**Developing Audio Applications**

Jared Owens

February 20, 2017

**Abstract:**

Audio applications and modules are implemented in many modern hardware and software. To program these one needs an appropriate understanding of the structure of audio and how it works in the digital realm. This leads to Digital Signal Processing. It is an idea not exclusive to audio, but without it practically no digital audio would be as it is today. With Digital Signal Processing, there are many obstacles to overcome and understand but doing so allows a more cohesive audio program to be developed. Using this technology along with tools in place to aid with audio development, the developer has the capability to create numerous different audio functions and implement ideas to process, manipulate, and create audio all within the digital realm.

# Introduction

Today’s modern society revolves greatly around electronic devices, many of which have the capability to produce sound. This must come from some form of developed audio software. The focus here will be on how digital audio applications are developed

To introduce how audio applications work and are developed, some context will need to be provided, the first being what type of software is appropriate to look at to understand the appropriate context for the audio application development process. The other subject to focus on to gain an understanding of how audio applications are developed is to look at how sound works.

To better understand the process of developing audio applications the focus will mostly be kept on audio plugins. These encapsulate most of the audio functions that would be developed on any program using digital audio. Audio plugins are often used to resemble realistic audio hardware [1]. These are even getting close enough to be nearly indistinguishable from hardware units. Some of the examples these plugins emulate would include equalizers, compressors, delay units and many more. Due to this wide array of emulated hardware this is a great subject to focus on when trying to encapsulate all available digital audio functions.

# 2. Background

Audio applications come in a vast array of different types of programs [2]. This is displayed both in varying functionality and even beyond that, many applications will function the same but have vastly different implementations. The reason behind the disparity between similarly functioning applications or audio modules would be due to two factors. One is obviously how the developer(s) decides to go about creating their program. The other, and more important one is focusing on selling to an audience the same concept exhibited in a unique manor that allows different functionality. This would especially be focused in on how the GUI of the program is structured as well as more minor details that add features and affect how the application would perform. These changes may not be present in other applications but the core concept remains the same.

One classic example to point to is the vastly popular idea of subtractive synthesis. This is easily the most market dominate avenue chosen by audio engineers and producers to create sound. A large reason for this is because it follows the more common structures of the early analog synthesis machines used in the 60’s and 70’s before the digital age of audio. This has resulted in numerous variants in this type of product that at first glance may appear to be completely unrelated. However, at the core structure, it follows the same fundamental principles, in this case, subtractive synthesis, to produce the desired sound.

# 3. Properties of sound and audio

One important thing in programming audio applications is simply understanding exactly how sound and audio work. Sound is comprised of waves. As The Physics Classroom [3] explains, “A [wave](http://www.physicsclassroom.com/Class/waves/u10l1b.cfm) can be described as a disturbance that travels through a medium, [transporting energy](http://www.physicsclassroom.com/Class/waves/u10l1b.cfm#energy) from one location to another location. The [medium](http://www.physicsclassroom.com/Class/waves/u10l1b.cfm#medium) is simply the material through which the disturbance is moving; it can be thought of as a series of interacting particles.” The typical medium known for a sound wave being air is not exclusive. Most materials with the right conditions can be mediums for sound waves such as metal, water, etc.…

# 4. Digital Signal Processing and Sampling Audio

Computers are digital machines. This means that information must be broken down into discrete packets rather than an analog signal which is continuous [4]. To get audio sounding close enough to analog there are some techniques that must be used. However, there first needs to be a stronger understanding of how digital audio is represented. This is the idea of digital signal processing. This is the very fundamental of audio development and without it one would be hard pressed to build a very marketable or even useful application. That is why it is crucial to at the very least understand the core concepts that goes behind digital signal process and especially those relating to audio.

