GatorVerb: Modeling Algorithmic Reverb in a Virtual Studio Technology Plugin

Alex Blair Whitler
Spring 2011
Magna Cum Laude
Bachelor of Science in Software Engineering

Advising Committee Members

Chair
Dr. Douglas. D Dankel II

Dr. Eric M. Schwartz

Dr. Prabhat Mishra

Table of Contents

1 - Introduction

1.1 - Abstract

GatorVerb is a Virtual Studio Technology (VST) audio effect plugin that models three reverberation algorithms; feedback delay networks and two variations of Schroeder's reverberator. The plugin consists of the underlying signal processing logic that creates the reverb model of the chosen algorithm and a GUI used to control the effect's parameters. The implementation is constructed from delay lines, allpass filters, and comb filters. GatorVerb processes audio in real time, rather than using batch processing to create a modified signal to be used separately. Due to this, parameters can be chosen or changed at any time during execution in order to create the desired level of reverb for the current project.

1.2 - Project Overview

The use of VST plugins has become immensely popular in recent years. As audio production becomes increasingly computer dependent, it becomes necessary to create digital models of valuable analog effects, such as reverb, delay, or tremolo. GatorVerb is a host-dependent algorithmic reverb modeler. It requires any host program that supports Steinberg's VST architecture. For the purposes of this project, the host program is Steinberg's Cubase SX3 audio production software.

The creation of GatorVerb stems from the need to migrate analog music to the digital realm. GatorVerb attempts to recreate three standard analog reverb algorithms digitally. These three algorithms consist of the Feedback Delay Network (FDN), Manfred Schroeder's Allpass algorithm, and Schroeder's original algorithm. These algorithms are discussed in more detail in Section 1.3.

The goal of GatorVerb is to provide a simple, easy to use interface for a user to create any level of reverb they desire, including some levels which would be unachievable in the analog realm. It does this by using only three controllable parameters; Wet Mix, Room Size, and Dampening. The creation and purpose of these parameters is discussed in Section 2.4.

1.3 - Past Work and Research

1.3.1 – Reverb Types and Brief History

A number of reverb types exist, most of which are still in regular use today. There are two main categories of reverb; algorithmic and convolution, which usually contains more specific types, such as hall, plate, and spring reverb.

Algorithmic reverb, as its name suggests, makes use of varying algorithms to define the specific reverb type to be modeled. Typically, algorithmic reverbs provide a more customizable mix, so a single reverb unit or plugin can be used for multiple purposes. GatorVerb is an algorithmic reverb model.

Convolution reverb is used for specific applications. This is because it makes use of samples of actual acoustic spaces to model the impulse response of that space. Since each convolution reverb models a particular space, they are generally very limited in application, especially in music production, where variety is a necessity. Hall, plate, and spring reverb are convolution types, and each models what its name suggests. Hall creates a large hall acoustic space. Plate actually sends sound through a hanging metal plate which then reverberates to create the effect. Spring reverb closely resembles a slap-back echo and uses springs to create a short, quick reverb sound.

During the 1920's, reverb began being used in recording. This reverb was created by physically recording at a distance from the sound source, so various rooms were

required to create different reverb styles. Due to technology being in its infant stages, most recordings were left dry because of playback problems caused by reverb.

In 1957, Elektromesstecknik (EMT) released the first ever plate reverb unit, shown in Figure 1.1. This breakthrough in creating reverb without a specific acoustic space paved the way for future designs. Reverb became truly popular when Fender created a compact reverb for Dick Dale to use on his debut album. This compact reverb led to Fender's spring reverb which is still desired more than any other analog unit today.

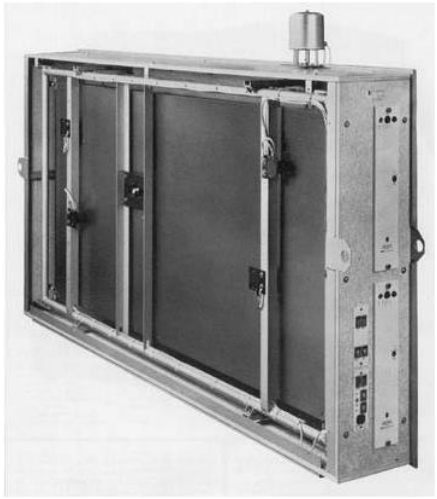

Figure 1.1: EMT's original 140 plate reverberator

Lexicon released the first digital reverb in 1971. As computers became commonplace, audio production moved from analog to digital, and software based reverbs were developed. Today, VST has become an industry standard, creating everything from reverb to auto tune digitally.

1.3.2 Reverb Algorithms

Artificial reverb primarily involves the use of delay lines. At the most basic level, reverberation is simply multiple echoes or delays of different lengths decaying over time. This idea became the foundation for artificial reverberation.

The idea of artificial reverberation was primarily fathered by Manfred Schroeder in the 1960's. Schroeder's algorithm centers around a parallel bank of feedback comb filters. The output of this filter bank is then fed into cascading allpass filters. The output of these allpass filters is then sent out as the resulting output. A diagram of this algorithm as defined by Schroeder can be seen in Figure 1.2.

