Logan Bancroft Boucher
12/10/2015
Capstone Project Final Paper

Project Description:

The goal of my capstone was to create an audio plugin that could be used by an audio engineer inside their digital audio workstation (DAW). As most audio engineers know, audio plugins are small pieces of software that are called by a bigger piece of software (usually DAW software) in order to manipulate an audio signal in real time. This technology is used by recording studios and sound engineers around the world to modify audio so that it's either more pleasing to the listener or the audio now works for a specific purpose. These plugins use digital signal processing (DSP) algorithms to receive the desired aural effects. Practically all professionally recorded audio one hears today has had some processing performed on it before it is released to the public, and most likely that process was completed by an audio plugin within a DAW environment.

When I first began this project, my main goal was not only to make a functional plugin, but to also look into the development of DSP algorithms. I really wanted to go "under the hood" to see how digital audio works and how software can manipulate this audio through mathematical equations. I hoped that maybe by the end of this project I would have enough understanding of DSP to write my own algorithm. Unfortunately, this did not happen and I had to rescope my project accordingly based on the amount of time I had and the limitations of my own technical expertise. Through my analysis and reading of DSP algorithms, I found that often the process involved math that was beyond me, as well as a deeper understanding of electronics than I currently have. When I Googled resources for people interested in studying DSP, I was given links to books which are hundreds of pages long. People who specialized in DPS reported that it takes years of study and practice to become skilled at writing DSP algorithms and code, and that programming for DSP is actually one of the most difficult programming disciplines one can study.

With this in mind, I switched my focus from writing my own algorithms to instead implementing existing algorithms and code into a plug in program that can be used by an audio engineer or even a normal person desiring to manipulate an audio track in their DAW. My final project is a working multi effects plugin that implements several effects to manipulate audio.

The first effect is the Stereo Widener effect which I showed as my first draft. Just as the name implies, this plugin either widens or narrows the stereo field, depending upon the parameter set by the engineer. This is a very interesting effect that is almost immediately apparent to the listener when the user manipulates the parameter.

The second effect I implemented was a reverberation effect. It has several parameters that can be adjusted to change the sounds of the echos, while considering room size, wetness, dryness, damping, reverb width, and an interesting parameter called freeze mode. The reverb effect can be used to simulate what a source of audio might sound like in a particular room or space, depending upon what parameters the user inputs into the plugin. This effect is immediately apparent to the listener. If one modifies the parameters enough, they can experience some unique sounds from this plugin. If the engineer turns the freeze mode parameter up enough, the reverb will never fade as it creates a really eerie sound.

The final effect of the plugin was suppose to simulate a compressor. A compressor is used to reduce the dynamic range of an audio signal. To put it simply, a compressor is suppose to take the quiet parts of a signal and make it louder while taking the loud parts of the signal and making them more quiet. This means that overall volume level of the signal should become more consistent throughout. While this final effect was suppose to be a compressor, I failed in properly implementing an audio compression algorithm. As a result, the effect distorts the audio and makes it sound “crunchy”. While this could be a useful effect in certain situations, it is not what was originally intended. This compressor takes in two parameters, threshold and ratio. The threshold was intended to determine the decibel level that the compressor kicked in at, while the ratio was intended to determine how much the signal was compressed Unfortunately, due to time constraints and my own technical limitations, I will be leaving this effect as is. I spent days trying to make this effect work properly, and honestly I need a deeper understanding of how actual compressors directly manipulate signal in order to program this effect properly.

Since the DSP parts of the project proved to be a bit beyond my capability, I instead focused on creating a graphic user interface which could be used by the engineer to manipulate the parameters of an effect. I made sure that when the engineer pressed a button or moved a slider, it actually changed how the effect sounded in real time. This required the user interface to be synced properly with the DSP effects so that the values a user passed via the user interface were properly communicated. The actual values shown by the sliders and buttons on the screen were the actual values being used in the mathematical equations directly manipulating the audio samples. In order to do this, I used the JUCE framework for C++. The framework helped me create my graphic user interface in a way that made the interface communicate effectively with the audio effects I was using.

