb{R} \mid 0.0 \leq u \leq 1.0\}$, spherical linear interpolation will find the shortest distance with constant angular velocity between them, as follows:

$$\text{Slerp}(q_1, q_2, u) = q_1(q_1^{-1}q_2)^u \tag{8}$$

In the visualiser, this is simply achieved by using glm's `glm::slerp` function, and since the special quaternion-based camera designed in [21] is used, it all magically "just works". In the end, the entire animation can be roughly condensed into the following (with `progress` being calculated as described above):

```
// lerp position
auto pos = glm::mix(anim.begin.pos, anim.end.pos, progress);
// slerp angle
auto orientation = glm::slerp(anim.begin.orientation, anim.end.orientation, progress);
// update camera
camera.setEyePoint(pos);
camera.setOrientation(orientation);
```

This produces a really nice result, and goes to show the power of simple, effective and widely used interpolation techniques!

4.5. Camera shake

In order to illustrate the intensity of the song at certain points, a camera shake effect was added to the `cosc::CameraAnimationManager`, which is computed alongside the existing camera move system.

Camera shake is essentially applying pseudorandom perturbations to the camera's orientation quaternion (in other words, semi-randomly rotating the camera each tick). It turns out that *actual* randomness does not look very natural, so instead, Simplex noise [24] is applied. Simplex noise is a very popular procedural gradient noise technique designed by Ken Perlin, who also designed the seminal Perlin noise technique. These types of gradient noise techniques are often used in video games for procedural terrain generation, e.g. Minecraft. Compared to Perlin noise, Simplex noise has the advantage of lower computational cost and better visual results.

Raw Simplex noise on its own is not flexible enough to control the camera shake as desired, so I also sum it together using fractal Brownian motion (fBm), using the technique described in [4]. The C++ implementation of both fBm and Simplex noise was provided by [25]. To actually shake the camera, I generate a noise value for each Euler axis, and apply it every frame.

The result of Simplex noise combined with fBm is shown in Figure 11. This is a two dimensional image, where the noise value is seeded based on the x, y pixel position. In our case, the noise value is seeded based on the time since the application has started, and a value of 1.0-3.0 for the roll, pitch and yaw Euler axes respectively.

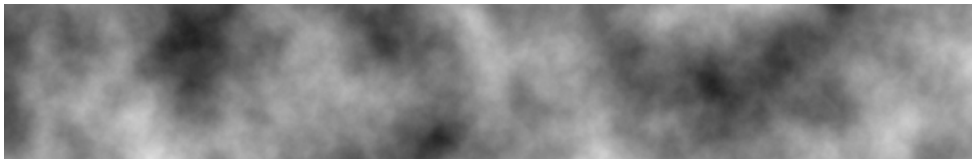

Figure 11: 2D Simplex noise with fractal Brownian motion [4]

The magnitude of the shake is based on the spectral energy ratio, as defined in Equation 6. That means that more "intense" moments in the song have more screen shake, and less intense moments have less shake. This scaling is applied separately to the fBm parameters.

4.6. Skybox cubemap

The visualiser also draws a skybox using OpenGL's cubemap system. A cubemap is a 3D texture that is mapped to a cube volume that surrounds the environment (Figure 12). Assuming the texture has no visible seams on the edges of the cube volume, the cubemap gives the impression of a detailed space environment. This is a very common technique used in almost all video games. The space cubemap texture itself was generated using the tool in [26], which uses procedural techniques, coincidentally also using GLSL shaders.

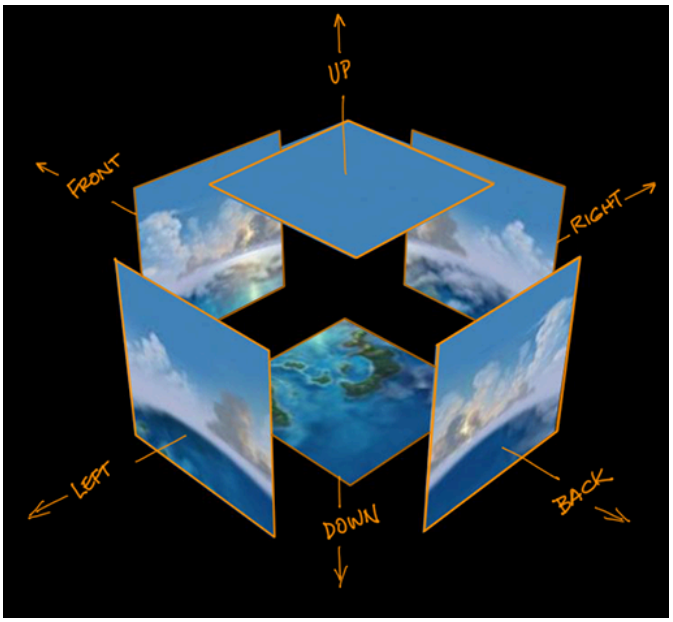

Figure 12: Demonstration of an OpenGL cubemap [5]

