Multichannel Delay VST Plugin: applying delay effect in Dolby Atmos using JUCE Framework and DSP techniques.

By

Gabriel Traini

Supervisor:

Eugenio Donati

School of computing and engineering

Audio Digital Engineering 2022-2023

UNIVERSITY OF
WEST LONDON

ABSTRACT

In recent years, rapid digital technological advancements have ushered audio into a period of significant evolution. Technologies such as Ambisonics, binaural audio, or object-based audio, which were not recently discovered and never found significant practical use in the industry, are now regaining popularity thanks to the increased accessibility provided to the public through tools like home speaker setups, soundbars, advanced multidrive headphones, and more. This thesis will delve into the realm of immersive audio, particularly in the context of music production, with the aim of contributing to its standardization within this industry. After extensive researc200h, a gap has been identified in the lack of tools for processing 3D music for producers and engineers, especially of plugins for time-based effects, which are of crucial importance in this context. As a solution, the decision was made to create a Multichannel Delay Plugin for use in Atmos Music, a new workflow made available by Dolby Laboratories. This workflow allows users, in collaboration with some of the existing Digital Audio Workstations, to create their own Immersive Music. The program has been successfully built using the JUCE framework. The project encompassed all phases of development, from research to program implementation, culminating in the presentation and evaluation of results. It also outlined the current limitations of the project and potential future work and further enhancements.

Table of Contents

Table of Figures

1. INTRODUCTION

1.1 Introduction to Immersive Audio

The pursuit of immersive audio experiences is nothing new. We've long understood that sound has a powerful impact on our emotions. Adding a sense of three-dimensionality to sound intensifies the experience even more. Over the years, various methods have been suggested to achieve this goal. Recently, these methods have gained renewed attention, thanks largely to advancements in digital technology. This drive for immersive audio isn't limited to one area. In fields like gaming and cinema, efforts to standardize immersive audio are nearly complete. In the world of movies and video games, immersive audio formats have been in use for quite some time (Justin Paterson, 2019). However, the music realm is a bit different. Despite the innovation, stereo sound still dominates. While some songs have undergone recent remixing and a few streaming platforms offer 3D versions, most tracks are still produced and recorded in stereo (Sun, 2021). The catalyst for potential change lies with Dolby, who, for the past five to six years, has offered software that lets anyone experiment with immersive music (Ciniero, 2022). Dolby Atmos technology uses the object based audio technology, but also supports channel based tracks and also take advantage of any other 3D audio format through the use of renderers and decoders. (Ciniero, 2022). This subchapter will go through the history of Spatial Audio Technologies and main relevant formats.

1.1.1 Ambisonics

Ambisonics technology, developed by British engineer Michael Gerzon in the 1970s, revolutionized immersive audio experiences. It captures and reproduces sound in a 360-degree sphere, utilizing spherical harmonics to represent sound waves' complex patterns of amplitude and phase (Sun, 2021).Unlike traditional surround sound, Ambisonics covers both horizontal and vertical directions. A microphone array captures sound from all angles, encoding it into spherical harmonics, which are divided into Ambisonics channels of different orders. Typically, third order Ambisonics (W, X, Y, and Z channels) are common. Playback systems must match the Ambisonics order used during recording, and a decoder translates Ambisonics channels into speaker feeds to recreate the original 360-degree sound field. (Nicol, 2018)

1.1.2 Surround sound (channel-based audio)

The term "surround sound" is commonly used to denote sound reproduction involving multiple loudspeakers, aiming to envelop the listener with audio from various directions. Originally coined as a marketing concept to enhance spatial appeal beyond 2-channel stereo, it has been somewhat overtaken by terms like spatial audio, 3D, or immersive sound (Rumsey, 2018). Nonetheless, it remains valuable to describe audio systems with more than two loudspeakers in the horizontal plane., based on conventional stereo principles, encompassing three or more channels while excluding height information.

The evolution of surround sound layouts and techniques traces back to 2-channel stereophony's inception, rooted in ideas from the 1930s by pioneers like Bell Labs and Blumlein (Rumsey, 2018). The spatialization of sound in basic stereo involved introducing time and level differences between channels, achieved through microphone relationships or pan pots. These techniques were expanded to cater to more speakers in surround setups, often considering the relationships between pairs or trios of channels. As loudspeakers multiplied from four to over ten, the challenge shifted to effectively distributing signals across speakers to create convincing spatial illusions. Within Channel-Based Audio (CBA), multiple sound sources are combined, typically within a digital audio workstation (DAW), to craft a final mix tailored for a predetermined loudspeaker arrangement. Playback of CBA is straightforward as the signals are pre-mixed for each designated loudspeaker. For instance, in a 5.1 surround sound system, a set loudspeaker layout is fixed, including Front Left (L), Front Right (R), Center (C), Left Surround (LS), Right Surround (RS), and a low-frequency channel (LFE). With the addition of two rear surround channels, a 7.1 configuration builds upon the 5.1 framework (Sun, 2021). According to the SMPTE ST 428-12 standard, the channel sequence for 5.1 is L, R, C, LFE, Ls, Rs, and for 7.1, it becomes L, R, C, LFE, Lss, Rss, Lrs, Rrs (Sun, 2021).This progression expands the immersive audio experience by augmenting the rear surround channels and enriching the auditory space.

1.1.3 Object based audio

The existing pre-mixed channel systems have limitations when it comes to adaptability and interaction according to user preferences. An innovative solution to these limitations is the object-audio approach. This approach involves representing media content using individual assets and accompanying metadata that defines their associations and connections. Originating from the domain of game audio, object audio has emerged as the fundamental basis for modern immersive audio frameworks (Tsingos, 2018).

At the heart of object audio lies metadata, which typically includes details about an audio signal's desired position and volume. Positional metadata employs a Cartesian coordinate system. For professional applications like the film industry, these coordinate values are standardized relative to a cube that represents an idealized cinema model. This metadata is then transmitted to the rendering side, and it's the role of the renderer to translate the object's audio position within the cube into suitable playback settings, such as cinema loudspeakers.

Object audio's rendering technique is primarily rooted in panning (Tsingos, 2018). This method orchestrates sound in a way that generates a perceptible sense of direction (azimuth/elevation) across multiple speakers or headphones. It achieves this effect by manipulating the amplitude and phase relationships of the audio signals.

Compared to object based, the conventional pre-mixed channel systems lack adaptability and interactivity. The object-audio approach revolutionizes this by using distinct assets and associated metadata to address these limitations (Tsingos, 2018).

Binaural

The widespread use of headphones has made converting immersive audio to stereo or binaural signals crucial. Binaural sound, mirroring human audio perception, originated in the 1880s for live performance transmission. Notably, in the 1930s, Bell Telephone Laboratories engineers pioneered the use of dummy heads to precisely replicate sound vibrations for distant listeners via headphones (Roginska, 2018). Binaural sound uses two audio channels to emulate human localization cues, adjusting sound with time, intensity, and spectral cues, either naturally or through signal processing. Head-Related Transfer Functions (HRTFs) apply these cues before reaching the ears, capturing sound's direction-specific spectral changes. Essential cues include Interaural Time Difference (ITD) and Interaural Intensity Difference (IID), central to source localization. HRTFs encapsulate these cues, but perceptual

studies suggest selective acoustic information use in judging direction. Room transfer functions or impulse responses represent environmental cues, encompassing acoustic characteristics between the source and receiver, influencing the overall auditory experience with factors like reflections, absorption, diffraction, and resonance (Sun, 2021).

1.2 Introduction of Delay Effect

1.2.1 History of delay effect in music

The general function of delay involves recording the input sound and playing it back with a specific time delay. Typically, the delayed sound is added to the original signal rather than replacing it, resulting in an overall effect similar to echo. The concept is analogous to reverb, with the main difference being the range of time scales available to the musician. Reverb reproduces the original sound with minimal delay, typically less than a tenth of a second, while delay can produce delayed sound for several seconds.

The delayed sound can also be reintroduced into the delay system, creating a sequence of echoes. Usually, the delayed sound is played back at a lower volume than the original, causing the echo sequence to fade over time, similar to a physical echo under specific acoustic conditions.

Echo/delay is one of the standard effects frequently used in rock music. Among the artists who have traditionally made prominent use of echo effects are Pink Floyd, who based the sound of many of their songs on echo effects, from "One of These Days" (Meddle, 1971) to "Another Brick in the Wall" (The Wall, 1979). Perhaps the most famous example is guitarist Brian May, who develops a solo exclusively based on delay in the song "Brighton Rock" by Queen. (Wikipedia, n.d.)

Brian Eno, which was re-adopted by The Edge of U2 in the album "The Unforgettable Fire." Chuck Hammer also used it effectively with his guitar synth harmonizer on David Bowie's "Ashes to Ashes" and "Teenage Wildlife" from the album "Scary Monsters (and Super Creeps)." (Wikipedia, n.d.)

The history of delay effects in music is a fascinating journey that aligns with technological advancements. It all began with magnetic tape players, where early experiments in tape-based

echo and looping effects emerged in the avant-garde electronic music scene of 1940s Europe, championed by composers like Pierre Schaeffer. (Justin Guitar, n.d.)

In popular music, Les Paul played a pivotal role, not only as a guitarist and innovator but also for his groundbreaking work in the studio. His invention of multitrack recording in 1945, achieved by modifying a reel-to-reel tape machine, marked a monumental moment in music history. Songs like 'Lover' (1948) showcased his use of tape machines to create innovative guitar textures through varying recording speeds. In 1951, 'How High The Moon' became the first instance of multi-tracked vocals, further demonstrating Les Paul's musical and technological influence.

Live performances saw a breakthrough with Ray Butts' Echosonic Combo in 1953, introducing the 'slapback' echo effect to the stage, embraced by artists like Chet Atkins and Scotty Moore.

The Echoplex, designed by Mike Battle in 1959, improved tape echo control and became popular among renowned guitarists. The Watkins Copicat, another tape delay unit from around 1958, left its mark on songs like Johnny Kidd's 'Shakin' All Over' (1960).

Roland's Space Echo, introduced in 1973 and perfected with the RE-201 model in 1974, added spring reverb and offered musicians a wide array of creative possibilities. However, these tape-based echo units presented maintenance challenges.

In the 1970s, solid-state analog delay pedals emerged with Bucket-Brigade Device (BBD) chips. The Electro Harmonix Memory Man, MXR Analog Delay, and Boss DM-1 became iconic pedals of this era.

The 1980s brought digital effects, offering precise replication of analog delays or entirely new sounds. Rack units from Ibanez, Eventide, Lexicon, TC Electronic, and Korg showcased programmable digital delay, while the Boss DD-2 (1983) brought digital delay into pedal format. (Justin Guitar, n.d.)

Advancements in digital memory gave rise to looping pedals, extending delay possibilities, and reshaping live music performance.

The journey of delay effects demonstrates how technology and innovation have continually reshaped the sonic landscape of music.

1.2.2 Delay VST Software Plugins

Starting in the 1980s, the market rapidly replaced analog delays with digital delays that digitize the incoming sound and store it in a circuit with characteristics similar to computer RAM. This technique has the advantage of virtually no component wear, allows for more faithful sound reproduction of the original, and can easily integrate with other sound processing tools through digitalization. For example, many digital delays also provide functions like flanger, phaser, chorus, or other waveform modification effects.

The natural evolution of traditional pedal delay, as with almost all other effects, involves the use of specific software that allows for the effect within a computer-based music processing program. (Wikipedia, n.d.)

VST plugins are typically used within digital audio workstations (DAWs) to enhance audio production capabilities. While there are standalone plugin hosts that support VST, most users employ them within a DAW. VST plugins come in two primary categories: instruments (VSTi) and effects (VSTfx), although there are other types like spectrum analyzers and meters. VST instruments receive MIDI data as input, translating it into digital audio output. On the other hand, effect plugins take digital audio input and process it before producing audio output. Some effect plugins can also accept MIDI input for tasks like tempo-synced modulation. MIDI messages can control parameters in both instrument and effect plugins.

A Delay VST plugin, or Virtual Studio Technology delay plugin, is audio processing software designed to replicate and create delay effects within a digital audio workstation (DAW) or music production software. They bring versatility and precision to the art of creating delay effects, both classic and experimental. (Wikipedia, n.d.)

One of their notable features is their ability to emulate various delay types, including analog tape, and digital delays. This versatility enables music producers to choose the specific character of delay that complements their creative vision. These plugins seamlessly integrate with digital audio workstations (DAWs), becoming an integral part of the modern production workflow. With easy integration, users can effortlessly add and manipulate delay effects within their projects, making experimentation and creativity more accessible.

1.3 Project's aim and objectives

The release of the software package by Dolby can certainly be regarded as a turning point, as it facilitates access to the Atmos workflow, enabling users to experiment with a simple setup comprising a laptop, a digital audio workstation (DAW), and a pair of headphones, albeit not for a complete Dolby Atmos experience. (Laboratories Dolby, 2018)Nonetheless, beyond the release of the Plugin along with its accompanying 3D panners, which can position various mono and stereo tracks within a three-dimensional space, the development of specialized tools for immersive audio production and mixing is undoubtedly progressing at a relatively gradual pace (Leonard, 2018).

In the context of digital audio workstations (DAWs), by the term "tools" it primarily refers to VST Plugins – encompassing a range of advanced effects, equalizers, and dynamic processing tools designed to align with the Dolby Atmos workflow. Moreover, the significance of time-based effects like reverb and delay cannot be understated. (Justin Paterson, 2019)The absence of innovative and novel plugins in this domain would indeed pose a considerable constraint for producers, engineers, and even beginners eager to explore this realm. (Leonard, 2018)

The aim of this project is to conceive a Multichannel Delay Effect plugin. This plugin aims to take a stereo track within an Atmos mix, which could function either as part of the bed or an object and apply the effect to a 7.1.2 format group-track employed as the bed. In doing so, it contributes to the expanding toolkit for immersive audio production, filling a vital gap and addressing the distinct needs posed by the Dolby Atmos ecosystem. For the realization and implementation of the program, the choice fell upon the JUCE framework. The concept behind this Delay Plugin is to adapt commonly used delay effects in music—such normal echo Delay, Ping Pong Delay, and Stereo Delay (Imagine Line, n.d.)which—within a three-dimensional context. The aim is to create a tool that enables the application of these effects into an Immersive Music Production context, in a simple and direct manner, making it accessible for beginners, in order to promoting the standardization of Immersive Audio in music production. In terms of complexity, the software doesn't introduce novel delay algorithms and can certainly undergo further refinement. Additional control parameters, for instance, could be integrated. However, this will be discussed in the "Future Works" chapter.

Below are listed the project objectives:

1. Achieving the 7.1.2 output configuration of the Plugin and gaining access to the 10 channels of the group track (Up-mixing) using the JUCE libraries.
2. Adapting the algorithms of standard stereo delay to accept stereo input as source track and direct the output signal to the ten channels, establishing which channels to involve and how to utilize them effectively in a creative manner.
3. Incorporating functionality to switch between the three delay types and to select the desired input track plus more functionality (Equalization, etc.), while also designing and implementing the plugin's graphical user interface (GUI).
4. Testing the plugin within the Nuendo DAW using the Dolby Atmos Renderer and assessing the achieved outcomes.

