Audio Programming in Creating Audio Plugins

by

Agustin Aguilera

Submitted to the Department of Mathematics and Computer Science

School of SUNY Purchase

in partial fulfillment of the requirements

for the degree of Bachelor of Arts

Purchase College

State University of New York

May 2022

Sponsor: Irina Shablinsky

Second Reader: Knarik Tunyan

Acknowledgements

I would like to thank my mentor, Dr. David Jameson for his extremely well taught lessons, I would not have been able to do this without his charisma and enthusiasm pushing me to go past my limitations. I would also like to thank Irina Shablinsky for taking me on as her student and also helping me in my endeavors. Finally, I would like to thank Professor Knarik Tunyan for being a part of my successful experience working on my thesis and its spectacular results.

Table of Contents

Abstract

Not too many computer science students delve too deep in audio programming, possibly because it doesn't have a huge background in contrast to software engineering, game design or game development, or even web programming. However, audio programming can be related to all of these types of programming since in almost every category, audio is a very useful tool for the user of any program to feel attached and engaged with the content. For this thesis, I have constructed an essay to describe my experiences building an audio plugin and witnessing firsthand the difficulties of audio programming. In this paper, I explain the process of building an audio plugin, a tool that is used inside software that artists use to create music. This tool allows the user to add, distort, and amplify sound, as well as use samples of sound, for example, a drum loop, the sound of drums playing over and over again. In my thesis, I detail every one of the components that make up my audio plugin and also its practical uses.

Keywords: DAW, VST, IDE, decibel, ADSR

1. Introduction

Audio plugins are software components that can add or enhance audio-based functionalities in a computer program. They use digital signal processing as a form of imitating studio equipment inside a Digital Audio Workspace, or the DAW. The DAW is a program that enables the user to record audio on a computer. It allows the user to record and edit audio, play virtual instruments, such as pianos, guitars, trumpets, etc., and much more. The audio plugins serve as a tool to change the way the sound in the DAW is heard, in accordance with how the user decides to change it.

My interest in learning audio programming and being able to build my own audio plugin comes from my love for melodies. As someone who has used plugins before, I can fully understand the different components that my plugin needs and the different functions they must have in order to be user friendly. My plugin is able to help me create beautiful melodies in my DAW, and will hopefully help others do the same, perhaps even do much more. The aim of this paper is to understand how an audio plugin can enhance the sound quality of audio files. In order to get the best results, I have constructed an audio plugin that consists of features such as, gain control, a sampler. I also included a couple of sliders that change the amplitude values of the sound called, attack, sustain, decay, and release, or ASDR as an abbreviation.

1.1 JUCE: The Skeleton of Audio Programming

JUCE is software available to users as a tool for creating audio applications for desktop and mobile devices. Most importantly, JUCE supports the development of VST,

AAX, and AU plugins. These three listed refer to the format of the audio plugin, VST for example, refers to Visual Studio Technology and is the most used format since it is more widely supported by various DAWs. Whereas AU, or Audio Unit, which was created by Apple to be used only for their products, is only supported by a handful of DAWs and really only used with Apple's ecosystem. Since this project was done on a Windows operating system, the format chosen was VST3, the latest version of VST. JUCE uses a software program called the Projucer, which is JUCE's project management tool and basically allows the user to create either a basic audio plugin, a GUI audio application, a 3D rendering audio application, or even an animated audio application. However, the Projucer requires the use of an external IDE, or Integrated Development Environment, such as Visual Studio 2019, which is the one used in this project, in order to run and debug the audio plugin. In the Projucer, I am able to create GUI components and style them however I choose and then code functionality for those components in Visual Studio. These tools are quite handy for creating plugins and make every audio programmer's design innovative and aesthetically pleasing.

1.2 Gain Control

To fully understand what gain is, let me first introduce audio input and output. Audio input refers to any sound that a machine, such as, a computer, an amplifier, basically anything that can listen for a signal. The audio output, in contrast, produces sound and can amplify, and change the sound, truly anything can be done when sound is outputted. Gain can sometimes be confused with volume; however, they are not the same. Volume is the decibel (dB) output of a system; this basically means how loud the

sound is outputted. Whereas gain is the input level of the audio signal, meaning that it consists of making changes to the audio before it gets processed, which will change the tone of the audio. This is important because whether the volume is high after processing does not change the tone of the audio which is the goal of gain control. Only by changing the volume before processing, does the tone change and create a new sound outputted through the sound system.