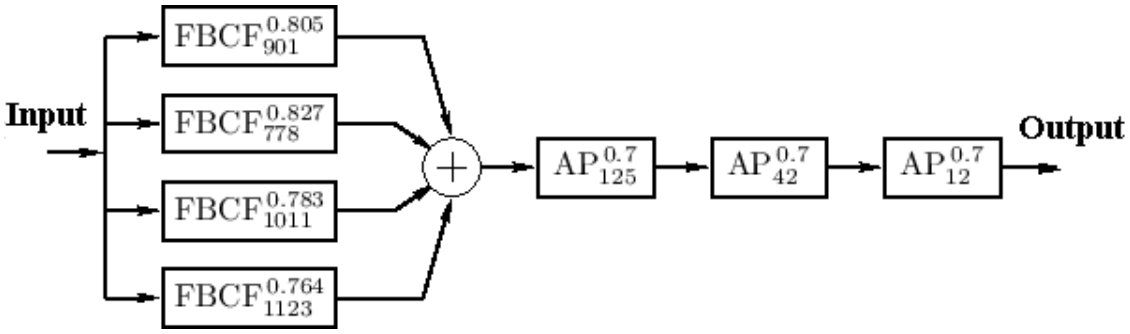

Figure 1.2: Manfred Schroeder's original reverberation algorithm

Comb filters work by adding a delayed version of the original signal to itself. In feedback comb filter, the output of the filter is actually fed back and added to the input signal. When the filter is stable, the signal is seen as a series of identical impulses decreasing in amplitude. Thus, the response sounds like the original signal decaying over time.

Allpass filters pass all frequencies through equally, just as lowpass filters allow low frequencies through, so the amplitude response of an allpass filter is one at all frequencies. Typically, allpass filters have unity gain across the frequencies. Interestingly, an allpass filter can be constructed from a feedback comb filter and a feedforward comb filter. Essentially, the input signal is added to a delayed version of itself, while simultaneously, the output is added to the original input and fed into the internal delay line. This can be seen more clearly in Figure 1.3.

Arguably the cleanest and best sounding algorithmic reverb is the Feedback Delay Network (FDN). Originally pioneered by Michael Gerzon, the FDN is a surprisingly simple algorithm, but is noted as being one of the best sounding algorithms currently. The name defines the algorithm in its entirety. A FDN is simply a network of parallel, mutually prime delay lines added together. The idea is to recreate the basic reverb idea noted above: to create multiple echoes of different lengths that decay over time. By making the delay times mutually prime the corresponding echoes of each delay line fall separately creating an artificial impulse response. The user hears this as multiple echoes decaying at different rates, just as one would hear from a physical acoustic space.

2 - Design, Development, and Implementation

Before going over the development of each algorithm and the effect's necessary parameters, we will first go over the basics of the VST development kit and its necessary classes used in GatorVerb's development.

The most important class for our purposes is *AudioEffectX*. This class provides methods for all of the following tasks:

- Processing audio signal
- Getting/setting the program, which gets/sets the name and settings of any parameter presets
- Getting/setting parameter values
- A number of documentation methods, which get the effect name and vendor name for display in the host program

The processing methods are the heart of the entire plugin. *AudioEffectX* actually provides two versions of this method, *process()* and *processReplacing*(). Both methods perform the same function except for the final resulting output. The *process()* method is used when the VST is placed as a send effect. A send effect is essentially an outside bus that processes the input signal separately. This processed signal is then mixed with the original unaltered signal. In *processReplacing*(), the signal is in series with the effect, meaning that the original input is altered immediately and the processed audio becomes the resulting output. GatorVerb uses the *process()* method. Due to the fact that reverb makes use of mixing wet and dry signals, we want to process the input signal separately and mix it later. This provides a much more customizable sound.

GatorVerb does not make use of the get/set program methods because no parameter presets are provided. This was a personal design choice. From personal experience, the preset values of an effect never get used, except for rare occasions. I chose to omit preset values and allow the user to set the sound they want.

The get/set parameter methods are self explanatory, and work just like standard accessor and mutator methods to retrieve or change the desired parameter's value.

The documentation methods are only used in Steinberg's records. Steinberg requires any VST that is going to be distributed publicly to be registered with them under a unique name. When a VST programmer is issued this unique name, a method *getProductString()* is used to set this unique ID. Other documentation methods include *getVendorString()*, which is used to identify the creating company or organization, and *getEffectName()*, which simply gets a string that the VST GUI can display to the user. None of these methods affect the plugins output and merely provide ways to identify the creator and name of each effect.

The main class of GatorVerb is an extension of the *AudioEffectX* class, so that it can access all of these methods. As simple as it seems, this is the only class necessary to perform the signal processing.

