Computer Music: Languages and Systems - Homework 2

Francesco Colotti *(francesco.colotti@mail.polimi.it)*
Gioele Fortugno *(gioele.fortugno@mail.polimi.it)*
Matteo Gionfriddo *(matteo.gionfriddo@mail.polimi.it)*
Marco Granaiola *(marco.granaiola@mail.polimi.it)*
Emanuele Greco *(emanuele2.greco@mail.polimi.it)*

May 5, 2023

In this report, we will briefly analyze *Fire*, a multi-band distortion plugin made with JUCE by jerryuhoo.

1 Introduction

Fire is practically divided in two tabs, the *Band Effect* part and the *Global Effect* part. Both share a representation of the frequency spectrum of the audio, an Output and Mix knobs and a graph visualizer. The first part permits to divide the audio spectrum up to a maximum of four bands, for each of which one can set the distortion parameters. The second part instead allows the user to control three filters using a graphic equalizer, which are a low-pass filter, a band-pass filter and a high-pass filter. Finally, in the top region of the plugin an interface is used for parameters and presets management.

For the code analysis we will first explore the DSP starting from the function ProcessBlock() in which can the order of signal processing blocks is defined. Then we will concentrate on the function ProcessOneBand(), where the single band distortion process is organized. This is followed by a discussion on the leftChain and rightChain objects, which are crucial elements for implementing the Global Effect part, then a description of the functions within the ClippingFunctions.h file (the core of the distortion processes) and a brief explanation of the function getStateInformation(), setStateInformation(), important for handling presets and current parameters. We also spend some words on the HQ and the Downsample functionalities of the plugin.
Finally there is a general view of The Graphical User Interface.