1.3 Attack

The attack of the tone of any sound is best described as the increase in amplitude over the time it takes to reach its maximum level. In short, attack is the length of time measured for the sound to reach from an amplitude of 0 to an amplitude of 100 percent. The user can change the attack level so that the highest amplitude of the sound reaches a certain point. This will change the frequency of the sound as the attack bar is shifted.

1.4 Decay

If the attack is the increase in amplitude, then the decay is the decrease in amplitude over time. Unlike attack, however, decay decreases its amplitude until it reaches the sustain stage. It starts at the maximum amplitude level, the attack level, and falls to the sustain level, where the decay ends. In other words, decay is best defined as that bar in which the frequency of the sound decreases from the highest amplitude percentage to a constant level that is set by the user.

1.5 Sustain

Imagine a piano and whenever a key is pressed for a certain duration, the length of time the sound is played is what we refer to as sustain. Sustain controls how long a note will sound for a constant length of time, which means the amplitude remains the same and the user can play with the tone of the sound as much as they would like. The benefit to sustain is the freedom of placing the sustain bar at whichever frequency level the user prefers, thus changing the overall texture of the tone of the sound.

1.6 Release

The release is the moment the sustain bar ends and the wavelength or the amplitude decreases to an absolute zero percent amplitude. This happens as soon as the key that was pressed on the piano key, as seen in the example above, has been released. The sound that is heard can be best described as a fading sound where the note plays softly into silence. This allows music to have an ambient sound when playing and also allows the notes that are played to give off a more natural tone rather than ending instantly. For a complete visualization of the ADSR relationship see Figure 1.

1.7 Sampler

A sampler is a virtual tool that stores a sound file, for example, a mp3, aif, wav, etc., all files that contain some form of sound and once stored, the file gets converted into MIDI notes for the user to play on their virtual instrument, such as a keyboard. MIDI stands for Musical Instrument Digital Interface and it allows for devices such as a piano

keyboard to connect to a computer. It then sends out these messages to the computer which receives and records them in some software, in this case, the DAW. Thus, any time an action is performed on the keyboard, the sound is actually sent to the computer and gets output through the speakers instead of the actual keyboard. It is important to note that when the user picks a sound file that is then stored in the sampler, that sound will be set to its pre-established tonal center, the key that sounds clear to our ear, which is normally set to the key of C4. However, the user is able to change the pitch of the sound by playing the note in a different key.

2. Literature Overview

The book I read to familiarize myself with the programming language C++ is called *C++ Primer* [1]. In this book, I learned new things I hadn't already learned from other programming languages such as Java, JavaScript, or Python. Since I am completely new to pointers and references, I hadn't any clue as to what they were nor what they did. The first thing I found to be new to me was the idea of pointers which are variables that can hold the address of an object or zero, which simply points nowhere. A reference is the name of an already existing variable or object and is used to assign values to pointers. Additionally, references can be used as aliases, which means that you can have different expressions or symbols pointing to the same object. Pointers in C++ are denoted by an asterisk '*', whereas a reference is denoted by an ampersand '&'. Since in my project, I was working with vectors to create a sort of wavelength of a sound file, I really paid attention to the iterator arithmetic. Iterator arithmetic are operations on vector or string iterators such as adding or subtracting an integral value

which an iterator yields an iterator that is many elements ahead of or behind the original iterator. Subtracting one iterator from another yields the distance between them. Iterators must refer to elements in, or off the end of the same container, or class of vectors. Another crucial implementation of a C++ element that I took from the Primer is dynamic_cast, it is used in combination with inheritance and run-time type identification. In C++, dynamic casting is mainly used for casting a base class pointer or a reference to a derived class pointer/reference. This has allowed me to make some necessary conversions of objects that were pointed at different types. Finally, perhaps the most important element I had to learn and use for my project was a lambda expression. A lambda expression is a callable unit of code. A lambda is somewhat like an unnamed, inline function. A lambda starts with a capture list, which allows the lambda to access variables in the enclosing function. Like a function, it has a possibly empty parameter list, a return type, and a function body. A lambda can omit the return type. If the function body is a single return statement, the return type is inferred from the type of object that is returned. Otherwise, an omitted return type defaults to void. In my project, I use a lambda expression to assign a function to a button, specifically, the sampler button. The lambda expression allows for the audio processor to load a file when the button is pressed.