Overall, the final product was very close to what I envisioned creating. I have a functioning plugin that can be used by an audio engineer to manipulate audio in a noticeable way. The program runs nicely, and as far as I can tell there are no noticeable errors. The program is easily called by a DAW host program and can be used by just

about anyone with a limited knowledge of audio software use. While I am disappointed that I don't have the understanding of DSP I wanted, I am glad that I took a good look at the field; I now have a better idea about how digital audio works on a really low level. Additional work needs to be done is in the compressor algorithm. A proper algorithm needs to be implemented, and potentially the overall program needs more class and header files to properly implement a good compressor effect. The GUI could also be made to look a bit more attractive, but it is still an effective GUI that allows users to easily manipulate the program parameters. I also could have added a feature that would allow for saving and loading of preset parameter files. This would make it so a user could easily save settings they may want to use later.

Technical Description:

To go over the basics, a plugin is a small program that doesn't run on its own; instead, it runs inside a larger program (AKA the host) in order to do processing on a piece of audio. There are several different formats that audio plugins through which audio plugins could be constructed. I chose to use the most common format, Virtual Studio Technology (VST). Often people refer to audio plugins as VSTs. This is a reference to the format through which the plugins are constructed. Since the VST format is so common, this makes it so these plugins can be implemented in a wide variety of environments, or with many different hosts. VST plugins are often used with DAW software, in order to aid in the manipulation of digital audio.

The plugin I created was primarily built in C++ using the JUCE framework and the Steinberg VST Software Development Kit which the JUCE framework is dependent upon. There is some XML used in the project as well. The JUCE framework provides a nice structure for building audio plugins, and has some nice built-in features and functions that make the overall process easier and more organized. It also helps the programmer build a useful GUI for their project.

In my project I ended up using pre-existing DSP classes and then used JUCE to create a GUI and link it to these classes. While originally I wanted to make my own DSP code, this proved to be beyond my capabilities as I do not know nearly enough DSP theory in order to code something that works. DSP looks at digital audio at a really low level, and while I learned a lot from analyzing DSP source code, I am still far from being able to program DSP algorithms. The proof of this can be seen in my compression effect code. I tried hand-writing the code for an algorithm I found online, and in the end the resulting audio sounded distorted and nothing like the "compressed" audio I was aiming for. The main reason why is because I really don't understand conceptually what was fully being done to the audio. I had much better results leaving the DSP to people

who knew what they were doing and instead focusing on making it so that there was a way for the everyday user to use the DSP code.

One of the nice features about the JUCE framework is that when creating a new project it makes four files: “Pluginprocessor.cpp”, “Pluginprocessor.h”, “Plugineditor.cpp”, and “Plugineditor.h”. These four files form the core of the plugin with the Pluginprcessor.cpp being the class for the audio processor object and plugineditor.cpp being the class for the GUI. JUCE automatically sets the code up in these structures so there are “areas” in the file to insert certain features of code. For example in “Pluginprocessor.h” there is a section in the code specifically marked for declaring helper methods and private data. These sections are surrounded by comments explaining what is suppose to go into the section so the programmer has an easier time properly placing their code within the files.

When initially building this plugin I followed a tutorial online in order to learn the basics of the JUCE framework and how to build a VST using the framework. The plugin is pretty much divided into two main components the GUI that gives the user something to look at and manipulate and the audio processor that actually performs calculations and manipulates the audio. The first thing one needs to do is create a connection between the GUI class and audio processor class. JUCE makes this easy by allowing the creation of a JUCE GUI component which then can be easily linked with the Audio Process Class through some minor changes in the project set up.