The GUI is a separate entity. Although Steinberg provides its own GUI SDK, which GatorVerb makes use of, it is not required to develop a GUI. In fact, some plugins have such elaborate interfaces that the creators chose to make use of OpenGL and develop their own GUI. The creation of the GatorVerb's GUI and the SDK classes used are discussed in Section 2.3.

2.1 – Creating the Filters

The main task in creating these algorithms is to translate the response of each filter type into C++ in order to be used with the VST SDK. To do this, we turn the difference equation into a simple calculation using an individual buffer for each filter. These internal buffers are used to model the delay lines used for feedback.

Creating the comb filter is the first step in implementing Schroeder's algorithms. Figure 2.1 shows the signal path of a feedback comb filter.

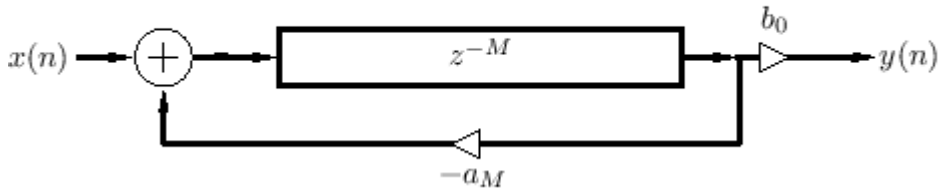

Figure 2.1: Feedback Comb Filter

The output, ***y***, is created from the buffer's delayed output. Each buffer is modified by a feedback value, ***a***. To create a realistic reverb response, feedback must be strictly less than one to provide decay. If feedback is set equal to one, it creates an infinite reverb response, while a value greater than one will create an increasing response. The coefficient ***b***, acts like ***a***, so in the GatorVerb implementation, it will be set equal to one to provide an unaltered signal being output from the buffer. The resulting equation for the feedback comb filter's output is:

(1) $Y(n) = b*z(n)$

Since the gain, ***b***, will be kept as one to preserve the signal amplitude, the delay line's input can be seen as:

(2) $Z(n) = x(n) + -a * z(n-1)$

In equation two, the variable ***x***, is the original signal, as shown in Figure 2.1.

Surprisingly, the allpass filter is actually simpler to convert to a C++ usable form. Figure 2.2 shows the diagram for an allpass filter. This image was obtained from a Camtasia screen capture created by an unknown professor.

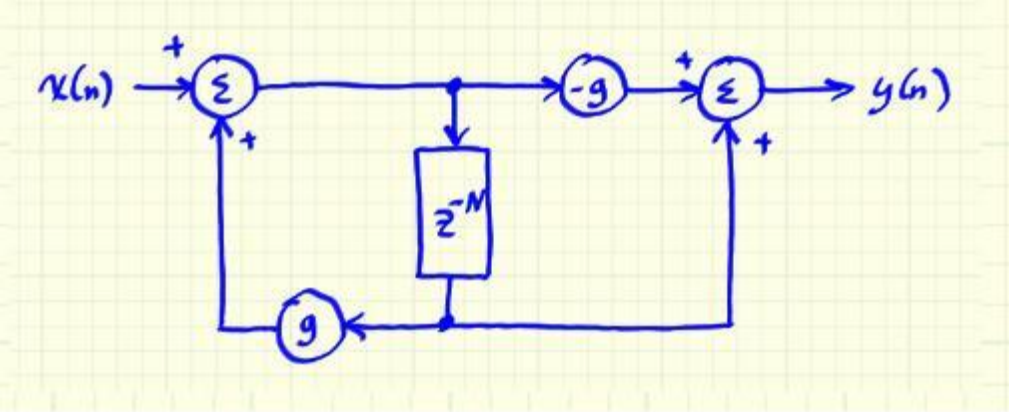

Figure 2.2: Allpass Filter

In this diagram, ***g*** represents gain, as ***a*** did for the comb filter. The delay line is represented by ***z***, like it was in the comb filter. For simplicity, we set gain equal to one. In doing this, we achieve an equation for the output, ***y***.

(3) $$Y(n) = x(n) * -g + z * g$$

For the purposes of reverberation, we want to alter the amount of feedback received from the delay line. Again, if we simply took the unaltered delay line output, it would never decay. This decay is accomplished by simply adding in a feedback parameter, which will be called ***f***, resulting in the equation for the internal delay line input.

As the signal is processed, the delay line's input is continuously recalculated based on the equation:

(4) $$Z(n) = x(n) + z(n - 1) * f$$

Now that we have the necessary equations, converting everything to C++ is a relatively simple task. First, the delay lines must be created.

Using the VST architecture, audio signals are passed in as floating point values. This makes creating the delay lines a simple task; they are represented as arrays of float values. To implement the delay, every time an input sample is taken, the input signal is stored in the current array position. The array pointer is then incremented. The output signal is taken directly from the array position. For example, if the audio sample rate is 44.1 KHz, as is standard, then a one second delay would require an array of size 44,100. After one second, 44,100 samples will have been sent, so the buffer will be full. When the end of the buffer is reached, the pointer is set back to zero. On the next sample, the output will take the first sample stored in the buffer, which occurred one second ago, so we achieve a one second delay. Thus, creating a delay of one second in C++ would translate to:

```
float* buffer = new float[44100];
```

Now that the delay line has been created, we will make use of it to create the two filters discussed above, as well as creating network of delay lines for the FDN. The comb filter will be created first.