3. Process and Results

To begin, when I created my JUCE project, there were four types of files that were used to create the actual audio plugin. That is, the PluginEditor.h, the PluginEditor.cpp, the PluginProcessor.h, and the PluginProcessor.cpp. Of course, there

are more files that can be added onto the project, for example, one could create a component file, where all the user does is create GUI components. Within the PluginEditor.h file, I began declaring all the variables that I was going to need. Sliders are bars whose values are changed by the user, like a volume bar, or a Youtube video time-lapse bar. To start, in the private class, I declared my Slider class variables, these are objects that are inherited from the Slider::Listener class, this means that the audio processor will 'listen' or rather check for a Slider, and then it is up to me to tell it what kind of Slider it is, what the Slider does, etc. In my case, I had various Sliders where one changed the gain of the sound being played and the others changed the amplitude and the frequency of the sound waves. Then I declared my Labels for the Sliders using the Label class. I had a TextButton which does exactly what it sounds like, it creates a button that contains text, in this case the sampler button, the text reads, "Pick a file". Then, I declared a vector of type float, this would be what draws the wavelength of the sound when it is loaded into the plugin. Finally, I added a boolean type variable called mShouldBePainting and set it to false, this will draw the wavelength when true. See Figure 2.

In the public class, JUCE is really efficient and creates a constructor and destructor with built in functions like the paint() function and the resized() function. The paint() function basically draws the canvas for the audio plugin's UI, and the resized() function essentially updates the canvas, redrawing any changes in the audio and graphics. Next, I declare a Boolean function, isInterestedInFileDrag, that is already created by JUCE, however, I have to define it to overwrite the method. The method takes a reference of a StringArray type and this function will allow the user to not only

access a file via the file explorer but also drag and drop the file into the plugin. The other function I declare is of course the filesDropped() function, this method also takes in a reference to a StringArray type and an x and y integer value. It allows for the user to click the “Pick a file” button and choose a file from the file explorer. Finally, I declare the sliderValueChanged() function, this will be the most important function; it decides what happens to the sound file once the user has applied a change to each of the Sliders. This function takes a Slider pointer type as a parameter and it is a pointer because the actual Sliders are declared in the private class and I need to point to them in the public class. See Figure 3.

In the PluginEditor.cpp, I begin with a lambda expression that returns the loadFile() function I created earlier whenever the button is clicked. I then make it visible in the audio plugin with the addAndMakeVisible() function from JUCE. The labels and the sliders are then given values such as their SliderStyle, the gain slider is a vertical bar whereas the ADSR sliders are all rotary sliders. They are given color, font, and other values that give them a bit of aesthetic. Note I only included the Attack Slider in Figure 4, this is because the Decay, Sustain, and Release Sliders are written in the same format. See Figure 4.

The paint function is initialized in the PluginEditor.cpp file as well and it contains a couple of interesting elements. Primarily the wavelength painting whenever the sound file is dragged, if shouldBePainting is true, then the plugin creates a Path, which is similar to a path in processing. The Path gets drawn by taking the vector points of the wavelength from the audio sound file. The other event that happens in the paint()

function is the actual painting of the plugin's GUI, where I set the color of the background, the text, the font, etc. See Figure 5.

The resized() function essentially sets the bounds of all the components so that all elements are relative to the size of the window, in other words they are locked in place so that there is no way to expand the window. Next I implement the sliderValueChanged() function which points to those sliders I created in the private class of the PluginEditor.h file. Earlier I mentioned that gain is not the same as volume. If I had made the gain be multiplied by some number so that when the user increases the value of the gain slider, the gain increases linearly, then the sound would be too powerful when increased and could damage the eardrums. This is because sound is not perceived linearly but logarithmically, therefore, I instead made the gain be the equation, $N_{dB} = 20\log_{10}\left(\frac{I_2}{I_1}\right)$, where the $\frac{I_2}{I_1}$ is represented by the midiVolume and the rawVolume respectively. The rest of the sliders are all updated using a method named updateADSR(). See Figure 6 and 7.

The next two functions I initialized were the filesDropped() and the isInterestedInFileDrag() methods. The isInterestedInFileDrag() method checks to see if the file picked has an extension of either a .wav, .mp3, or a .aif, and if it does, it returns true. The filesDropped checks to see if the file is a sound file, that it has one of the previously mentioned extensions, and sets the boolean mShouldBePainting to true, then it loads the file into the audio processor. Finally, it updates the plugin by running the repaint() method. See Figure 8.

