Computer Music - Languages and Systems

Homework N.2 - Analysis of an existing Juce plugin

BYOD: Build Your Own Distortion

Professor: **Fabio Antonacci**

Academic year: 2022-2023

Group 7 - Lemonjuice:
Guglielmo Fratticioli (Matr: 214322)
Chiara Lunghi (Matr: 233195)
Alessandra Moro (Matr: 102283)
Elia Pirrello (Matr: 223050)

In the following report we will provide an overview of the software's features and dive into an analysis of the software model, including the frameworks used and the structure of the processors and audio buffer handling. Finally, we will draw conclusions about the complexity of the software and discuss key implementation strategies, as well as provide a review of some of the presented DSP algorithms.

Contents

1 Introduction

BYOD (Build Your Own Distortion) is a virtual pedalboard designer that emulates a physical guitar pedalboard through the use of virtual cables connecting various processors. Some of these model specific real circuits, replicating accurately the sound. Key features of BYOD digital software are unlimited effects, complex routing, and a preset manager which enhances its flexibility. Additionally, the software is available as a free open-source standalone desktop version, an iOS standalone version, and a VST plugin.

A virtual pedalboard designer offers several advantages over traditional analog equipment. It allows guitarists to digitally model analog circuits, enabling them to achieve vintage sounds without the costs and maintenance needs of physical gear. Moreover, virtual pedalboards offer greater portability, allowing users to experiment with various effects and settings both in the studio and on stage.

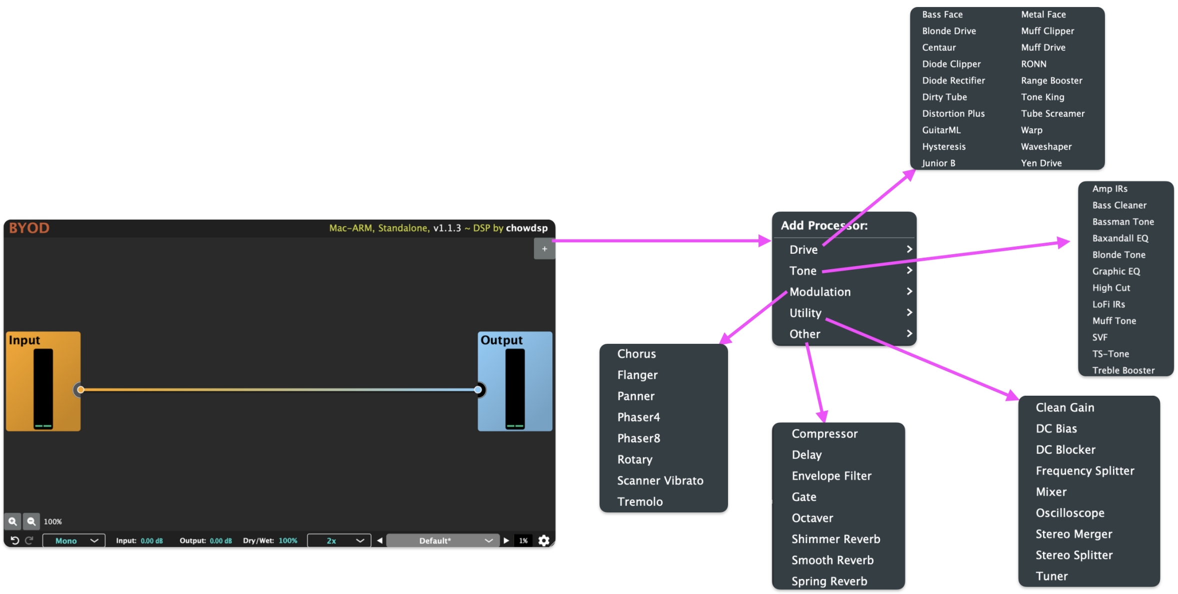


Figure 1: BYOD Graphical Interface

2 Features Overview

- **Patchable Grid Layout**

  One of the most noteworthy features of BYOD is its patchable grid layout, which offers users a modular and visual way to connect different modules and processors. With the ability to drag and drop cables and modules onto the grid, users can create intricate and personalized signal chains.

- **Stereo Audio and Oversampling**

  BYOD's DSP core is designed to support Mono and Stereo audio, while also offering oversampling capabilities of up to x16. This feature proves especially valuable when using nonlinear processors, as it helps to minimize aliasing problems.