Based on a technique described in LearnOpenGL, the cubemap is able to be drawn last in the scene as an optimisation. This is achieved by the following snippet in the vertex shader:

```
gl_Position = pos.xyww;
```

In the perspective division stage, the use of `xyww` ensures that the vertex of the skybox always appears behind any other vertices in the scene. For our case, this means that we can guarantee the skybox is drawn behind the spectrum bars, and that we don't *overdraw* - which saves some calls to the fragment shader where other geometry in the scene obscures the skybox.

The skybox is shown in-app in Figure 13:

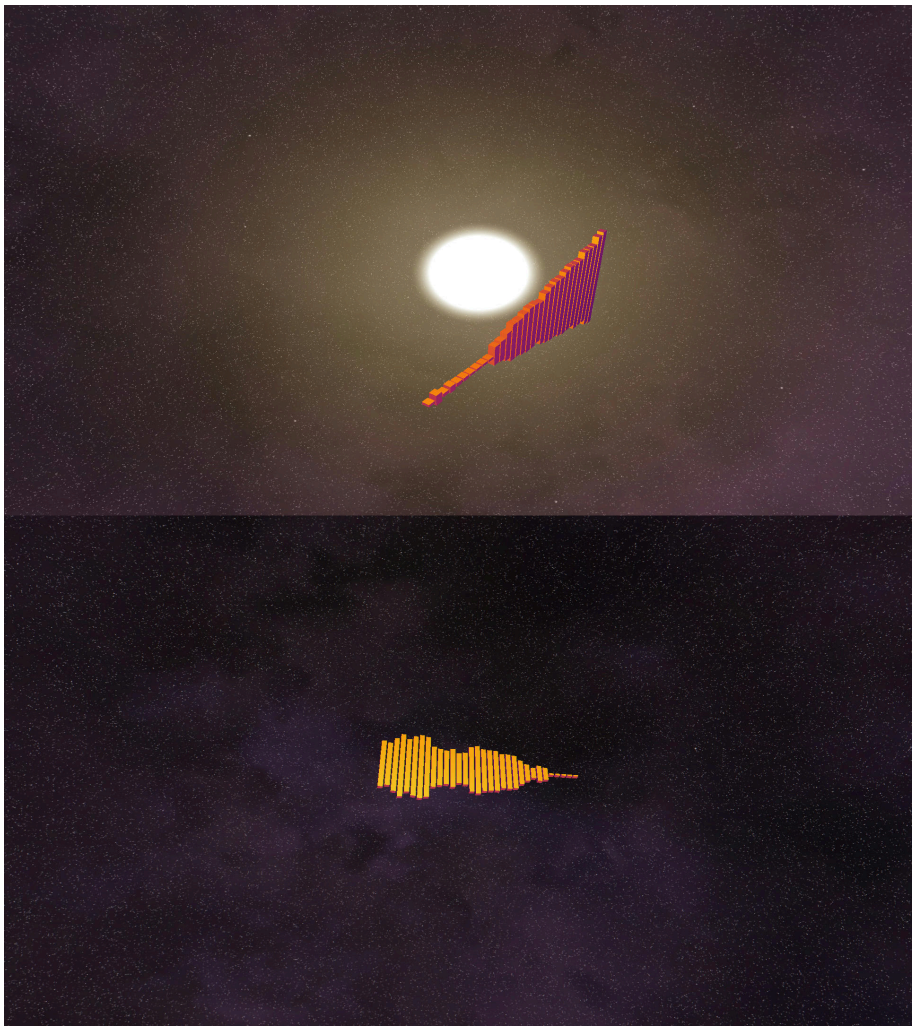

Figure 13: Screenshot of two angles of the skybox in the visualiser

4.7. Post-processing effects

The visualiser implements a simple post-processing pass that is run after the main scene is drawn. Currently, the only effect rendered is chromatic aberration. This is an "artistic touch", if you will, that I added to emphasise the *intensity* of the song at certain points. The intensity is again based on the spectral energy ratio from Equation 6.