In the PluginProcessor.h file, I declare all the necessary variables and functions in the public class, in this case I declare the rawVolume float type variable, this is the

volume the audio processor initially registers in the beginning. I declared the loadFile() functions, one has no parameters and the other accepts a String reference as a parameter. The getADSRParams() function is a really useful method because JUCE actually has pre-set variables for the ADSR values. This will allow for the ADSR sliders to get the value directly from the getADSRParams() function. The getNumSamplerSounds() method uses a function from JUCE that returns the number of sounds that have been added to the sampler. The AudioBuffer creates an empty buffer with 0 channels and 0 length. The function getWaveForm() returns the vector path of mWaveForm that is created when the user drags a sound file into the sampler. See Figure 9.

In the private class, the first object declared is the Synthesiser type object called mSampler, this creates our Sampler which in JUCE is referred to as a Synthesiser. Since no parameters have been set, the sampler is currently empty until the voices and sounds from the addVoice() and addSound() methods are added to it. We create a constant integer type variable that tells the sampler how many voices are in the channel. Next, the AudioBuffer of type float called mWaveForm is declared, previously I mentioned that the buffer is empty and here I only declare it so it at this point, the AudioBuffer is still empty. The next two objects declared are JUCE audio managers and readers that will take the audio and return the format of the sound file, for example, a wav file will return .wav. See Figure 10.

Finally, in the PluginProcessor.cpp file, I write a for() loop that adds the voices that get added to the sampler and also make sure that the format manager for the sampler returns the file extension to the sampler to ensure the file is indeed a sound file.

Remember that a voice is essentially something that plays a single sound at a time and a synthesiser, or our sampler to be more precise, holds an array of voices so that it can play polyphonically. See Figure 11.

The next function I initialize is the loadFile() method. This function takes in a reference to a String type object I named path as this will be the path to the file we are trying to load. The first step in this method is to clear all the sounds before we load the sound into the sampler, this is in the case there is already a sound loaded into the sampler. This way it doesn't load our sound file onto another audio file already placed within the plugin. Once that is done, I create a File object that is made from the path called in the parameters of the function. The format manager takes the file and then it gets pointed to the format reader. This way, the format reader can receive the length of the samples and get casted as an integer. Now that we have the sample length, the waveform can finally take shape. I also create an if statement to add sounds whenever the format reader reads a sound file, in short whenever a sound file is dragged into the sampler, the sound will get added to the sampler's memory. See Figure 12.

The last method implemented is the updateADSR() method. This is the function that sets the ADSR values to the envelope. An envelope is the box, or rather the boundary that the attack, decay, sustain, and the release all work in (see Figure 1). First, I create a for() loop to get all the sounds from the sampler and then I specify the parameters of the Envelope to be that of the mADSRParams we set earlier. Now, whenever the user makes any changes to the ADSR slider values, the change is reflected by the updateADSR() function. See Figure 13.

4. Discussion

Creating this audio plugin has been a real pleasure and it has brought me tons of new and interesting experiences I would have never had the chance to explore this topic if it wasn't for the support of my advisors and peers encouraging me in this long process. The beginning was a rough and bumpy road but it was all well worth it as I ended up not only succeeding in creating an audio plugin but also finding answers to questions that came to mind along the way. Questions such as the importance of audio plugins to musicians and music creators, as well as questions like how difficult it is to learn audio programming, and is it worth pursuing. The answers to these questions I have realized came with time and effort. The truth is that audio programming is not studied enough by people studying computer science and general programming. Yet, I feel that the best plugins are created by companies full of talented individuals, all of whom give one hundred percent to polish and shine their plugins. My plugin came with many problems and tons of troubleshooting on my end. The biggest obstacle I came across was implementing the sampler into the plugin since I had no idea how to create the Synthesiser, SamplerSound, and SamplerVoice classes. The Synthesiser class is the base class for playing notes, the SamplerSound class represents sampled audio clips, and the SamplerVoice class holds the number of voices the sampler can play. However, through the JUCE documentation, I was able to construct the sampler, yet there was still the part of actually loading an audio file into the sampler which gave me tons of headaches. This part was not easy and in no way could I have achieved a successful sampler in my plugin if it was not for the aid of other audio programmers from the JUCE community. These programmers are very professional and have offered