Once the basic GUI component is linked with the audio processor data needs to be passed between the two. VSTs built in JUCE are dependent on a timer function that performs certain actions at a programmer defined time interval. This is referred to as the timer callback function and the plug in calls this function over and over again at a set time interval. JUCE provides an easy way to implement a timer callback function. In my project this function was called every 200 milliseconds and was used as a way to pass up to date information on arguments between the GUI and the audio processor that was actually manipulating the Audio data. This made it so that what the numbers the user was seeing via the GUI were the numbers that were actually being used in the DSP calculations. It kept the GUI in sync with processing.

Once the timer was in place I started building the actual GUI with the JUCE framework. This was a tedious project, but it was made a lot easier with the JUCE framework which actually provides a GUI for building GUIs! I created different slider objects to allow a user to click on the visual representation of my plugin and drag the slider in order to change a numerical value between a minimum value (the left side of the slider) and a maximum value (the right side of the slider). A few bypass buttons were also added in order to allow users to turn certain effects on and off. The code for these objects is inserted into “plugineditor.cpp” along with the various properties of these objects such as the range of values that a slider covers.

I created some parameters for which GUI components can be associated with. I ended up making several enumerators that are associated with the parameters I wanted the user to manipulate via the GUI and then pass to the audio processor. I also created a boolean which serves as a flag that indicates whether or not the GUI needs to be updated. Methods were also implemented in order to set the flag to true when a GUI update is needed or to set the flag to false once a GUI update is complete. This ensured that the GUI was only updated when a parameter was actually changed and that what the GUI is reflected were the actual values of what's being put into the audio processing calculations.

At this point I created two objects, one associated with the Stereo Widener class "StereoWidthCtrl2.cpp" and one associated with the reverb class located in "juce_Reverb.h". The StereoWidthCtrl2 object I named "mWidthControl" and the Reverb object I named "areverb". When the program is widening the audio mWidthControl is handling the processing while when the program is creating the reverb effect areverb is handling the processing.

How audio is actually passed to these objects is in the processblock() method in my code. This is where the actual audio calculations occur. What comes out the other side is our modified audio signal. Any DSP calculations or calls to object methods that perform DSP calculations are done here. For example if I want to run the audio through the mWidthControl object in order to actually affect the audio we are hearing we call mWidthControl.Clockprocces() (the method within mWidthControl that does the DSP processing) here within processBlock(). I can also do the DSP calculations directly in the processBlock() method, as I tried to do with the compression effect.

Before I get more into how processBlock() works I just want to explain some basic concepts about digital audio. An audio signal can be visually represented as a waveform that shows the variation in amplitude (how loud the sound is). In order for computers to approximate the waveform and process it, they break the wave up into discrete chunks called samples. Each sample represents how loud the signal is at that discrete moment of time. Typically CD quality digital audio is sampled at a rate of 44.1 kHZ which means that for every second of digital audio there is 44100 samples that are taken to approximate the waveform or ,in other words, 44100 different amplitude values are taken over that one second of audio. The higher the sampling rate the more accurate the digital approximation of the wave is to the actual sound wave or the higher the quality of the digital audio. For my project I used the typical sample rate of 44.1kHz.

With the knowledge that digital audio is made up of a finite number of samples one can see how the processblock() works. The total number of samples of audio that needs to be manipulated is loaded into a buffer called AudioSampleBuffer and then passed into the processblock() method. The buffer is made up of a series of pointers to floating point values that represent the amplitudes of each samples. The process block

then iterates through every single sample that exists in the buffer and performs the proper calculations to modify the sample to the user's liking. Since there are 44100 samples in one second of audio these calculations are performed 44100 times a second, once for each sample of audio.