2.1.1 – Comb Filter

For this discussion, assume that the internal delay line is simply an array called buffer[].

A dampening factor, which we will call ***d***, is added to the equation as a parameter that allows for extra reverberation decay, if necessary. As with the feedback parameter, dampening is strictly less than one to create a rapidly decaying signal.

Looking at equation one, we can see that the filter's output is simply the output of the buffer, so:

Output = buffer[i]

Now, looking at equation two, the buffer's input can be created as:

*Buffer[i] = input + - a * output*

If we want to add a dampening factor, which will simply cause the reverb to decay at a faster rate, we simply multiply the buffer output by ***d*** to dampen what comes out, so we end up with:

*Buffer[i] = input + -a * output * d*

Now we will create the allpass filter.

2.1.2 Allpass Filter

Equation three tells us the output of the allpass filter. By translating this to C++, we get:

*Output = -input * buffer[i]*

To create the decaying reverb, as with the comb filter, we put equation four into C++.

*Buffer[i] = input + buffer[i-1] * feedback*

Fortunately, once the filter's response is understood, translating to C++ is a simple task.

Now that the two necessary filters and the delay lines have been created, the reverberation algorithms can be implemented by putting these pieces together.

2.2 – Schroeder Algorithms

The development of each of Schroeder's algorithms using Steinberg's VST Software Development Kit is discussed in this section. The primary structure for each algorithm's program is identical, but each algorithm's processing method is different, since this is where the work is done.

To make piecing these algorithms together simple without filling these sections with code, I will simply use each filter's *output()* method, which takes in the input signal, to retrieve the output. In this method, the output signal is returned and the internal buffer is incremented, setting the pointer to zero if the end of the buffer has been reached.

2.2.1 – Original Schroeder Reverberator

Recall that Schroeder's original algorithm (Figure 1.2) is created by building a parallel bank of comb filters and feeding their output into cascading allpass filters. We have four comb filters and four parallel filters, as in Schroeder's original algorithm.

For the parallel comb filters, we can see that the output of each filter is added to create one final output that will go into the allpass section. Using a for loop, we can quickly achieve this output. *OutputCF* will represent the output of the comb filter section.

```
for(int i = 0; i < 4; i++)
{
        outputCF += comb[i].output(input);
}
```

As this for loop executes, the final comb filter section output is calculated from each individual filter's output.

Now this input is passed into the allpass section. The final output, *result*, will come out of this portion. Again, since we have four filters, we use a for loop. However, these filters are in series, so their respective outputs are not added together, only the final filter's output is sent as the resulting signal.

```
result = outputCF;

for(int i = 0; i < 4; i++)
{
        result = allpass[i].output(result);
}
```

This loop processes the input signal sequentially, and the output of each filter is sent into the next filter down the line.

Each of the filters' internal delay lines has to be a different delay length in order to create the reverb effect.

2.2.2 –Cascading Allpass Filter Algorithm

Schroeder also proposed using only allpass filters to create an artificial reverberator. This modified algorithm is shown in Figure 2.3.

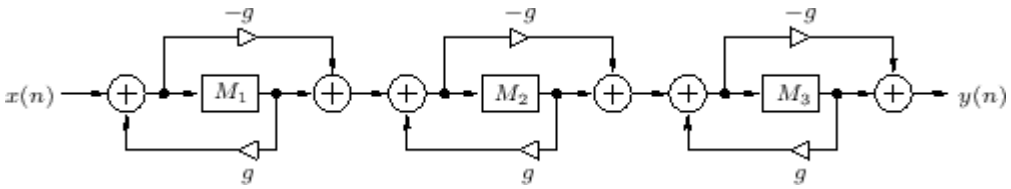

Figure 2.3: Schroeder's Cascading Allpass Filter algorithm

Schroeder originally suggested five of these filters in series, so this is the algorithm GatorVerb aimed to create. Like the allpass section of the original algorithm, the resulting output comes from sequentially processing the input signal. In this case, it will be processed five times in total before being sent out to the host program.

```
result = input;

for(int i = 0; i < 5; i++)
{
        result = allpass[i].output(result);
}
```

2.3 – Feedback Delay Network

The original idea behind GatorVerb was to recreate Schroeder's original algorithm. However, once that was done, my musical interest wanted to try another algorithm. After doing some research on other reverb algorithms, I decided to try the FDN. A FDN using three delay lines is shown in Figure 2.4.