This quote by Steven W. Smith [5] shows how digital signal processing relates with computers dealing with representing realistic signals, “Most of the signals directly encountered in science and engineering are continuous: light intensity that changes with distance; voltage that varies over time; a chemical reaction rate that depends on temperature, etc. Analog-to-Digital Conversion (ADC) and Digital-to-Analog Conversion (DAC) are the processes that allow digital computers to interact with these everyday signals. Digital information is different from its continuous counterpart in two important respects: it is sampled, and it is quantized. Both of these restrict how much information a digital signal can contain.” Sampling takes the independent variable of the input signal and changes it from continuous to discrete. Likewise, quantization is the same, but instead taking the dependent variable and converting it from continuous to discrete.

## 4.1 Quantization

Quantization can afford an error of +/- ½ of the Least Significant Byte (LSB), distance between the quantization levels adjacent to each other, of the sample at most. In figure 1, the quantization error, which closely resembles random noise, is shown in section (d). This is a direct result from subtracting section (b) from section (c). Another way to look at this is to understand that (c), digital output, is realized by the continuous signal being input and adding (d), the quantization error.

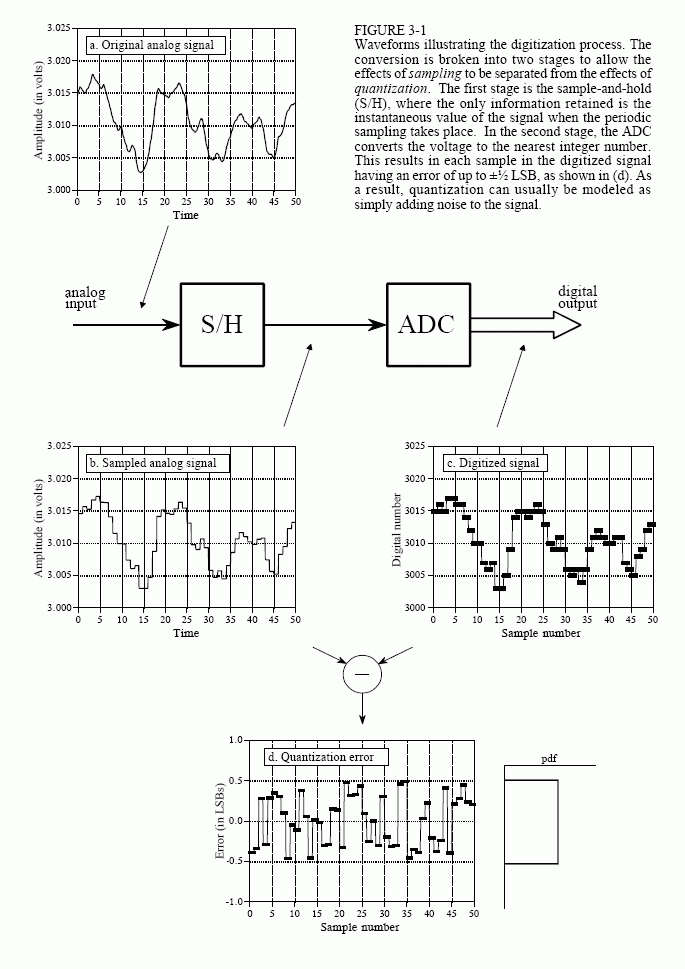


Figure : Demonstration of the electronic waveforms of a standard analog-to-digital conversion