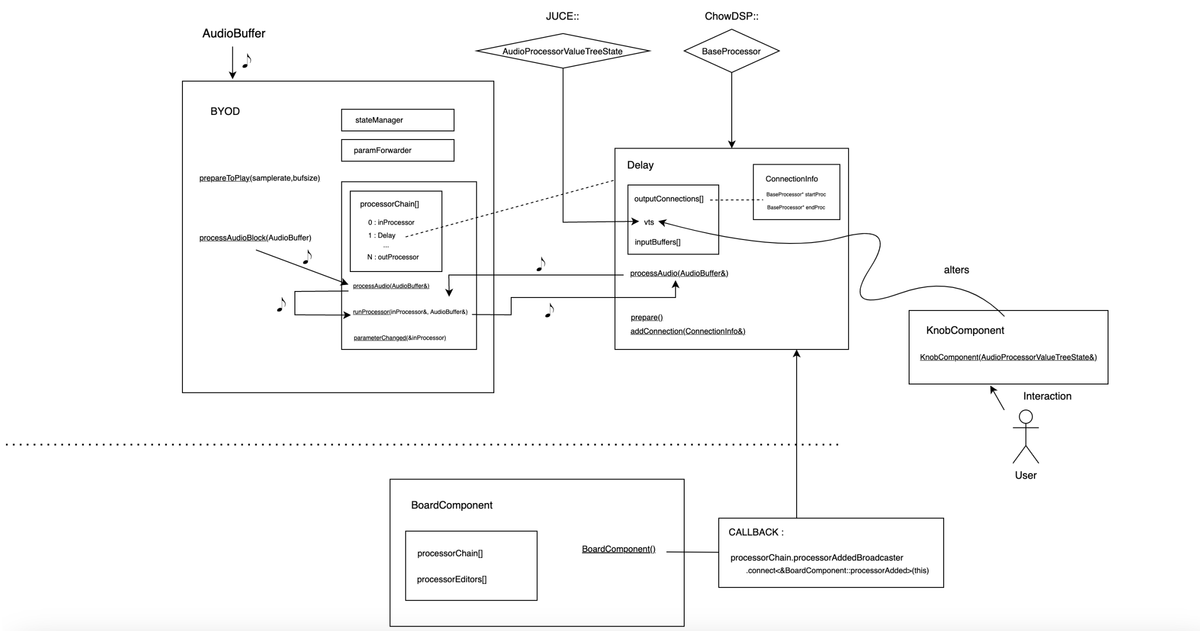
- **Preset Manager**

  An Audio Software with a preset manager feature enables users to save and load their preferred settings for the software's processors. This feature is especially beneficial for musicians looking to access their favorite sounds quickly, without the need to manually adjust each parameter every time they use the software.

- **Processors**

  The BYOD software offers a wide range of processors that can be used to design virtual pedalboards:

  - *Distortion processors* to add overdrive and distortion effects.
  - *Tone/EQ processors* to adjust tonal characteristics by boosting or attenuating specific frequencies.
  - *Compression processors* to control the dynamic range.
  - *Delays and Reverbs* to create spatial effects such as echos.
  - *Chorus/Phaser/Flangers* to create modulation effects.
  - *Visualizers* to visualize the audio signal in real-time.
  - *Signal Mixers* to mix multiple audio sources together into a single output.



BYOD Block Diagram

3 Software Model

BYOD's digital architecture employs open source libraries from ChowDSP, which utilize the JUCE C++ framework, resulting in a complex and modular design. This report will primarily analyze the code of the Processor's Hosting, which serves as the basis for processor modules and enables the implementation of specific effects. We will not delve into the details of the code for building and testing, but instead provide a general overview of the UI builders.

In addition to analyzing the Processor's Hosting code, we will also examine the base class for a processor module and provide examples of how it can be implemented to create specific effects. This will provide a more comprehensive understanding of how the software model is structured and how to implement a custom additional module.

3.1 Processors Hosting

The Software Model for BYOD is based on a modular design, the main `BYOD` class, which is an extension of `JUCE::AudioProcessor`, manages the hosting of individual modules and the software state. The `chowdsp::ProcessorChain` object handles the processor modules, while the `chowdsp::StateManager` object manages the state. Additionally, the BYOD class takes care of handling incoming `JUCE::AudioBuffer` in the `processAudioBlock()` function, which forwards the buffer to the `ProcessorChain` to begin the audio processing.

3.1.1 Processor Add/Delete

In the code, adding and deleting processors follows a callback design pattern. The GUI's `BoardComponent` class stores a reference to the `processorChain` and assigns callback functions to handle user interaction events.

```
callbacks += {procChain.processorAddedBroadcaster
        .connect<&BoardComponent::processorAdded> (this)
```

When a new processor is added to the chain in BYOD, the `processorAdded` method is called by the `Broadcaster` to set it up and create its interface. Moreover, BYOD supports Dry/Wet mixing by using a bypass buffer to store the clean signal, which introduces a delay to match the sample latency of the processed signal.

3.1.2 AudioBuffer Handling

In order to gain a comprehensive understanding of how the signal is transmitted through the processor chain, it is necessary to examine the implementation of the processors located in the `BaseProcessor.cpp` file.