2. LITERATURES REVIEWS AND RELATED WORKS

2.1 Up-mixing techniques

In up mixing, the goal is to generate surround sound using more channels than are present in the source material. The process of converting audio content between different spatial formats, known as up-mixing and downmixing, involves using matrices or algorithms to achieve the transformation. Many of these systems for up-mixing have been driven by the consumer demand to experience multichannel surround sound from older content (Rumsey, 2018): in the context of music, it refers to the process of converting a finished mix with a low number of channels or speakers to a higher number. Considering for example a process like stereo-to-5.1 up-mixing, in this procedure, a finished two-track stereo mix is transformed into a 5.1 surround sound format. This transformation involves various techniques such as replication, separation, or creation of audio data. The goal is to distribute the sound content into the different speaker channels that exist in a 5.1 setup, even if the original audio didn't have this spatial arrangement. (Sun, 2021)

This is often done by extracting ambiance from the difference information between left and right channels and using it to drive rear channels. Directional steering techniques are applied to enhance stereo separation in the rear channels. This is what Carlos Avendano and Jean-Marc Jot tried to propose on their paper “A Frequency-Domain Approach to Multichannel Up-mix” (2004). In their paper, the authors explore techniques for transforming stereo audio recordings into multichannel audio. It uses an analysis framework based on comparing the short-time Fourier transforms of left and right stereo signals. This framework employs measures to identify ambience components and panning coefficients of various sound sources in the mix.

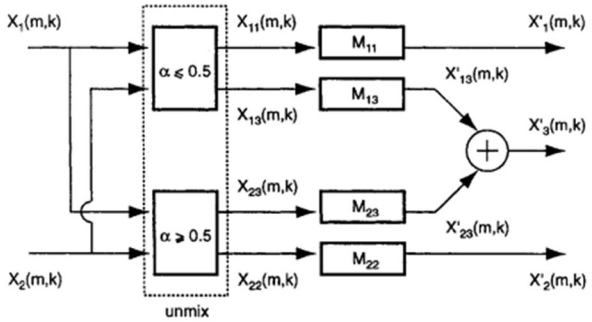


Figure 1: 2-3 Upmix block diagram, sources extraction and redistribution

By Ambience Extraction and Synthesis, the ambience signal is extracted for the generation of surround (or rear) channels, and then all pass filters ($G_{(z)}$ in the picture), with different phase characteristics, are used to reduce their correlation with the ambient components in the front channels. This helps minimize the formation of phantom image on the sides (Avendano, 2004). To prevent audio localization issues, a delay (z^{-D}) is added to surround signals.

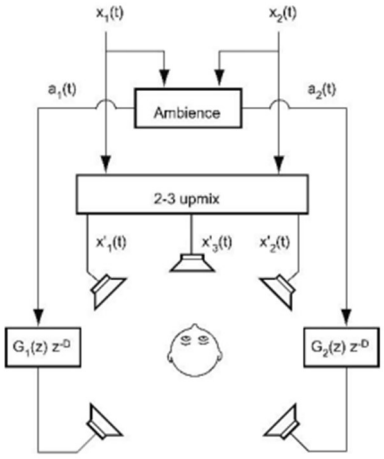


Figure 2: Stereo to 5 Upmix block-diagram

The front channels are generated by extracting the center-panned signal and subtracting it from the left and right channels, preserving stereo quality for listeners in the ideal position. However, the project faces challenges with side-panned sources collapsing off-axis, requiring further investigation. Although informal listening tests confirm effectiveness, errors due to signal overlap in time are inevitable. Panning sources with delay, especially with noncoincident microphone pairs, introduces complex relationships between Short-Time Fourier Transforms (STFTs) compared to the simpler amplitude panning case (Avendano,

2004). Aalternativelly, portions of front sound might be placed in the rear channels with appropriate delay and filtering to prevent unwanted effects.

Other solutions, that aims to address issues with the real time implementation of a stereo to multichannel to improve audio realism, has been studied from the authors Chan Jun Chun, Yong Guk Kim, Jong Yeol Yang, and Hong Kook Kim in their work "Real-Time Conversion of Stereo Audio to 5.1 Channel Audio for Providing Realistic Sounds" (2009). They review four methods for decorrelating the stereo signal for converting it into 5.1 channel audio. These methods include passive surround decoding, least-mean-square based up-mixing, principal component analysis based up-mixing, and adaptive panning. They then implement a simulator to test these methods and conduct a MUSHRA test to compare the quality of the up-mixed 5.1 channel audio with the original stereo audio.

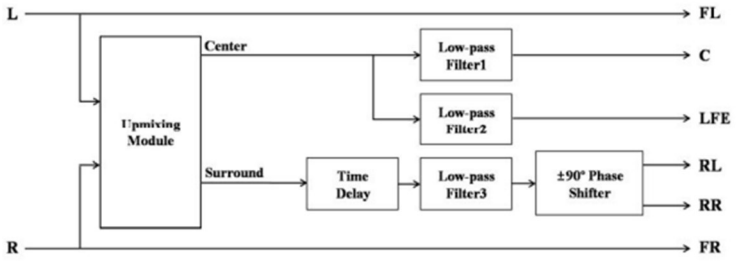


Figure 3: Complete block diagram of Up mix method with processing of each channel

Similarly, to what was done by Carlos Avendano and Jean-Marc Jot, phase rotation was implemented for the rear channels, along with a delay, as shown on figure 3. In this case, a low-pass filter was applied to the center channel, as it is often used for vocals, both in movies and songs. Another filter was used for the LFE channel to emphasize low frequencies below 200Hz, and finally, a third low-pass filter was used to simulate a high-frequency absorption effect. Here below are shown the equations of one of the fours methods of decorrelation, the passive surround decoding method

$$\text{Center}(n) = (x_L(n) + x_R(n))/\sqrt{2}$$

Equation 1: Centre signal upmix equation

$$\text{Surround}(n) = (x_L(n) - x_R(n))/\sqrt{2}$$

Equation 2:Surround signals upmix equation

where x (n) L and x (n) R represent the left and the right samples at the time index n, respectively.

Certain algorithms in home cinema and surround systems introduce "effects" to conventional stereo to simulate a surround impression. These effects, such as "Hall" or "Jazz Club," add reverberation on top of existing audio, altering the acoustic characteristics of the original recording. This approach has been taken from Sebastian Kraft, Udo Zölzer, “Stereo signal separation and up-mixing by MID-SIDE decomposition in the frequency-domain” (2015). The authors presents a algorithm that estimates azimuth directions in a stereo signal, enabling the separation of direct and ambient signal components and the remixing of the original stereo track. This method uses a band wise mid-side decomposition in the frequency domain for efficient processing, delivering a high-quality surround experience with low computational costs in a stereo-to-five-channel up-mix. They added more ambience using a Large Hall impulse response reverb from a Bricasti M7 stereo reverb unit.

However, subjective experiments have shown that many 2-channel materials experience a degradation in front image quality when converted to 5-channel reproduction. This could result in a narrower image, altered perceived depth, or loss of focus. Adrian Tregonning1 and Bryan Martin tried to face this issue in their paper “The vertical precedence effect: Utilizing delay panning for height channel mixing in 3D audio” (2015); the study focuses on understanding psychoacoustic cues for 3D sound reproduction, with a specific emphasis on the vertical aspects of acoustics and psychoacoustics, which are crucial for 3D audio. The research explored vertical inter-channel time differences (ICTDs) in the Auro-3D 9.1 loudspeaker setup and found that these ICTDs play a significant role in creating perceived audio images. The study revealed that a 5 ms threshold is important for achieving maximum source elevation perception. Beyond this threshold, elevation effects become less pronounced, but ICTDs notably increase both the width and vertical spread of phantom audio images. The findings from this research can be applied to enhance the creation of immersive audio content.

In conclusion, up-mixing and downmixing involve converting audio content across spatial formats, each with its challenges and benefits. While up-mixing can enhance spatial qualities, careful consideration is needed to maintain front image quality.

2.2 State of the art: Immersive Music production tools and mix techniques

2.2.1 Importance of effects in Immersive Music

In the realm of immersive audio, engineers and content creators have a wide range of creative possibilities for designing audio mixes and environments. We're constantly pushing the boundaries of cutting-edge immersive audio technology, giving us the freedom to experiment and innovate. Much of the work in this field relies on the same foundational skills that make engineers successful in traditional mono, stereo, or surround sound production. These skills include meticulous attention to detail, preserving the natural qualities of sound sources, and maintaining an appropriate dynamic range, regardless of the audio format (Leonard, 2018).

According to “Producing 3-D Audio” (2017) of Justin Paterson and Gareth Lewellyn, expanding audio into 3D offers creative possibilities but comes with challenges. Composers can embrace this medium by adding new instrumental layers, capitalizing on the newfound spatial canvas. Alternatively, sound engineers can employ mixing techniques to expand the original sonic space subtly, preserving the composition's core. Decorrelation can be achieved through slight variations in EQ settings, delay times, or pitch. Reverb systems and Impulse Response convolutions can also help. Time-based effects, like delays and echoes, can enhance spatial modulation over front-back and median planes. (Paterson, Llewellyn, 2019)

In fact, the use of delays and phase shifts can also position sounds effectively in 3D space, especially considering the precedence effect. The surrounding ambience might need adjustments, particularly for moving sources. Striking a balance between these effects and the overall mix requires artistic judgment and technique. Smaller room settings with short early-reflection times and shorter reverb tails often work well when excessive reverb might disrupt the mix's cohesion. (Paterson, Llewellyn, 2019)

Beyond these fundamental aspects, there are countless creative opportunities to explore. Notably, as larger immersive audio systems, especially those incorporating height elements, become more prevalent, the role of reverberation becomes increasingly significant. (Leonard, 2018) To create a truly authentic three-dimensional auditory experience, it's

essential to replicate lifelike ambient sound that complements and enhances the core audio. Engineers face limitations when reproducing recordings originally intended for mono playback in a mono-object playback system (Paterson, Llewellyn, 2019). To create a convincing placement of audio objects within a periphonic system using amplitude panning, it's crucial to incorporate realistic early reflections. These early reflections play a significant role in how our brains perceive the source's location. However, generating satisfactory reflections for object-based systems has been a challenge. Until recently, the computational demands of creating scalable reflection and reverb systems often outweighed their practical use. Additionally, this approach could negatively impact the overall sound balance and spectral characteristics in different listening environments. Philip Coleman, Philip J. B. Jackson, Luca Remaggi , Frank Melchior, Richard J. Hughes tried to address this challenge in their paper "Object-Based Reverberation for Spatial Audio" (2017). They proposed a method for achieving object reverberation, against the current common approach, which is to use a set of (localized) objects in conjunction with channel- tracks containing the ambience or reverberation; authors state that this method of reverberation has inherent limitations that can be categorized into three key aspects. First, the spatialization of the reverberation is fixed, assuming an ideal playback setup, which restricts adaptability to various environments. Second, any dialogue spoken within this setup becomes an integral part of the reverberation, making it challenging to separate speech from the ambient sound. Lastly, control over the level of reverberation is constrained to a basic wet/dry mix, limiting the ability to fully exploit the advantages of object-based audio. The proposal introduces a framework called the Reverberant Spatial Audio Object (RSAO) that generates reverberation within an audio object renderer. An example implementation using this framework is presented, backed by listening test results. These results demonstrate that the approach effectively preserves the perception of room size compared to convolved audio, and adjustments to RSAO parameters can modify the perceived room size and source distance. However, their project showed to have many limitations, and authors say that much research has still to be done.

2.2.2 Plugins for Immersive Music

Currently, the common way to apply effect in a object based system, remains to allow reflections and reverb to be generated in (channel-based) beds, whilst the objects themselves are rendered as 'dry' objects at playback (Justin Paterson, 2019) (Leonard, 2018).

Furthermore, as expansive immersive systems become more common, we can expect the development of advanced three-dimensional reverberation tools for studio use. Some interesting 3D reverbs has been recently released in the market. Liquid Sonics released Cinematics Room, which allows the user to design convincing rooms, and supports multichannel tracks, up to 9.1.6 Atmos (Liquid Sonics, n.d.), and Lustrous Plates, a Plate reverb which can be used in the Atmos Bed tracks as well (Sonics, n.d.). Izotop also released Symphony and Stratus reverbs while Fiedler Audio has Spacelab Interstellar (Fiedler Audio, n.d.). Two other plugins that deserve to be mentioned are “Hornet Samp” and “Spheric Immersive Limiter&Compressor” by Waves Audio: the first one has been designed on purpose for Dolby Atmos, and it addresses the challenge of not having a Master Bus within the workflow. This plugin combines four crucial master bus tools into one interface, including an equalizer, compressor, clipper, and limiter. These tools are interconnected, so any modifications you make to one of them will be instantly mirrored in all the others. This streamlines the process of fine-tuning your overall mix, making it more efficient and less time-consuming (Wagner, 2022). “Spherix” instead, is a plugin package comprising two tools: a compressor and a brickwall limiter, meticulously crafted to enhance the audio quality of 7.1.2 and 7.1.4 bed channels within immersive audio configurations like Dolby Atmos. Spherix operates seamlessly in Multi Mono mode, facilitating rapid adjustments without requiring channel swapping. Moreover, it empowers you to process speaker zones collectively by organizing them into channel groups, streamlining your workflow (Waves, n.d.). The use if of Ambisonics Effect Plugins is also a technique currently in use, thanks also through the use of Encoder and Decoder tools, capable of convert the Ambisonics format to any speaker array setup. Richard Furse’s Blue Ripple toolset (2017) has an interchange plugin which can transform an Ambisonics mix to a 20 channel A-format that allows the use of standard stereophonic-type effects processing, followed by a re-encode to HOA.

However, the current state of immersive audio still presents challenges, primarily stemming from the lack of specialized production tools. Advanced tools for tasks just mentioned like reverberation, delay, panning, and multi-channel equalization and dynamics processing are being slow to evolve and reach content producers (Leonard, 2018). This can lead to slower workflows for engineers, confirming the concerns of those who believe immersive audio projects are costly and time-consuming. Until this integration occurs, it's crucial for successful immersive audio content creators to be resourceful and inventive using the existing toolset. By mastering the current techniques, also mentioned in the previous

chapters of this paper, high-quality results can be achieved with minimal reliance on additional processors or specialized production tools (Leonard, 2018).

The next sub'section will mention some of the last plugins released and show techniques currently used for achieving delay effect in Immersive audio, especially, in a Dolby Atmos project.

2.2.3 Immersive Delay Plugins

In specific regard to the delay effects plugins, the subject of this project, it appears that there is a lack of literature addressing this topic. Simultaneously, the availability of Delay plugins designed for Immersive Music is extremely limited, if not non-existent. As mentioned in the previous chapter, for this effect, the common practice is to create a bed channel containing a delay plugin used for Stereo mixes, separate from the affected object, which is left "dry." However, this approach can become cumbersome when attempting to create more intricate effects or involve multiple speakers. The only plugin available in the market that allows for this and supports multichannel tracks up to a 7.1.2 setup, which can serve as a bed channel for Atmos, is Slapper Delay.