2 Block diagram

2.1 Overview

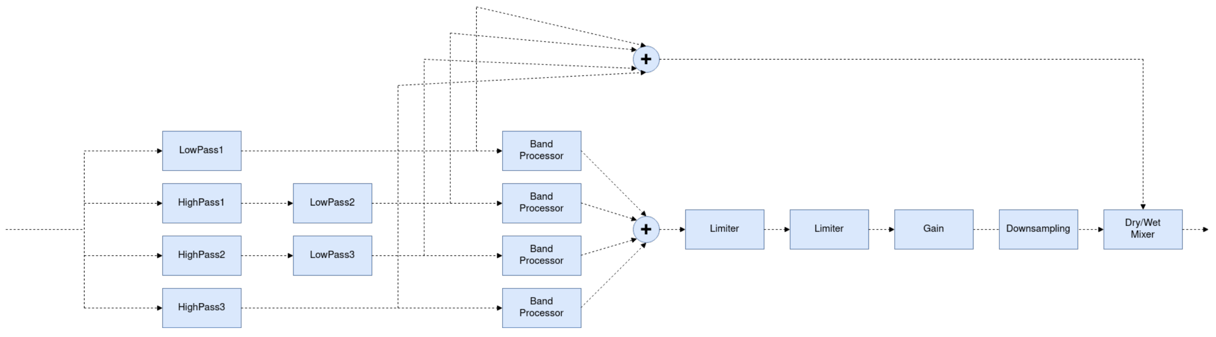

Figure 1: Block Diagram of the behaviour of the plugin

2.2 Band Processor

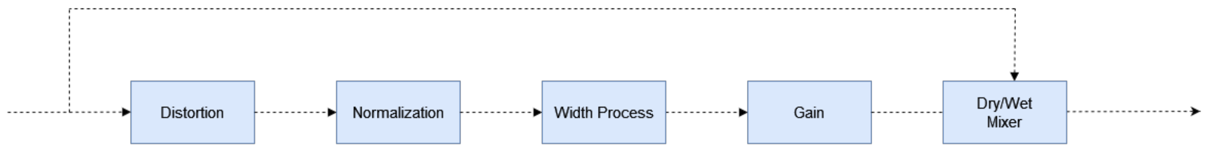

Figure 2: Block Diagram of each Band component

3 DSP

3.1 ProcessBlock()

As we anticipated, to understand the processes the signal goes through, we need to look at the function `processBlock()` located inside `PluginProcessor.cpp` file. In this there are numerous processes that are used in a well-defined order but we will analyze only the main ones that provide us a broad idea of how the signal is processed. A general block diagram of the signal path is visualized in Figure 2.1. Initially it is established whether the effect is bypassed or not, then the `ScopedNoDenormals` class is used to avoid ”denormal numbers” problems in the audio signal. With the `AudioBlock` class we access the audio data in the buffer to enable audio processing. Following a multiband process, the audio signal is divided into up to 4 separate frequency bands and with a different processing for each. The frequency breakpoints are stored in `freqArray` and sorted in ascending order, and the number of breakpoints is stored in `lineNum`. The audio buffer is split into four auxiliary buffers, `mBuffer1`, `mBuffer2`, `mBuffer3`, and `mBuffer4`, one for each band. The plugins keeps track of which band is enabled and if some band is in solo mode. The left and right channel identifiers are set according to the total number of input channels. For each band, an AudioBlock object is created from the corresponding `mBuffer` and a `ProcessContextReplacing` object is passed as well. The `lineNum` variable determines the number of frequency breakpoints: together with a series of `if` statements the function performs the necessary frequency filtering. If `lineNum` is greater or equal than 1, then the buffers are filtered. For example, for `lineNum`=2 a low-pass filter with cutoff frequency `freqArray0` is applied to the first buffer (`mBuffer1`) to obtain the first band. Then a high-pass filter with the same cut-off frequency is applied to the second block of buffer to get the complementary band. The dry buffer of each band is then copied into `mDryBuffer` for later processing. If the `multibandEnabled` option is true for the current frequency band, the `processOneBand()` function is called to process the buffer. This function does the actual processing of the selected band through the use of values retrieved from the `treeState` object. The values are retrieved using the IDs passed to the function. Finally, the RMS values for each left and right input and output channel are calculated and stored in variables for the current frequency band. If all of these variables are false, the `mLatency` latency value is set to 0 and the `setLatencySamples()` function is called with this value. Next, the output buffer is initialized to zero, and then populated with data from the four input buffers: `mBuffer1`, `mBuffer2`, `mBuffer3`, and `mBuffer4`, depending on the `shouldSetBlackMask()` and `lineNum` variables. The `shouldSetBlackMask()` function is used to check if the relevant frequency band of the current multiband has been activated. After each band has been processed the complete output buffer is processed some more. First, the left and right channel of the buffer are processed through the same chain composed of a low-pass, high-pass and peak filter, and then the output is processed through a limiter. Both of these process can be bypassed based on user input. Finally the audio block goes through a gain processor. The filters, limiter and gain parameters values are retrieved from the `treeState` parameter tree. Next, the audio signal is mixed with the dry signal according to the `MIX` parameter ID, using a dry-wet mixer. The mixer blends the audio signals while taking into account the plugin's latency. After mixing, the audio buffer is copied to the `mWetBuffer` buffer, which is used to calculate the frequency spectrum of the audio signal through the FFT. The code then computes the RMS values of the left and right channels of the output signal, which are used to drive the VU meter. Finally, the `mDryBuffer` buffer is cleared, preparing it for the next audio block to be processed.

3.2 ProcessOneBand()

It represents the chain through which each band passes and is located inside `PluginPr-ocessor.cpp` file. A general block diagram of the signal path is visualized in Figure 2.2. Other than the actual buffer and the context it takes several arguments, most of which are string ID used to retrieve values from the parameter tree. First it copies the buffer `bandBuffer` given as a parameter to another one called `dryBuffer`. Then it calls the following functions, each with `bandBuffer` as a parameter.

3.2.1 processDistortion()

Here is where the actual distortion is applied. First an AudioBlock for the input (from the pointer to the array of channels passed as argument) and for the output are created. Each indication about the plugin settings (e.g. number of band considered, type of distortion ecc...) are retrieved by parameters of a `treeState` object and by the predefined constants. Based on these values operations like oversampling or usage of the safe modality are performed, often in a high level, calling the related specified functions defined in the other files. In particular, the pointer `overdrive` to a `BiasProcessor` object is first used to set the distortion amount, the chosen waveshaping (calling one of the function present in`ClippingFunctions.h`)and the bias and rectification amount. Finally the processing is applied to the current block.

3.2.2 normalize()

Normalization process is really simple. For each input channel is retrieved a writable pointer of the current audio, from which are calculated the minimum and maximum values and stored in a `range` object. To each sample of the buffer, thanks to the use of some `SmoothedValue` objects, is applied a first linear smoothing that for the retrieving of the next value follows the rule : $(\texttt{RangeMax} + \texttt{RangeMin})/2$, and then is done a multiplication to a second linear smoothing factor.

3.2.3 WidthProcessor

`WidthProcessor.process()` is applied in case of a stereo signal to make it ”wider” or ”narrower”. It starts by computing the `mid` and `side` component of the buffer, obtained through the sum and the difference of the two signal, divided by the square root of 2. The function then amplify the different components by a factor of `1-width` and `width` respectively before reconstructing the left and right channel from the 2 components.

3.2.4 processCompressor() , processGain() , mixDryWet()

. In each case are simply retrieved the associated parameters from the `TreeState` object and, based on them, applied the compression, the gain or the dry-wet process by respectively a `CompressorProcessor`, a `GainProcessor` object or a `DryWetMixer` object .In `mixDryWet()` wet latency is set to a particular value when we are in HQ mode.