Post-processing is achieved using OpenGL's framebuffer and renderbuffer system. The framebuffer is attached to the colour buffer, which is sampled in the fragment shader. The colour and stencil buffers are bound to the renderbuffer, which does not need to be sampled.

The fragment shader, which implements the post-processing effects, is passed the final rendered image along with the spectral energy ratio. To compute chromatic aberration, the fragment sampling position is adjusted on the horizontal axis by an amount relative to the spectral energy ratio, performed separately for the R, G and B channels. This is implemented as follows (based on the shader in [27]):

```
vec4 colour;
float amount = 0.06 * spectralEnergyRatio;
colour.r = texture2D(screenTexture, vec2(TexCoords.x + amount, TexCoords.y)).r;
colour.g = texture2D(screenTexture, TexCoords).g;
colour.b = texture2D(screenTexture, vec2(TexCoords.x - amount, TexCoords.y)).b;
colour.a = 1.0;
```

The result is shown in Figure 14:

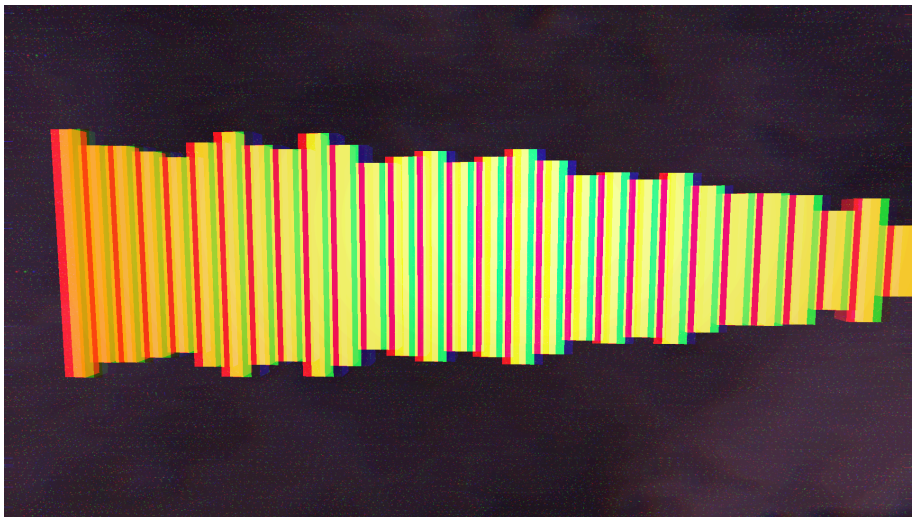

Figure 14: Screenshot of visualisation application showing chromatic aberration effect

Figure 15 also shows a close-up picture of the chromatic aberration effect. Notice the separation of the individual colour channels.

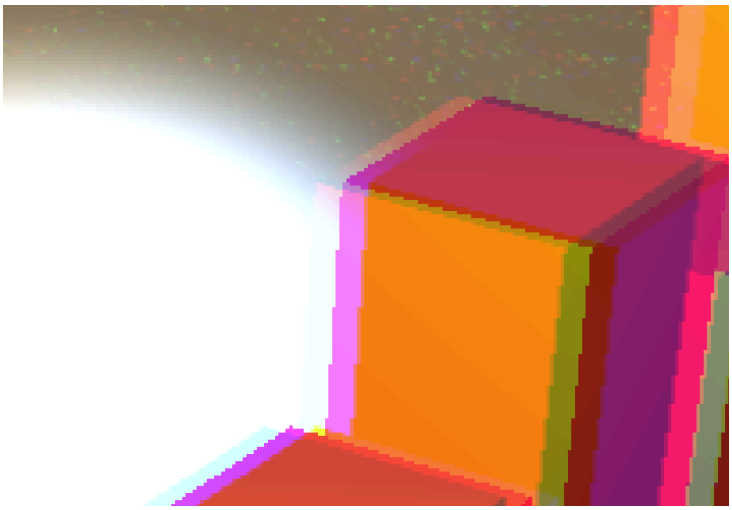

Figure 15: Close-up of chromatic aberration effect

In the above figures, you will notice that the chromatic aberration effect is applied to the whole screen, including the background skybox. This could be avoided if desired by rendering the bars to a separate framebuffer and compositing the result together. However, I was inspired by the album art to the song "Saturn's Air" by Animadrop (which was, at one point, going to be used in the visualiser[6]) to make the chromatic aberration effect fullscreen. The album art is reproduced in Figure 16. It also served as an inspiration for the space skybox.

