Ilari Suhonen
Year Group 2025
Student ID 102869275
2026-04-25

CS-C2120
B.Sc. and M.Sc. (Comp. Sc.)
School of Science
Aalto University

FINAL REPORT

on Ang, a real-time audio effect program in Scala[1]

1 OVERVIEW OF THE PROJECT

The purpose of the project was to implement a program called Ang, initially intended to be a real-time audio manipulation program. No significant changes in the intent or use cases of the program occurred during the course of the development.

The first goal of Ang was to advance my understanding of the project-oriented software development process, architecture planning and UML. This goal was moderately successful; Ang was my largest complete project to date, and the first time I used UML to design and plan out a personal project. It was also the first time I set clear goals for a project beforehand.

The second goal was to gain an understanding of the signal processing algorithms required to implement audio effects. This goal was achieved beyond expectations, and I developed a much deeper understanding of concepts like the fast Fourier transform and its inverse (FFT and IFFT respectively), phasors, filter banks, the reasons for using window functions and the concrete problems that the mathematical tools solve. Particularly fruitful was the implementation of a complex windowed overlap-add algorithm, which was necessary to reduce artefacts in the output audio.

The third and final goal was to create a useful tool for visualising and manipulating audio. It can be argued that Ang in its current state is not exactly the most useful of programs for applying real-time effects. However, it should be noted that the framework that Ang builds for implementing effects makes it trivial to add new ones and get the program quickly to a more useful state; this is an exercise for further development.

The program's complexity and difficulty of implementation matched my expectations well; it was indeed of intermediate complexity as set out in the general plan[2], and ended up with slightly over two thousand lines of code.

Overall, the project was a success and I am happy with the resulting program. All features specified as mandatory in the general plan have been implemented, and the user interface matches the draft user interface closely.

2 USAGE OF THE PROGRAM

The program is launched on any of three supported operating systems (Linux, Windows, macOS) either by double-clicking the assembled Java Archive (.JAR) (with limited functionality), or from the source files with the command `./sbtx run` (full functionality). The program will first download the correct JavaFX binaries for the user's operating system, caching them in a system-appropriate cache directory. After this, it will load the relevant classes and launch the program. When the program is first opened, the user is greeted with the program's main view, consisting of the Ang logo, a sidebar with three sub-panes for input, effects, and output respectively, as well as an empty pane with the text "No effect selected."

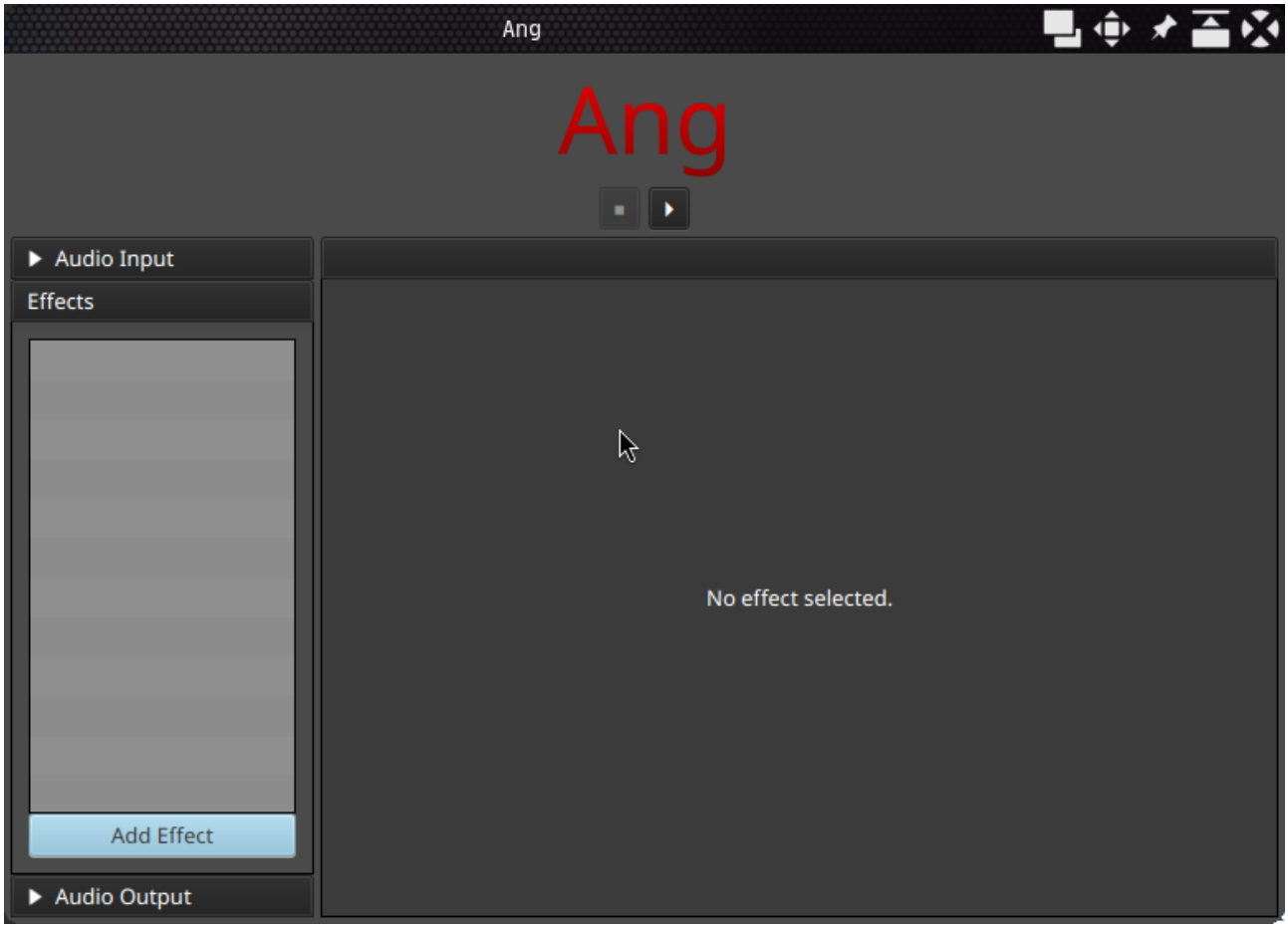

Figure 1: The main view of the Ang program when no effects have been added.

The default audio input and output are usually perfectly fine choices and do not need to be changed. The user may add effects by clicking the "Add Effect" button, at which point a pop-up dialogue will be shown with a drop-down menu of available effects.

When "OK" is clicked, the effect will be added to the effect chain and selected automatically, at which point the effect's configuration will be available in the main effect pane.

Each type of effect has different configuration options, and each effect may be added multiple times, in which case these instances all have separate configurations.