Each processor can handle audio buffers and have references to the following processors in a vector of struct called `ConnectionInfo`. Additionally, it supports several audio inputs in an Array of `JUCE::AudioBuffer` called `inputBuffer`. The `ProcessorChain` manages the signal transmission across the chain, utilizing this information.

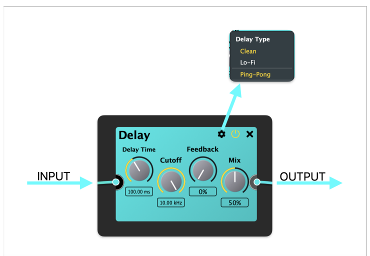
The `runProcessor` method is a recursive function utilized by the `ProcessorChain` to apply processing to the `JUCE::AudioBuffer`. The initial call to this method passes in the first processor in the chain, which is the `inputProcessor`. The function then retrieves the reference to the next processor at each step by calling the `getOutputConnection()` method on the current processor. The recursion is performed as :

```
runProcessor (nextProc, copyNextBuffer, outProcessed);
```

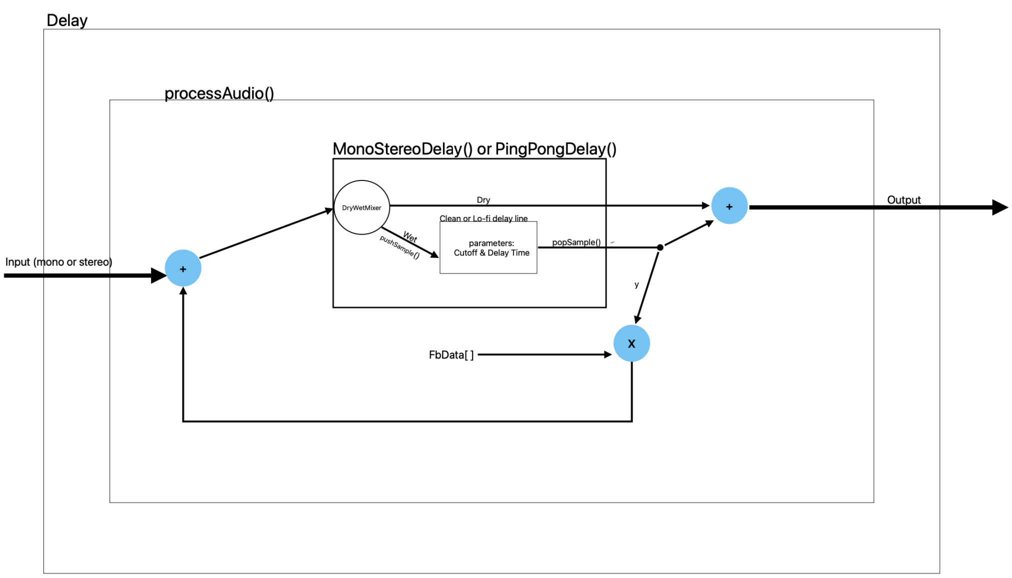
4 Audio Processors

To support different types of audio processors, the BYOD platform requires each processor to extend the `BaseProcessor` class. The implementation of the `createParameterLayout()` method is necessary to define the parameters of the processor. Additionally, it is essential to define the `prepare()` function and integrate the DPS algorithm into the `processAudio()` method.
In this report, we will focus on the implementation of two specific processors: Delay and Blonde Drive.

4.1 Delay



The ‘`Delay`’ module is a plugin effect that offers adjustable parameters for creating a delayed version of the input audio. It also features a lowpass filter with customizable settings, and the option to add feedback by routing the output back to the input. Additionally, the module provides two popup menu parameters, `delayTypeTag` and `pingPongTag`, which allow users to choose between lo-fi or clean mode and enable a ping-pong effect.



Audio Chain Scheme

4.1.1 Implementation

The functionality of the processor is heavily dependent on the `chowdsp::DelayLine` module, which utilizes a circular buffer to perform its function. The circular buffer is a vector that has two pointers - one for writing and one for reading. This buffer enables data to be written and then read at different times, creating a delay effect.
Interpolation is used to calculate the output sample when the delay value is not an integer number of samples. The implementation includes several algorithms for fractional delay calculation, but in this case the option used is the 5th-order Lagrange interpolation.
The feedback mechanism of the DelayLine module involves extracting a sample 'y' from the module's circular buffer and then feeding it back into the same buffer. To prevent the feedback signal from becoming too loud and unstable, the sample 'y' is multiplied by a feedback attenuation gain value contained in an array called 'fbData'.

```
delayLine.setDelay (delaySmooth.getTargetValue());
        delayLine.setFilterFreq (freqSmooth.getTargetValue());

        for (int ch = 0; ch < numChannels; ++ch)
        {
            auto* x = buffer.getWritePointer (ch);
            for (int n = 0; n < numSamples; ++n)
            {
                auto y = delayLine.popSample (ch);
                delayLine.pushSample (ch, x[n] + y * fbData[n]);
                x[n] = y;
            }
        }
```