3.3 leftChain and rightChain

The `leftChain` and `rightChain` are `ProcessorChain` objects that join together the chain of processors that implements the 3-band parametric equalizer, they are instances of `MonoChain` that is defined inside the `FilterControl.h` file. The resonant low-pass(high-pass) filter is implemented by using and IIR high order low-pass(high-pass) filter made with Butterworth method summed with an IIR peak filter, while the band-pass filter needs obviously only an IIR peak filter. As a consequence each process chain is made by 3 peak filter and 2 cut filter. In particular each cut filter is a *ProcessingChain* of 4 filters. Only one of them is active at a given time, depending on the chosen filter slope (12, 24, 36 or 48 dB).

Before the processing happens the function `updateFilter()`is called (inside `processBlock()`). This function gets the filters parameters from *treeState* and calls one helper function for each filter to set the appropriate coefficients. In particular, the corresponding update function of each filter works by first computing the correct coefficients from the parameters obtained through the user input, and then calling the function `UpdateCoefficients()` to set these values inside the filter in the processing chain. In the case of the low-pass filter and the high-pass filter is also updated the related cut filter by the function `UpdateCutFilter()`. After the update the filters are ready to be used and the `process()` function is called on both the chains.

3.4 ClippingFunctions.h

In this file are defined all the waveshaping functions designed to distort the signal. The concept behind distortion functions is to alter the incoming audio signal to add harmonics not present in the original signal. This effect is typically achieved by non-linear saturation of a circuit, which distorts the original audio signal waveform. In this way, richer and more complex sounds can be obtained, ranging from slight saturation to extreme distortion. The distortion functions in this code use various nonlinear saturation algorithms to create distortion effects, and each function has a unique sonic characteristic that can be suitable for different types of sounds and musical styles. The functions take the input signal x, apply mathematical formulas and return a distorted signal x. This code is a template T that is inserted into the `pluginprocessor.h` and then called from the `pluginprocessor.cpp`. Specifically we have 12 different functions grouped into 3 categories (SoftClipping, HardClipping, Foldback): **ArctanSoftClipping**: this function applies the arctangent function to the input variable and divides the result by 2.0, thus applying "soft clipping" via the arctan. The input value is limited between -1 and 1. **ExpSoftClipping**: this function applies a "soft clipping" via the exponential: if the input value is greater than 0, it is subtracted from 1 and the negative exponential of this result is calculated, otherwise the input value is added to -1 and calculated the exponential of this result. The output value is limited between -1 and 1. **TanhSoftClipping**: this function applies a "soft clipping" through the hyperbolic tangent function. **Cubic-SoftClipping**: This function applies "soft clipping" via a cubic function. If the input value is greater than 1, the output value is set to 2/3; if the input value is less than -1, the output value is set to -2/3; otherwise, the output value is calculated by subtracting from the input value its third power divided by 3. The output value is then multiplied by 3/2. **HardClipping**: this function applies "hard clipping" via a step function: if the input value is greater than 1, the output value is set to 1; if the input value is less than -1, the output value is set to -1; otherwise the output value is equal to the input value. **SausageFattener**: This function applies "soft clipping" similar to a "bouncy sausage", where the output value is multiplied by 1.1 and then a cubic function is applied to "crap" peaks with amplitude greater than 1.1. The output value is then limited between -1 and 1. **SinFoldback**: This function returns the unbounded value of sin(x). **LinFoldback**: this function applies a "foldback distortion" in which the input value is folded on its own function (y = x) after a certain point. In particular, if the input value is greater than 1 or less than -1, the absolute value of the input value is calculated, then the remainder of division by 4 is calculated, this value is then subtracted from 2, and finally the the absolute value of this result is subtracted from 1. The output value remains unchanged if the input value is between -1 and 1. **limitClip**: This function limits the input value between -0.1 and 0.1. **SingleSinClip**: This function applies a distortion via a single sine wave, where the input value is multiplied by a sine wave. The output value is then limited between -1 and 1. **LogicClip**: this function applies a "hard clipping" where the input value is compared to 0: if the input value is greater than 0, the output value is set to 1, otherwise the output value is set to -1. **TanClip**: this function applies "hard clipping" via the tangent function: if the input value is greater than 1, the output value is set to 1; if the input value is less than -1, the output value is set to -1; otherwise the output value is equal to the tangent of the input value. The output value is then limited between -1 and 1.

3.5 getStateInformation(), setStateInformation()

This methods are used to store and recall parameters in the memory block and are located inside `PluginProcessor.cpp` file. This can be done either as raw data, or using the XML or `ValueTree` classes as intermediaries to make it easy to save and load complex data.