[6]It was removed due to not looking very interesting on the spectrum.

Figure 16: Album art for *Saturn's Air* by Animadrop, inspiration for fullscreen chromatic aberration

5. Discussion

Overall, the application was completed to a very functional standard and could be considered stable enough for real live presentations.

One problem that I encountered was with anti-aliasing. In the application configuration section, I configure SDL2 to use multi-sample anti-aliasing (MSAA) with 4x samples. Unfortunately, as can be seen in Figure 15, the anti-aliasing didn't end up working correctly. In the time given, I wasn't able to figure out what exactly is causing this discrepancy. I originally thought it was an issue with Wayland, but the issue also occurs under XWayland. Instead, I believe the issue is that, while the *screen target* is configured to use MSAA, the off-screen framebuffer has not been configured to use MSAA.

One other issue I noticed was that, although any songs will *work* on the visualiser, not all songs look *good* on the visualiser. This particularly seems to be related to how the spectral energy ratio is distributed across the length of the song, which in turn depends on how the song is mixed and also how the bassline is written. Songs with rolling, constant bass and songs that are mixed with low dynamic range (i.e. highly compressed) will be regarded by the visualiser as being "intense" for their whole duration, even if they don't *sound* intense to our human ears. This means that the camera will shake, move quickly, and chromatic aberration will be present throughout the entire song. Instead, songs with short, "blippy" basslines seem to work the best.

Figure 17 shows the spectral energy ratio for the song *Pure Sunlight* by Laura Brehm, AGNO3 and MrFijiWiji, which was the main showcase song for the visualiser. The X axis is block index and the Y axis is spectral energy ratio at that instant. This graph shows that *Pure Sunlight* works really well because it has a varied, "blippy" spectral energy ratio, with a slow intro section and more intense second section.

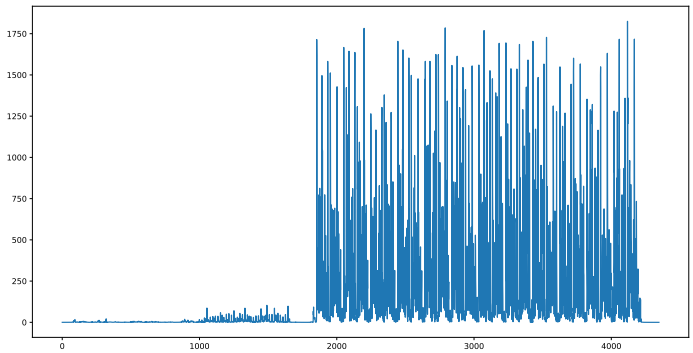

Figure 17: Spectral energy ratio graph of *Pure Sunlight* by Laura Brehm, AGNO3 and MrFijiWiji.

As always with these sorts of projects, there's much that can be improved. Here's a list of good targets for future improvement and research:

- Performing DC offset removal in spectral processing on the Python side
  - DC offsets can significantly bias the spectrogram for certain inputs
- Mel sampling in spectral processing
  - Currently, the sampling is linear which may not be accurate to human hearing
  - Mel sampling (hence the "Mel spectrogram") resembles human musical hearing much more closely
- Refactoring the render pipeline, especially `main.cpp`, to have a `Renderer` interface with `IntroRenderer` and `MainRenderer` sub-classes; plus a proper state machine to switch between them
- Make "state" less ugly - we would like to avoid having as many globals as we currently have
  - Not always easy in graphics
- Add the ability to easily stack multiple post-processing passes. Ideally, the `Framebuffer` class would be written into a `PostFX` class that has a `addPass(const Shader &shader)` method.
- Serialise camera animations to disk using JSON, make a proper editor for them with a timeline, and make it animations depend on the song being played
- More advanced and dynamic lighting for the bars
  - Including reflection mapping: Projecting the cubemap skybox onto the bars
- Use the symmetry of the skybox more effectively, by having a reflective ground plane
  - This would look like those photos taken of lakes at ground level with a "mirror" effect
- Most post-processing effects, on the glitchy side of things
- Bloom post-effect
  - Requires HDR rendering pipeline (separate task)
- Colour correction and tonemapping
  - Also requires HDR rendering pipeline
  - Tonemapping can be done for example using the ACES curve [28]
- Animate camera using splines rather than two linear interpolation points
  - Would allow for smoother trajectories with multiple points in them