4.1.2 Ping Pong Delay

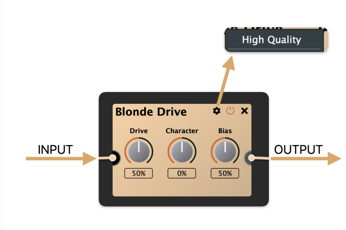
The PingPong Delay option in the processor generates a stereo effect, creating an illusion of the audio bouncing between the left and right channels. It does so by separately applying the delay effect to each channel, with the left channel having a shorter delay time than the right channel. The output of each channel is then combined into a single audio buffer, resulting in the desired stereo effect.

```
auto yL = delayLine.popSample (0);  auto xR = delayLine.popSample (1);

auto xL = dataL[n] + dataR[n] + fbData[n] * yR;
auto xR = fbData[n] * yL;

delayLine.pushSample (0, xL);  delayLine.pushSample (1, xR);
dataL[n] = yL; dataR[n] = yR;
```

4.2 Blonde Drive



The `Blonde Drive` module is based on the Joyo American Sounds circuit, which utilizes an operational amplifier and passive components to generate a warm distortion effect. This distortion is achieved by overdriving a Zener diode circuit, which acts as a voltage limiter. In addition, a State Variable Filter (SVFBell) is applied before the circuit to manipulate the distortion's character. The Zener diode has an asymmetrical characteristic, but this circuit allows a bias signal to be applied to adjust the amount of symmetrical and asymmetrical clipping.

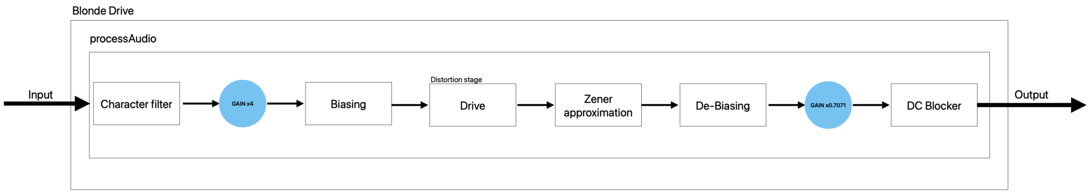
4.2.1 Modeling of the Zener Diode circuit

The Characteristics of the Zener Diode is estimated through the `zeiner_clip_combined()` function :

```
inline auto zener_clip_combined (xsimd::batch<double> x) noexcept
{
    x = mult * x;

    const auto x_abs = xsimd::abs (x);
    const auto x_less_than = x_abs < max_val;
    const auto exp_out = expApprox (beta_exp * -xsimd::abs (x + c));
    const auto y = xsimd::select (x_less_than , x*oneOverMult, chowdsp::Math::sign(x)*(-exp_out+bias)*oneOverMult);
    const auto y_p = xsimd::select (x_less_than , xsimd::broadcast (one), exp_out + betaExpOverMult);

    return std::make_tuple (y, y_p);
}
```



Audio Chain Scheme

The response of the circuit is estimated through the Newton-Rapson algorithm:

```
template <int maxIter = 8>
void processDrive(double* left, double* right, const double* A, xsimd::batch<double>& y0, const int numSamples) noexcept
{
    for (int n = 0; n < numSamples; ++n)
    {
        stereoVec[0] = left[n];
        stereoVec[1] = right[n];
        auto x = xsimd::load_aligned (stereoVec);

        // Newton-Raphson loop
        for (int i = 0; i < maxIter; ++i)
        {
            const auto [zener_f, zener_fp] = Zener::zener_clip_combined (x + A[n] * y0);
            const auto F = zener_f + y0;
            const auto F_p = zener_fp * A[n] + 1.0;
            y0 -= F / F_p;
        }

        xsimd::store_aligned (stereoVec, -y0);
        left[n] = stereoVec[0];
        right[n] = stereoVec[1];
    }
}
```

The number of iterations of the Newton-Raphson method per sample can be chosen by setting the Quality parameter, High Quality corresponds to 12 iterations per sample while by default we have 8 iterations:

```
if (hiQParam->get())
    processDrive<12> (leftData, rightData, driveData, state, numSamples);
else
    processDrive (leftData, rightData, driveData, state, numSamples);
```

5 Conclusions

Analyzing a sophisticated software system that is modular and built on a framework can be challenging. As a result, our report did not delve deeply into the implementation strategies employed in the ChowDSP library for managing data (such as vectors and smooth parameter buffers) and for managing plugin state and presets. We only provided a cursory overview of certain classes within the ChowDSP library that handle these tasks.