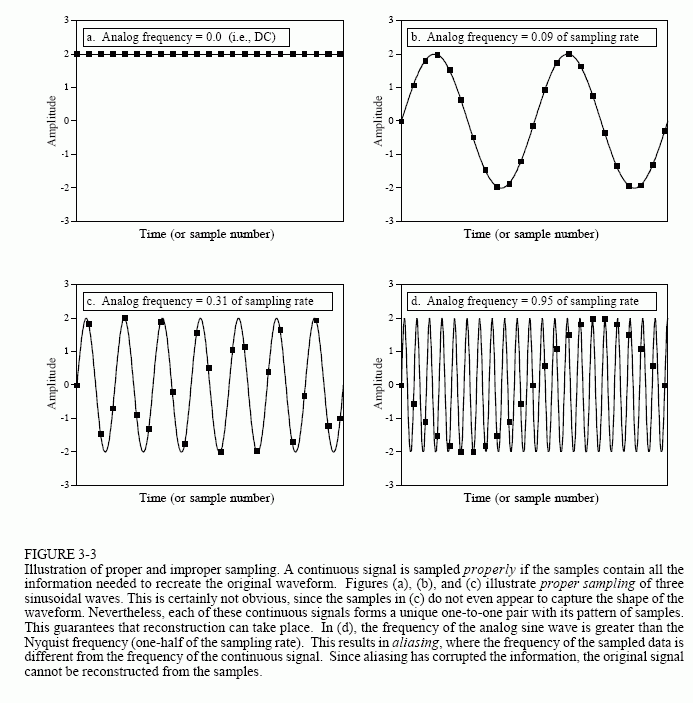
Because quantization brings about random noise through quantization error, finding ways to reduce this noise can be very beneficial. One important thing to keep in mind is that the number of bits relates to the precision of the information. The higher the number of bits the greater the precision in the information. After looking at an example afforded in *The Scientist and Engineer’s Guide to Digital Signal Processing*, going from an 8-bit system to a 12-bit system the additional noise comparing the digital signal to the original analog signal goes from around 50% to not losing any information from quantizing the signal.

To determine the number of bits one would need in a system it must be asked, how much noise is already in the analog signal and how much is acceptable in the digital signal. Even after all of this, the quantized signal still can cause problems in certain circumstances, creating distortion through thresholding the signal. To solve this one would need to consider utilizing dithering.

## 4.2 Sampling

Sampling is being able to take a signal, store it in digital format (samples), and reproduce the signal. Sampling is done properly if the signal can be reproduced exactly as it started. Looking at figure 2 it shows several sinusoids before and after becoming a digital signal. Section (a) of figure 2 shows an analog signal with a constant value (this example being DC) which results in a cosine wave of zero frequency. Because it is a straight line one could properly sample this resulting in a perfect copy when output from the digitally sampled version. Likewise figures (b) and (c) are both able to be properly sampled due to having the frequency of the sample rate in a position that is manageable.

However, example (d) is not able to be properly sampled. This is because the analog frequency is close to .95 of the sample rate with only 1.05 samples per sine wave cycle. This results in a change of .05 frequency in the digital signal resulting in a concept known as aliasing



**Figure 2: ProcessDoubleReplacing function implementation with additional comments**

## 4.3 Aliasing

Aliasing is an important concept to understand in DSP. This occurs when the sinusoid (in this case being the overall sound waves in focus) changes frequency during sampling, meaning it is improperly sampled. This ties in to the Nyquist sampling theorem, which states that a continuous sample can only be properly sampled if the sampler rate is at least twice the given frequency components (cycles/second) or hertz. If the frequencies go above this threshold the information originally there will be altered by aliasing, sending back unwanted artifacts.

In the example given by Steve Duda, [6] he demonstrates a real-world example of digital aliasing in music frequencies. The current and most commonly used sample rate being used in this demo demonstrates that the Nyquist limit is 22.5 kHz. If one would try to produce a frequency higher than the Nyquist limit, it would result in reflections, pieces of information bouncing back and adding information not wanted.

In the example given by him in figure 3, it demonstrates an audio application, Native Instruments Massive, with much less than perfect sampling because they did not write algorithms to compensate for the Nyquist theorem, especially at higher frequencies. As seen by this graph the higher peaks are the information that is supposed to be included with the fundamental (lowest key frequency) around 1kHz. However, there is much additional information (noise) that should not be there due to reflections from producing samples above the threshold acceptable by the sample rate given. It even produces information below the fundamental frequency which alters the way the sound is comprised entirely.

Compare this Figure 3 to Figure 4; there is quite a significant difference as one can easily see.