Slapper is a versatile multi-tap delay plugin designed for multi-channel audio. with the release of the 2.0 version. The first release was introduced in 2015 and remains the sole multi-channel delay plugin currently on the market. Slapper offers eight distinct delay units, each with its own separate controls, including the ability to position them within a three-dimensional audio space (refer to the image for details).

While Slapper's user-friendly interface effectively manages the complexity of dealing with eight delay units, finding the desired effect can still be challenging due to the abundance of presets, especially for a beginner. Many of these presets are geared towards replicating various acoustic environments.; this emphasis on environmental presets stems from the plugin's initial release, which primarily targeted film post-production for Surround 5.1 and 7.1 audio formats (Magazine, 2016). It's worth noting that when Slapper was introduced in 2015, there were no software options from Dolby for accessing the Atmos workflow, and digital audio workstations (DAWs) had yet to support the surround formats with height channels.

Nevertheless, Slapper is a versatile tool that can be used to achieve various delay effects and can be effectively employed in musical contexts, delivering impressive outcomes.

Another way to achieve delay effects in immersive music context is by using ambisonics decoder plugins, as discussed in previous chapters of this document. One of these plugins is called AB Delay. This plugin is natively compatible with mono, stereo, and Ambisonics signals. It features four delay units, each of which has independent low-pass and high-pass filters, feedback controls, and delay time settings, all equipped with Spatial Transformation for three-dimensional delay positioning. Well, this plugin can become compatible with various multichannel surround formats by using a Decoder Plugin, such as the AB decoder. The plugin is compatible with any Ambisonics order and decodes in all surround configurations like 5.1, 7.1, and Atmos bed configurations such as 7.1.2, up to 9.2.6. The algorithm will distribute the signal across all channels of the bed channel. With the four independent delays, which can be used either in series or parallel and with possibility to three dimensional pan them, allows for many creative choices.

2.2.4 Dolby Atmos

Dolby Atmos is the latest surround audio technology developed by Dolby Laboratories, introduced since 2012. According to Dolby's website, it 'enhances entertainment across all consumer devices.' It goes beyond conventional listening, immersing the audience in a revolutionary spatial audio experience that captivates them more deeply, enabling heightened sensation and stronger emotions. It offers 'a richer, multidimensional sound experience and helps you connect more intensely with your favourite shows, games, and music, with greater depth, clarity, and detail than ever before.'

This concept translates into the ability to 'position and move sounds in a three-dimensional virtual space with the introduction of "audio objects"' with greater precision than traditional surround sound. Additionally, with the incorporation of overhead speakers, a sonic sphere envelops listeners, immersing them in the consumption experience. 'Dolby Atmos automatically optimizes content based on the number of speakers in the device or environment, delivering the best performance for cinema, as well as enabled mobile devices, home theatres, soundbars, and headphones (Ciniero, 2022).

The Dolby Atmos workflow inherits from traditional standard approaches of older surround technologies, utilizing both object-based and channel-based concepts, introducing in addition to the previous surround formats, the height channels; Atmos introduces two innovative concepts:

1. Bed: Static sound sources anchored to a predefined speaker configuration. Adding two overhead speakers allows various setups up to 7.1.2, enabling the instantiation of up to 10 audio beds.
2. Audio Objects: Individual sound sources that can be positioned dynamically anywhere in the 3D listening space, independent of beds. These objects have metadata that allocates them based on X, Y, Z coordinates and the number of available speakers in the room. Up to 118 audio objects can be instantiated.

The workflow commences with the Digital Audio Workstation (DAW) used to record, mix, and master the content. The DAW needs Dolby Atmos compatibility, such as Pro Tools Ultimate1 and Nuendo2, which offer native integrations to facilitate smoother routing and panning of sources.

Subsequently, the 7.1.2 bed, audio objects, and metadata are directed from the workstation to the Dolby Atmos Renderer, software used for monitoring, recording, and playback of produced content. (Ciniero, 2022) The Renderer processes audio and metadata based on the speaker configuration used as monitors (actual speakers or headphones).

2.3 Reflections

In this chapter, it has been examined the issue of up mixing stereo sources into a multichannel format in 2.1 and the typical challenges that arise, since one of the objectives of this project is to convert a stereo track into a 7.1.2 format. As it is studied, the main challenges primarily involve signal decorrelation and extraction of the signal and after that the redistribution of it across the numerous available channels. Through this research, we have come to understand that decorrelation and extraction issues are primarily related to adapting a finished and mixed stereo track to a playback system with multiple channels. However, when it comes to this plugin, its nature as a delay is to create echo effects, typically for a single sound or multiple sounds. Therefore, concerns such as decorrelation and the extraction of ambiance and main sources take a back seat, while the main concern consists on the redistribution of it after processing. Nevertheless, they have still served as inspiration in utilizing the various channels to understand their role and the effects they can produce, such as which channels use for time Offset (see chapter 3), how to use height channels, and how filters and reverbs for specific channels may enhance the Immersive feeling (see chapter 5).

2.2 helped to understand the importance of effect for the new way of mixing Immersive Audio and to individuate the gap in the availability of plugins and software tools for applying effect processing in immersive audio, in particular of Multichannel Delay Plugins which up-mix a stereo source and output it to a multichannel track in Dolby Atmos.

3. METHODOLOGY

The proposed project heavily relies on the JUCE framework, an open-source C++ framework known for its cross-platform compatibility and widely used for its Graphical User Interface (GUI) and audio plugin libraries. The following sub-sections in this chapter will delve into GUI and DSP development, providing insights into the usage and development of the up-mixing method and the implementation of the delay algorithms and some other signal processing function within the context of Music Production.

3.1 Graphical User Interface (GUI)

The interface is straightforward and user-friendly, featuring typical parameters found in a standard stereo delay plugin. At the top left, there's a combo box that allows the user to select their desired delay type. The "normal" delay includes the usual control parameters for delay, such as delay time, feedback, and mix, presented as knobs. Additionally, there are input and output gain, as well as high pass and low pass filters accessible via knobs.

If the user chooses to utilize the MidSide algorithm, the interface remains the same, with identical controls, plus the Offset button; with the PingPong choise, the Balance button appears.

In the Editor Processor JUCE allows the performance of the creation of the GUI, and all its variables and functions for generating each component, in this case rotatory sliders for all the delay parameters and high pass and low pass filter, vertical slider for Input and Output gain and the combo box for the type of Delay choise. The class AudioProcessorValueTreeState is used, for attaching sliders to values of parameters. In the process block it is ensured that the value of each parameter is read in real time. To ensure that the GUI changes depending on the selected type of effect, an if statement is used within the "paint()" function. The subsequent sections will delve into the development the Up-mixing method, the two delay algorithms and the other functions and parameters for manipulating the affected signal.

3.2 Up-Mixing

The first objective of the project is the up-mixing of the stereo track. The Plugin is supposed to work on a 7.1.2 track, where it outputs the affected signal; the track that aims to affect is an object of a Atmos mix, which is normally a stereo or a mono track. Therefore, the first thing that has been done is the creation of a stereo bus for the input and the creation of a ten channel bus for the output. For that purpose, the AudioChannelSet JUCE Class has been explored and for the input, has been left the static member function AudioChannelSet::stereo(), from the default JUCE template for VST Plugin, while for the creation of the Output bus, has been used the AudioChannelSet::Create7point1point2(). This function generates the default 7.1.2 Atmos bus with 10 channels: (left, right, centre, leftSurroundSide, rightSurroundSide, leftSurroundRear, rightSurroundRear, LFE, topSideLeft, topSideRight). Also, it is needed to ensure that the layout is supported within the host: it is used an "If" condition:

```
if (layouts.getMainOutputChannelSet() == juce::AudioChannelSet::create7point1point2()){
return true; }

    else { return false; }
```

The project involved several algorithm experiments, and the final implementations were selected based on criteria like efficiency, complexity, and time limitations.

The approach taken for up-mixing was quite straightforward, considering the available time and the specific type of processor that needed to be developed. Since the plugin in question is a Delay Effect, many of the up-mixing challenges studied in chapter 2.1 become less significant. Given the nature of the plugin in question, which is to create echo effects, issues like source extraction have been set aside, also due to complexity so the original signal is left as unchanged. Each delay algorithm will then decide how to use the channels differently by introducing time offsets, differences in amplitude, and the distribution of dry and wet signals.. That being said, the approach for both delays is nearly identical: for the first delay, the signal used for all channels is the same but reduced by 3dB by dividing the input signals by the square root of 2 (equation 1 and 2), while for the second delay, the same signal is used for all channels.

3.3 Normal Delay Algorithm implementation

For the initial delay algorithm, the objective was to adapt a standard delay to function within a 10-channel output configuration. With a focus on simplicity and user-friendliness, this effect is designed to introduce delay into the various channels of the bed channel, and their usage is predefined. However, the user retains control over essential parameters such as Delay Time, Feedback, Mix, as well as input gain, multichannel output, and high pass and low pass filters.

The signal is replicated and delivered with slight intensity variations to the remaining channels. This distribution is facilitated through the utilization of a for loop to assign the input signal across all nine channels.

The first input signal serves as the input for front right, surround left, rear right, and top left,

```
{
    for (int channel = 0; channel < 10; ++channel)
    {
        float* inputData;
        if (channel != 3)
        {
            //Duplicate original stereo channels to multi-channel
            if (channel == 1 || channel == 4 || channel == 7 || channel == 8)
            {
                // Left Channel Data
                inputData = buffer.getWritePointer(0);
            }
            else if (channel == 0 || channel == 5 || channel == 6 || channel == 9)
            {
                // Right Channel Data
                inputData = buffer.getWritePointer(1);
            }
            else
            {
                // Center Channel Data
                inputData = buffer.getWritePointer(0);
            }

            float* channelData = buffer.getWritePointer(channel);
            float* delayData = delayBuffer.getWritePointer(channel);
            localWritePosition = delayWritePosition;
```

Figure 4: Upmix. Copying input data to all 9 channels

while the second input signal is dedicated to the remaining front left, surround right, rear left, and top right channels. The use of one signal for one group and the other signal for another group is responsible for the difference in amplitude between the two groups. The choice of

which channels to group together was made with the aim of creating the greatest sense of movement possible, through the sum of all channels.

Following this, pointers for each channel are established: a set of pointers for the main buffer and another for the delay buffer.

```
for (int sample = 0; sample < buffer.getNumSamples(); ++sample)
{
    const float in = (1/sqrt(2))*inputData[sample];
    float out = 0.0f;

    float readPosition = fmodf((float)localWritePosition - (currentDelayTime) + (float)delayBufferSamples, delayBufferSamples);
    int localReadPosition = floorf(readPosition);

    if (localReadPosition != localWritePosition)
    {
        float fraction = readPosition - (float)localReadPosition;

        float delayed1 = delayData[(localReadPosition + 0)];
        float delayed2 = delayData[(localReadPosition + 1) % delayBufferSamples];
        out = delayed1 + fraction* (delayed2 - delayed1);

        channelData[sample] = in*(1-currentMix) + (currentMix * (out - in));
        delayData[localWritePosition] = in + out * currentFeedback;
    }
```

Figure 5:Delay Algorithm processing

During this phase, the input signal undergoes a 3dB attenuation to maintain consistent loudness levels (Kraf, 2015); this is because with this delay mode, more speakers plays sound at the same time. Another for loop is executed, this time iterating through each sample, to generate the wet signal. The "read position" variable contains information regarding the position within the delay buffer where the "writing" process occurs. This position is dynamically adjusted by the "delay time" variable. Essentially, the delay time corresponds to the temporal interval within which the dry signal and its subsequent repetitions occur. Finally, we arrive at the output equation for the main buffer, enabling the adjustment of dry and wet signal levels via the "mix" parameter. Additionally, there is an equation governing the output into the delay buffer, which is responsible for producing the desired "echo" effect. The quantity of this echo is modulated by the feedback parameter.

3.4 Mid Side Delay Algorithm

This algorithm is slightly more advanced than the first one. It introduces the "Offset" parameter, allowing the user to alter the delay time between the central, right, and left zones. Through this line of code, as soon as the user activates the control, they will add (or subtract) delay time to the right speakers (or left speakers). Compared to the previous algorithm, there is more control over the use of speakers, and the user will be able to create more interesting solutions. To enable individual adjustments on each channel, a different approach was employed compared to the first algorithm, where each channel was used in the same way, and this process was handled by the for loop.

```
float* leftchannelData = buffer.getWritePointer(0);
float* rightchannelData = buffer.getWritePointer(1);
float* centerchannelData = buffer.getWritePointer(2);
float* surroundleftchannelData = buffer.getWritePointer(6);
float* surroundrightchannelData = buffer.getWritePointer(7);
float* rearleftchannelData = buffer.getWritePointer(4);
float* rearrightchannelData = buffer.getWritePointer(5);
float* topleftchannelData = buffer.getWritePointer(8);

float* toprightchannelData = buffer.getWritePointer(9);

float* leftdelayData = delayBuffer.getWritePointer(0);
float* rightdelayData = delayBuffer.getWritePointer(1);
float* centerdelayData = delayBuffer.getWritePointer(2);
float* surroundleftdelayData = delayBuffer.getWritePointer(6);
float* surroundrightdelayData = delayBuffer.getWritePointer(7);
float* rearleftdelayData = delayBuffer.getWritePointer(4);
float* rearrightdelayData = delayBuffer.getWritePointer(5);
float* topleftdelayData = delayBuffer.getWritePointer(8);
float* toprightdelayData = delayBuffer.getWritePointer(9);
```