me tons of support for my project. Thanks to them, I was able to figure out how to create the loadFile() function that enables the user to load an audio file into the sampler inside the plugin. After having made the sampler, however, I came across a major hiccup that prevented me from using both the gain slider and the sampler at the same time. I fixed this later on by changing a couple of settings in my JUCE file in the Projucer. Specifically, the Plugin Characteristics in the plugin settings, the Plugin is a Synth and the Plugin MIDI Input settings must be checked, otherwise the project won't operate properly. In my case, the sampler would not work but the gain slider would function properly and after having checked these two settings, the sampler, the gain control, and the ADSR sliders operated in perfect symphony. The debugging process was a bit of a challenge as well. While I had hoped to be able to use my preferred DAW which is called FL Studios, unfortunately this DAW needs to have a full paid edition to make full use of my plugin. Therefore, I looked at other DAWs such as Ableton Live, however I also encountered issues in Ableton where my plugin would load an audio file but it would not play any sound. I checked with my debugging tool in Visual Studios that allows me to use the plugin without the need of a DAW and there were no issues with the actual plugin. This may have been an issue related to Ableton Live and my inexperience with the software, however, I chose to find another DAW to test my plugin in. Luckily, Dr. Jameson had provided me with his very own DAW, one he co-founded called Gig Performer 4, this DAW is very difficult to use but thankfully Dr. Jameson explained in detail how to make it work with my plugin. The phrase 'all's well ends well,' couldn't be truer, my plugin was able to work perfectly as it should in Gig Performer and my testing was very successful. See Figure 14 to take a look at what the plugin looks

like after I completed my audio plugin. See Appendix 14 for the image of the actual plugin and see Figure 15 to find the link for the short video demonstration of the plugin I created.

5. Conclusion

Creating an audio plugin from scratch was no easy task but my love and admiration for audio programming pushed me to my limits in working on an amazing senior project. The time I dedicated to this project, I believe was well spent, perhaps not managed quite as efficiently as others, but learning to manage one's time is part of the experience. I also had an amazing time learning of the various different uses that audio plugins have in the music industry and in the recording studio, especially among big studio companies and producers. I sincerely hope that I can continue to work on this project, even after graduation, to sharpen my audio programming skills. It is also a goal of mine to make more students aware, or at least for them to take interest in the topic of audio programming. Very few audio programmers actually share their experiences with others, however through searching around, I was able to find a whole community full of audio programmers skilled enough to show off their skills. It is these very programmers that shape the music creation in today's industry of musical production. I only hope to be able to follow along the same path and perhaps pursue a career in audio programming as well as production.

6. Appendix

1.

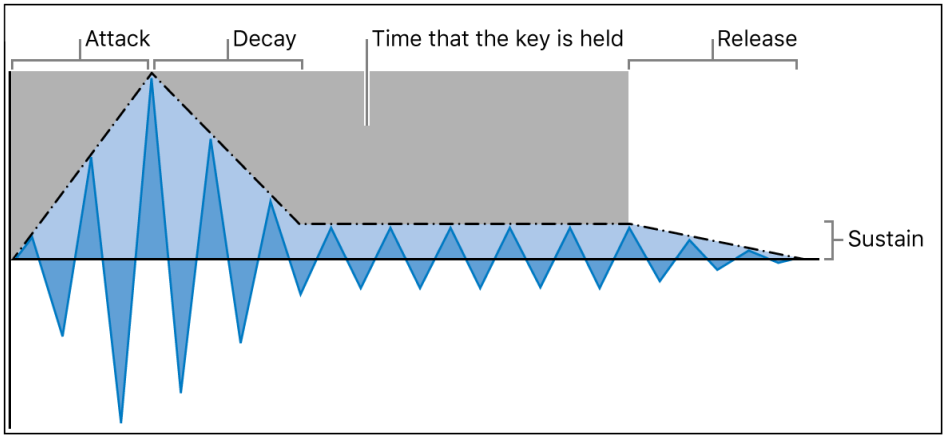


Figure 1. ADSR Representation (retrieved from https://support.apple.com/guide/logicpro/attack-decay-sustain-and-release-lgsife419620/mac)

2.

```
private:
    // This reference is provided as a quick way for your editor to
    // access the processor object that created it.
    ThirdExperimentAudioProcessor& audioProcessor;

    juce::Slider midiVolume;
    juce::Slider mAttackSlider, mDecaySlider, mSustainSlider, mReleaseSlider;

    juce::Label midiVolumeLabel, mAttackLabel, mDecayLabel, mSustainLabel, mReleaseLabel;

    juce::TextButton mLoadButton{ "Pick File From Folder" };

    std::vector<float> mAudioPoints;
    bool mShouldBePainting{ false };

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (ThirdExperimentAudioProcessorEditor)
```

