Programming II - 7CTA1130-0905-2020

***Ashley Alphonso* – 14167168**

1.Introduction

The purpose of this task was to create an AU/VST3 plugin using the JUCE library and Projucer application.

The plugin designed in this report was a distortion filter. Made combining a ladder filter from the JUCE dsp module and some nonlinear processing functions.

2.Method

1. Implementing JUCE dsp ladder filter
2. Adding distortion dsp code
3. Creating Parameters
4. Creating Sliders and customising GUI

2.1 Implementing filter

JUCE::dsp::LadderFilter<foat>

```
//=========Creating DSP Objects================
juce::dsp::LadderFilter<float> moogFilter;
juce::dsp::Gain<float> driveGain;
```

Figure 1: Creating Ladder Filter in pluginProcessor.h

There is a lot of precursor work that needs to be done before using the filter from the dsp module. It needs a process spec, and audioBlock and a context replacing object.

The Process spec is a dsp class that creates an object that can hold information about the audio data that will be processed by the dsp algorithm.

```
{
}

//==============================================================================
void ProgrammingIIAudioProcessor::prepareToPlay (double sampleRate, int samplesPerBlock)
{

    //====Creating and populating process spec=======
    juce::dsp::ProcessSpec spec;
    spec.maximumBlockSize = samplesPerBlock;
    spec.sampleRate = sampleRate;
    spec.numChannels = getTotalNumOutputChannels();

    //=========Reset and preparing filter=============
    moogFilter.reset();
    moogFilter.prepare(spec);

    //============Reset and preparing Gain===========
    driveGain.reset();
    driveGain.prepare(spec);
    driveGain.setRampDurationSeconds(0.1);
    //if (
    //moogFilter.setMode(juce::dsp::LadderFilterMode::BPF24);
```

Figure 2: Creating and using Process Spec

Dsp objects use the information curated by the spec via the prepare function.

The final steps in using the dsp algorithms is in creating an audio block and a context object.

```
auto audioBlock = juce::dsp::AudioBlock<float>(buffer);
auto context = juce::dsp::ProcessContextReplacing<float> (audioBlock);

moogFilter.process(context);
driveGain.process(context);
```

Figure 3: Creating Audio block and Context Replacing

The audio block behaves as the audiobuffer for the DSP filter and the process context replacing behaves like a buffer write pointer, which replaces the original signal with the processed signal from the filter.

2.2 Nonlinear Processing Function

The Hyperbolic tangent function was adapted I to create distortion to the samples.

Name	Acronym	Equation (x = in, y = out, k = saturation)	Notes
Arraya	ARRY	$y=\frac{3x}{2}\left(1-\frac{x^2}{3}\right)$	No saturation, very mild
Sigmoid	SIG	$y=2\frac{1}{1+e^{-kx}}-1$	
Sigmoid2	SIG2	$y=\frac{(e^x-1)(e+1)}{(e^x+1)(e-1)}$	No saturation, very mild
Hyperbolic Tangent	TANH	$y=\frac{\tanh(kx)}{\tanh(k)}$	Good for diode simulation
Arctangent	ATAN	$y=\frac{\arctan(kx)}{\arctan(k)}$	
Fuzz Exponential 1	FEXP1	$y=\mathrm{sgn}(x)\frac{1-e^{-\lvert kx\rvert}}{1-e^{-k}}$	

Figure 4: Nonlinear function source. (Pirkle, 2019)

```
for (int channel = 0; channel < totalNumInputChannels; ++channel)
{
    auto* channelData = buffer.getWritePointer (channel);

    for (int sample = 0; sample< buffer.getNumSamples(); sample++)
    {
        float headroom = 0.5;
      // channelData[sample]=channelData[sample]/(1+fabs(channelData[sample]));
        channelData[sample]= tanh(headroom * channelData[sample]) / tanh(headroom);
    }
}
}
```

Figure 5: Equation in code form

2.3 Creating Parameters

Using the JUCE class audio processor value tree state. A value tree of the plugins parameters were created.

```
juce::AudioProcessorValueTreeState::ParameterLayout ProgrammingIIAudioProcessor::createParameters()
{
    juce::AudioProcessorValueTreeState::ParameterLayout params;

    params.add(std::make_unique<juce::AudioParameterInt>("cFreq",
                                                          "cutOffFrequency",
                                                          130,
                                                          22000,
                                                          500));
    params.add(std::make_unique<juce::AudioParameterFloat>("Q",
                                                            "Resonance",
                                                            0.0f,
                                                            1.0f,
                                                            0.0f));
    params.add(std::make_unique<juce::AudioParameterFloat>("DRIVE",
                                                            "Drive",
                                                            0.0f,
                                                            24.0f,
                                                            0.0f));
    params.add(std::make_unique<juce::AudioParameterFloat>("OUTPUT",
                                                            "Output",
                                                            0.0f,
                                                            100.0f,
                                                            50.0f));
```