Figure 6:Upmix, creating a different float pointer for each channel

Here, pointers were created for each channel, along with a corresponding variable for each channel on both buffers. In this scenario, three read positions are generated to accommodate three different delay times.

```
// Obtain the position to read and write from and to the buffer
float centerReadPosition = fmodf((float)localWritePosition - currentDelayTime + (float)delayBufferSamples, delayBufferSamples);
float rightReadPosition = fmodf((float)localWritePosition - (currentDelayTime - currentOffset) + (float)delayBufferSamples, delayBufferSamples);
float leftReadPosition = fmodf((float)localWritePosition - (currentDelayTime + currentOffset) + (float)delayBufferSamples, delayBufferSamples);


int centerLocalReadPosition = floorf(centerReadPosition);
int rightLocalReadPosition = floorf(rightReadPosition);
int leftLocalReadPosition = floorf(leftReadPosition);

if (centerLocalReadPosition != localWritePosition)
{
    //==============================PROCESSING DELAY=======================================//
    float centerFraction = centerReadPosition - (float)centerLocalReadPosition;
    float rightFraction = rightReadPosition - (float)rightLocalReadPosition;
    float leftFraction = leftReadPosition - (float)leftLocalReadPosition;

    float delayed1L = leftdelayData[(leftLocalReadPosition + 0)];
    float delayed1R = rightdelayData[(rightLocalReadPosition + 0)];
    float delayed1C = centerdelayData[(centerLocalReadPosition + 0)];

    float delayed2L = leftdelayData[(leftLocalReadPosition + 1) % delayBufferSamples];
    float delayed2R = rightdelayData[(rightLocalReadPosition + 1) % delayBufferSamples];
    float delayed2C = centerdelayData[(centerLocalReadPosition + 1) % delayBufferSamples];

    float delayed3SL = surroundleftdelayData[(leftLocalReadPosition + 0)];
    float delayed3SR = surroundrightdelayData[(rightLocalReadPosition + 0)];
    float delayed4SL = surroundleftdelayData[(leftLocalReadPosition + 1) % delayBufferSamples];
    float delayed4SR = surroundrightdelayData[(rightLocalReadPosition + 1) % delayBufferSamples];

    float delayed5RL = rearleftdelayData[(leftLocalReadPosition + 0)];
    float delayed5RR = rearrightdelayData[(rightLocalReadPosition + 0)];
    float delayed5TL = topleftdelayData[(centerLocalReadPosition + 0)];
    float delayed5TR = toprightdelayData[(centerLocalReadPosition + 0)];

    float delayed6RL = rearleftdelayData[(leftLocalReadPosition + 1) % delayBufferSamples];
    float delayed6RR = rearrightdelayData[(rightLocalReadPosition + 1) % delayBufferSamples];
    float delayed6TL = topleftdelayData[(centerLocalReadPosition + 1) % delayBufferSamples];
    float delayed6TR = toprightdelayData[(centerLocalReadPosition + 1) % delayBufferSamples];

    leftsampleOutput = delayed1L + leftFraction * (delayed2L - delayed1L);
    rightsampleOutput = delayed1R + rightFraction * (delayed2R - delayed1R);
    centersampleOutput = delayed1C + centerFraction * (delayed2C - delayed1C);
    surroundleftsampleOutput = delayed3SL + leftFraction * (delayed4SL - delayed3SL);
    surroundrightsampleOutput = delayed3SR + rightFraction * (delayed4SR - delayed3SR);
    rearleftsampleOutput = delayed5RL + leftFraction * (delayed6RL - delayed5RL);
    rearrightsampleOutput = delayed5RR + rightFraction * (delayed6RR - delayed5RR);
    topleftsampleOutput = delayed5TL + centerFraction * (delayed6TL - delayed5TL);
    toprightsampleOutput = delayed5TL + centerFraction * (delayed6TL - delayed5TL);
```

Figure 7: Creation of three read position where the performance of right-left Offset control parameters occurs and performing delay processing. Generation of delayed Output

Through the "offset" variable, users can add or subtract an amount of delay time for right channels while subtracting or adding to left pair. This manipulation results in the remaining centre and height channels having a third distinct value. The front channels will be used differently: the right and left channels will transmit only the dry signal, as no data is

```
//=========================MIX AND OUTPUT FOR CURRENT SAMPLE================================//
leftchannelData[sample] = sampleInput * (1 - currentMix) + currentMix * (leftsampleOutput - sampleInput);
rightchannelData[sample] = sampleInput * (1 - currentMix) + currentMix * (rightsampleOutput - sampleInput);
centerchannelData[sample] = currentMix * (centersampleOutput);
surroundleftchannelData[sample] = sampleInput + currentMix * (surroundleftsampleOutput - sampleInput);
surroundrightchannelData[sample] = sampleInput + currentMix * (surroundrightsampleOutput - sampleInput);
rearleftchannelData[sample] = sampleInput + currentMix * (rearleftsampleOutput - sampleInput);
rearrightchannelData[sample] = sampleInput + currentMix * (rearrightsampleOutput - sampleInput);
topleftchannelData[sample] = currentMix * (topleftsampleOutput);
toprightchannelData[sample] = currentMix * (toprightsampleOutput);

//leftdelayData[localWritePosition]          = centersampleInput       + rightsampleOutput         * currentFeedback;
//rightdelayData[localWritePosition]         = centersampleInput       + leftsampleOutput          * currentFeedback;
centerdelayData[localWritePosition] = sampleInput;
surroundleftdelayData[localWritePosition] = (sampleInput)+surroundleftsampleOutput * currentFeedback;
surroundrightdelayData[localWritePosition] = (sampleInput)+surroundrightsampleOutput * currentFeedback;
rearleftdelayData[localWritePosition] = (sampleInput)+rearleftsampleOutput * currentFeedback;
rearrightdelayData[localWritePosition] = (sampleInput)+rearrightsampleOutput * currentFeedback;
topleftdelayData[localWritePosition] = sampleInput + topleftsampleOutput * currentFeedback;
toprightdelayData[localWritePosition] = sampleInput + toprightsampleOutput * currentFeedback;
```

Figure 8: Final Mid-Side delay output, for main buffer and delay buffer

written to the delay buffer. Meanwhile, the center channel will create only one repetition of the dry signal, resulting in a feedback value of 0. The height channels plays same sound with same delay time effect and will have more wet signal, unlike the front centre channel. This is done with the intention of enhancing the central phantom image (Borzym, 2020).

3.5 Ping pong type Delay Algorithm implementation

As the name suggests, this processor is inspired by a stereo Ping Pong algorithm, which normally mixes the left Output with the right Input and viceversa The main idea is to incorporate the Ping Pong effect into the algorithm, adapting it to 3D, mixing any right and left Input and Output for each pair of speaker. This mode of the plugin will be slightly more advanced than the other, allowing the user to adjust the panning of the two input signals and adjust the delay time offset between the front, surround, and rear speakers. This way, there is the possibility of having more control over the desired effect, providing opportunities for many creative solutions. To enable mixing the left input with the right output channels and vice versa, same approach of MidSide algorithm was taken during the input assignment compared to the first algorithm.

```
for (int sample = 0; sample < buffer.getNumSamples(); ++sample)
{
    // Input samples for each channel
    const float leftsampleInput = (1.0f - currentBalance) * leftchannelData[sample];
    const float rightsampleInput = currentBalance * rightchannelData[sample];
    const float centersampleInput = (leftchannelData[sample] + rightchannelData[sample]) / 2;
```

Figure 9: Creation of right and left sample Input variables

With these two lines of code, input samples are created, and users can control their panning using the "balance" variable. Simultaneously, this line ensures that the front centre channel remains fixed when adjusting the balance.

Three read positions are generated to accommodate three different delay times. Through the "offset" variable, users can add or subtract an amount of delay time for rear pair of channels while subtracting or adding to another pair. This manipulation results in the remaining channels having a third distinct value.

```
// Obtain the position to read and write from and to the buffer
float frontReadPosition = fmodf((float)localWritePosition - currentDelayTime + (float)delayBufferSamples, delayBufferSamples);
float midReadPosition  = fmodf((float)localWritePosition - (currentDelayTime - currentOffset) + (float)delayBufferSamples, delayBufferSamples);
float rearReadPosition = fmodf((float)localWritePosition - (currentDelayTime + currentOffset) + (float)delayBufferSamples, delayBufferSamples);


int frontLocalReadPosition = floorf(frontReadPosition);
int midLocalReadPosition = floorf(midReadPosition);
int rearLocalReadPosition = floorf(rearReadPosition);

if (frontLocalReadPosition != localWritePosition)
{
    //==============================PROCESSING DELAY=======================================//
    float frontFraction = frontReadPosition - (float)frontLocalReadPosition;
    float midFraction   = midReadPosition - (float)midLocalReadPosition;
    float rearFraction = rearReadPosition - (float)rearLocalReadPosition;

    float delayed1L = leftdelayData[(frontLocalReadPosition + 0)];
    float delayed1R = rightdelayData[(frontLocalReadPosition + 0)];
    float delayed1C = centerdelayData[(frontLocalReadPosition + 0)];

    float delayed2L = leftdelayData[(frontLocalReadPosition + 1) % delayBufferSamples];
    float delayed2R = rightdelayData[(frontLocalReadPosition + 1) % delayBufferSamples];
    float delayed2C = centerdelayData[(frontLocalReadPosition + 1) % delayBufferSamples];

    float delayed3SL = surroundleftdelayData[(midLocalReadPosition + 0)];
    float delayed3SR = surroundrightdelayData[(midLocalReadPosition + 0)];
    float delayed4SL = surroundleftdelayData[(midLocalReadPosition + 1) % delayBufferSamples];
    float delayed4SR = surroundrightdelayData[(midLocalReadPosition + 1) % delayBufferSamples];

    float delayed5RL = rearleftdelayData[(rearLocalReadPosition + 0)];
    float delayed5RR = rearrightdelayData[(rearLocalReadPosition + 0)];
    float delayed5TL = topleftdelayData[(frontLocalReadPosition + 0)];
    float delayed5TR = toprightdelayData[(frontLocalReadPosition + 0)];

    float delayed6RL = rearleftdelayData[(rearLocalReadPosition + 1) % delayBufferSamples];
    float delayed6RR = rearrightdelayData[(rearLocalReadPosition + 1) % delayBufferSamples];
    float delayed6TL = topleftdelayData[(frontLocalReadPosition + 1) % delayBufferSamples];
    float delayed6TR = toprightdelayData[(frontLocalReadPosition + 1) % delayBufferSamples];

    leftsampleOutput           = delayed1L     + frontFraction * (delayed2L - delayed1L);
    rightsampleOutput          = delayed1R     + frontFraction * (delayed2R - delayed1R);
    centersampleOutput         = delayed1C     + frontFraction * (delayed2C - delayed1C);
    surroundleftsampleOutput   = delayed3SL    + midFraction   * (delayed4SL - delayed3SL);
    surroundrightsampleOutput  = delayed3SR    + midFraction   * (delayed4SR - delayed3SR);
    rearleftsampleOutput       = delayed5RL    + rearFraction * (delayed6RL - delayed5RL);
    rearrightsampleOutput      = delayed5RR    + rearFraction * (delayed6RR - delayed5RR);
    topleftsampleOutput        = delayed5TL    + frontFraction * (delayed6TL - delayed5TL);
    toprightsampleOutput       = delayed5TL    + frontFraction * (delayed6TL - delayed5TL);
```

Figure 10: Creation of three read position where the performance of rear-front Offset control parameters occurs and performing delay processing. Generation of delayed Output

Following this, wet signals are created in the delay buffer, and subsequently, the outputs are generated.

```
leftchannelData[sample]            = leftsampleInput   *   (1 - currentMix)      + currentMix   * (leftsampleOutput- leftsampleInput);
rightchannelData[sample]           = rightsampleInput  *   (1 - currentMix)      + currentMix   * (rightsampleOutput - rightsampleInput);
centerchannelData[sample]          =                       currentMix   * (centersampleOutput);
surroundleftchannelData[sample]    = leftsampleInput   + currentMix   * (surroundleftsampleOutput-leftsampleInput);
surroundrightchannelData[sample]   = rightsampleInput  + currentMix   * (surroundrightsampleOutput - rightsampleInput);
rearleftchannelData[sample]        = leftsampleInput   + currentMix   * (rearleftsampleOutput - leftsampleInput);
rearrightchannelData[sample]       = rightsampleInput  + currentMix   * (rearrightsampleOutput - rightsampleInput);
topleftchannelData[sample]         =                       currentMix   * (topleftsampleOutput);
toprightchannelData[sample]        =                       currentMix   * (toprightsampleOutput);

//leftdelayData[localWritePosition]          = leftsampleInput       + rightsampleOutput         * currentFeedback;
//rightdelayData[localWritePosition]         = rightsampleInput      + leftsampleOutput          * currentFeedback;
centerdelayData[localWritePosition]        = centersampleInput;
surroundleftdelayData[localWritePosition]  = (leftsampleInput)     + surroundrightsampleOutput * currentFeedback;
surroundrightdelayData[localWritePosition] = (rightsampleInput)    + surroundleftsampleOutput  * currentFeedback;
rearleftdelayData[localWritePosition]      = (leftsampleInput)     + rearrightsampleOutput     * currentFeedback;
rearrightdelayData[localWritePosition]     = (rightsampleInput)    + rearleftsampleOutput      * currentFeedback;
topleftdelayData[localWritePosition]       = rightsampleInput      + toprightsampleOutput      * currentFeedback;
toprightdelayData[localWritePosition]      = leftsampleInput       + topleftsampleOutput       * currentFeedback;
```

Figure 11 Final Ping-Pong delay output, for main buffer and delay buffer. Here where the Ping Pong effect is caused

3.6 Gain, filters and other functionalities

In the process block, all the functions for signal processing are called. In addition to the delay effects described in the previous chapter, the Input gain, Output Gain, Low pass filter, and High pass filter functions are then invoked. The "Input gain" function is called before the delay, while the "Output" and "Filters" functions are called after. To allow the user to choose between the two effects, two string variables corresponding to each of them are created. Through the following if statement, the function corresponding to each effect will be called.

The high pass and the low pass filters are created by utilizing the integrated Infinite Impulse Response (IIR) Filter function in the JUCE framework. In contrast to the Finite Impulse Response (FIR) Filter, IIR is considerably more efficient because it demands fewer coefficients, resulting in reduced memory space requirements for computation and filter production.

$$y(n) = b_0 + b_1 x(n-1) + \cdots + b_M(n-M) - a_1 y(n-1) - \cdots - a_N y(n-N)$$

Equation 3: IIR filter general differential equation

Where a_i are the filter feedback coefficients, b_j are the filter feedforward coefficients, N is the filter order and x_i and y_j are the input signal and the output signal. By the use of MakeLowPass() and MakeHighPass() JUCE obtains the filter's coefficients; the Q value is fixed to 0.8.

An UpdateHighPassFilter() and UpdateLowPassFilter() filter functions have been created.

Figure is the image of the process block.

```
void Atmos3DDelayAudioProcessor::processBlock (juce::AudioBuffer<float>& buffer, juce::MidiBuffer& midiMessages)
{
    //========= Variables ===================================//
    ScopedNoDenormals noDenormals;

    auto dTime      = parameters.getRawParameterValue("delayTime");
    auto mix        = parameters.getRawParameterValue("mix");
    auto feedback   = parameters.getRawParameterValue("feedback");
    auto balance    = parameters.getRawParameterValue("balance");
    auto choice     = parameters.getRawParameterValue("delay_option");
    auto offset     = parameters.getRawParameterValue("offset");

    currentDelayTime = (dTime->load()) * (float)getSampleRate();
    currentMix          = (mix->load());
    currentFeedback     = feedback->load();
    currentBalance      = balance->load();
    currentChoice       = choice->load();
    currentOffset       = (offset->load()) * (float)getSampleRate();

    int localWritePosition = delayWritePosition;

    //========== Processing ================================//

    // Gain control of input signal
    inputGainControl(buffer);

    // Perform DSP below
    if (currentChoice == 0)
        PingPongDelay(buffer, localWritePosition);
    else if (currentChoice == 1)
       SlapBackDelay(buffer, localWritePosition);
    else if (currentChoice==2)
       MidSideDelay(buffer, localWritePosition);


    lpFilter(buffer);
    hpFilter(buffer);

    // Gain control of output signal
    outputGainControl(buffer);

    // This is here to avoid people getting screaming feedback when they first compile a plugin
    for (auto i = getTotalNumOutputChannels(); i < getTotalNumOutputChannels(); ++i) { buffer.clear(i, 0, buffer.getNumSamples()); }
}
```

Figure 12: Whole process block function, where all the Signal Processing operations occurs.