Figure 2. Code Snippet of PluginEditor.h Private Class

3.

```
#pragma once

#include <JuceHeader.h>
#include "PluginProcessor.h"

//==============================================================================
/**
*/
class ThirdExperimentAudioProcessorEditor  : public juce::AudioProcessorEditor,
                                             public juce::Slider::Listener,
                                             public juce::FileDragAndDropTarget
{
public:
    ThirdExperimentAudioProcessorEditor (ThirdExperimentAudioProcessor&);
    ~ThirdExperimentAudioProcessorEditor() override;

    //==============================================================================
    void paint (juce::Graphics&) override;
    void resized() override;

    bool isInterestedInFileDrag(const juce::StringArray& files) override;
    void filesDropped(const juce::StringArray& files, int x, int y) override;

    void sliderValueChanged(juce::Slider* slider) override;
```

Figure 3. Code Snippet of PluginEditor.h Public Class

4.

```
#include "PluginProcessor.h"
#include "PluginEditor.h"

//==============================================================================
ThirdExperimentAudioProcessorEditor::ThirdExperimentAudioProcessorEditor (ThirdExperimentAudioProcessor& p)
    : AudioProcessorEditor (&p), audioProcessor (p)
{
    // Make sure that before the constructor has finished, you've set the
    // editor's size to whatever you need it to be.
    mLoadButton.onClick = [&]() { audioProcessor.loadFile(); };
    addAndMakeVisible(mLoadButton);

    midiVolume.setSliderStyle(juce::Slider::SliderStyle::LinearVertical);
    midiVolume.setTextBoxStyle(juce::Slider::NoTextBox, true, 50, 25);
    midiVolume.setRange(-10.0, 0.0);
    midiVolume.setValue(-10.0);

    midiVolume.setPopupDisplayEnabled(true, false, this);
    midiVolume.setTextValueSuffix(" dB");
    midiVolume.setColour(juce::Slider::trackColourId, juce::Colours::deepskyblue);

    midiVolume.addListener(this);
    addAndMakeVisible(midiVolume);

    midiVolumeLabel.setFont(15.0f);
    midiVolumeLabel.setText("Gain", juce::NotificationType::dontSendNotification);
    midiVolumeLabel.setJustificationType(juce::Justification::centredTop);
    midiVolumeLabel.attachToComponent(&midiVolume, false);

    //Attack Slider
    mAttackSlider.setSliderStyle(juce::Slider::SliderStyle::RotaryHorizontalDrag);
    mAttackSlider.setTextBoxStyle(juce::Slider::TextBoxBelow, true, 40, 20);
    mAttackSlider.setRange(0.0f, 5.0f, 0.01f);
    mAttackSlider.setColour(juce::Slider::ColourIds::rotarySliderFillColourId, juce::Colours::deepskyblue);
    mAttackSlider.addListener(this);
    addAndMakeVisible(mAttackSlider);

    mAttackLabel.setFont(10.0f);
    mAttackLabel.setText("Attack", juce::NotificationType::dontSendNotification);
    mAttackLabel.setJustificationType(juce::Justification::centredTop);
    mAttackLabel.attachToComponent(&mAttackSlider, false);
```

Figure 4. Snippet of Code from PluginEditor.cpp

5.

```
//==============================================================================
void ThirdExperimentAudioProcessorEditor::paint (juce::Graphics& g)
{
    // (Our component is opaque, so we must completely fill the background with a solid colour)
    g.fillAll(juce::Colours::darkgrey);
    g.setColour(juce::Colours::black);
    g.setFont(15.0f);
    g.drawFittedText("Zambayon", 0, 0, getWidth(), 30, juce::Justification::centred, 1);

    if (mShouldBePainting)
    {
        juce::Path p;
        mAudioPoints.clear();

        auto waveform = audioProcessor.getWaveForm();
        auto ratio = waveform.getNumSamples() / getWidth();
        auto buffer = waveform.getReadPointer(0);

        for (int sample = 0; sample < waveform.getNumSamples(); sample += ratio)
        {
            mAudioPoints.push_back(buffer[sample]);
        }
        p.startNewSubPath(getWidth() / 2, getHeight() / 2 + 50);

        for (int sample = 0; sample < mAudioPoints.size(); ++sample)
        {
            auto point = juce::jmap<float>(mAudioPoints[sample], -1.0f, 1.0f, 200, 0);
            p.lineTo(sample, point);
        }

        g.strokePath(p, juce::PathStrokeType(2));
        mShouldBePainting = false;
    }
}
```