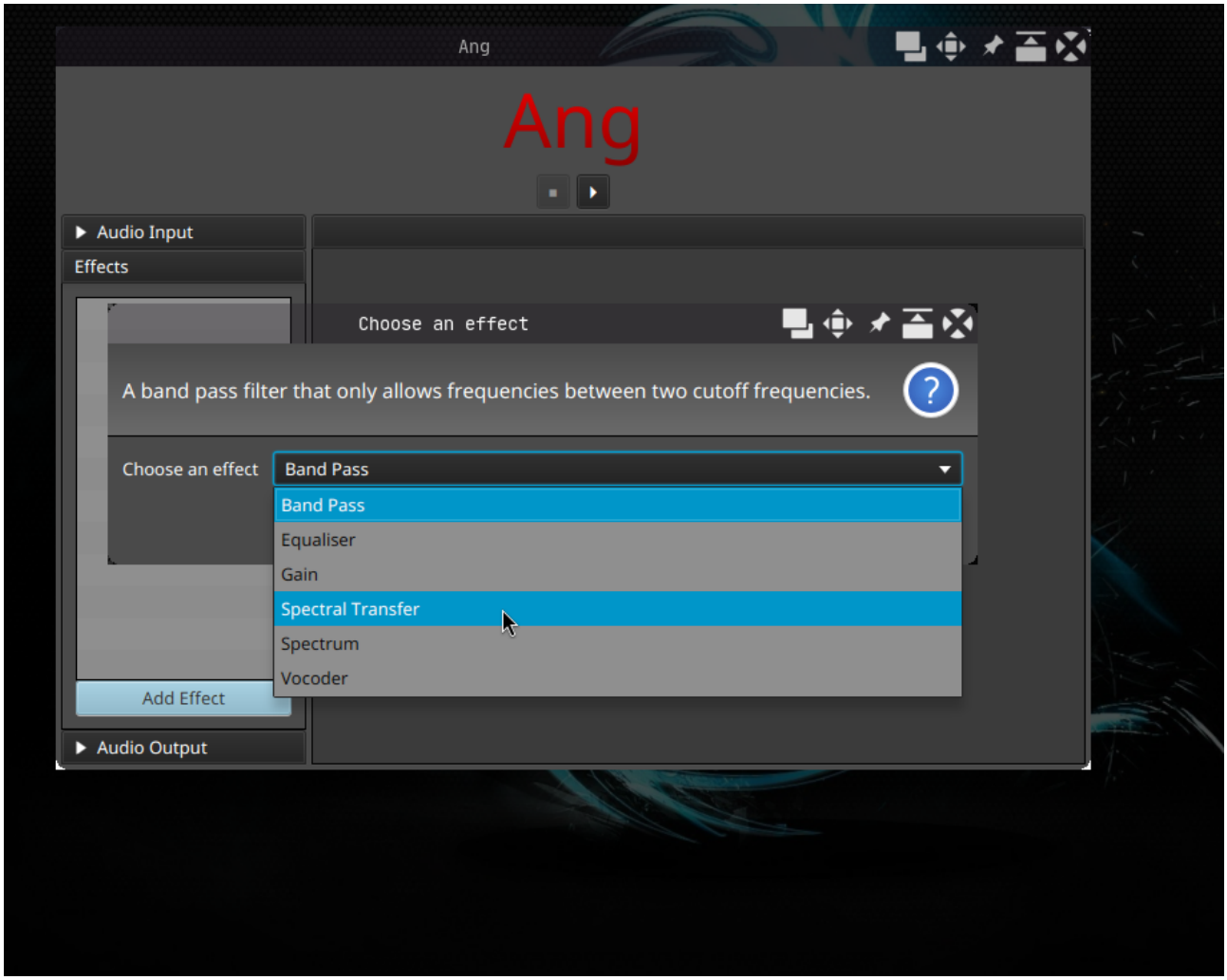

Figure 2: Selecting an effect to add in Ang is easy.

After all desired effects have been selected, the Play (►) button may be pressed. When pressed, the button instructs Ang to initialise playback, which allocates hardware resources for the audio input and output and instantiates the chain of effect instances into a chain of handles. At the end of the chain is the output handle, which constructs—for its lifetime—an operating system thread that polls new audio frames from the effect chain and sends them to the hardware output device (e.g. the speakers).

Ang is completely hotswappable; effects may be re-configured, added, or even removed from between two other effects. An effect may be removed by right-clicking on it in the effect list and clicking on the “Delete” button in the context menu.

Modifications that alter links in the effect chain (removal and insertion) trigger a re-chaining; each effect’s input is updated to be the previous effect such that the effect chain matches the effect list. If a link-altering modification is done while playback is active, playback is stopped and restarted in order to facilitate the change.

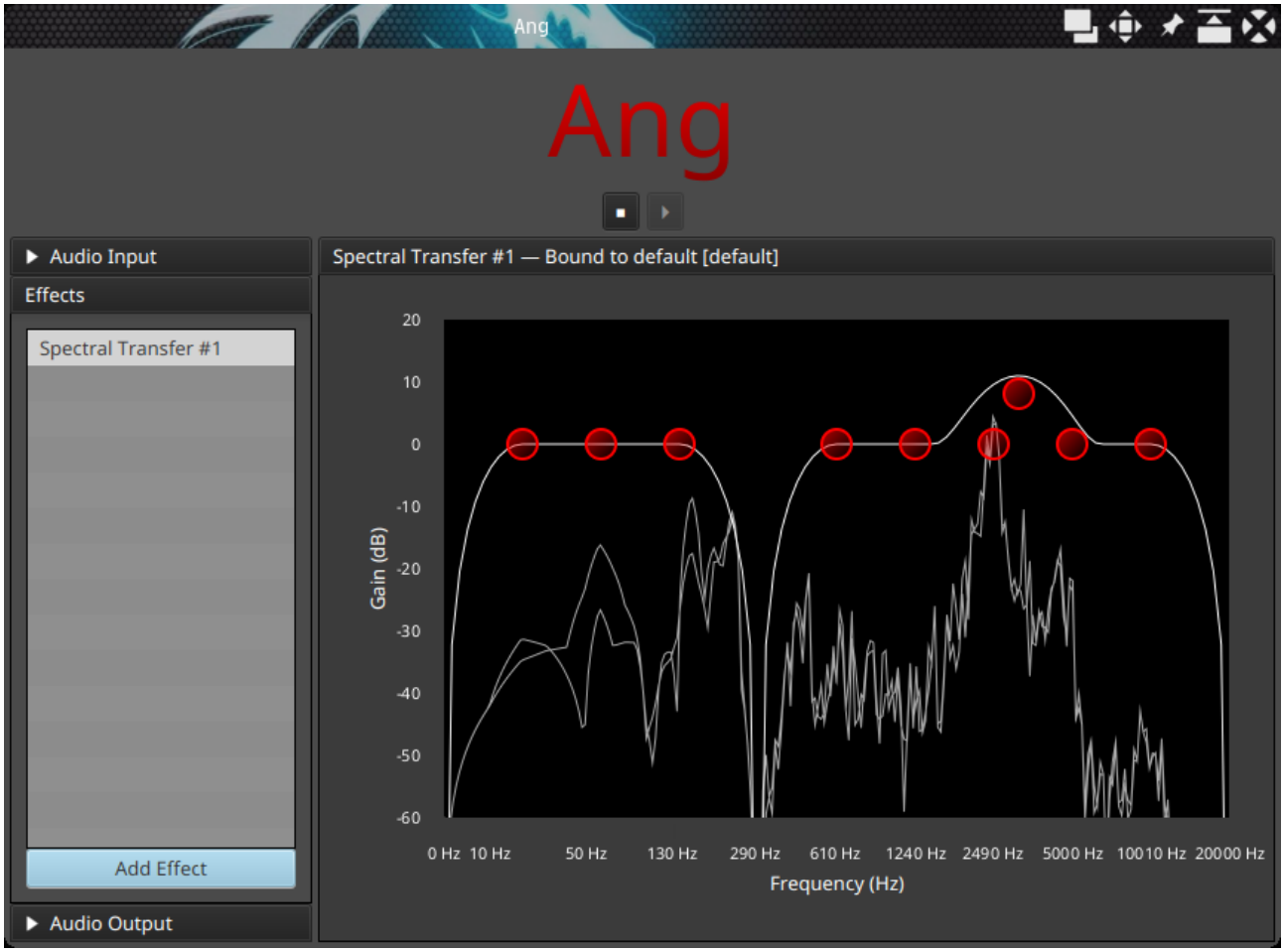

Figure 3: Ang can apply effects to popular songs such as Streets of Laredo by Marty Robbins.