Additionally, here are some more advanced ideas for long-term future improvement:

- Dynamically render intro text based on song being played, using FreeType
- Move to a fully online audio architecture all in C++, capture sound from loopback device using PipeWire
  - This would eliminate the need for offline audio processing entirely
- Song lyrics displayed as 3D text that flies around
  - Requires additional serialisation and animations
- More content in the background
  - Hard to describe, but I was considering a particle system based on the human head which disintegrates and re-forms based on the spectral energy.
- Use Vulkan instead of OpenGL

6. Self-assessment

I'm really proud of the work I achieved for this project. The visualiser turned out better than I was hoping for. I was able to achieve everything I set out to, and I was even able to complete a lot of the extension tasks I set myself. This was all made possible because I started extremely early (just after the Graphics Minor Project was finished), and worked on the project all basically all semester. The visualiser is stable, and I was able to try it with a wide variety of inputs - some songs work better than

others - but all songs were fully functional. The code is relatively clean and performant, which I'm happy about.

I was able to draw from a wide variety of advanced topics to construct this visualisation, including from digital signal processing, linear algebra in regards to quaternions and vectors, statistics for fractal Brownian motion, and finally general computer graphics for Simplex noise and chromatic aberration post-processing. By drawing from these topics, I always able to learn a huge amount of cross-discipline knowledge that I couldn't have gotten otherwise.

I don't think this will be the final music visualiser I ever make, but nonetheless, I'm really happy with the result I achieved.

7. Conclusion

In this paper, I present a 3D music visualisation application using OpenGL and the Discrete Fourier Transform. The offline signal processing, using the Fast Fourier Transform as the mechanism for computing the DFT, is implemented in Python. The visualisation application, which loads the computed spectral data, is implemented using C++. It has graphics features such as camera animations, camera shake, a cubemap skybox, and a post-processing path that implements chromatic aberration, and an intro section. The application can be configured to display any song with audio available.

8. Appendix A: Special thanks

- **Angus Scroggie** and **Henry Batt**, for always listening to my nonsense inside and outside the COSC pracs. It was great to have beers (or apple cider, technically) with you guys - thanks for being real ones, and the best of luck in your own projects! They are looking fantastic!
- **Sarah Hurley** for being the greatest cinema genius of the 21st century. Thank you for helping me film :)
- **Joey de Vries** of LearnOpenGL, for providing one of the best graphics programming resources.
- **Laura Brehm**, **Mr FijiWiji**, **AGNO3**, **Eastern Odyssey** and **Animadrop**, for producing great music.
- **The authors and contributors** of: SDL2, glad, glm, Cap'n Proto, dr_flac, stb_image, NumPy, SciPy, spectrum.py, Simplex.h - such a project could not even be remotely achieved without the generous efforts of these talented free software programmers.
- **Paul Houx** of The Cinder Project, for the quaternion camera. The camera animations would not have been at all possible without this extremely polished perspective camera implementation.

This document was typeset using Typst.

References

[1] Monstercat Music, "[Electronic] - Mr FijiWiji, Laura Brehm & AgNO3 - Pure Sunlight [Monstercat Release]," 2014. https://www.youtube.com/watch?v=F_QrKYAv1NE (accessed Mar. 30, 2024).

[2] Allen D. Elster, "Fourier transform." https://mriquestions.com/fourier-transform-ft.html (accessed May 23, 2024).

[3] CHRobotics LLC, "Understanding Euler Angles." https://web.archive.org/web/20210415160916/http://www.chrobotics.com/library/understanding-euler-angles (accessed May 23, 2024).

[4] Christian Maher, "Working with Simplex Noise," 2012. https://cmaher.github.io/posts/working-with-simplex-noise/ (accessed Apr. 28, 2024).

[5] "Scali", "Cubemaps," 2013. https://scalibq.wordpress.com/2013/06/23/cubemaps/ (accessed May 23, 2024).

[6] Léon McCarthy, "Live Visuals: Technology and Aesthetics," *Live Visuals*. Routledge, pp. 194–207, 2022.