Figure 6: Parameter Value tree

Creating this class allows you to link parameters with their sliders in the pluginEditor. The class has slider and button attachments.

```
*/
class ProgrammingIIAudioProcessorEditor  : public juce::AudioProcessorEditor
{
public:
    ProgrammingIIAudioProcessorEditor (ProgrammingIIAudioProcessor&, juce::AudioProcessorValueTreeState&);
    ~ProgrammingIIAudioProcessorEditor() override;

    typedef juce::AudioProcessorValueTreeState::SliderAttachment SliderAttachment;
    typedef juce::AudioProcessorValueTreeState::ButtonAttachment ButtonAttachment;
    typedef juce::AudioProcessorValueTreeState::ComboBoxAttachment BoxAttachment;
```

Figure 7:multiple valuetree attachments

2.4 GUI Design

After creating and positioning the sliders I created a user interface class to manipulate the look and feel functions for the sliders. I downloaded a texture off of the internet and used it as the background of the plugin.

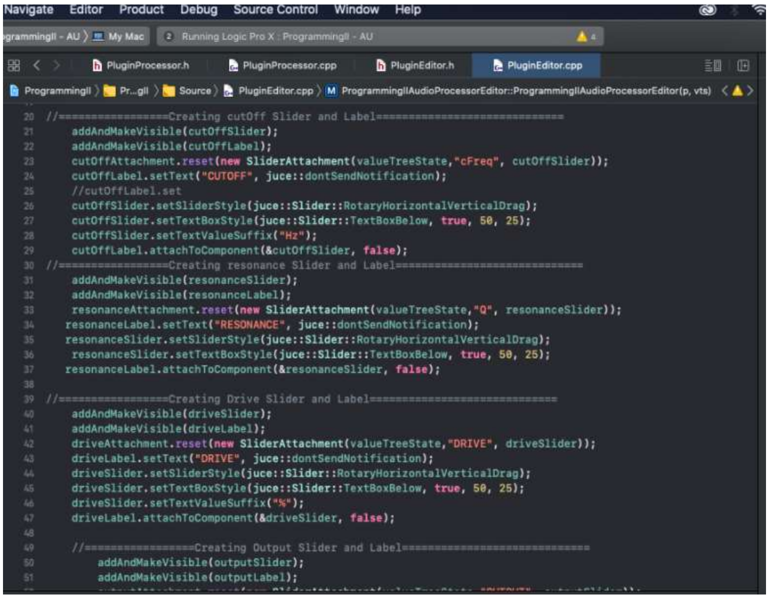

Figure 8: Creating and labelling Components

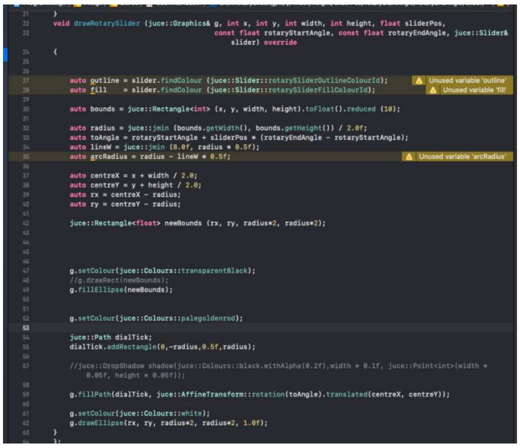

Figure 9:UserInterface class with customiised look and feel fuction

Figure 10: Paint function and background (Wallpaper.dog,2020)

3. Results

The plugin functions on multiple platforms. The pre sets menu would have been a modern addition to the plugin as many plugins today have a selection of presets accessible from the UI of the plugin. The look and feel of the buttons and combo box could have been redesigned to ft the entire aesthetic of the rest of the plugin.

Bibliography

Antonia, D. (2020) Audio Programming Lecture. [Online] 23rd February 2021. Available from: https://uh.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=690b3a1d-eb97-49e5-8d1a-acd8014712ca [Accessed on 13th May 2021]

Pirkle, W. (2019) Chapter 19: Nonlinear Processing: Distortion, Tube Simulation, and HF Exciters. In: Pirkle, W. *Designing Audio Effect Plugins in C++ For AAX, AU, and VST3 with DSP Theory* – second Edition. New York: Routledge.

The Audio Programmer (2020) JUCE 6 Tutorial 10 – State Variable Filter and the DSP Module [YouTube] home. Available from: https://www.youtube.com/watch?v=CONdIj-7rHU&list=PLLgJJsrdwhPyNsICl0_gSGF7owlow_cfA&index=14 [Accessed on: 10th May 2021]

The Audio Programmer (2020) Juce Tutorial 06- Customizing Dials and Sliders Using The Look And Feel Class (Pt 1 of 2) [YouTube] home. Available from: https://www.youtube.com/watch?v=po46y8UKPOY [Accessed on: 10th May 2021]

Juce.com (n.d) JUCE: Tutorial: Customise he look and feel of your app. [Online] Available from: https://docs.juce.com/master/tutorial_look_and_feel_customisation.html [Accessed on 18th May 2021]

Wallpaper.dog (n.d) WallpaperDog-17248257.jpg [Online] Available from: https://wallpaper.dog/black-metal-texture-wallpapers [Accessed on 18th May 2021]