3 STRUCTURE OF THE PROGRAM

The program was, in general, structured according to the specifications set out in the technical plan from February[3]. The architecture is split into three key components;

- the model, which implements the logic of the audio manipulation effects
- the system interface, which communicates with the underlying computer hardware to integrate with available microphones and speakers, and
- the user interface, which defines the ScalaFX components used to show the state of the program to the user and allow the user to add, remove, and configure the effects, pause and resume playback, and switch between hardware devices.

There are three especially notable changes to the initial plan. Firstly, in early development of the model it was identified that the approach of using variable-size windows caused problems with repeated audio data and missed samples. The audio chain was adapted to instead have each input provide an arbitrary number of samples at its own discretion, while the output handles providing samples to the hardware device at the correct rate, reporting any underflows.

The second notable change is the shift from modelling the entire program as part of the model component to having the model represent individual high-level structures, which are then orchestrated by the user interface component. This allows for the user interface component to be more tightly coupled with the effect chain and reduces the complexity of ensuring the user interface matches the actual state of the program.

The third, final and largest change is to the lifecycle of the effects and user interface elements; in the technical plan it was outlined that the effects are singleton objects which then construct effect instances, which are tied to specific hardware resources. However, the need appeared to represent the configuration for an effect instance while hardware resources are not allocated. Thus the lifecycle of the effect instances was split; an effect creates an instance, and the instance creates a new type of object, a *handle*.

A handle will always be linked to the same underlying hardware source for the duration of its lifetime, meaning a constant sample rate and channel count. These invariants are crucial for the operation of effects, as many involve filter banks and channel-specific operations that would be difficult and resource-intensive to implement in such a way that they respond well to dynamic changes in the order, number, or sample rate of audio channels.

For user interface resources, a handle has a slightly different relationship to the effect as effect instance handles do, but the same idea is conveyed. A user interface handle is present when the user interface element is visible, i.e. when the effect instance is selected in the sidebar. The purpose of this separation is to allow for user interface elements to have two kinds of lifetimes for state; one for the entire existence of the effect, and another for only when the effect is actually visible.

To give an example, the handle of the Spectrum effect’s user interface is what requests the Fourier transform be computed in the first place. If the spectrum is not visible, then no FFT is computed, which improves performance. We contrast this with the control points of the Equaliser effect (graphic EQ). These points should retain their position even when the effect is not selected, but they cannot be determined from the effect instance configuration. The configuration for the Equaliser effect accepts only a function that converts frequencies to decibel gain, and thus retains no information about where specifically control points were placed.

A high-level overview of the structure of Ang is given below (Fig. 4).

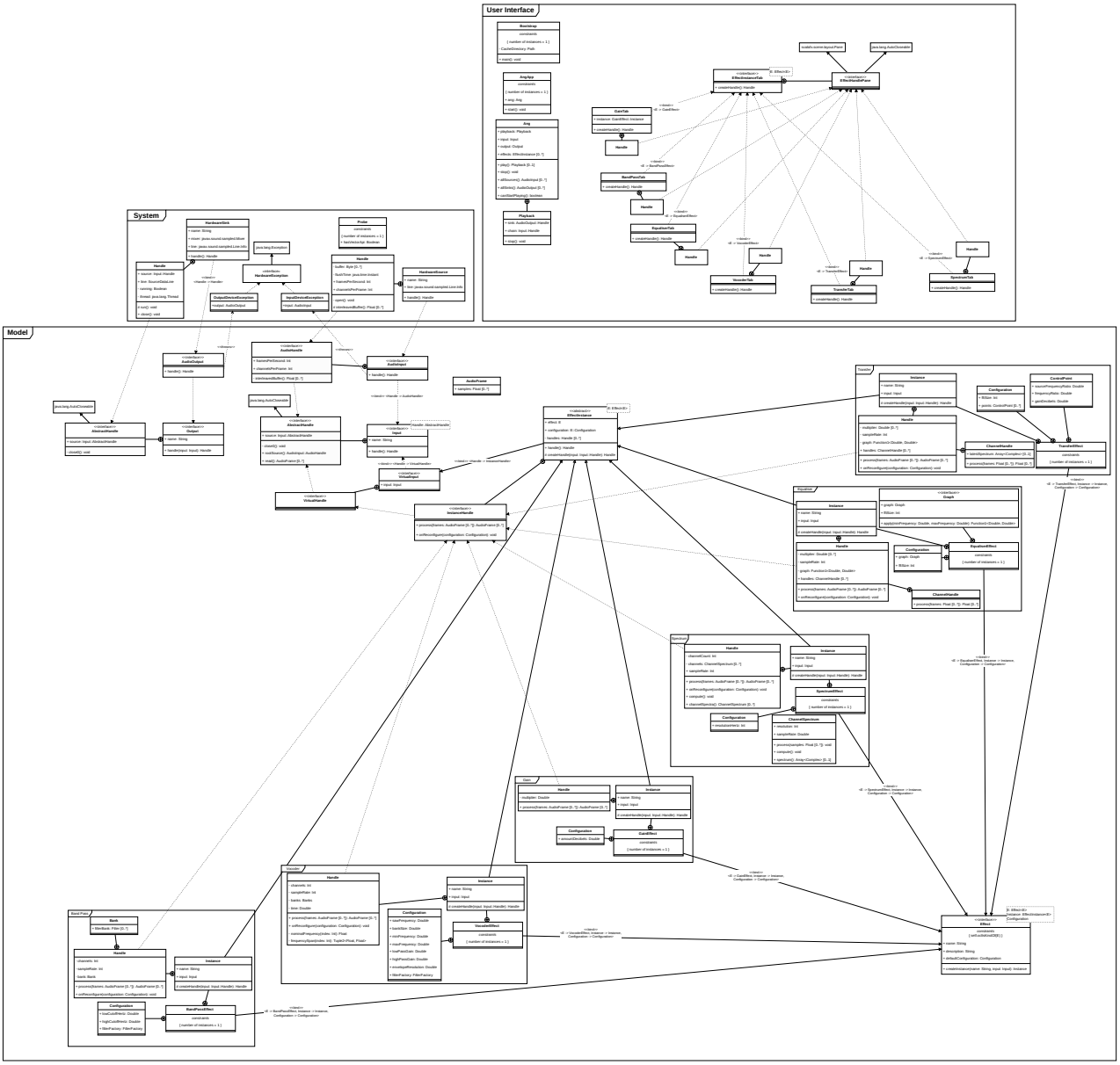

Figure 4: UML 2.4 diagram showing the basic structure of Ang (best viewed electronically by zooming in).

4 ALGORITHMS

The program relies heavily on mathematical algorithms used for processing the input audio signal and applying effects. These algorithms, such as the Fourier transform, frequency transfer, gain, inverse Fourier transform, et cetera are crucial for Ang's functionality. We describe a pipeline that processes audio samples, and the algorithms it uses to do so.