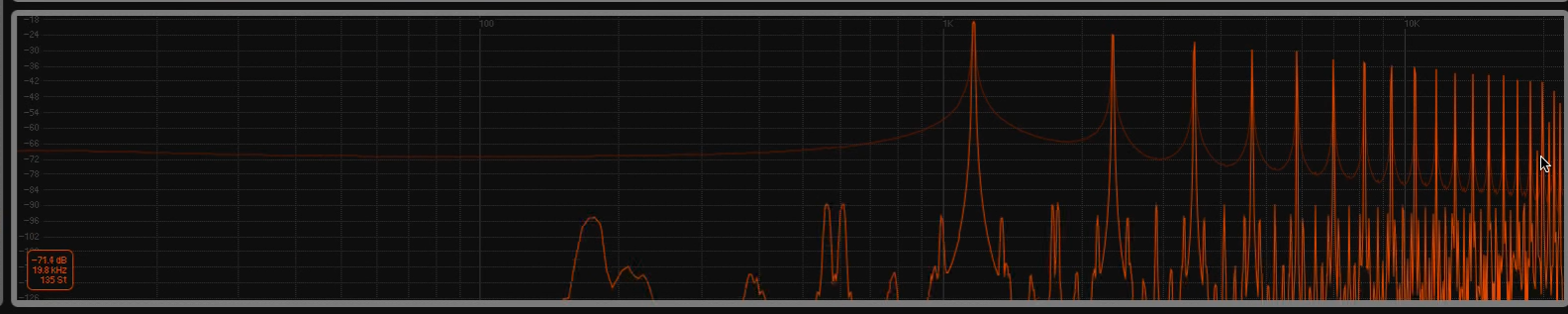


Figure 3: Audio application not compensating for aliasing

The other example given was the audio synthesizer Serum programmed by Steve Duda. This is demonstrated by Figure 4. The spectral analyzer tool used here shows the old reflections from the Massive application but these are not actually present when Serum is processing the same audio source at the same fundamental note. As one can see it is very clean because it was programmed with safeguards to sample enough to avoid conflicts and reflections resulting from avoiding frequencies going past its Nyquist theorem limit.

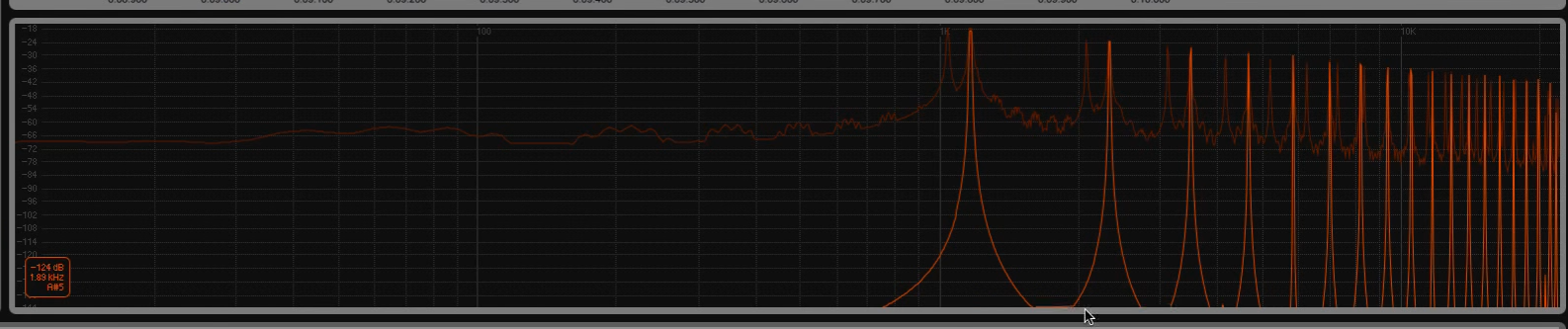


Figure 4: Audio application that is compensating for aliasing

## 4.3 Digital with proper sampling techniques compared to Analog

As pointed out by deadmau5, with audio there are things where you absolutely cannot get perfect representation of analog [7]. He states that with more oversampling, which is essentially adding more sampling capabilities artificially, you can pass through the more refined steps one can achieve in the software getting closer to representing an analog signal. However again, there is no way to get digital audio the same as analog because an analog signal is an infinite value and computers cannot because they can only run in a finite stream of data.