Figure 5. Snippet of Code from PluginEditor.cpp

6.

```
void ThirdExperimentAudioProcessorEditor::resized()
{
    const int border = 250;
    const int width = getWidth() / 2;
    const int height = getHeight() / 2;

    const auto startX = 0.3f;
    const auto startY = 0.7f;
    const auto dialWidth = 0.1f;
    const auto dialHeight = 0.2f;

    midiVolume.setBounds(width - border, height - 145, 50, 300);

    mLoadButton.setBounds(width - 50, height - 100, 100, 100);

    mAttackSlider.setBoundsRelative(startX, startY, dialWidth, dialHeight);
    mDecaySlider.setBoundsRelative(startX + dialWidth, startY, dialWidth, dialHeight);
    mSustainSlider.setBoundsRelative(startX + (dialWidth * 2), startY, dialWidth, dialHeight);
    mReleaseSlider.setBoundsRelative(startX + (dialWidth * 3), startY, dialWidth, dialHeight);

}
```

Figure 6. Snippet of resized() Function

7.

```
void ThirdExperimentAudioProcessorEditor::sliderValueChanged(juce::Slider* slider)
{
    if (slider == &midiVolume) {
        audioProcessor.rawVolume = pow(10, midiVolume.getValue()/20);
    }
    else if (slider == &mAttackSlider)
    {
        audioProcessor.getADSRParams().attack = mAttackSlider.getValue();
    }
    else if (slider == &mDecaySlider)
    {
        audioProcessor.getADSRParams().decay = mDecaySlider.getValue();
    }
    else if (slider == &mSustainSlider)
    {
        audioProcessor.getADSRParams().sustain = mSustainSlider.getValue();
    }
    else if (slider == &mReleaseSlider)
    {
        audioProcessor.getADSRParams().release = mReleaseSlider.getValue();
    }
    audioProcessor.updateADSR();
}
```

Figure 7. Snippet of sliderValueChanged() Function

8.

```
bool ThirdExperimentAudioProcessorEditor::isInterestedInFileDrag(const juce::StringArray& files)
{
    for (auto file : files)
    {
        if (file.contains(".wav") || file.contains(".mp3") || file.contains(".aif"))
        {
            return true;
        }
    }
    return false;
}

void ThirdExperimentAudioProcessorEditor::filesDropped(const juce::StringArray& files, int x, int y)
{
    for (auto file : files)
    {
        if (isInterestedInFileDrag(file))
        {
            mShouldBePainting = true;
            audioProcessor.loadFile(file);
        }
    }
   repaint();
}
```

Figure 8. Snippet of filesDropped() and isInterestedInDrag() Functions

9.

```
void loadFile();
void loadFile(const juce::String& path);

void updateADSR();

juce::ADSR::Parameters& getADSRParams() { return mADSRParams; }

int getNumSamplerSounds() { mSampler.getNumSounds(); }

juce::AudioBuffer<float>& getWaveForm() { return mWaveForm; };

float rawVolume{};
```

Figure 9. Public Class of PluginProcessor.h

10.

```
private:
    juce::Synthesiser mSampler;
    const int mNumVoices = { 16 };

    juce::AudioFormatManager mFormatManager;
    juce::AudioBuffer<float> mWaveForm;
    juce::ADSR::Parameters mADSRParams;
    //juce::AudioFormatReader* mFormatReader { nullptr };

    //==============================================================================
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (ThirdExperimentAudioProcessor)
};
```

Figure 10. Private Class of PluginProcessor.h

11.

```
{
    mFormatManager.registerBasicFormats();

    for (int i = 0; i < mNumVoices; i++)
    {
        mSampler.addVoice(new juce::SamplerVoice());
    }
}
```

Figure 11. Snippet of PluginProcessor.cpp

12.

```
void ThirdExperimentAudioProcessor::loadFile(const juce::String& path)
{
    mSampler.clearSounds();
    auto file = juce::File(path);
    std::unique_ptr<juce::AudioFormatReader> mFormatReader(mFormatManager.createReaderFor(file));

    auto sampleLength = static_cast<int>(mFormatReader->lengthInSamples);

    mWaveForm.setSize(1, sampleLength);
    mFormatReader->read(&mWaveForm, 0, sampleLength, 0, true, false);

    auto buffer = mWaveForm.getReadPointer(0);

    for (int sample = 0; sample < mWaveForm.getNumSamples(); ++sample)
    {
        DBG(buffer[sample]);
    }

    if (mFormatReader)
    {
        juce::BigInteger range;
        range.setRange(0, 128, true);

        mSampler.addSound(new juce::SamplerSound("Sample", *mFormatReader, range, 60, 0.1, 0.1, 10.0));
    }

}
```