The signal processing pipeline is driven by the output; it continually reads data from its input (being, most likely, an effect instance handle). This input then reads data from its own input, which reads from its input, and so forth until a hardware audio source is reached. Since the only contract satisfied by an input is that it returns *some* number

of audio frames, the buffer size is decided entirely by the audio source. Notably, this is always equal to the size of the output's hardware buffer, as both sizes are dictated by a shared constant.

Each effect in the pipeline applies some processing to its received audio frames before returning them to its caller, and finally the audio output sends the audio frames to the hardware source.

THE FAST FOURIER TRANSFORM

Perhaps the most foundational algorithm used in Ang's processing pipeline is the Fourier transform. Initially described in the technical plan was a radix-2 decimation-in-time fast Fourier transform (DIT FFT). However, this algorithm was not used, and was instead replaced by the (perhaps simpler) decimation-in-frequency transform (DIF FFT) of the same radix. The decimation-in-frequency FFT of a complex, time-domain signal is computed recursively by applying the following steps:

1. Each complex value is paired with the value $N/2$ samples in the future (where N is the length of the signal).
2. The left value is replaced with the sum of the two values.
3. The right value is replaced with the difference of the two values, multiplied by the *twiddle factor* for the left value. The twiddle factor is computed as $e^{j2\pi\frac{k}{N}}$, where e is Euler's number, $j=\sqrt{-1}$, and k is the time-index of the left value in the signal (such that $0 \leq k < N$).
4. The operation is applied recursively for the latter half of the resulting time-domain signal
5. The operation is applied recursively for the former half of the resulting time-domain signal

After these steps are completed, the signal is transformed to a frequency-domain signal in *bit-reversed order*, such that the frequency bin that should be at index 0b1100 is actually at index 0b0011 (for a signal of length 8). A series of swaps is performed to move the bins back into their rightful place, and the final frequency-domain signal is obtained. This algorithm differs from DIT in two ways: first, the samples are combined differently, and second, the *output* is bit-reversed, rather than the input.

SPECTRAL TRANSFER

One of the most interesting and novel applications of the Fourier transform in Ang is the *Spectral Transfer* effect. As well as the standard graphical equalisation features (namely, increasing or decreasing the volume of frequency bands), it allows the user to shift the frequencies of *specific bands* up or down as desired.

Shown in Figure 3 is an application of Spectral Transfer to the song *Streets of Laredo* by Marty Robbins[4]. The frequency band spanning roughly from 130 to 610 Hertz is shifted up to the 2500 Hz range, such that any sound within the band will be played around 2500 Hz instead. The volume of the band is also increased, as might be done in a regular graphic equaliser.

Internally, the frequency shift is implemented using a bank of *phasor arrays*. Consider a 130,2 Hz signal in *Streets of Laredo*. When we perform a frequency transform, we find that the 130 Hz bin has a complex value of some magnitude. A fraction of a second later, we observe that the value has *rotated by some amount*. This amount is a function of the *precise* frequency of the signal, as the 130 Hz and 130,2 Hz signals share a bin in the spectrum; it is only by the rotational speed of that bin that we identify the true frequency.

We first calculate the expected rotational speed for that bin, given an exact 130 Hz signal. By

comparing this with the observed rotational speed, we obtain a *deviation.* If we take the deviation for the 130 Hz input bin and apply it to the 2500 Hz output bin, we include in our output signal a true frequency of slightly over 2500 Hz—corresponding to the true input frequency of 130,2 Hz.

In this way, we can shift the frequencies from each of the control points. We use the window of the control point to determine the influence of each point on the *amplitude* of each output phasor; the control point that was originally between 130 Hz and 610 Hz has no effect on the output phasors at, say, 18 000 Hz, because the window has zero values there.

BUFFERING STRATEGIES

Some signal processing algorithms, such as the FFT, require the input size to satisfy specific properties in order for the algorithm to work. In particular, the DIF FFT used in Ang requires that the input size is a power of two. However, setting the hardware buffer size to a power of two, particularly a large one, has consequences of its own; namely delay.

There are two solutions to this problem in Ang, applied separately based on the specific scenario.

The first solution is the *fixed window*, which receives audio samples up to a specific window size before expelling old values. It is a fixed-capacity first-in-first-out (FIFO) queue. Importantly, the fixed window is defined as having any data at all iff its buffer is at capacity, meaning that it will either provide no samples at all or exactly the number of samples defined in its window size.

The fixed window is an excellent solution in cases where the processed data is not used afterwards, for example the spectrum visualiser effect (Spectrum). However, it makes no guarantee that any audio sample appears exactly once in its output, so it is not suitable for streaming applications.

The second solution, more complex, is the overlap-add algorithm. This algorithm is defined by two numbers, the *chunk size* and *hop size*. It consists of a pending input queue, input chunk, output chunk, and pending output queue.

The overlap-add algorithm implements a sliding window over the input, with a window size of *chunk size* and a step of *hop size*. Each window is processed separately, and overlapping windows are combined by adding their results. The pending input and output queues are used to ensure that exactly the number of input samples is output, and the input and output chunks contain the input data and processed output respectively. Crucially, we will later see the following: by choosing the processing algorithm and hop size carefully, we can ensure that the sum of the overlap of two windows matches exactly to the result we would obtain by applying the processing algorithm only once to the overlap region, and in this way no artefacts are left behind from the overlap-add algorithm. This property is called constant overlap-add (COLA).

The guarantees of the overlap-add algorithm are thus more powerful than those of the fixed window; not only do we only have to be able to process inputs of a desired size, but the output is streamed correctly as if we were able to process an input of any length.

HANN WINDOWS

One problem introduced by performing the Fourier transform to a signal in real-time is that the result suffers from artefacts caused by the nature of the transform. The Fourier transform makes the assumption that the signal is periodic, i.e., if we were to follow the signal past the slice that is given to the Fourier transform, it would repeat indefinitely.

For most real-world signals this is not the case, and the jarring change from the end of the signal back to the start of the signal causes broad artefacts in the frequency spectrum. Since we cannot process entire songs at once (the user would have to wait for the

song to end before listening to it, which is no fun), we cannot solve this easily.

This issue, however, can be mitigated by applying a *window function* to the input signal. By smoothly reducing the signal's amplitude to zero near the edges, it *does* become periodic in some sense of the word. The artefacts are reduced, and since altering a signal's amplitude does not change its frequency, the spectrum remains otherwise very similar.

It happens that a specific window function, the *Hanning window*, works especially well. The Hanning window is obtained from the following formula:

$$w[n]=\frac{1}{2}-\frac{1}{2}\cos\frac{2\pi n}{N}$$

where n is the sample index and N is the total length of the signal. In Ang, the equivalent form $w(x)=\sin^2(\pi x)$ where $0<x<1$ is used.

The usefulness of the Hanning window comes from the fact that it also satisfies the COLA property. By applying the Hanning window to our input before computing the Fourier transform, we reduce edge artefacts of the transform. After processing the frequency spectrum and computing the inverse Fourier transform, we get the Hanning windowed output, which satisfies COLA. Thus, our algorithm is a valid processor for overlap-add and we get two benefits at the cost of one window. Neat.

AUDIO FILTERS

For many audio processing effects it is necessary to keep only specific frequencies while reducing others. However, the processing cost of a full Fourier transform is not always necessary. Ang contains two alternative techniques for filtering audio frequencies.

DIGITAL BIQUAD FILTER

The first, perhaps more complex of the two, is a *digital Biquad filter*. The origin of its name is rather complex (in many senses of the word), so we shall not dwell on it. However, the filter can be implemented simply by keeping the two previous input samples and two previous output samples in memory.