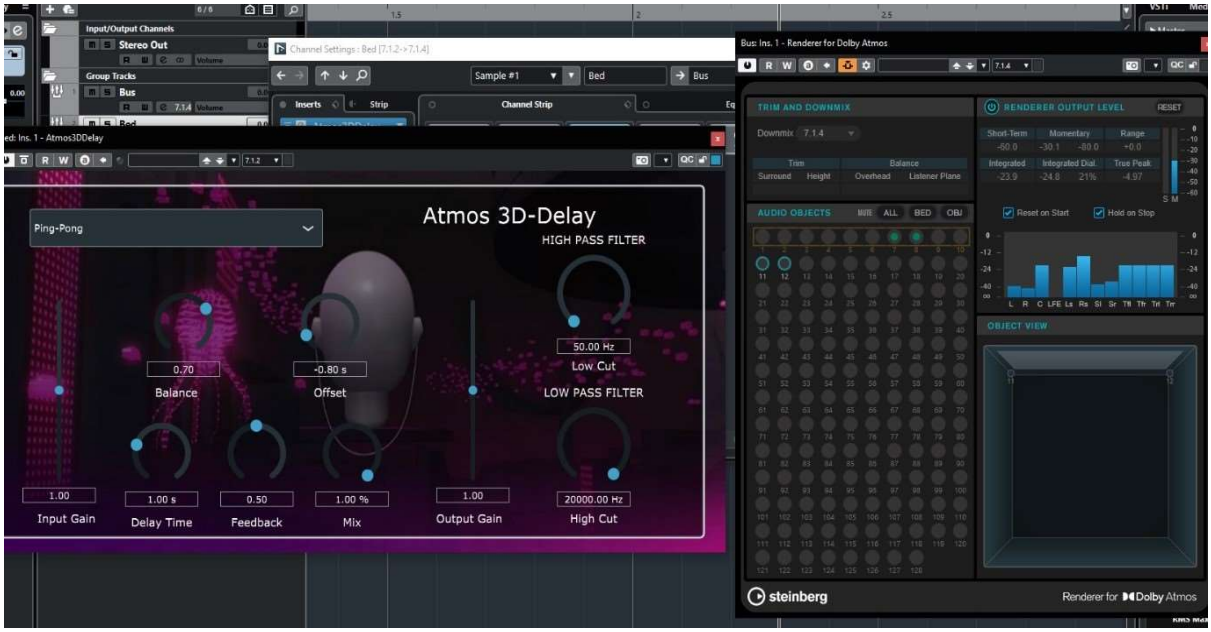
4. RESULTS AND DISCUSSION

For testing the plugin, it has been used the software Nuendo 12, a DAW capable of supporting multichannel audio and equipped with an internal Dolby Atmos Renderer. A project in the Atmos format was created, which involves setting up a 7.1.4 track, called bus (figure 13) the standard for Atmos, with stereo output since there is no speaker array available. In this track, the Dolby Atmos Renderer Plugin was loaded.

Figure 13: Nuendo 12 mixer Interface. Bed Track channel settings shows the Atmos 3D-Delay inserted

Subsequently, a 7.1.2 track representing the project's Bed channel was created, and the Atmos 3D-Delay plugin was loaded onto it. As shown in image 14, the plugin loaded correctly. To conduct the test, we selected a track featuring a snare drum hit to analyse the effect's results in the most effective way.

From picture it is noticeable that the first 10 tracks within the red box are the bed track, where the plugin is inserted, while the 11 and 12 tracks are the snare stereo signal.



In the following paragraphs, the results of the signal affected by the plugin in the various channels will be shown. The following table displays the channel nomenclature of the analysing program.

1	2	3	4	5	6	7	8	9	10
L	R	C	LFE	RSL	RSR	SSL	SSR	TL	TR

Table 1: 7.1.2 Channels naming legend

4.1 Normal Delay

Delay time	Feedback	Mix
1.25 s	0.2	0.75 (75%)

Table 2: First set of parameter's value used for testing normal delay mode

As we can observe from the figure 15, all channels except for the LFE are involved. They all emit sound simultaneously, with variations in intensity due to the input settings in the for loop. Specifically, channels 1 and 2, which contain the stereo sources, are different from the others.

As a result, even though there is not control on how the sound is moving, it randomly does, as it is noticeable from picture 17.

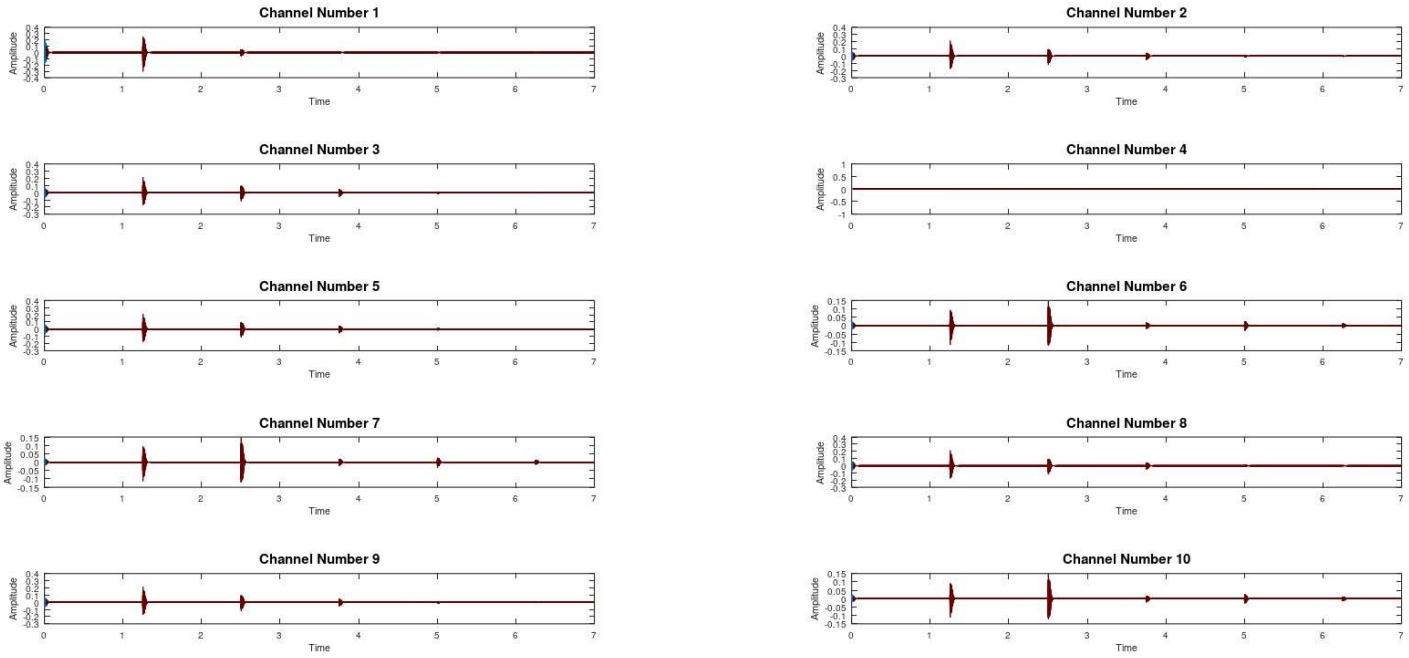


Figure 14: Normal Delay's channels graphs results with first set of parameter's values

Delay time	Feedback	Mix
0.3	0.6	0.75 (75%)

Table 3: second set of parameter's value used for testing normal delay mode

Figure 16 shows the result with a lower delay time and more feedback. The parameters performes correctly and the results changes accordingly: bigger number of repetitions and smaller intervals between them.

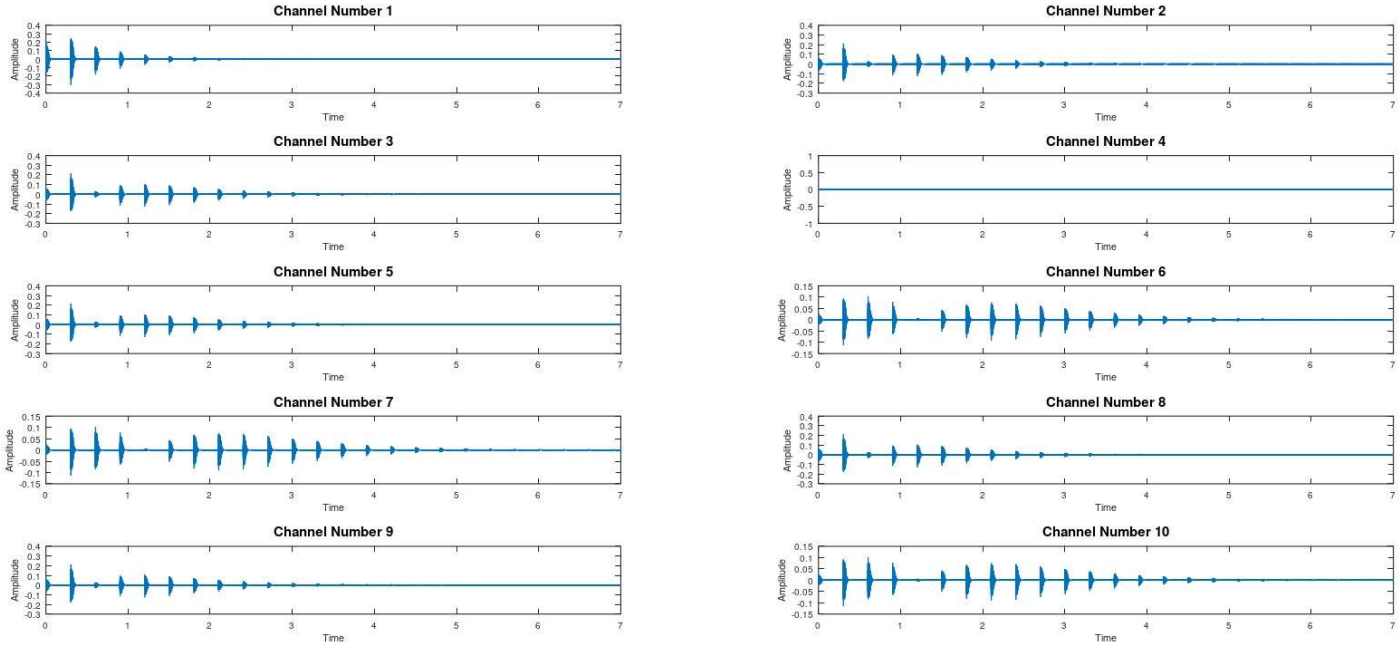


Figure 15:Normal Delay's channels graphs results with second set of parameter's values

Figure 17 shows the sound field of the Normal Delay, using Insight 2 plugin, which offers a set of meters which allows the monitoring of spectral balance, loudness, intelligibility for both stereo and multichannel. For this representation has been used the same snare hit sample and the result shows the sound field at a certain time instant. The location of the dot represents the summed surround locations of all channels.

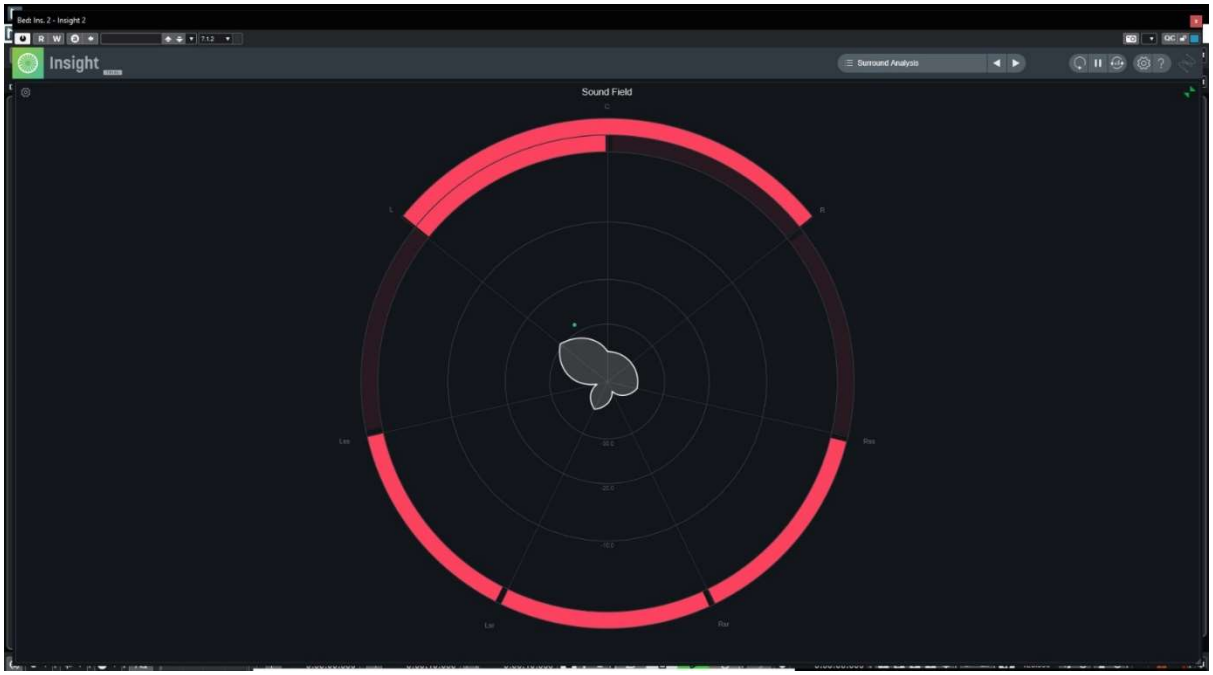


Figure 16: Sound field of Normal Delay with first set of parameter's values

Overall, the Normal Delay delay playback over the 10 speakers has been achieved, but performance is not perfect. The red lines that figure 17 shows on the circle borders, are representing a phase issue within the specified channel. Further research will be done in order to solve the problem.

Furthermore, not having the control of the speakers might be limiting for a producer in some cases, especially with long feedback, if the mix is busy. One suggested use might be a short echo delay, such a SlapBackDelay type, with Delay Time within 0.05s and 0.2s, with no feedback or very little, less than 0.2, and high mix level plus making space using High Pass and Low Pass filters.

4.2 Mid-Side Delay

Delay time	Feedback	Mix	Offset
0.5s	0.3	1 (100%)	0.3

Table 4: First set of parameter's value used for testing Mid-Side delay mode

According to the parameters' value, it is noticeable that the processors work as it is meant to. Channel 1(left) and channel 2(right) plays once, while the centre channel plays once but with 0.5s of delay time. The offset button, set at 0.3s, also works correctly: channels 6(surround rear right) and 8(surround side right) plays each 0.2s, which comes from delay time - offset, while channels 5(surround rear left) and 7(surround side left) plays each 0.8s, which comes from delay time + offset. Top channels 9 and 10 have 0.5s delay time as expected.