In Processblock() I pass the first set of samples in the audio buffer to the mWidthControl object for processing. There the mWidthControl object does its own modifications through DSP calculations to the samples. Once the samples have gone through the mWidthControl object they proceed to the areverb object where once again I pass the pair of samples in order for areverb. After the modified samples are outputted by areverb they then go into the compression calculations. Since there is no compressor object and the calculations are all directly done in the processblock() method there is no need to call any external methods. After the samples have gone through the compressor calculations they have now reached their final form and are returned to the host program who then plays the modified audio through the computer's speakers. The audio can now be heard in its final form after it has been processed. In essence the processblock() method forms the core of the plugin and produces the much desired output that this whole project revolves around.

Some final technical details I want to go into is the DSP classes that I implemented to do the processing. As I mentioned earlier I tried to learn enough about DSP algorithms to try to make my own, but to do so proved beyond my current capabilities as I do not have enough understanding of the signal processing theory involved. When I did attempt to hand code an algorithm I ended up getting my "compressor" effect which doesn't act as compressor at all, but instead just distorts the audio. Since I i was not pleased with the results of the compressor, I instead I ended up using C++ class files that I found to do the reverb and stereo widening processing for me. I then had my plugin pass audio and parameters into the the mWidthControl and areverb objects that were made from these classes.

In the StereoWidthCtrl2.cpp and .h files the StereoWidthCtrl2 class is declared and implemented. The class consists of two methods ClockProcess() and SetWidth(). SetWidth()in a value use that values calculate how "wide" our stereo field is going to be. Two values come from this process. These two values are used by the Clockprocess() method. Clockproccess() takes in two samples fed to it by processblock(), the left audio sample and the right audio sample. It then performs calculations on the two samples using the two values that came out of the setwidth() method. The resulting samples are part of the final signal that can be heard from the speakers!

The reverb class that I used actually was included in the JUCE framework. It incorporates a relatively complex algorithm that uses theoretical concepts from DSP theory that I am not that familiar with. I'm not going to go into the details of the algorithm as it's far more complex than the stereo widener algorithm described above and

honestly I do not completely understand everything that is going on and it goes beyond the scope of my project. Instead I am going to focus on how parameters are passed into this class as that was the main challenge I encountered while building this project. Earlier I mentioned that my object areverb that's built from the Reverb class defined in juce_reverb.h took parameters packaged in a custom defined struct type called parameter. To put it simply I can't just change one parameter and then pass that single parameter into the areverb object. Instead I had to declare a struct of type parameter in the audio processor class. I called this parameter struct newparameters and packaged all the parameters that areverb needed to function into this struct. Once the struct was made, I then had to modify the individual fields of the struct that was associated with the parameter I wanted to modify. Once I modified that single parameter in the struct, I had to pass the whole struct (including all the values I didn't modify) into the areverb object . From there the different values would propagate to each of the parameters in the struct to their appropriate variables within areverb.

I initially had trouble with the above parameter passing concept as it was a lot different than how I passed the stereo width parameter into the mWidthControl object. I had issues trying to use the parameter struct defined in the juce_reverb.h in the audio processor class. I had to experiment a lot with C++ syntax in order to get access to the defined parameter struct type. Once I figured it out, I could easily declare my own parameter struct and then modify the encapsulated values within the struct with the values that were present in the parameter enumerators. Once I had modified all the values in the struct I could easily pass it into the into areverb's setParameter() method and correctly modify the relevant values.

Finally I want to discuss my attempt at a compression algorithm. I based the algorithm on a compressor class I found online and on a post I found on an online forum specializing in DSP. I tried implementing concepts from both in order to make my own; I also tried to simplify them and try to skip over things that I either did not understand or thought was unnecessary. My end result was disappointing, to say the least, but it does affect the aesthetic qualities of the audio. Even though it is not the desired compressor effect, it is still an effect. The simplified algorithm above is still quite complex. Since my compressor doesn't work, the algorithm is obviously incomplete and possibly completely wrong. Future work could be done in fixing the algorithm and doing further research into how compression algorithms actually work.