A digital Biquad filter is defined by six parameters $(a_0, a_1, a_2, b_0, b_1, b_2)$. However, since a_0 is simply a scaling factor, we represent this in the normalised form where the first parameter is implicitly defined as 1 and all others are scaled down by a_0 to match.

Then for any sample input sample, the corresponding output sample may be computed using the following difference equation:

$$\begin{aligned} y[n] &= b_0 x[n] + b_1 x[n-1] \\ &+ b_2 x[n-2] - a_1 x[n] - a_2 x[n-1] \end{aligned}$$

where $y[n]$ and $x[n]$ represent the output and input signal values at time n respectively. Since $x[n]$ is known trivially (it is the sample being processed), only four samples must be kept in memory and the Biquad filter is very computationally inexpensive.

CONVOLUTION FILTER

In contrast to the digital Biquad filter, which kept four samples of memory, a convolution filter may keep an arbitrary number of samples. A convolution filter is defined by a set number of weights, called *taps* (similar to the windows discussed earlier). Each sample's filtered value is the weighted sum of the previous values, where the weight of each value is the corresponding tap value. Mathematically, this is given by

$$y[n]=x[k]*t[k]=\sum_{k=0}^{K-1} t[k]x[k-1]$$

where $x[k]$, $t[k]$ and $y[k]$ represent the input signal, convolution taps, and output signal

respectively, and K is the filter's *order,* i.e. the number of taps in t.

Convolution filters get their name from the fact that they represent the mathematical operation of *convolution.* The taps of a convolution, conventionally, are called the *convolution kernel*, and by designing clever kernels we can achieve a variety of effects.

For example, when sampling at a rate of F_s, we may implement a low-pass filter for frequencies under F_c by choosing as our convolution kernel the sinc function $\frac{\sin 2\pi\omega\Delta x}{\pi\Delta x}$. Here, $\omega = \frac{F_c}{F_s}$ and Δx is the distance from the tap index to the centre of the kernel. Longer kernels improve filter accuracy at the cost of performance.

An implementation note regarding Ang is that the convolution filter process is extremely optimised; convolution is implemented using the Java Development Kit's (JDK) experimental Vector API, allowing the code to compile to vectorised single-instruction-multiple-data (SIMD) instructions on processors that support them. The reasoning behind this optimisation is largely historical; effects that predate Ang's inverse FFT implementation—such as the vocoder—make heavy use of convolution filters.

HUMAN SOUND PERCEPTION

A very wide range of sound frequencies and amplitudes occur in nature: A gunshot has an amplitude over a thousand times that of, say, a snake's hiss. It is useful to be able to hear both, and so humans perceive volume on a *logarithmic* scale. The same applies also to pitch; we perceive the notes on a piano to increase roughly linearly, while in reality each octave doubles the previous octave's frequency.

Since Ang is primarily intended for use by humans, the program visualises pitch and volume using these scales. In addition, we involve some more nuanced adjustments to account for the fact that humans hear some frequencies better than others, and that human pitch perception is not *precisely* logarithmic, but only approximately.

The naïve logarithmic volume is trivially computed; we take $10 \cdot \log_{10}(x^2)$ for any sample in the range $[-1, 1]$, or rather the equivalent form $20 \cdot \log_{10}|x|$. The unit we get is decibels (really, deci-Bels for one tenth of a Bel) relative to the *full scale*, i.e. decibels such that 0 dB corresponds to the maximum possible sample (-1 or 1). This unit is notated dBFS.

To compute the influence of frequency on the perceived volume, we use two filters helpfully recommended to us by the Broadcasting Service of the Radiocommunication Sector of the International Telecommunications Union (ITU-R BS) in recommendation ITU-R BS.1770.

These filters are specified in the recommendation using parameters for two Biquad filters, specifically by their normalised filter coefficients. In Ang, the combined frequency response of these filters is derived from the given coefficients and the frequency response is computed. This is then used directly to influence the gain (in decibels) on any specific frequency.

We notate the coefficients such that the first sub-index is the filter number (zero-indexed), the second sub-index is the coefficient number (zero-indexed) and the variable is the coefficient type (*a* or *b*). Thus, we obtain two biquads:

$$H_0(z) = \frac{b_{00} + b_{01} z^{-1} + b_{02} z^{-2}}{1 + a_{01} z^{-1} + a_{02} z^{-2}}$$
$$H_1(z) = \frac{b_{10} + b_{11} z^{-1} + b_{12} z^{-2}}{1 + a_{11} z^{-1} + a_{12} z^{-2}}$$

Evaluating $|H_0(z) H_1(z)|^2$ on the unit circle $(z = e^{j\omega})$, we (eventually) obtain[5] the form

$$\frac{\boldsymbol{B}_{00}\boldsymbol{B}_{10} + \boldsymbol{B}_{01}\boldsymbol{B}_{11}\phi + \boldsymbol{B}_{02}\boldsymbol{B}_{12}\phi^2}{\boldsymbol{A}_{00}\boldsymbol{A}_{10} + \boldsymbol{A}_{01}\boldsymbol{A}_{11}\phi + \boldsymbol{A}_{02}\boldsymbol{A}_{12}\phi^2}$$

where

$$\begin{aligned}
\boldsymbol{B}_{k0} &= \left(\frac{b_{k0} + b_{k1} + b_{k2}}{2}\right)^2 \\
\boldsymbol{B}_{k1} &= -\left(4 b_{k0} b_{k2} + b_{k1}(b_{k0} + b_{k2})\right) \\
\boldsymbol{B}_{k2} &= 4 b_{k0} b_{k2}
\end{aligned}$$

$$\begin{aligned}
\boldsymbol{A}_{k0} &= \left(\frac{a_{k0} + a_{k1} + a_{k2}}{2}\right)^2 \\
\boldsymbol{A}_{k1} &= -\left(4 a_{k0} a_{k2} + a_{k1}(a_{k0} + a_{k2})\right) \\
\boldsymbol{A}_{k2} &= 4 a_{k0} a_{k2}
\end{aligned}$$

$$\phi = \sin^2 \frac{\omega}{2}$$

Converting this to decibels gives us the final gain for a given angular $\omega = 2\pi f$.