3.6 HQ

The user has the possibility to work with higher sampling rate in order to reduce the artifacts introduced by the distortion at the cost of higher computation complexity. Thus, if the HQ mode is enabled, a 4x oversampling is applied inside the `processDistortion()` function for each block right before doing the processing of it. After the non-linear processing, in order to re-sincronize the output signal with the internal latency of the whole oversampling behaviour, downsampling is also performed.

3.7 Downsample

In the Global tab a downsample effect is available. It can be used to obtain a rougher sound and/or to obtain an old school chiptune-ish sound. We chose to include the code for the downsampling of a channel below.

```
//this is repeated for every channel (2 channels for stereo)
//for simplicity we only consider one channel now
//to see the full code check out
auto* channelData = buffer.getWritePointer(channel);
```

```
for(int sample=0; sample<buffer.getNumSamples(); ++sample){
    if (rateDivide > 1){
        if (sample % rateDivide != 0)
            channelData[sample] =
            channelData[sample - sample % rateDivide];
    }
}
```

4 The Graphical User Interface

For a more detailed analysis of the implementation of the GUI we will consider the plugin as made by three panels : Top Panel (the one that allows presets management), Spectrogram Panel (the one in which the audio frequency spectrum is displayed) and Control Panel (the one which are presents graph equalizer and control knobs).

4.1 Top Panel

The code for the Top Panel is located in */Panels/TopPanel*, in the files *Preset.cpp* and *Preset.h*. As it has been previously anticipated, the Top Panel manages presets (class *StatePresets*) and how the user can interact with them via the GUI (class *StateComponent*); this includes both user presets and temporary presets for doing A/B comparisons.

Presets are saved in a *.fire* format, which is actually just a XML file containing a single elements (*WINGSFIRE*) with all the effect parameters as its attributes. For this reason, the first part of *Preset.cpp* is dedicated to handling XML files. In this part the class StateAB is also implemented. Presets for A/B comparisons are treated as normal presets, just saved in a Temp folder. A PresetNameSorter class, whose job is to sort the presets in the UI, is also implemented. Interestingly, the sorter takes a generic `const juce::String &attributeToSortBy` as its input and sorts based on it; this means that multiple sorting criteria could be implemented, though the author only used the preset names. A second parameter defines whether to invert the order of the elements or not, but this is also not used and always set to true (do not invert).

The second part of the file (lines 208-433) is dedicated to the StatePresets class. It handles loading, saving and displaying presets in the Menu.

The third part of the file (line 438-837) is dedicated to the StateComponent class which actually implements all functionalities in the GUI. This means an A/B button, a "Copy" button for AB comparisons, a preset menu, where presets are listed in alphabetic order, divided in folders, then a "Save" button and finally the "Menu" button, which can be used to load the default preset and open or rescan the preset folder and its subfolders.

4.2 Spectrogram Panel

The Spectrogram Panel part is dedicated to the part of the GUI when the Spectrogram panel is shown. It is highly integrated with the Control Panel, in fact it is possible to use this part of the GUI to create up to four bands, control their boundaries and eventually turn bypass the effect in them if needed.

We will not comment this part of the code any further since it goes out of the scope of this report.

4.3 Control Panel

Four maximum bands means four times the controls to implement, and in fact that's what happens in the Control Panel part. This part of the code is responsible for "spawning" controls in the GUI, both for individual bands and for the whole plugin (global controls); the code for the four visualisers (distortion input-output response, oscilloscope, meter and a polar vectorscope) in the bottom left corner is also included in a special folder in this section.

We'll focus on the oscilloscope as an example and we'll omit the rest in order not to overflow this report with extra informations.

The general idea is to get the data from a buffer calling `processor.getHistoryArrayL` (also repeating it for the right channel with getHistoryArrayR if the signal is stereo), find the maximum amplitude `maxValue`; if it's bigger than a minimum (in order to avoid divisions by very low number or even zeros), normalise the values in the array and then do a for loop to draw the waveform using the `strokePath` function of a `juce::Graphics` object received as an input. The function uses a scaling factor on the for loop index in order to extract amplitude values from a smaller chunk of samples distributed uniformely in the array. Since `strokePath` requires a `juce::Path` object as an input, one is created (two for stereo) and is (are) constantly updated with each iteration using values from the extracted samples.

5 Conclusion

Thanks to the correct implementation of the overdrive processor and the many waveshaping functions the *Fire* is a viable open-source option to the many multi-band distortion plugins on the market. It does its job with great results and the intuitive graphical interface makes it easy to use. A plus point of the code is that it is written following a modular approach, such that the whole audio process can be seen in an high level by a main function while the actual work is divided between low-level blocks. At the same time it could benefit from a code cleanup (as we already said there are files like *diodeWDF.h* and *WDF.h*, but also *Delay.h* that are practically not used). Another future improvement could be more flexibility in the *Global Effect* part, particularly giving the user the freedom to have more than 3 filters and decide for each the filtering type.