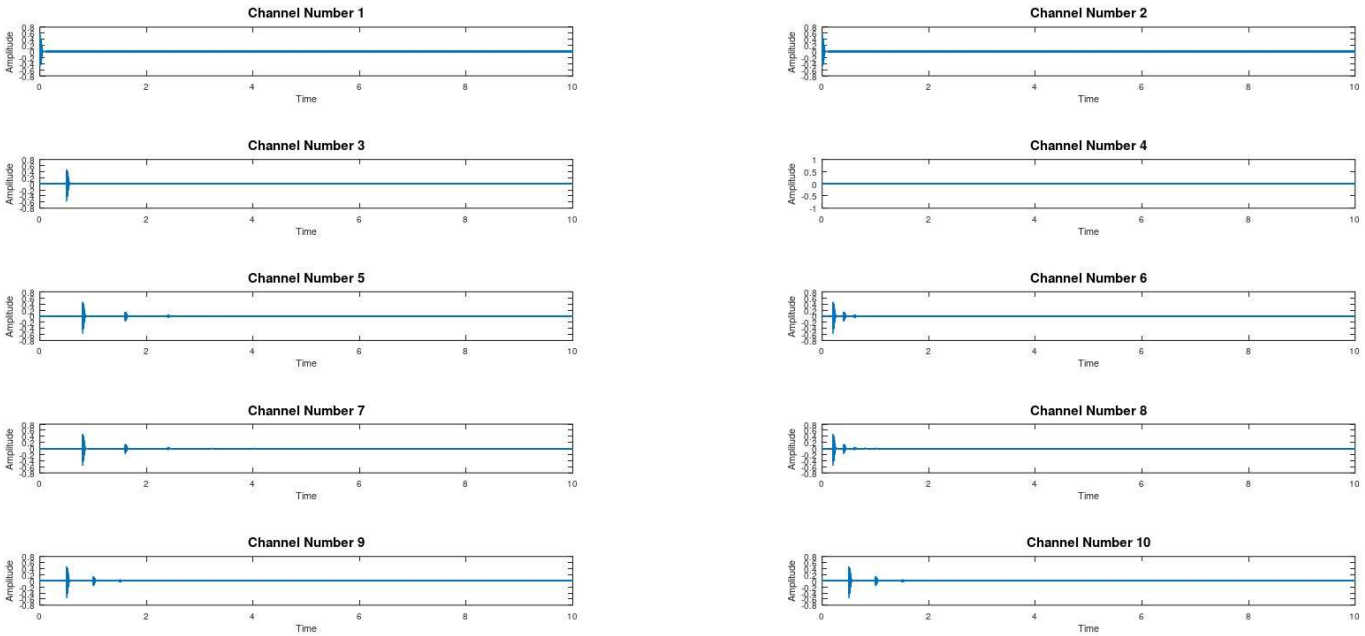


Figure 17: Mid-Side Delay's channels graphs results with first set of parameter's values

Delay time	Feedback	Mix	Offset
1s	0.5	1 (100%)	-0.8

Table 5: Second set of parameter's value used for testing Mid-Side delay mode

Figure 19 shows another scenario where the offset between the right pairs of surround speakers and the left pairs are more noticeable due to the Offset Value set to -0.8s: in this case, due to the "minus" sine, channel 4 and channel 6 has longer delay time, while channel 5 and 7 has shorter delay time and the number of repletion is also bigger, because of the

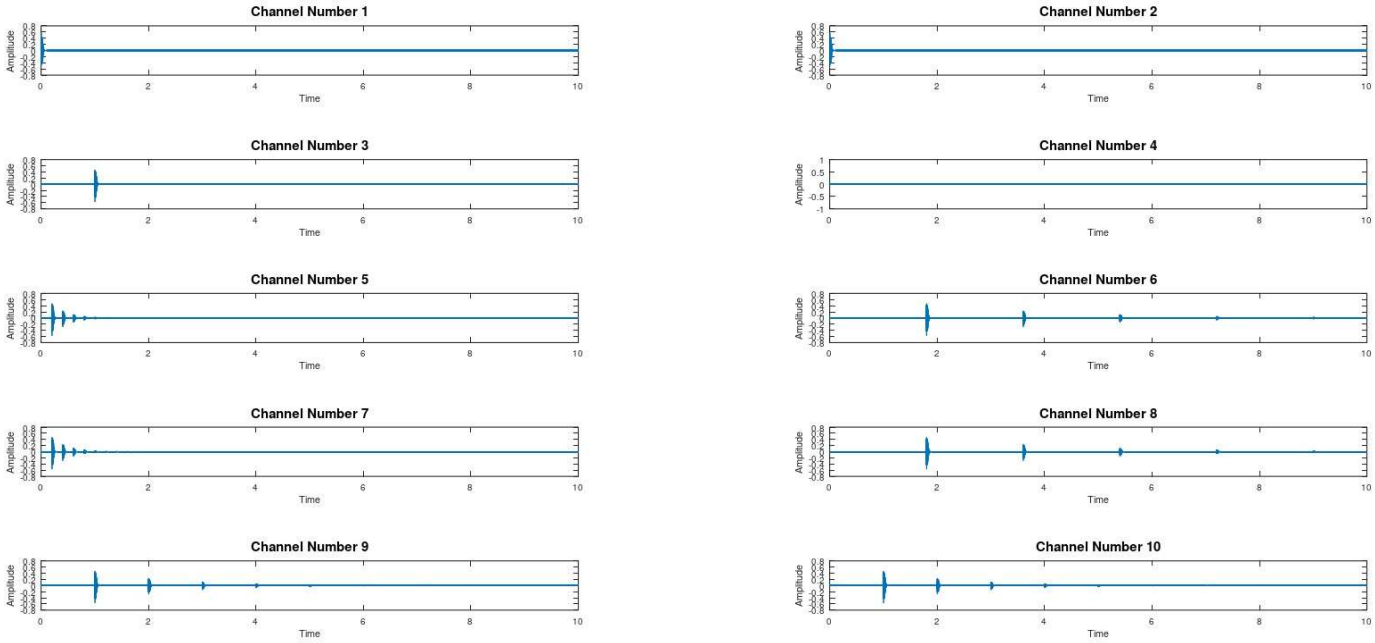


Figure 18:Mid-Side Delay's channels graphs results with second set of parameter's values

feedback set at 0.5.

Figure 17 shows the sound field of the Mid-Side Delay, using Insight 2 plugin. For this representation has been used the same snare hit sample and the result shows the sound field at a certain time instant. The location of the dot represents the summed surround locations of all channels. The summed surround location in a fixed time instant, with that set of parameters, it is luckily to be found on the left speaker area, as it correctly happens and can be seen in figure 20. The Mid side algorithm performs well and does not present phase issue. Thanks to the Offset control, the perception of movement and spatialization is higher than the

previous delay mode and the user can easily create interesting patterns and have control of the summed surround location.

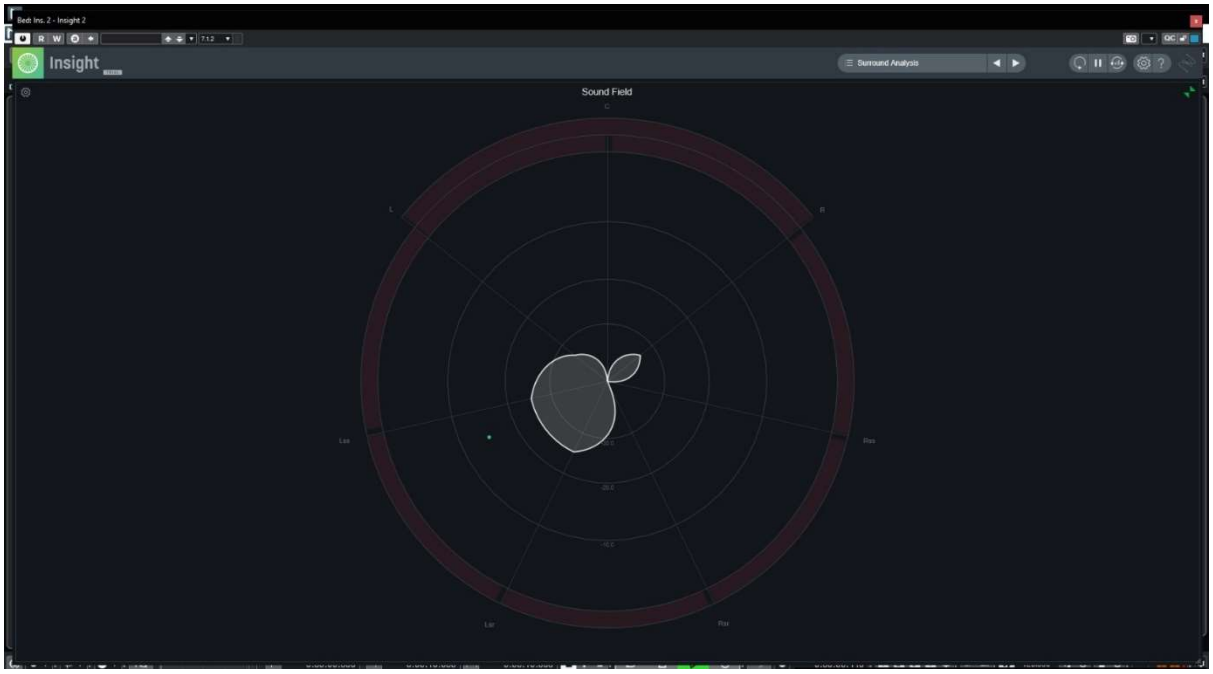


Figure 19: Sound field of Mid-Side Delay with second set of parameter's values

4.3 Ping-Pong Delay

Delay time	Feedback	Mix	Offset	Balance
1	0.5	1(100%)	0.66	0.7

Table 6: First set of parameter's value used for testing PingPong delay mode

In order for showing the performance of the third delay algorithm, three cases scenario has been chosen. Figure 20 shows how delay time of the surround side pair of speakers, corresponding to channel 7 and 8 change with Offset value 0.66, compared to channel 5 and channel 6, the rear channels. With this specific value, it is obtained a three-time pace pattern of repetitions for first two, and a third of delay time for the others. Again, front channels works correctly and height channels as well, having same behaviour of previous

shown processor. Compared to the Mid-Side, this delay creates different delay time patterns along centre and back, while the previous between left and right. Also, it is noticeable that with a value of 0.7, the balance value differences the level of input between the two channels: channel 1 has an amplitude value minor than 0.15, while channel 2 reaches 0.4.

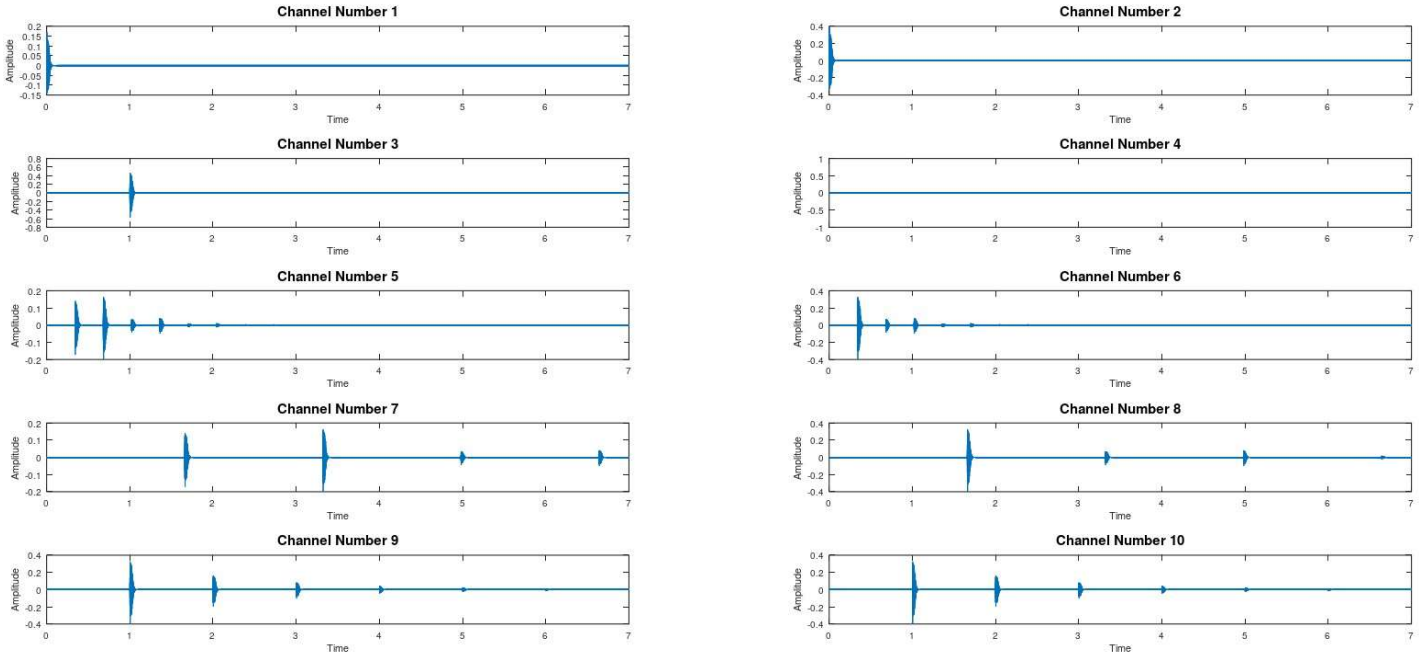


Figure 20: Ping-Pong Delay's channels graphs results with first set of parameter's values

As a result, the two remaining right channels appears to have higher amplitude values than left ones, while left decreases slower along the time than the right ones. This behaviour confirms the correct performance of the ping pong effect.

The figure 21 shows the processor acting with following parameters' values:

Delay time	Feedback	Mix	Offset	Balance
0.6	0.21	1(100%)	0.66	0.7

Table 7: Second set of parameter's value used for testing PingPong delay mode

In picture 21 shows results with higher repletion rate; it is better noticeable the ping pong effect and the interesting path done by the signal through all the speakers.