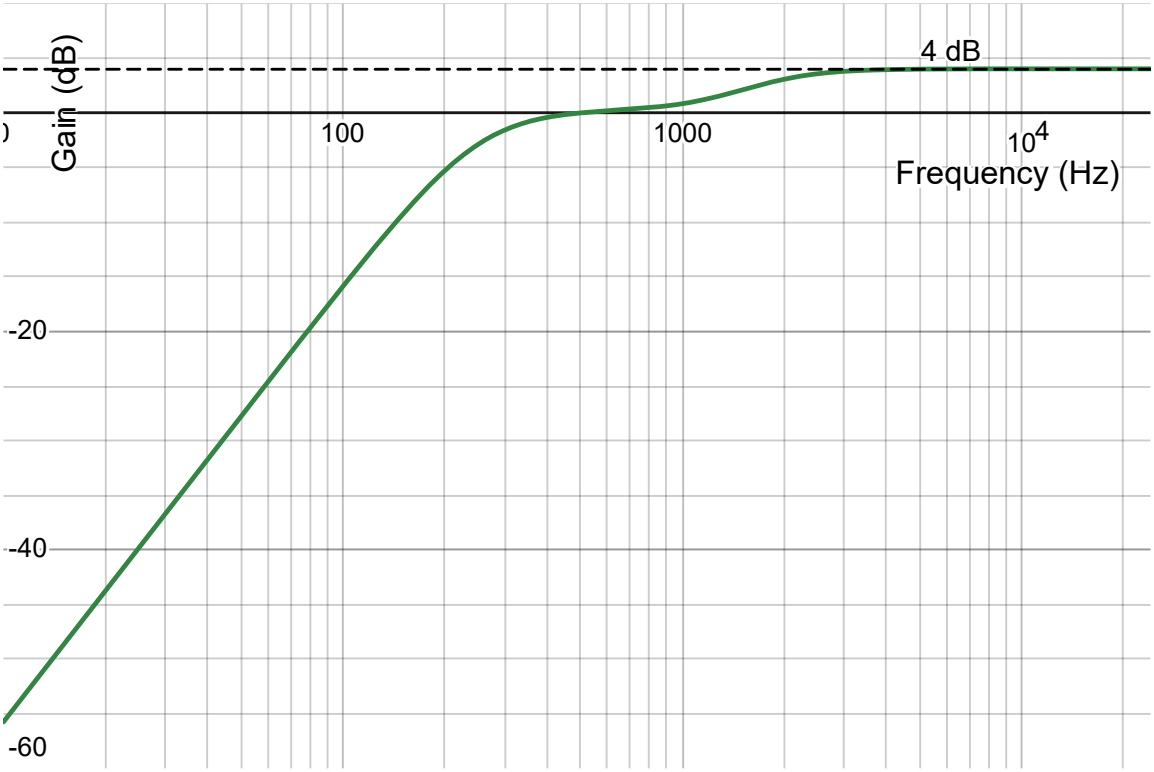

Figure 5: Frequency response of the combined ITU-R BS.1770 K-weighting filter.[6]

In the case of frequency, human perception is not quite logarithmic at lower frequencies. Rather, it appears split into bands, where each band has distinct perception characteristics. The widths of these bands have been measured empirically; important for us is the fact that the bandwidths seem quite constant for lower frequencies and begin to increase for higher frequencies. The Mel scale (short for melody) is one way of approximating human audio perception; we seem to perceive roughly mels as roughly linear in frequency. A function to convert a frequency in Hertz to mels is given in [7], Ch. 4:

$$m = 2595 \log_{10}(1 + f/700)$$

In Ang, a normalised approximation of the Mel scale is used. For any frequency *f*, the normalised Mel counterpart is $M_n(f/F_s)f$, where F_s is the sampling frequency and L is given by

$$M_n(x) = \frac{2}{3}\log_{10}(1 + 30x).$$

This function was identified by normalising the values of the Mel scale from 0 Hz to 20 000 Hz down into the range $[0, 1]$, and finding a function that approximates the resulting graph well by inspection. The inverse transform for the normalised approximation is found trivially to be

$$M_n^{-1}(x) = \frac{1}{30}(10^{\frac{3}{2}x} - 1).$$

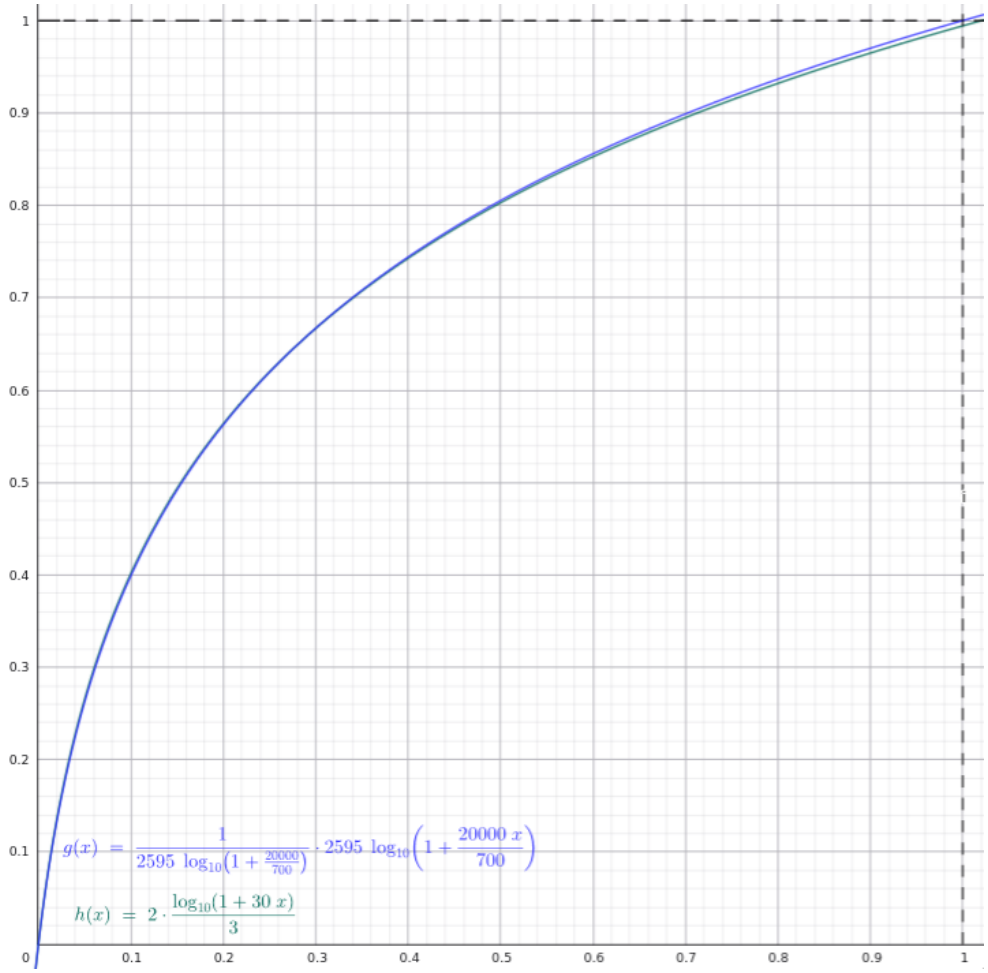

Figure 6: Graphs of the normalised Mel scale (g) and the approximation used in Ang (h), from 0 to 1.

5 DATA STRUCTURES

We discussed the data structures of the signal processing pipeline in *Buffering Strategies* (p. 8). Ang does not consist only of signal processing, though, but also has a user interface. Perhaps the most interesting data structures used in Ang outside the context of signal processing are the observable structures provided by ScalaFX and the *reactive mapping* for these structures implemented in the program. ScalaFX allows programs to manage their state through *properties*, which are mutable, named structures that can notify other code when their value changes. Also supported are *bindings*, which derive their value from other observable values, and generic observable values and buffers, which are similar to properties but are not named. For example, the effect chain in Ang is implemented as an observable buffer of effect instances, such that any code can subscribe to changes to the observable buffer and be notified when they happen (using a callback function). Nearly all user interface elements provided by ScalaFX support these kinds of observable values, which makes reactive state trivial.

One notable shortcoming of the ScalaFX observable system is the lack of *stateful reactive mappings*. Ang represents user interface elements using a concept of *instance tabs*, which are always bound to a specific effect instance, and can be created by a function that dispatches to the right type's constructor based on the type of the effect instance. We might wish that we could, then, implement a collection of instance tabs by simply asking ScalaFX to create an instance tab for each effect instance that is added to an observable buffer, and drop the reference to the tab when the instance is removed from the buffer.