What I Learned:

I learned a lot by creating a plugin. The main thing I learned about was how audio is stored digitally and how it can be manipulated by mathematical equations. I have been wanting to do programming with audio since before I came to American University and this project was a chance for me to satisfy this desire.

One of the major concepts I now understand is that digital audio is stored in a buffer of samples. These buffers are represented by an array of floating point values, a data structure that programmers commonly use when programming. Each value in the buffer corresponds to the amplitude of an audio signal taken at a discrete moment in time. If one wants to modify the overall signal, the individual samples stored in the buffer need to be modified. Learning this was huge in developing my understanding of how digital audio works.

Once I learned how audio is stored digitally I did research into how a computer program can manipulate this data through various mathematical equations and algorithms. These manipulations fall under a field called Digital Signal Processing. While I understood some of the math that was going on, a lot of it proved to be over my head and beyond my current education level. Still, I learned some basic concepts about how audio can be changed mathematically and I now feel like I have a base understanding that I can build upon with further education.

Another great experience I got from this project was building a graphic user interface. Before this project I had practically no experience with building any sort of user interface as most of my assignments have been command line based. Being able to make an interface and then sync it up with the processing going on behind the scenes was an extremely informative experience. Going forward I now feel like I have a basic understanding of how one can build graphic user interfaces that interface with the overall program.

I also gained a more solid understanding of object oriented programming from this project. While I have done object oriented programming before, I was a bit out of practice before starting this project. I hadn't really used object concepts in about a year and building this plugin was a nice refresher in the concepts I previously learned. This project reminded me of the power of programming under the object oriented paradigm as I saw how useful it was to treat the user interface and the audio processor as objects that could interact with each other. I also saw how I could encapsulate certain audio processing algorithms in objects that could then interface with the audio processor object, such as I did with the reverb and stereo widener objects. Overall I feel more solid in my understanding of object oriented concepts than I was before taking on this project.

Finally this project gave me a nice review of the C++ programming language. I had done some programming in C++ before, but like object oriented programming I had

not used it in quite a while. Building this project made me remember my C++ syntax. It also reminded me of some of the features of the language and how classes are declared in the language. At times I struggled with the syntax, but through hard work and perseverance I was able to accomplish what I wanted to do with the language. I still have a lot to learn though before I become a proficient programmer using the language are their are plenty of features and syntax that I am still not proficient with. My comfort with the language will only grow the more I use it.

Compilation Instructions

Since my plugin is not a stand alone program, some other software will need to be downloaded in order to compile and run the program. Below I am going to to lay out the basic steps that will allow the program to be run.

First, download a host program that can interface with my plugin in order for the plugin to run is needed. I recommend downloading Reaper, a digital audio workstation (http://www.reaper.fm). Once this is installed,a host program that can run my plugin is available.

Once Reaper is installed, it is necessary to download an IDE that works with the JUCE framework. For a computer running Windows, Microsoft Visual Studio(https://www.visualstudio.com/downloads/download-visual-studio-vs#d-2010-express), should be downloaded; Mac will need to download Apple's Xcode.

Once an IDE is installed, it is necessary to download the Steinberg VST Software Development Kit (SDK) and install it in the proper location the computer. The download can be found at http://www.steinberg.net/en/company/developers.html and instructions on how to install it can be found at http://www.redwoodaudio.net/Tutorials/juce_for_vst_development__intro2.html

Finally it is necessary to download and install the JUCE framework which can be found at http://www.juce.com/download. After the framework is downloaded the required software to compile the code is available.

The first step is to open up the Introjucer program that came with the JUCE framework. A window should appear that asks the user to create a new project with several options for different types of applications that can be created using the framework. Select the Audio Plug-In option. A window should appear asking the user to name the project and to choose a preferred IDE. Name the project "StereoWidthCtrl2" and chose the IDE that will be used.