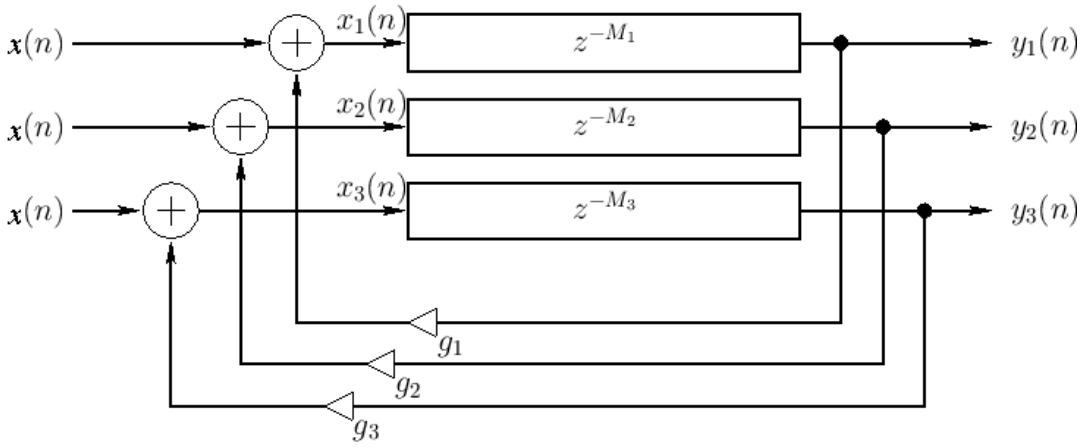

Figure 2.4: Feedback Delay Network

In this algorithm, each delay line's output is altered by an individual gain, ***g***, which acts as a decay modifier, before being fed back into itself. To create reverberation, each of the delay line outputs, ***y1 – yn***, are added together. In addition, GatorVerb adds in the original input signal, ***x***, to the final output. The outputs and input are each modified by the wet mix parameter to provide a customizable mix of wet and dry signals.

For GatorVerb, the FDN consists of eight separate delay lines. The delay values, which determine how long the signal is delayed, are unique for each line in order to provide a decaying echo. Unfortunately, there is no concrete algorithm to calculate perfect values for each line, other than knowing that they should be mutually prime or at least have very few common factors. Once initial values are chosen, they can be tweaked based on listening tests to create a desired level of reverberation.

Prime numbers were chosen as a starting point for the delay values. For GatorVerb, I was working with a sample rate of 44.1 KHz, since this is standard for most purposes. Due to this fact, a delay of 44,100 is equal to one second, as explained above, so each of the chosen delay values will represent milliseconds. The starting values chosen were: 43, 53, 59, 67, 73, 79, 89, and 97. To convert these values into the correct values in terms of sample rate, we use a simple equation:

(5)
$$\frac{X \text{ ms}}{1000} = \frac{delay}{44100}$$

After performing the necessary calculations, the following delay line buffer values are found in the following table.

Milliseconds	Delay line size (based on 44.1 KHz)
43	1896
53	2337
59	2601
67	2955
73	3219
79	3484
89	3925
97	4277

Since we know how to create the delay lines already, we can create this network without any issues. First, the initialization of each delay line.

Buffer1 = new float[1896]*
Buffer2 = new float[2337]*
Buffer3 = new float[2601]*
Buffer4 = new float[2955]*
Buffer5 = new float[3219]*

Buffer6 = new float[3484]*
Buffer7 = new float[3925]*
Buffer8 = new float[4277]*

To create the desired delay, each time a sample is taken, we set the current buffer position equal to the input and increment the pointer. If the pointer equals the buffer size, we set it back to zero to point at the base of the buffer. The entire process looks like the following in C++.

```
Output = buffer[i];
Buffer[pointer++] = input + output * feedback

If(pointer >= buffer_size)
{
        pointer = 0;
}
```

This same process is done eight times in parallel, resulting in the final altered output. This final output from the network is then added to the original input, with each being altered by the wet mix parameter to create the wet/dry signal mix.

2.4 – Plugin Parameters

In this section, we will briefly discuss each of the user-controllable effect parameters and their purpose. All parameters range from zero to one.

2.4.1 – Wet Mix

Wet mix controls the signal level of the original dry signal mixed with the altered wet signal. This parameter is used in the final step of each algorithm. The wet signal is multiplied by the wet parameter directly. To obtain the dry level, we simply take (1 – wet) and multiply it by the dry input signal. In the FDN algorithm, for example, the final output equation is:

*Output = (output * wet) + (input * (1 – wet))*

2.4.2 Room Size

Room size is GatorVerb's name for the feedback parameter discussed in each algorithm. Like all parameters, it ranges from zero to one, and to create reverb, it must be strictly less than one. Room size is simply multiplied by a delay line's output before being fed back into the input of the line.

2.4.3 Damp

Dampening provides a more rapid decay to the reverberation. If dampening is zero, the full reverb will decay at its normal rate.

2.4.4 Bypass

Bypass is GatorVerb's simplest parameter. Its only purpose is to allow the user to quickly turn off the reverb effect through the GUI. When bypass is turned on, the original input signal is allowed to pass through to the output. Otherwise, the signal is altered by the chosen algorithm before being sent out.

2.4.5 Mode

The mode parameter allows the user to choose one of the three available algorithms. It is a three-position switch. Each position chooses a different algorithm.