This is, however, not functionality provided by ScalaFX or the underlying JavaFX library—so we implement it ourselves. Ang defines class, *ReactiveMapping*, which holds an internal reference to a backing observable buffer, a function that generates derived values, and a hash-map that stores the derived values.

When a reactive mapping is created, it subscribes to changes to the underlying buffer and automatically adds and removes values from its hash-map, calling the converter function when necessary. The mapping exposes only a single method, *get*, which returns the mapped value for any value from the underlying observable buffer.

Notably, the reactive mapping does not generate values on every call to *get*, but only when the observable buffer's values change. This means that effect instance tabs created by the observable mapping can safely store state, and the state won't be reset by a subsequent call to *get*.

6 FILE SYSTEM ACCESS

The technical plan detailed two reasons for Ang to access the file system; the first reason was to cache the downloaded JavaFX binaries after the boostrap phase in order to avoid downloading them on each startup while also allowing Ang to work cross-platform without having to include system libraries for each supported operating system. The second was in order to store configuration files for the operating system user with information such as the effect chain configuration that was in use when Ang was last closed, allowing the program to retrieve these data and apply them automatically when it is started again.

Only the former of these two features was implemented, with the files being stored in the operating system cache directory. The correct cache directory for the operating system is determined by the bootstrap using the *os.name* system property. There is also a sequence of fallbacks for cases when detection fails, ultimately ending in storing files in

the working directory under a subdirectory called *.cache* if nothing else is available.

The second feature, storing configuration parameters for restoration when the program is started again, was not implemented. It would be an interesting feature to add in the maintenance phase.

7 TESTING

The testing phase outlined in the technical plan consisted mostly of the process of implementing automated tests for the model and user interface components. The particular subjects of the tests were the effect chain, properties of the Fourier transform, as well as the user interface for choosing effects.

The test runner included with SBT was used for executing the tests. Two separate testing frameworks are used; ScalaCheck[8] for property-based tests and MUnit[9] for unit and user interface tests.

PROPERTY-BASED TESTING

The Fourier transform is tested with *property-based tests*, i.e. tests that are defined only by some property that the output of the code must satisfy. Three properties of the FFT are tested to ensure that it is functioning correctly.

The first property is that the inverse transform of a transformed input signal is, itself, the input signal On each test run, ScalaCheck generates 100 cosine signals of some arbitrary length and frequency, then checks that the property is satisfied (within the uncertainty of double-precision floating point arithmetic).

The second property is that the frequency spectrum of a real cosine signal that has a wavelength that is a harmonic of the FFT size will be zero everywhere except the bins corresponding to the frequency of the signal and its conjugate frequency. The energy of the spectrum should be the FFT size in total, and it should be split evenly between the two bins. In the case of a signal at the Nyquist frequency, the bins will be the same. In this case, the total energy is contained entirely in one bin.

The third and final property relates to the sequence of swaps performed as the last step of the DIF FFT; it is that no two values are swapped to the same position in the output (i.e. the swaps are unique).

IMPERATIVE TESTING

The other two types of tests—unit and user interface—are specified imperatively, so each test is a sequence of instructions followed by checks that the behaviour matches what is expected.

The test on the effect chain is not particularly interesting; we create an input that feeds samples from an in-memory buffer, an effect that does nothing with the samples, and an output that records the received samples. We simply verify that the output receives the samples that were sent by the input.

The user interface tests are more technically interesting. They work by creating a complete Ang user interface window and simulating mouse movements, clicks, and key presses in order to determine the actual behaviour of the user interface. When the tests are run in *headful* mode, the window is shown on the screen, and the real operating system mouse is used to click buttons on the user interface one at a time. By setting a value for the environment variable *CI*, the tests can be run in *headless* mode using OpenJFX' Monocle[10]. Monocle is a software implementation of the Glass Windowing Toolkit, the JavaFX component which acts as communication interface between native operating system components and the JavaFX Platform.

Since Monocle is not connected to any real hardware resources, it can be run without any attached monitor, keyboard, or mouse, making it especially suitable for headless environments like continuous integration (CI) runners.

8 KNOWN BUGS AND MISSING FEATURES

There are is one known bug in Ang, as well as features that I would have hoped to add but have not done so yet. The bug is that in the Spectrum Transfer effect (shown in Figure 3 and 4), there are certain situations in which the transfer degrades into an unrecoverable invalid state, leading to an empty output signal, and playback must be stopped and re-started in order to recover the effect. Particularly, this occurs when playback is active but the audio signal is entirely quiet, for example when feeding audio from the output of a music player into Ang.

As discussed earlier, the spectral transfer effect is implemented as a collection of complex values, phasors, which are advanced based on the contribution of each input band. There are three outstanding issues with the implementation of the effect that I have not yet been able to fix. First, the effect has no windows for very low nor very high frequencies, leading to low and high frequencies being cut off entirely. Second, there are still some artefacts in the audio, particularly when shifting frequencies from the low end of the spectrum to the high end. Finally, moving a control point back to its original position leaves some artefacts in the audio, presumably because the output phasors are not reset correctly.

A great deal of features have not been implemented yet, but this also shows the potential that Ang has for additional development.

The first and perhaps most obvious to the user is the equaliser effect's user interface; it is currently a blank screen with large, single-coloured red circles. Contrast this with the transfer effect, which embeds a spectrum visualiser and transfer graph and has control points that are filled with a gradient rather than being a solid red. The transfer effect also has axis labels for both axes, whereas the equaliser effect has none. This would be trivial to fix, but I have not had time yet.

The second feature is the configuration file support that was planned initially in February. It was of low priority to me to begin with, so it's natural that it fell out of scope for the duration of the project. To implement this, it would be necessary to add a serialisable abstract type member for the configuration to EffectInstanceTab; implementors could override the type and use it to store their configuration state, which could then be retrieved automatically when instantiating the tab. The largest body of work re: this feature would be transferring state management into the configuration for each effect. Perhaps a day's work or so.

I would also like to re-implement the vocoder now that the inverse Fourier transform is available; it would improve both the performance and quality of the effect. Furthermore it would be a good idea to add a non-vectorised fallback for the convolution filter implementation; since jdk.incubator.vector is not available in the mainline JDK, the only way to use the convolution filter at this time is to start the program with the special Java Virtual Machine (JVM) flag *--add-modules jdk.incubator.vector.* A fallback implementation would also allow removing the *Probe* class entirely.

There are also many improvements to the structure of the codebase that could be made. For example, all user interface tabs are implemented in a single source file. This was done for historical reasons; it was initially the plan to have a *sealed trait* for the tabs, which is only possible if all implementors are in the same source file. Sealed traits would carry the benefit of compile-time safety for the user interface

dispatching match arm. Since *EffectInstanceTab* is not a sealed trait, Ang currently implements *the worst of both worlds*, so to speak. It gets neither the code structure of the non-sealed approach nor the type safety of sealed traits.

In Java, it is possible to have sealed interfaces with implementors from different source files, and, indeed, different packages. It could be worthwhile to look into implementing something like this into the Scala language.

On the topic of improvements to the Scala language, Ang currently implements its own *transpose* method for arrays of audio frames, shadowing the existing method from the Scala 3 standard library. This is done because the standard library's *transpose* has a bug that breaks the implementation for arrays of non-arrays.

The transpose method is documented as allowing a function, *asArray* to be provided, which gives instructions on how to convert the elements of the receiver array into arrays, if they are not already. However, regardless of whether this function is provided, the actual implementation of *transpose* assumes that the input is already a two-dimensional array, which is not the case for arrays of component type *AudioFrame* (since that is not an array).