An example given of this is adding higher oversampling to a low frequency oscillator that is controlling frequency modulation of the filter of the synthesizer he is using. While this gives much smoother results as you increase the frequency to be higher of the LFO (Low Frequency Oscillator), it is still having to step through each different sample, just in a faster manor. In opposition to this, when using an analog machine the signal is controlled by electricity so there are no steps. The incrementation is completely continuous. This leads to a perfectly smooth source to any kind of modulation one would use.

# 5. Audio programming tools

There are many ways to develop audio applications today. Many SDKs, frameworks, and toolkits are available for use [8]. Plenty of high level development options have been given. However, staying at a lower level affords many important advantages. These can include better performance, allowing control over fine details, and allowing one to more easily manage projects over time.

With audio applications, there are many ideal scenarios to keep in mind when developing them. Being able to use it on multiple platforms, such as OSX, Windows, and Linux is very advantageous. Programming it to include 32 and 64-bit versions can be very useful, especially when integrating with other software. However, adding all these features can result in needing to program the same thing multiple times. An example of this is creating different versions, such as making a 32 and 64-bit version of the software separately. Ideally, one would only need to code it once and all these features will be available to distribute.

Keeping the goal in mind of programming once, then having it work in multiple versions on multiple platforms, one library to go to would be WDL-OL IPlug. IPlug, developed by John Schwartz, is part of the company Cocko’s WDL C++ code library. It is free to use and has several modified versions available online, one being WDL-OL IPlug.

WDL-OL IPlug will provide use to VST2, VST3, AU, RTAS, standalone applications, and even IOS applications. This covers practically all major audio application formats that can be extended also by other software. It comes implemented with plenty of examples and helper scripts. It even comes with its own Build-chain with installer scripts. It also comes with efficient classes to manage memory allocations, lists, queues, strings, and other such objects [9].

# 6. Understanding important IPlug structures

## 6.1 Core IPlug files

WDL-OL IPlug is set up so one only must edit three files and the rest of the project is managed for the developer [8]. The first file to modify is resource.h. In this file the developer would specify properties of the audio unit being programmed. The first set of information allows the developer to specify the software’s name, manufacturer name, and version number.

Next the developer can specify the options of valid channel, I/O, side-chain. The developer would also need to specify the plugin category and type. It also needs to have the unique ID and manufacturer unique ID specified. The last thing needing to be set in the resource file is whether it uses chunks. Chunks are just a way for the application to store information in a current state. Without chunks each parameter value must be stored as a single floating point number.

Resource.h also contains the applications constants, flags, and image resources [4, 7]. In the example being looked at in Martin Finke’s blog the application has a simple knob. This is accessed by specifying the specific PNG file which is a sprite sheet to cycle through various positions of the knob so it looks as if it is turning and not just rotating a static image. The other unit specified in the walkthrough at this point is the GUI width and height constants.

The next file to manipulate in creating an audio application is the plugin interface (.h) [8]. This is the interface of the implementation in which the developer will be programming specific functions. In this the developer needs to specify the member variables. The other thing to be specified is which methods from the IPlugBase the developer needs to override. The developer at least needs to change the ProcessDoubleReplacing function and the OnParamChange function.

The only other file needing to be changed is implementation (.cpp). This is the implementation of the plugin class with the constructor and destructor. In the constructor one would initialize member variables, add on parameters for the application, create the GUI and add controls, and store preset configurations to select for the application.

## 6.2 Core functions to control the audio application

ProcessDoubleReplacing is crucial to the application, this is the processor for audio input [4]. As shown in Figure 5, this takes a sequence of double type samples containing the amplitude at any given time. The double\*\* inputs parameter takes in sequences of double type samples. This is accessed by in1 which points to the samples in the left channel of audio and in2 which points to the samples in the right channel. Then doing the same with the outputs double\*\* variable in respect to out1 and out2 one can process the sample buffers by iterating over them. This is shown in the for loop in Figure 5.