2.5 – Graphical User Interface

GatorVerb's GUI was developed using Steinberg's VSTGUI SDK. Much like the above mentioned *AudioEffectX*, the GUI SDK provides a main class, *AEffGUIEditor*. This class provides all necessary methods to alter the values of each parameter through the use of an interface. In addition to this class, GatorVerb also makes use of three control classes: *CAnimKnob, COnOffButton,* and *CVerticalSwitch*. As expected, these classes create an animated knob, a two-position button, and a switch, respectively.

The animated knob class takes in an image map file containing any number of identically sized images. The number of images in each file determines how many positions the knob will have. In GatorVerb, each knob has fifty positions, so there are fifty different images in a knob's image file. Each image in a file is nearly identical, but is slightly rotated from the previous in order to provide an animated feel.

The on-off button class takes in an image file that contains only two images, one for the on position and one for the off position. This class is used by GatorVerb's bypass parameter. When the button is pressed, bypass is turned on.

The vertical switch works like the animated knob. It has as many positions as there are images in its associated image file.

In order to set up each control device in the GUI, the process is relatively straightforward. The GUI has a main background rectangle, which can be blank or can contain a custom image. This rectangle determines the size of the VST GUI. Each knob, switch, etc, has its position in the rectangle set based on horizontal and vertical offsets, where offset (0, 0) is the upper left corner of the containing rectangle.

Each of these control devices also has a parameter called *tag*. Tag is the name used for each controllable parameter in the VST's main process file. For example, in GatorVerb, the Wet Mix parameter is tagged kWet. Thus, when creating the knob to control this parameter, its tag must be set to kWet.

The final step in initializing each control device is to add its *view*, which is the rectangle containing the knobs image, to the main rectangle frame. Once this is done, the device is visible in the GUI.

Controlling each parameter with the knobs and switches is a slightly less straightforward process. The *AEffGUIEditior* class provides two similar methods, *setParameter() and setParameterAutomated()*. The *setParameter()* method is actually used to set the value of the parameter in the VST plugin. However, this method cannot simply be called from the GUI. Instead, the automated version must be called. Below are the steps that occur when a parameter is to be changed. This example uses one of the animated knobs to demonstrate.

1. The knob is clicked and dragged to the desired level on the GUI.
2. After the image value is set to determine how to draw the knob, an internal variable, *valueChanged*, is set to be true.
3. When this variable is true, the *valueChanged()* method is called. This method calls the *setParameterAutomated()* method with the tag of the affected parameter.
4. The *setParameterAutomated()* method finally calls the effect's method to change the actual parameter and updates the knob's image.

As strange as this process seems, it saves the programmer from having to manually update the parameter and the knob's image. Otherwise, the programmer would have to manually set the knob's image map offset to draw the proper knob position after each parameter change.
In terms of the aesthetic design for GatorVerb's GUI, my interests in music and audio equipment were the main focus. Though the actual look of the GUI is the least important part, having an interface that is pleasing to the eye is still important. GatorVerb is designed to look like an old radio or tube amplifier head, with some unique features for switching algorithms and bypassing the plugin. The final design is shown below in Figure 2.5 on the next page.

Figure 2.5: Final GatorVerb GUI Design

3 - Results

In this section, I will discuss the results and look at some sample waveforms of affected input signals. In addition, I will look at each algorithm's result from the perspective of audio production, both in music and in sound effect production.

3.1 - Waveform Analysis

The majority of testing GatorVerb was done with one particular sound file. I happened across a dry recording of Freddie Mercury singing Queen's "Killen Queen." This provided a perfect input to test reverb algorithms, since there are no reflections at all in the original recording. Figure 3.1 shows a portion of the original waveform.

Figure 3.1: "Killer Queen" original waveform

Figure 3.2 shows the same waveform after being processed using the Feedback Delay Network with the following parameters:

- Room Size = 0.5
- Wet Mix = 0.5
- Dampening = 0
- Bypass = 0

Figure 3.2: “Killer Queen” waveform after FDN process

As we can see, there is a tremendous difference between the two. Recall that a Room Size of one equals infinite feedback, so having 0.5 is still a great deal of reverberation. Having half of the output signal equal to the wet processed input also causes much of the reverb to take over the original sound, which may or may not be desirable.

To get a better view of the reverberation, let us look at the before and after processing waveforms for a single snare drum hit. The original waveform is seen in Figure 3.3.