The bug has been present for over a decade now, but no bug report exists currently. I have already made the relevant fixes to the standard library, but have not opened a merge request as of yet.

I digress; there are many places in Ang that are opportunities for further development. Ang would also benefit from new effects, such as a compressor. Finally, it would be interesting to look into support for viewing multiple effect instance tabs simultaneously, though this would likely warrant a redesign of the user interface.

9 STRENGTHS AND WEAKNESSES OF THE SOFTWARE

I identify three strengths and three weaknesses of Ang, particularly from the point of view of the user stories outlined in the general and technical plans.

STRENGTHS

1. Real-time processing. Ang is one of only a few tools that can process and apply effects in real-time, with sufficient performance to apply complex manipulations without significant stuttering. For Linux, there exists a similarly capable tool called *EasyEffects*; aside from this, very few such programs exist.
2. Cross-platform interoperability. Ang is built on top of the Java Virtual Machine, using Java's built-in hardware abstraction layer for audio. It can operate on all major operating systems, processing practically any audio format: the conversion is done internally by Java. I imagine it would not be an insurmountable task to make Ang into an app for Android phones, for example.
3. Strong type safety. Ang's programming language of choice being Scala, with its robust type system, reduces errors and problems significantly. The user interface does not crash if an error occurs; if the error cannot be corrected automatically, it is simply shown to the user.

WEAKNESSES

1. Lack of effects and configurability. Despite significant effort, Ang is still in its early stages of development. There are six somewhat interesting effects, but this is not enough to be an outstanding piece of software, nor enough to compete with existing programs like *EasyEffects*.

2. Performance constraints. The Java Virtual Machine is a double-edged sword in terms of performance; while it can make meaningful, often non-obvious optimisations at runtime, it still carries significant overhead compared to a native implementation. A native binary built with something like Rust would perhaps be a more suitable choice, carrying both the type safety of Scala and the performance of native applications.
3. Poor development interest and funding. Ang is, in the end, only developed and maintained by me. It is an academic endeavour, and did not benefit from great advances in programming technologies like agentic programming using large language models because these tools are not in line with my academic goals. Development of Ang is neither profitable nor full-time work for me, and in fact not even my primary side project. It is unlikely that Ang will see much progress after the academic phase of this project is completed, unless I or someone else take a specific interest to its development.

10 DEVIATIONS FROM THE SCHEDULE

In the technical plan, a three-phase development plan was defined for the initial development of Ang. The plan consisted of two-week phases, with development of the audio processing pipeline in the first phase, user interface in the second phase, and testing and integration of the components in the third phase. The audio processing pipeline was mostly developed in phase one, in accordance with the plan. The second phase was started three weeks after the project began, rather than two. It also involved preliminary integration efforts and was not solely focused on the user interface. The third phase began ten days late, and consisted not only of developing automated tests and integrating components but also signifcant refactoring of the model and user interface as well as adding the *Spectral Transfer* effect.

Deviations from the initial plan and schedule occurred also w.r.t documentation, as the initial plan did not set any goals nor a timeline for documenting the project. While some documentation was written over the course of development, most in-code documentation was added only near the end of the third phase. The final report was written at the end of the third phase, starting on the 20th of April and ending on the 26th of April.

It is very likely that the third phase of development will be completed before the project deadline, the 28th of April, allowing for the project to be completed on-time despite the significant delays in phases two and three.

11 FINAL EVALUATION

Overall, the project was a great success. In the general plan, it was described as “an étude into graphical audio manipulation in Scala”.[2] It certainly was successful as an educational project, and I learnt a lot about signal processing, ScalaFX, and management of software projects. It successfully fulfilled the user stories set out in February, and the program ended up being rather interesting and even somewhat useful.

Perhaps the main shortcomings of the project were in regard to the number and quality of features as well as the timeline for the project. Phases two and three started late, and the fortnightly sprint meetings were neglected and scheduled only at half the

requested frequency. There was also significant overlap in the focus areas of development in each phase, which made the development process more ambiguous. That being said, it is perhaps unwise to attempt to segment the development process into distinct parts, as it should be a continuous and integrated effort.

I suspect that the missing features could be implemented without much work, and this capability is largely owed to the modular and encapsulated nature of the model component of the program; the project was especially successful in this regard.

12 SOURCES

As noted in the technical plan, I relied heavily on multiple sources for instruction, guidance, and reference during the implementation of this project. Relevant articles on signal processing, as well as the Scala SDK Documentation and the Java® Platform, Standard Edition & Java Development Kit Version 25 API Specification[11], especially concerning the javax.sampled API, were crucial. The structure of the program was based on previous experience from working on Virekuvain[12], which only allowed for visualisation of audio signals rather than real-time manipulation. Draft implementations for the normalised Mel scale and ITU-R BS.1770 -standard frequency response were made using an AI assistant; these were replaced with hand-written implementations because the AI was not sufficiently and could not, for example, derive the correct frequency response from the ITU-R BS.1770 filter coefficients. I implemented changes to the spectral transfer effect following an AI assistant's recommendation to use complex phasors to fix artefacts caused by phase discontinuities. In addition, recommendations for improvements to the UML diagram were generated by an AI assistant, while the diagram was drawn by hand using the web site draw.io[13]. Overall, large language models served as important tools but did not contribute to the development of the project due to issues of academic integrity and code quality.

Bibliography

1: École Polytechnique Fédérale, The Scala Programming Language, 2002, https://www.scala-lang.org/
2: Ilari Suhonen, General Plan for an étude into graphical audio manipulation in Scala, 2026
3: Ilari Suhonen, Technical Plan for Ang, an audio manipulation program in Scala, 2026
4: Traditional, arr. Marty Robbins, Streets of Laredo, 1960
5: Robert Bristow–Johnson, Answer to "How do I manually plot the frequency response of a bandpass Butterworth filter in MATLAB without freqz function?", 2014, https://dsp.stackexchange.com/a/16911
6: Ilari Suhonen, Desmos Graph for the combined Frequency Response of the ITU-R BS.1770 K-weighting filter, 2026, https://www.desmos.com/calculator/hfxppk0tga
7: Douglas O'Shaughnessy, Speech Communications: Human and Machine, 1999
8: Rickard Nilsson, ScalaCheck: Property-based testing for Scala, 2021, https://scalacheck.org/
9: Scalameta, MUnit: Scala testing library with actionable errors and extensible APIs, 2026, https://scalameta.org/munit/
10: Oracle Corporation and/or its affiliates, Monocle, 2026, https://wiki.openjdk.org/spaces/OpenJFX/pages/17957197/Monocle
11: Oracle Inc. and/or its affiliates, Java® Platform, Standard Edition & Java Development Kit Version 25 API Specification, 2025, https://docs.oracle.com/en/java/javase/25/docs/api/index.html

Ilari Suhonen
Year Group 2025
Student ID 102869275
2026-04-25

CS-C2120
B.Sc. and M.Sc. (Comp. Sc.)
School of Science
Aalto University

12: Ilari Suhonen, Virekuvain, 2023
13: JGraph, draw.io, 2026, https://draw.io/

APPENDIX 1

The source code for this project is distributed on the Aalto University Version Control System (VCS), Aalto Version. The URL for the Git repository is https://version.aalto.fi/gitlab/suhonei4/ang.

The source code is available to staff of the CS-C2120 course upon request; if you do not already have access, you may request it by sending me an E-mail. My e-mail addresses are ilari.suhonen@aalto.fi and ilari.suhonen@gmail.com.