[7] Ryan Geiss, "MilkDrop plug-in for Winamp," 2012. https://www.geisswerks.com/milkdrop/ (accessed Apr. 19, 2024).

[8] Josh Coalson and Xiph.Org Foundation, "FLAC - What is FLAC?," 2022. https://xiph.org/flac/ (accessed Apr. 10, 2024).

[9] Kenton Varda, "Cap'n Proto: Introduction — capnproto.org." https://capnproto.org/ (accessed Apr. 10, 2024).

[10] Eric Weisstein, "Fourier Series – from Wolfram MathWorld," 2024. https://mathworld.wolfram.com/FourierSeries.html (accessed Mar. 30, 2024).

[11] Eric Weisstein, "Fourier Transform – from Wolfram MathWorld," 2024. https://mathworld.wolfram.com/FourierTransform.html (accessed Mar. 30, 2024).

[12] Elias Rajaby and Sayed Masoud Sayedi, "A structured review of sparse fast Fourier transform algorithms," *Digital Signal Processing*, vol. 123, p. 103403–103404, 2022, doi: https://doi.org/10.1016/j.dsp.2022.103403.

[13] Gilbert Strang, "Wavelets," *American Scientist*, vol. 82, no. 3, pp. 250–255, 1994, Accessed: Mar. 31, 2024. [Online]. Available: http://www.jstor.org/stable/29775194

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

[15] Michael T. Heideman, Don H. Johnson, and C. Sidney Burrus, "Gauss and the history of the fast Fourier transform," *Archive for History of Exact Sciences*, vol. 34, no. 3, pp. 265–277, 1985, doi: 10.1007/bf00348431.

[16] H. Sorensen, D. Jones, M. Heideman, and C. Burrus, "Real-valued fast Fourier transform algorithms," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 35, no. 6, pp. 849–863, 1987, doi: 10.1109/TASSP.1987.1165220.

[17] Richard N. Youngworth, Benjamin B. Gallagher, and Brian L. Stamper, "An overview of power spectral density (PSD) calculations," in *Optical Manufacturing and Testing VI*, 2005, vol. 5869, p. 58690–58691. doi: 10.1117/12.618478.

[18] Bui Tuong Phong, “Illumination for computer generated pictures,” *Seminal graphics: pioneering efforts that shaped the field*, 1975, [Online]. Available: https://api.semanticscholar.org/CorpusID:1439868

[19] Joey de Vries, “LearnOpenGL - Basic Lighting,” 2020. https://learnopengl.com/Lighting/Basic-Lighting (accessed Mar. 15, 2024).

[20] Sean Kelley, “WebGL Colour Maps,” 2020. https://observablehq.com/@flimsyhat/webgl-color-maps (accessed Mar. 15, 2024).

[21] Paul Houx and Cinder Project, “cinder/src/cinder/camera.cpp,” 2012. https://github.com/cinder/Cinder/blob/master/src/cinder/Camera.cpp (accessed Apr. 24, 2024).

[22] Evan G. Hemingway and Oliver M. O’Reilly, “Perspectives on Euler angle singularities, gimbal lock, and the orthogonality of applied forces and applied moments,” *Multibody System Dynamics*, vol. 44, no. 1, pp. 31–56, Mar. 2018, doi: 10.1007/s11044-018-9620-0.

[23] Ken Shoemake, “Animating rotation with quaternion curves,” in *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, 1985, pp. 245–254. doi: 10.1145/325334.325242.

[24] Ken Perlin, “Chapter 2: Noise Hardware,” 2001. https://redirect.cs.umbc.edu/~olano/s2002c36/ch02.pdf (accessed Apr. 28, 2024).

[25] Simon Geilfus, “Collection of Simplex noise functions,” 2019. https://github.com/simongeilfus/SimplexNoise (accessed Apr. 28, 2024).

[26] Rye Terrell, “Space 3D — tools.wwwtyro.net,” 2021. https://tools.wwwtyro.net/space-3d/index.html (accessed Apr. 25, 2024).

[27] "bkhan", “Chromatic Abberation (With Offset) - Godot Shaders,” 2021. https://godotshaders.com/shader/chromatic-abberation-with-offset/ (accessed Apr. 25, 2024).

[28] Krzysztof Narkowicz, “ACES Filmic Tone Mapping Curve,” 2016. https://knarkowicz.wordpress.com/2016/01/06/aces-filmic-tone-mapping-curve/ (accessed May 12, 2024).