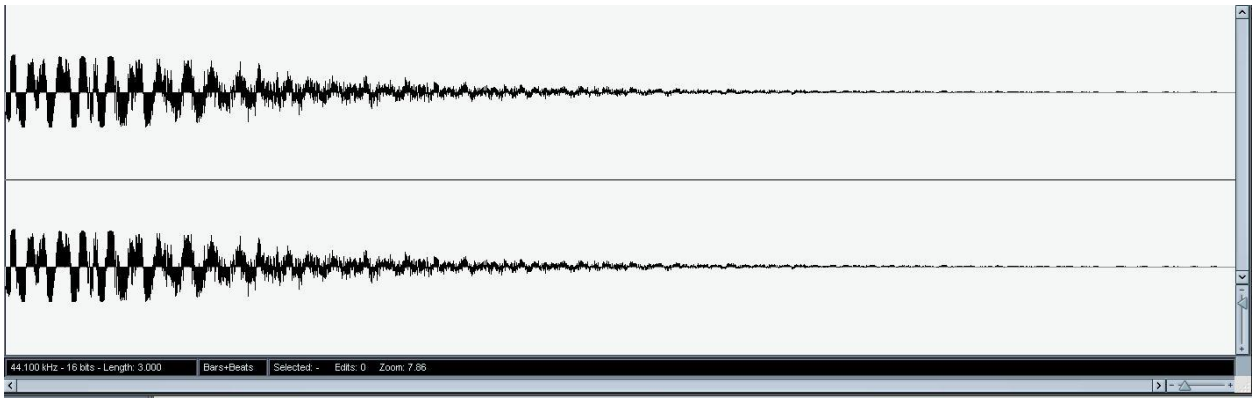

Figure 3.3: Snare drum hit original waveform

Now, Figure 3.4 shows the affected snare hit with the same parameters.

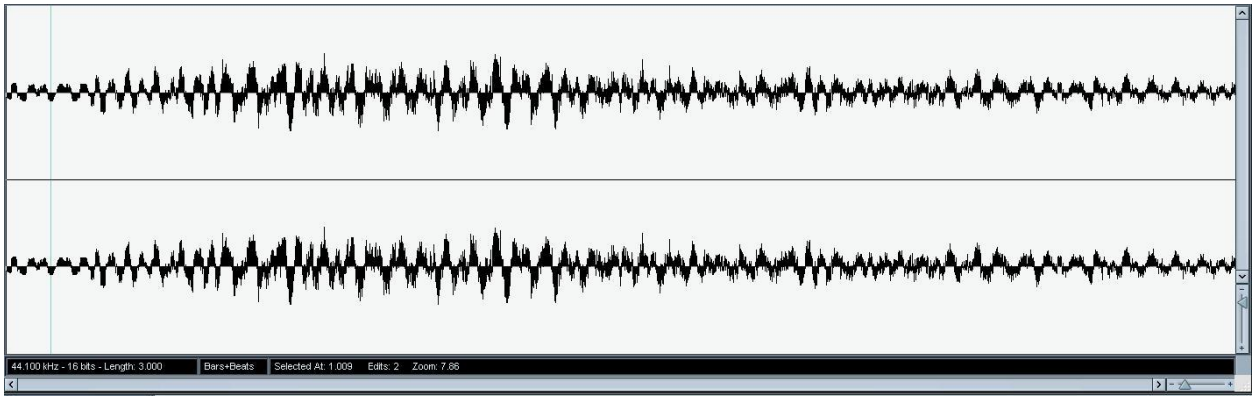

Figure 3.4: Snare drum hit waveform after FDN processing

Unfortunately, the FDN algorithm does not handle the sudden snare hit well. Due to this, the internal delay lines are much more noticeable on the FDN. However, Schroeder's original algorithm, which is mode three on GatorVerb, handles the drum hit better (Figure not shown). The internal delay lines for the filters are not nearly as noticeable, so the hit reverberation sounds much smoother.

To avoid overfilling this section with waveforms, I will discuss the algorithms further without figures.

3.1.1 – Feedback Delay Network

As stated above, the FDN does not handle the sudden impulse from the snare drum hit well, and causes the separate delay lines in the network to become noticeable. However, on vocals, as demonstrated with the "Killer Queen" sample, sound much better than expected. Typically, artificial reverb has a distinct metallic sound that can be heard when listening. Feedback Delay Networks are generally known to be one of the better sounding reverberation algorithms, but the resulting output was outstanding, considering the expectations I had. The FDN sounds surprisingly full, and could sound better by increasing the number of delay lines used in the network.

3.1.2 – Original Schroeder Reverberator

Schroeder's original algorithm, as expected, sounds good. However, when listening closely, it suffers from the metallic sound mentioned above. The reverberation tends to sound like a pipe, as opposed to an open acoustic space. The algorithm fairs well on the drum hit, but could sound better with a more efficient algorithm, and perhaps faster processing. The onboard soundcard of the laptop used for testing suffers from significant latency, so a dedicated audio interface would be much better suited to handling the signal processing.

3.1.3 – Cascading Allpass Filters

This algorithm produced disappointing results. Though it has been recorded as not being as successful as the original Schroeder reverberator or FDN, the results achieved through GatorVerb were not acceptable. The algorithm did not produce any noticeable reverb at all. Perhaps I did not fully Schroeder's theory on cascading allpass filters, but the signal comes out of the algorithm dry, as if it was never processed. Schroeder theory on using cascading allpass filters was unusual because the nature of allpass filters does not lend itself to reverberation of this fashion.