The variable mGain is the overall gain amount, that is basically understood as the overall volume level. The input buffers are multiplied by this to be assigned to the respective output buffers. Lastly, the variable nFrames tells how many samples per channel to tell how long the buffers are. A helpful safeguard included in this function is that IPlug guarantees the buffers will all be valid. If the host cannot connect yet, it fills the buffers with zeros.

void MyFirstPlugin::ProcessDoubleReplacing(double\*\* inputs, double\*\* outputs, int nFrames)

{

// Mutex is already locked for us.

/\*A lock or mutex(from mutual exclusion)

is a synchronization mechanism for enforcing limits on access

to a resource in an environment where there are many threads

of execution.A lock is designed to enforce a mutual exclusion

concurrency control policy.\*/

double\* in1 = inputs[0];

double\* in2 = inputs[1];

double\* out1 = outputs[0];

double\* out2 = outputs[1];

for (int s = 0; s < nFrames; ++s, ++in1, ++in2, ++out1, ++out2)

{

\*out1 = \*in1 \* mGain;

\*out2 = \*in2 \* mGain;

}

}

Figure 5: ProcessDoubleReplacing function implementation with additional comments

The next function to focus in on is OnParamChange. This is called each time a parameter changes, which will happen quite often, so this function too is very crucial to understand [8]. Parameter changes can occur on automation calls, user editing parameters via the GUI, and loading presets that include parameter changes.

As shown in figure 6 [4] this enables the IMutexLock to ensure thread safety. The rest of the code is a switch to handle which code to call on the appropriate parameter changes. The kGain parameter is a value between 0 and 100. This is divided by 100 and assigned to the private variable mGain.

void MyFirstPlugin::OnParamChange(int paramIdx)

{

IMutexLock lock(this);

switch (paramIdx)

{

case kGain:

mGain = GetParam(kGain)->Value() / 100.;

break;

default:

break;

}

}

Figure 6: OnParamChange function implementation

The reset function is also very important to understand. It is called each time the sample rate is changed [8]. It will also be called on the star of the process. After reset is called it is recommended to call the GetSampleRate function to get the new sample rate and update the DSP dependencies correctly.

If the application being developed uses MIDI information the developer would need to consider the ProcessMidiMsg function. This is called each time the audio application receives a MIDI event. The strongest point of this function is that when MIDI events are received they are sample-accurate and come timestamped with the sample offset to tell where the event should be occurring.

# 7. Programming specific audio modules

Using the given information of DSP and how audio plugins and software can be programmed these concepts can be combined to create specific helpful audio modules [10]. The first example that will demonstrate this is creating digital reverb. Digital reverb is a way to emulate a natural ambient space using algorithms. When sound produces noise, it bounces all around the space decaying over time, eventually decaying silence. Many algorithms used to emulate this can be very complex because of the actual real-world complexity being very difficult to be modeled in digital format.

Relating reverb to DSP, it can be done relatively simply [5], the complexity comes from more realistic algorithms [10]. Starting by simply adding the audio channels together the sound will be frail as if the music was being output in an outdoor environment giving it not echo warmth and depth that an inside acoustical environment would provide. Listeners can be greatly influenced by echo and reverb, this gives the feeling of a sound in a real space so it feels familiar and correct to the listener. To do this in a simple manor in digital signal processing a delay can be programmed with a few milliseconds depending on how much depth is wanting to be perceived

Another idea is that of distortion through limiting the audio signal being processed in the application [4]. This is the idea of applying clipping to the audio signal which means limiting the threshold of the audio down so it does not have the full range of amplitude of the original signal. This squashes the audio making it lose information and pushing it down to be a more fat and tight signal. One example to look at is using a sign wave and limiting it. This will push it down more towards a square wave which has more volume giving the signal a louder feel as shown in figure 7. This is going to create a distorting effect.