Figure 12. Snippet of loadFile() Function

13.

```
void ThirdExperimentAudioProcessor::updateADSR()
{
    for (int i = 0; i < mSampler.getNumSounds(); ++i)
    {
        if (auto sound = dynamic_cast<juce::SamplerSound*>(mSampler.getSound(i).get()))
        {
            sound->setEnvelopeParameters(mADSRParams);
        }
    }
}
```

Figure 13. Snippet of updateADSR() Function

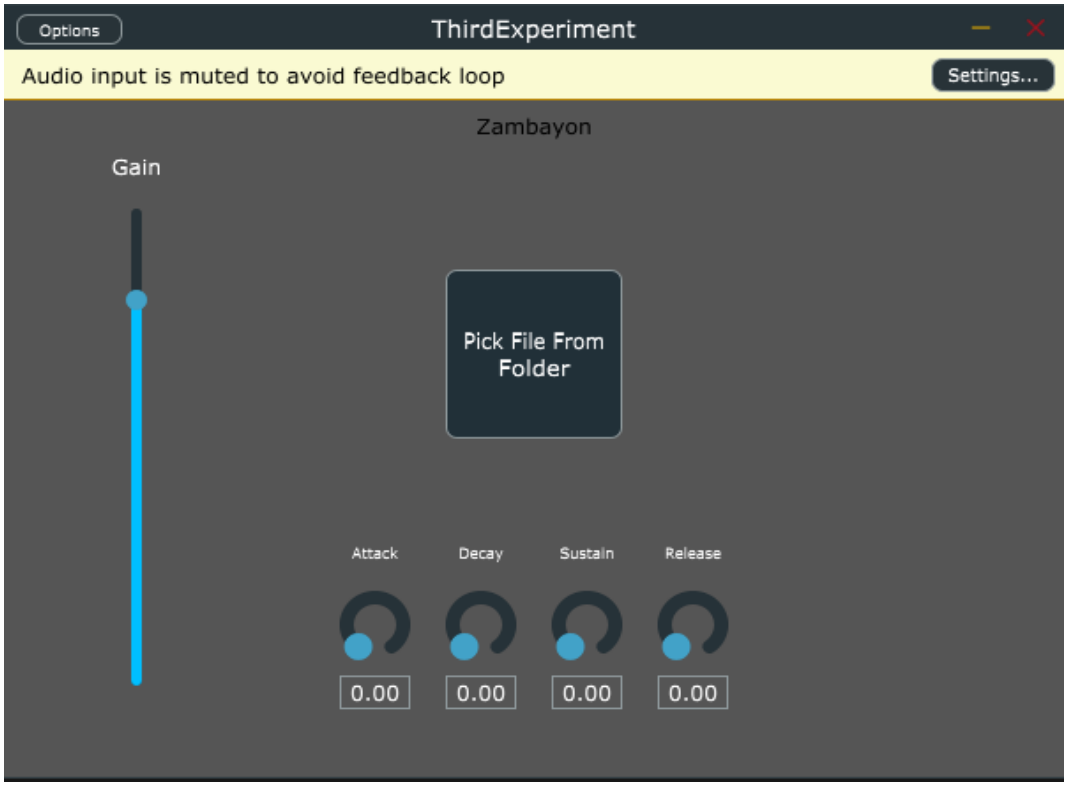
14. 

Figure 14. Image of my Audio Plugin

15. https://youtu.be/vd7z8kXcTg8

-This link will take you to a short Youtube clip demonstrating how my audio plugin works.

7. Bibliography

1. Lippman, Stanley B., et al. *C++ Primer*. Addison-Wesley, 2013.
2. Ekeroot, Jonas. “Audio Software Development: An Audio Quality Perspective.” *D-uppsats*, Luleå Tekniska Universitet/Musik Och Medier/Medier Och Upplevelseproduktion, 2008.
3. Raman, T. V. “Audio System for Technical Readings.” Cornell University, Dept. of Computer Science, 1994.
4. “Class Index.” *JUCE*, https://docs.juce.com/master/index.html.