The problem with this algorithm is puzzling, so part of my future work with VST programming will be to locate and resolve the issue that is causing the algorithm not to work. Some artificial reverbs have made use of the cascading allpass filters following a FDN. This produces good results, since the filters act to equalize the signal being sent out from the network.

3.2 - Analysis from an Audio Production Perspective

The motivation behind creating GatorVerb was the fact that music is my greatest interest, so it only makes sense to look at the reverb results from the perspective of audio production, specifically, music production.

When recording music, we want to record the driest sound possible when tracking. There should be no natural reverberation or echo, so studios are specially built to be sound deadening. The interesting part of music production is that we almost universally do not want a dry sound for the final product, so we add in effects later.

Some of my thoughts were mentioned in the analysis above, so there is no need to recount them in detail.

Out of the three algorithms, the FDN produced the most pleasing results. By customizing the parameters, a very full reverb sound can be produced, especially when working with vocals. Sudden, short sounds like drums do not fare as well, but these sounds generally use a very small amount of reverb, so the FDN is still more than useable in music production.

The cascading allpass algorithm was covered entirely in the above section, and since it produced no results, I will pass on discussing it here.

Schroeder's original algorithm, though metallic, produced a well rounded decay in reverberation. This was the most impressive part of the algorithm, since a smooth decay is important when replicating natural reverb. The metallic sound is not nearly as much of an issue when the algorithm is used on one piece of a complete song, since the listener will usually not be able to clearly distinguish one track's reverb from another.

Overall, GatorVerb produced exactly what was intended; a simple to use, pleasant sounding reverb that I can use in my own music production.

4 - Future Work with VST and DSP

As I mentioned above, my first task in moving forward with VST development is to rectify any mistakes I may have made in developing Schroeder's cascading allpass algorithm.

GatorVerb, as a whole, turned out to be quite successful. Additionally, it provides room to improve each algorithm, as well as add new algorithms for a user to employ. Specifically, I would like to attempt a convolution reverb algorithm. Convolution reverb is defined in Section 1.3.1. Though convolution is generally not very customizable, it models a specific space very accurately. Due to this, a desirable room can be modeled almost as if it were natural reverb.

In addition to putting more features into GatorVerb and improving the existing ones, other audio effect plugins can be created using the knowledge of Steinberg's VST SDK gained in completing this project. A standalone delay plugin would be a simple task, since the delay lines are already constructed. A particularly challenging plugin to implement is an equalizer. This involves extensive use of filters and requires extremely efficient coding to work well. However, it provides a great deal of experience with signal processing and VST programming in general.

In terms of digital signal processing as a whole, I have a long way to go before I can be called knowledgeable. There are many theories about different aspects of DSP, many of which I have no knowledge about at all, so I have a lot to learn in the future.

5 – Conclusion

The opportunity to combine two of my main interests in life, music and software engineering, was extraordinary. In university courses, you learn the basics of what you need to know to implement anything you want, but there is no specific focus. GatorVerb allowed me to put everything I have learned throughout my time here to real use in a field I enjoy. Digital signal processing is a field that I have great interest in, but no experience working with until this project. The ability to learn something brand new while employing skills I have previously learned is always an invaluable experience. As with all projects, especially software projects, GatorVerb will evolve over time. It may gain features or improve the existing ones, but it will grow in one way or another.

Although one of the algorithms was unsuccessful, the project was still a great experience and provided a tool that I can use for my own projects. Unfortunately, until sending an actual signal through the algorithm, there was no way to know if it would work or not. Despite this one negative result, the overall design and implementation was completely successful, and will lead to more positive results as it is expanded upon.

6 - Works Cited

[1] Smith, J.O. *Physical Audio Signal Processing,* http://ccrma.stanford.edu/~jos/waveguide/, online book, accessed February 15, 2011.

[2] Larsen, Chris. *Euphoria Audio,* http://www.euphoriaaudio.com/index.php/category/audio_software/, website, accessed February 14, 2011

[3] Gardner, William Grant. *The Virtual Acoustic Room*, 1982. Massachusetts Institute of Technology, Cambridge, Massachusetts.

[4] Hind, Nicky. *Sound Processing Techniques: Effects*, https://ccrma.stanford.edu/software/clm/compmus/clm-tutorials/processing2.html, website, accessed March 22, 2011.

[5] Costello, Sean. *Schroeder Reverbs: The Forgotten Algorithm*, http://valhalladsp.wordpress.com/2009/05/30/schroeder-reverbs-the-forgotten-algorithm/, website, accessed March 20, 2011.

[6] Scavone, Gary P. *Digital Reverberation*, http://www.music.mcgill.ca/~gary/307/week3/reverb.html, McGill University, website, accessed March 20, 2011.

[7] Unminger, Frederick. *Allpass Filters*. http://www.musicdsp.org/files/filters002.txt, online text document, accessed March 19, 2011.

[8] Unknown author. *Allpass Filter Impulse Response*. http://cnx.org/content/m15491/latest/snd_reverb-schroeder-apf-impulse-response.html, Camtasia screen capture lecture, accessed March 21, 2011.