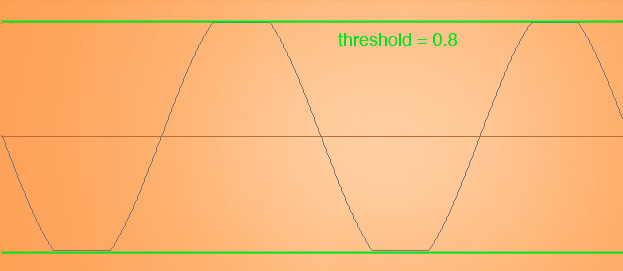


Figure 7: Limiting an audio signal to apply clipping

To program this the program would first set the threshold value. In the IPlug framework the mGain will be replaced with the desired threshold so it does not extend the amplitude to the max output level possible. Then when processing the audio signal the programmer checks the positive and negative values of the signal. If the signal is positive and is below the threshold’s max set amplitude it passes through. However, if it is above in amplitude it will pass the threshold value on and ignore the request higher value. The same is done for negative values except it goes with the negative set threshold value and only passes the threshold value instead if the signal being processed is below the set threshold level. In addition to these example audio functions implemented through programming with DSP there are numerous more examples to be utilized in a vast array of utilities and application

# 8. Conclusions

In audio programming the programmer will need to be very familiar with the chosen framework or library they chose to properly utilize its features. The features digital audio programming utilizes are built around digital signal processing. Without a good understanding of digital signal processing one would be hard pressed to create creative and unique audio software. Understanding the market and the place audio software has in the world alongside the proper programming techniques and theories for audio development, the audio developer will be able to program appropriate audio software for the desired results.

# 9. References

|  |  |
| --- | --- |
| [1] | Valve Corporation, "Steam," Valve Corporation, 2015. [Online]. Available: https://support.steampowered.com/kb\_article.php?ref=5828-WIFZ-6347. [Accessed 18 2 2017]. |
| [2] | Dawsons Music, "Dawsons," Dawsons Music, 2017. [Online]. Available: http://www.dawsons.co.uk/blog/types-of-synthesis. [Accessed 17 02 2017]. |
| [3] | The Physics Classroom, "Sound Waves and Music," The Physics Classroom, 2016. [Online]. Available: http://www.physicsclassroom.com/class/sound. [Accessed 19 02 2017]. |
| [4] | M. Finke, "Martin Finke's Blog," 10 2013. [Online]. Available: http://www.martin-finke.de/blog/articles/. [Accessed 18 2 2017]. |
| [5] | P. Steven W. Smith, "The Scientist and Engineer's Guide to Digital Signal Processing," California Technical Publishing, 1997. [Online]. Available: http://www.dspguide.com/. [Accessed 19 02 2017]. |
| [6] | S. Duda, "YouTube," Pyramid, 7 5 2015. [Online]. Available: https://www.youtube.com/watch?v=BjvrHOARC00&feature=youtu.be&t=661. [Accessed 20 02 2017]. |
| [7] | J. Z. (deadmau5), "Masterclass," Masterclass, 2017. [Online]. Available: https://www.masterclass.com/classes/deadmau5-teaches-electronic-music-production/chapters/digital-vs-analog-synths/watch?autoplay=true. [Accessed 20 02 2017]. |
| [8] | O. Larkin, "Oli Larkin blog," 14 01 2012. [Online]. Available: http://olilarkin.blogspot.de/2012/01/m4u-convention-iplug-workshop-slides.html. [Accessed 18 02 2017]. |
| [9] | Cockos Incorporated, "WDL (Whittle)," Cockos Incorporated, 2017. [Online]. Available: http://www.cockos.com/wdl/. [Accessed 19 02 2017]. |
| [10] | Retro Synth, "Retro Synth," Retro Synth, 11 06 2006. [Online]. Available: http://www.retrosynth.com/~analoguediehard/studio/effects/digitalreverb.html. [Accessed 20 02 2017]. |