After selecting the IDE and naming the project, a window should appear with two tabs, "Files" and "Config". Make sure you are on the Config tab. On the right there is the Project Settings form. Underneath are several different settings that JUCE uses to build the project accordingly. Make sure the option "Build VST" is checked and that none of

the other “Build” options are checked. Make sure that “Plugin Channel Configurations” is set to to “{1, 1}, {2, 2}”. Leave the rest of the settings as is.

Towards the bottom of the screen on the left underneath the list of modules you should see your target IDE name. Click on the name (for example Xcode if a Mac is being used) and a new pane should open up on the right with several fields. Make sure that the field that says “VST Folder” has the proper file path to the Steinberg SDK that you installed earlier as the program will not compile without it.

After the settings have been modified under the Config tab, click on the Files tab. A screen will appear that shows all the source code files contained within the project. By default 4 files are included: “PluginPorcessor.cpp”, “PluginProcessor.h”, “PluginEditor.cpp”, and “PluginEditor.h”. Select all these files, right click on them, and select delete. A window should appear asking if the files need to be moved to the trash, move them to the trash. These defaults files are not needed because the program is going to use the code that is provided instead.

Once the files have been deleted right click on the source folder in the Jucer window. An option should appear that says “Add Existing Files…”. Select this option and then add the source code files provided. In total there should be 6 l: “PluginPorcessor.cpp”, “PluginProcessor.h”, “PluginEditor.cpp”, “PluginEditor.h”, “StereoWidthCtrl2.cpp”, and “StereoWidthCtrl2.h”. Once the files are in place save the project by going up to the file menu at the top of the screen and selecting “Save All”.

The program is ready to be compiled! Click on the Config tab again. In the bottom left hand corner there is a button that saves the project and then opens the project in the the IDE of choice. Click this button and the IDE should open with the project already set up . Now hit compile! The first compilation might take quite a while to complete, but once it’s done the compiler should have outputted a VST file.

The final step is to open the host program and run the plugin! Open up Reaper. Once Reaper is open go to the top of the screen. There are several different menus, click on the Reaper menu and select Preferences. Under Preferences go down to the Plug-Ins section in list located in the left hand pane of the window. Under Plug-Ins click VST. A new pane should open up. Click the button that says “Re-Scan”. This should search the computer for new VST files that Reaper can use and execute.

Once “Re-Scan” has been clicked, click Ok and the window will close. The main window of Repear will now be at the forefront of the screen. Go up to Insert at the top of the screen. A drop down menu you should appear and select “Media File…”. Find an audio file on the computer and select it. Repear will now import this file and place it on a new “track” represented by a fader at the bottom of the screen. Above the fader that appears is a little white button with “FX” printed on it, click on that button. A new window will appear showing a list of plug-ins that Reaper recognizes. Select the plugin, “StereoWidthCtrl2” and click okay. A new window should now appear with the plugin

graphic user interface! The plug is now ready to manipulate the audio on the selected track.

Alternatively if one did not want to go through all the compilation steps the provided VST file can be saved to the computer. Reaper can then scan the system for new VSTs. This should save quite a bit of time.

Bibliography

The main tutorial I followed to build the plugin:
http://www.redwoodaudio.net/Tutorials/juce_for_vst_development__intro.html

Algorithm which the Stereo Widener effect I implemented is based upon:
http://musicdsp.org/showArchiveComment.php?ArchiveID=256

Reverb effect I implemented in my plugin:
https://github.com/julianstorer/JUCE/blob/master/modules/juce_audio_basics/effects/juce_Reverb.h

Compressor effect that I tried basing my Compression algorithm on:
https://github.com/music-dsp-collection/chunkware-simple-dynamics/tree/master/simpleSource

Another compression algorithm I referenced:
http://musicdsp.org/showArchiveComment.php?ArchiveID=169

JUCE, the framework I used throughout the whole project:
http://www.juce.com/

Steinberg VST SDK, the development kit that JUCE and thus my project is dependent upon:
http://www.steinberg.net/en/company/developers.html