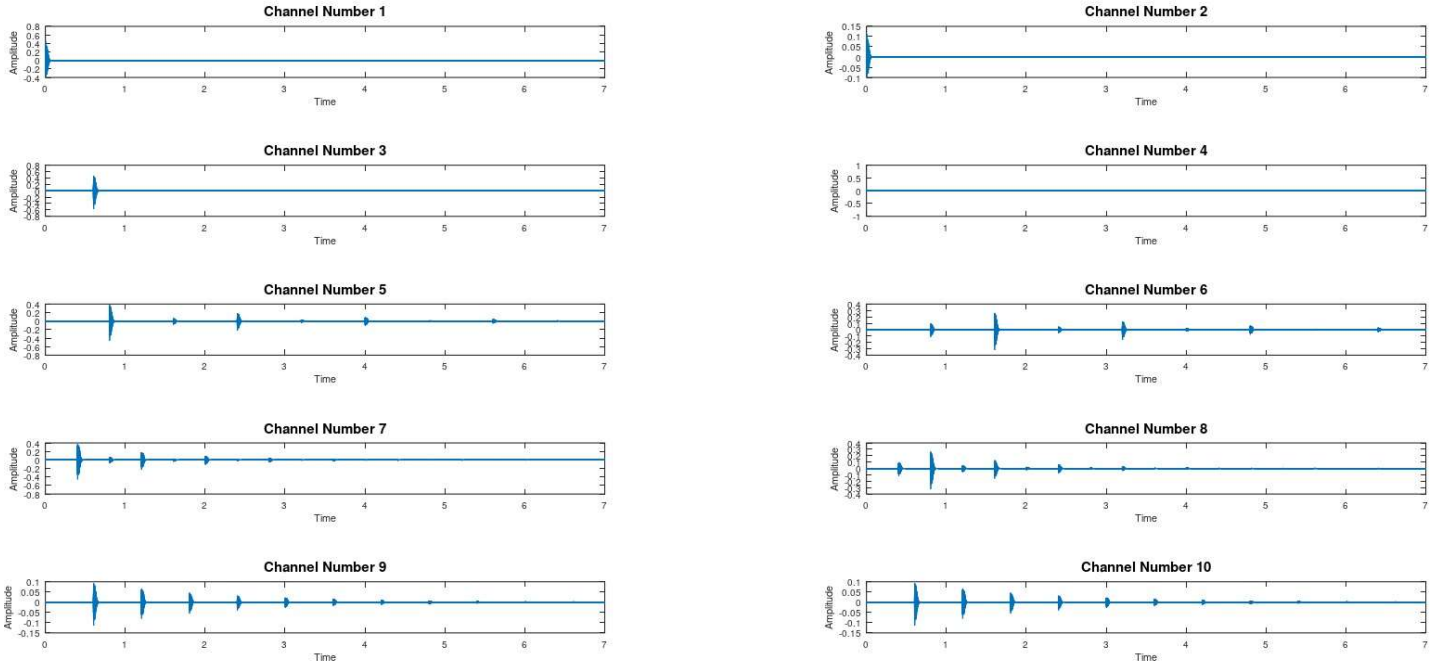


Figure 21: Ping-Pong Delay's channels graphs results with second set of parameters values

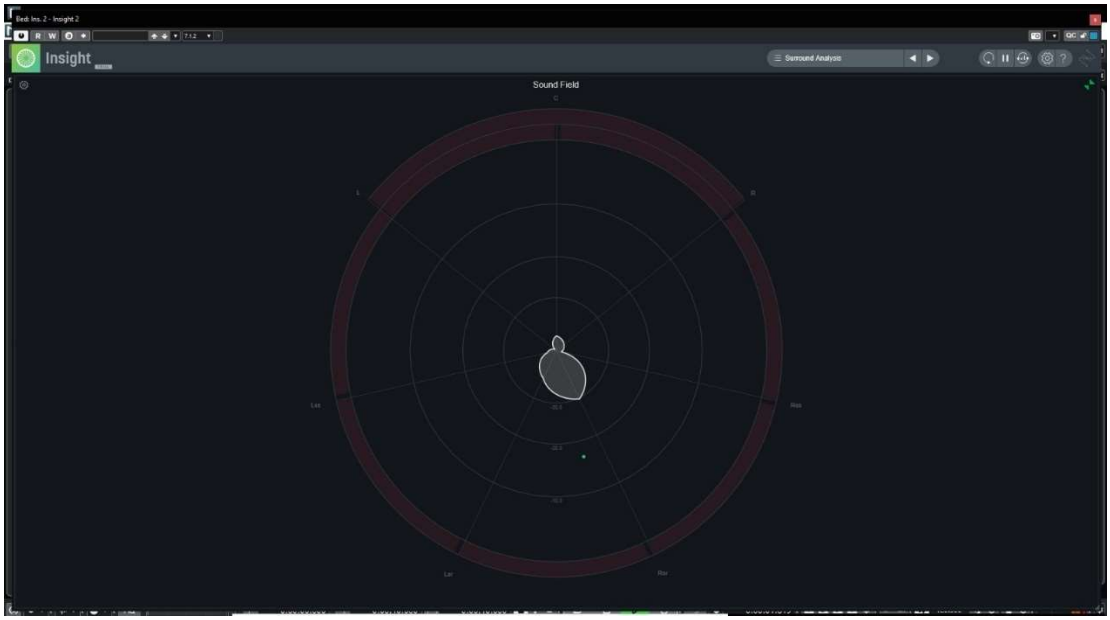


Figure 22: Sound field of Ping-Pong Delay with second set of parameter's values

Figure 22 shows the sound field of Ping pong delay mode. It is noticeable that the dot is located more in the right zona, confirming the input balance of 0.7 which make the left have a multiplier of 0.3 while the right of 0.7.

4.4 Graphical user interface GUI

The GUI looks user friendly; the small number of buttons makes the plugin very easy to use, compared to the infinite solutions and combinations of settings that a 10 channels processor might have.

Figures 23. 24 and 25, shows that the GUI performing in the three plugin mode.

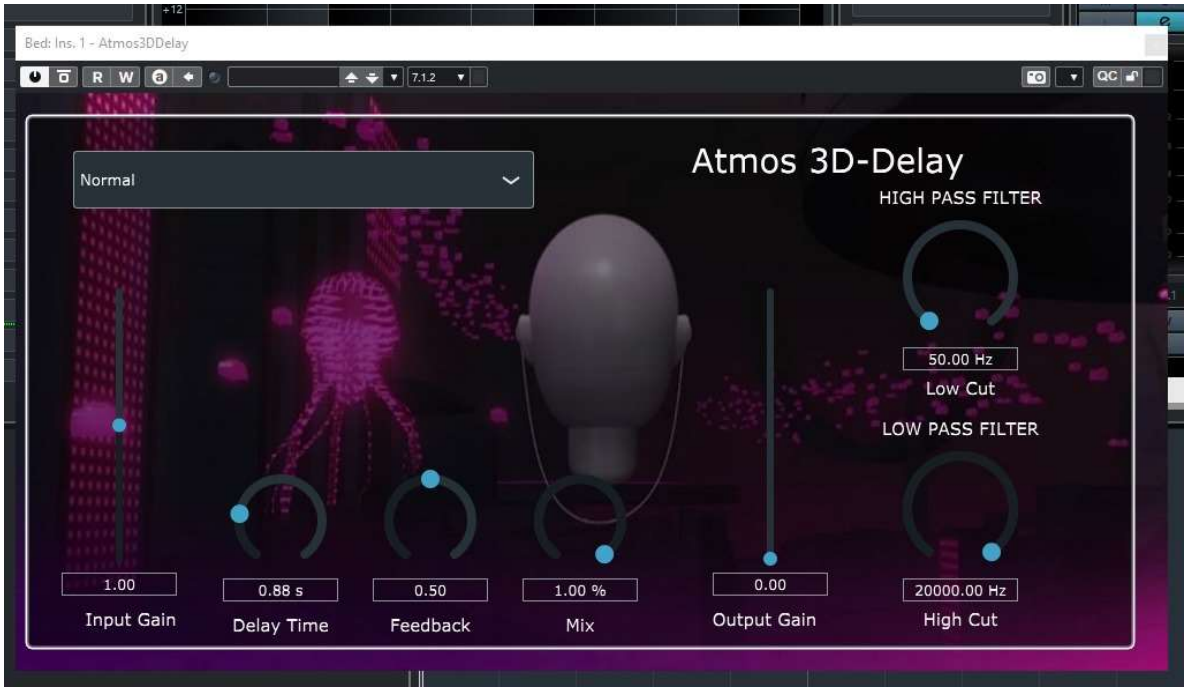


Figure 23: Atmos 3D-Delay GUI in Normal mode

Figure 23 shows the GUI in Normal mode, where it only shows Delay time, feedback and Mix controls for delay plus Input, Output and High Pass and Lows Pass Filter controls.

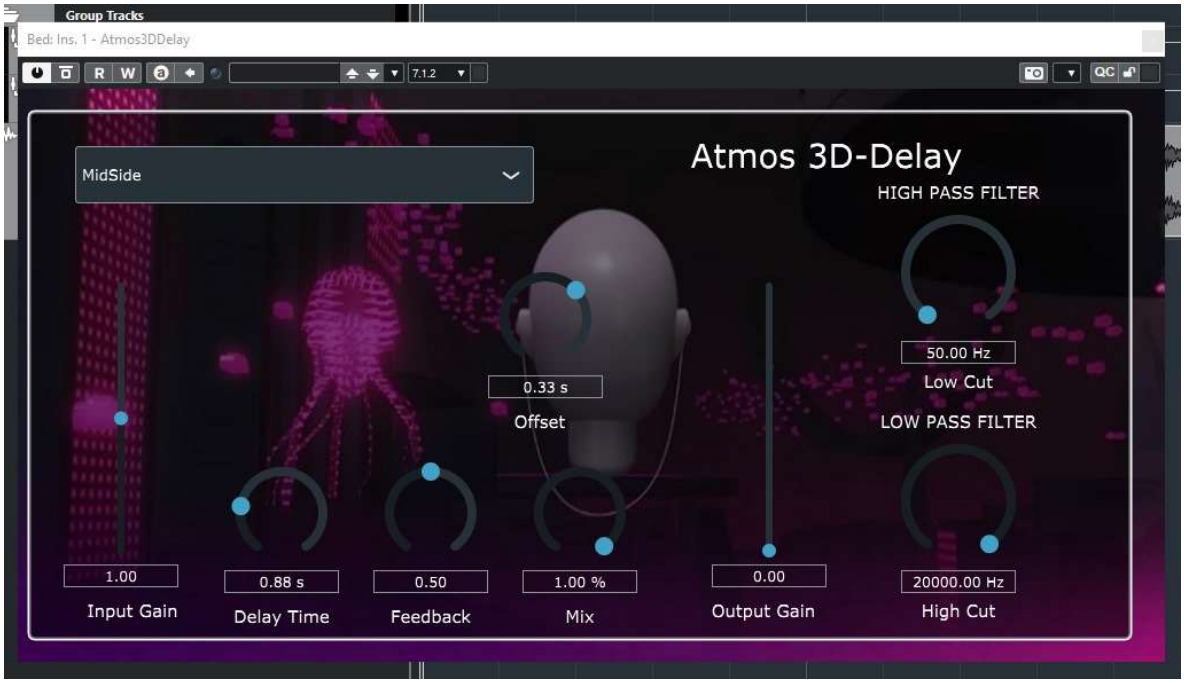


Figure 24: Atmos 3D-Delay GUI in Mid-Side mode

Figure 24 shows the GUI while performing Mid-Side mode of the plugin. As it is noticeable, Offset control is correctly added.

Figure 25: Atmos 3D-Delay GUI in Ping Pong mode

Figure 25 confirms the right performance of the GUI when selecting Ping-Pong mode: the Balance parameter is correctly added.

The only thing that might be better to change, in order to make the usage even more clear, is specifying the change of Offset type when switching between Mid-Side and Ping Pong. The modification would consist in make the GUI appears for example “Right-Left Offset” when Mid-Side is used and “Front-Rear Offset” when Ping Pong is used.

4.5 High Pass Filter, Low Pass Filter and Input/Output gain

To demonstrate the proper functioning of the LowPass and HighPass filters, which affect the entire buffer, we selected an instrumental track from a trending song, such as Drake's 'Middle of the Ocean.' This instrumental track covers the entire frequency spectrum. Figure 26 depicts the spectrum of the instrumental track before introducing the Delay plugin into the mix.

Figure 26: Original Sound Field and spectrum of Drake's Middle of the Ocean song with no processing at a fixed time instant

The following figures will showcase the instrumental track being affected first by one filter and then the other, at the same time instant.

Figure 27 shows the instrumental affected from High Pass filter with Low Cut frequency set at 10Khz. Comparing figure 27 to figure 26 it is clear that all frequencies below

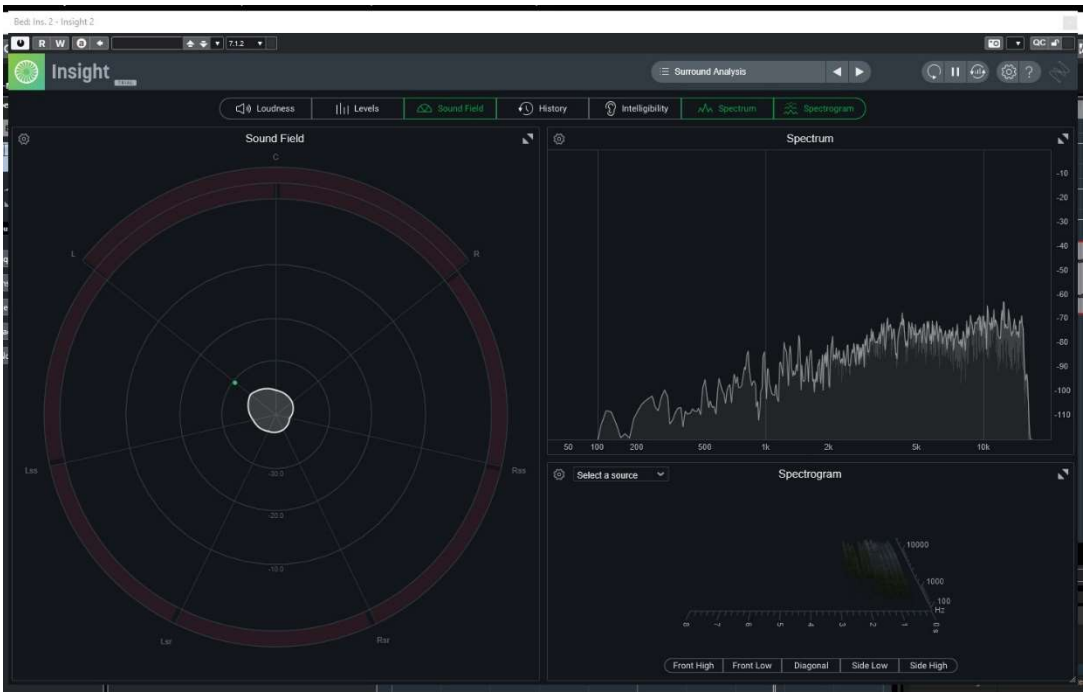


Figure 27: Sound Field and spectrum of Drake's Middle of the Ocean song with high pass filter processing at a fixed time instant

10Khz decreases, reaching the 0 amplitude value at 100hz, confirming the correct performance of the High Pass filter

Figure 28 shows the instrumental affected from Low Pass filter with High Cut frequency set at 10Khz. The second order High pass filter, with Q factor set to 0.8, works correctly; all the frequencies higher then 10Khz gets attenuated.

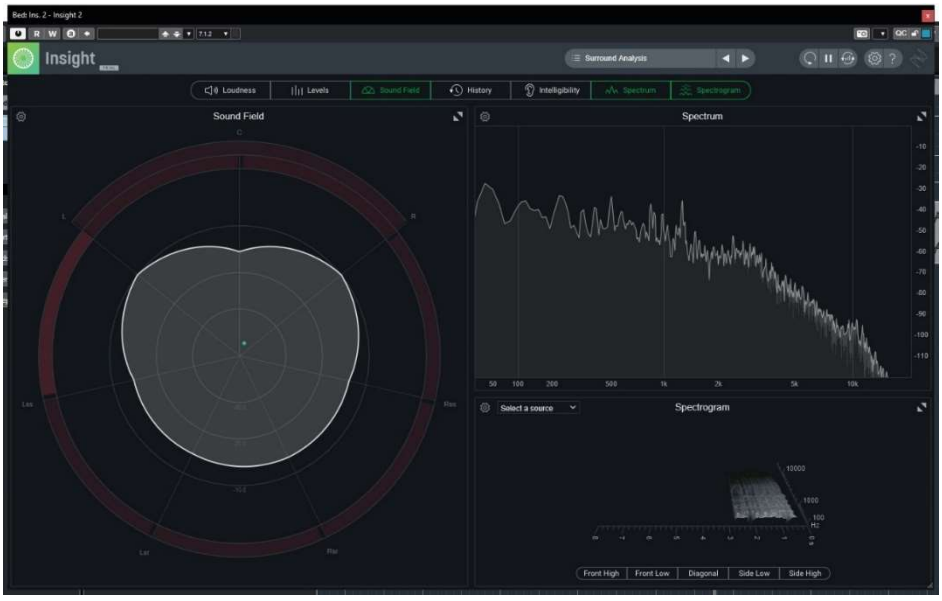


Figure 28: Sound Field and spectrum of Drake's Middle of the Ocean song with low passs filter processing at a fixed time instant

As it has been done for filters, first result figure 29 shows the signal when the Input and Output gain are not affecting the signal, with a value of 1. It has been selected the Normal Delay mode of the plugin, because all the channels are always playing, best showing the performance of the gain controls.

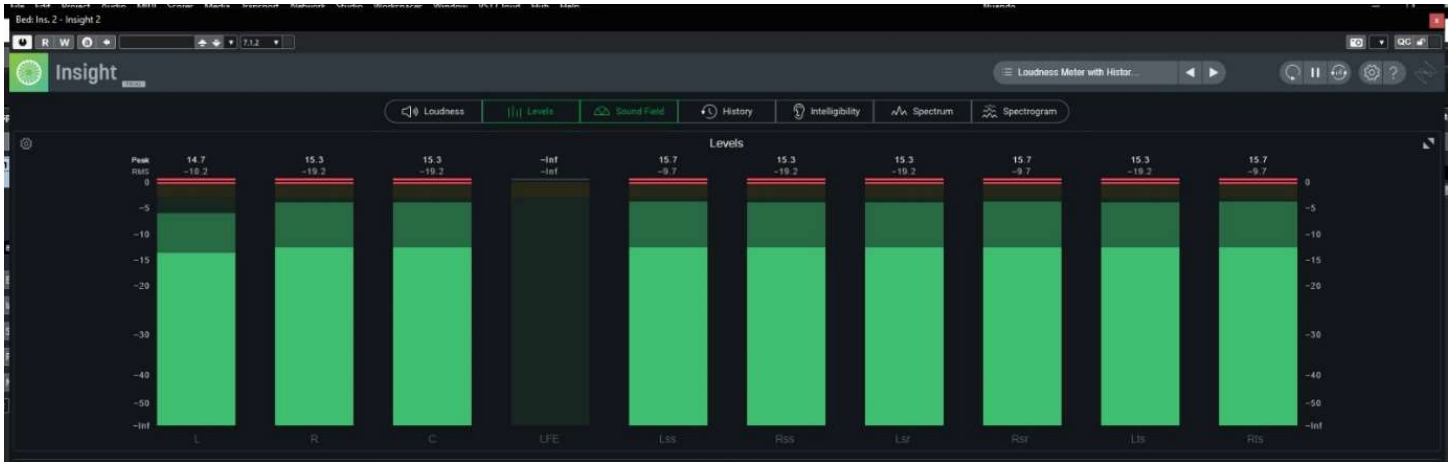


Figure 29: Channel meters with Input gain value 1 and Output gain value 1

Figure 30 shows results when only the input gain's value is changed to 0.3 and the output gain is left at 1, while picture 31 shows results in the reverse scenario, where output is rather changed to 0.1 and Input left at 1.

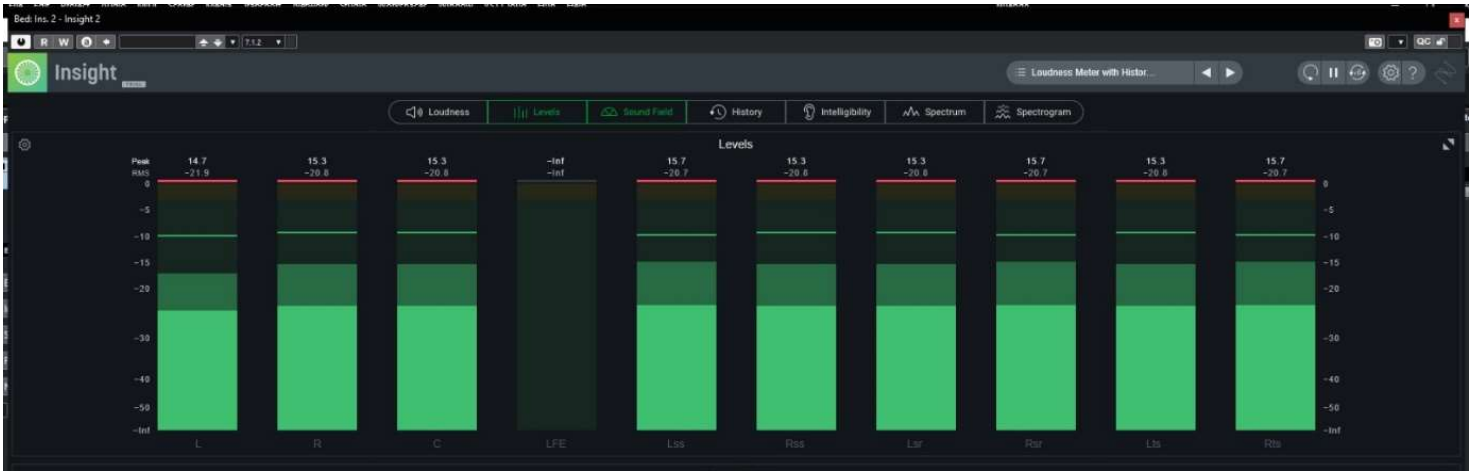


Figure 30: Channel meters with Input gain value 0.3 and Output gain value 1

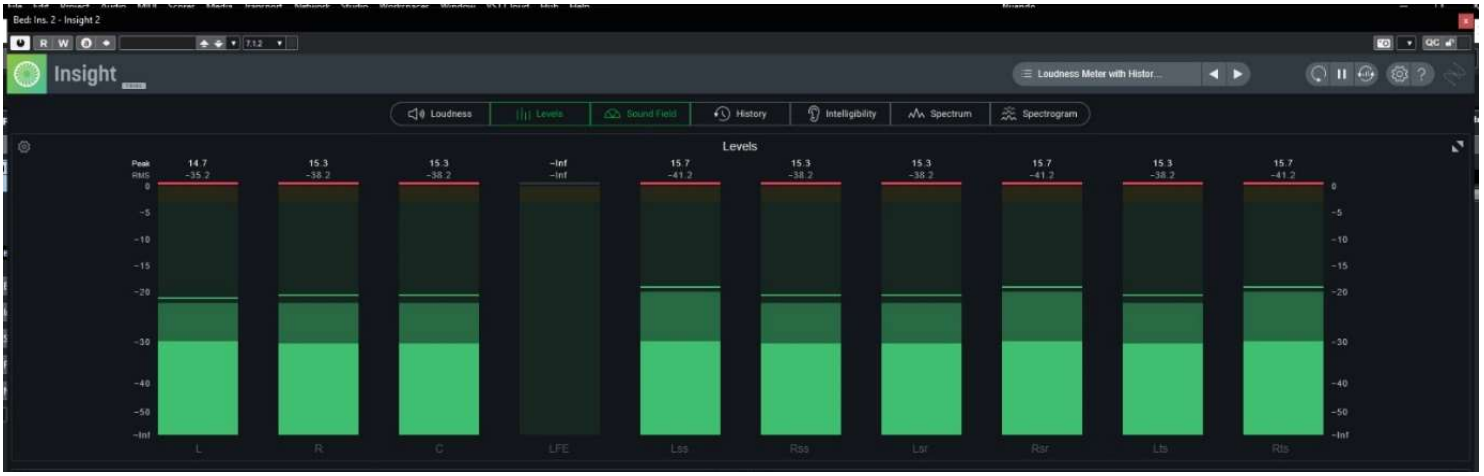


Figure 31: Channel meters with Input gain value 1 and Output gain value 0.1

Both the figures verified the correct performance of both Input and Output buttons.

5. CONCLUSIONS

5.1 Current limitations, future works and further implementation

When comparing the plug-in's final output to the initial proposal for this dissertation, it becomes apparent that certain features couldn't be implemented. This was primarily due to the complexity of these features and the limitations imposed by time constraints. The research phase and the up-mixing implementation's achieving has been more time consuming than what was expected, taking time from the designing of the creative part of the project and the implementation of more complex features and solutions. Features like bypass button and the option for inserting the source track directly from the plugin has not been achieved; although it has been proved that those actions can be taken using the DAW built in functions and the Dolby Atmos Renderer, the achievement of those features would have been helpful, according to the will to provide a plugin also for beginners and low-level skilled users.

Except from these two last features and the issues of the Normal Delay mode of the plugin previously discussed in Chapter 4, and that its solution will be object of further study and research, the project aim and the all the objectives have entirely been achieved.

Certainly, the plugin has large room for improvement. The LFE channel has not been involved. A possible solution might consist of doing research and finding a way of filtering the entire buffer and send all the low end information, usually below 120hz, to the LFE channel and the remaining range of frequencies to the rest of the speakers. Still regarding filters and equalization, another thing to be improved concerns the use of the height channels; in the realization of the program, it has been tried to use them for adding more phantom image to the centre sound field zone, using the centre channel only for one repetition of the affected signal without feedback, and using the height channels for the wet buffer only. A solution that has been explored during the research phase, but that unfortunately has not been found a way for achieving it, is applying a high pass filter to the height channels and a low pass filter to the centre channel for adding even more height feeling to the output result. The implementation of reverb for the rear channels, might also be a successful way for adding more depth and space feeling, which is the usual role of them.

5.2 Project summary

This project focuses on 3D immersive audio production within the realm of music. It comprehensively explores existing 3D audio technologies and highlights the recent advancements in Dolby Atmos Music technology, emphasizing its adaptability to harness the full potential of these technologies. The project also underscores the significance of time-based effects in the 3D context, with a specific focus on Delay effects, identifying a gap in the industry and a lack of solutions for producers and anyone entering the field of immersive music production.

As a proposed solution, a multichannel Delay plugin tailored for Dolby Atmos Music is introduced. This plugin can be seamlessly integrated into a 7.1.2 bed track while also affecting stereo tracks. The research investigates the up-mixing of stereo signals into multichannel formats, examining various methods that have been proposed for remixing stereo tracks into formats with more than two channels. Although no specific technique was adopted due to the complexity and the unique nature of the problem, particularly related to the specific processor in question, it was nonetheless valuable to revisit these methods to gain insights into the role of each speaker in the final audio result, how sound is perceived, and, consequently, how the three developed delay algorithms should leverage the various available channels.

The program was then implemented successfully using the JUCE framework, and the results were showcased through the use of external plugins such as Insight 2, along with a program developed in MATLAB. Results show that the program, despite some limitations which has been stated and discussed in the previous paragraph, achieved the aim of the being able to be a Multichannel Delay effect for a multichannel 7.1.2 bed track of Dolby Atmos workflow, offering the user to use three different types of delay successfully and also to control an high pass and low pass filter through a simple and user-friendly Graphic User Interface, which matches the will for making the program easy to use from any skill levels producers and pushing toward the standardization of the Immersive Audio within the Music Production world.

REFERENCIES

Anon., 2021. *What's the difference between beds and objects?*. [Online]
Available at: https://professionalsupport.dolby.com/s/article/What-s-the-difference-between-beds-and-objects?language=en_US
[Accessed 30 July 2023].

Borzym, H. L. a. K., 2020. *Creating Virtual Height Loudspeakers for Dolby Atmos and ,* Huddersfield: Audio Engineering Society.

Brewers, A., n.d. *ab Delay.* [Online]
Available at: https://www.audiobrewers.com/plugins/p/ab-delay
[Accessed 2023 06 12].

Carlos Avendano, J.-M. J., 2004. *A Frequency-Domain Approach to Multichannel Upmix,* Scotts Valley: Audio Engineering Society.

CHRISTOPHER J. KEYES, A. T., 2020. *Design and Evaluation of a Spectral Phase,* Hong Kong: Audio Engineering Society.

Ciniero, A., 2022. *Dolby Atmos Studios, applicazione del Dolby Atmos nelle sale di regia,* Torino: Politecnico di Torino.

ColemanPhilip, F. A. et al., 2017. *Object-Based Reverberation for Spatial Audio,* Southampton: Audio Engineering Society.

Ellis-Geiger, D. R. J., 2016. *Music Production for Dolby Atmos and Auro 3D.* Los Angeles, Audio Engineering Society.

Fiedler Audio, n.d. *Spacelab Interstellar.* [Online]
Available at: https://fiedler-audio.com/spacelab-interstellar/
[Accessed 17 September 2023].

Imagine Line, n.d. *Fruity Delay 3.* [Online]
Available at: https://www.image-line.com/fl-studio-learning/fl-studio-online-manual/html/plugins/Fruity%20Delay%203.htm
[Accessed 5 September 2023].

Izotope, n.d. *Insight 2, Intelligent metering for music & post.* [Online]
Available at: https://www.izotope.com/en/products/insight.html
[Accessed 19 September 2023].

Jon Downing, C. T., 2016. *REAL-TIME DIGITAL MODELING OF THE ROLAND SPACE ECHO,* Rochester: University of Rochester.

Juandagilc, n.d. *Audio-Effects.* [Online]
Available at: https://github.com/juandagilc/Audio-Effects/tree/master/Ping-Pong%20Delay
[Accessed 2 August 2023].

Juandagilc, n.d. *Audio-Effects.* [Online]
Available at: https://github.com/juandagilc/Audio-Effects/tree/master/Delay
[Accessed 2 August 2023].

JUCE, n.d. *AudioChannelSet Class Reference.* [Online]
Available at: https://docs.juce.com/master/classAudioChannelSet.html
[Accessed 23 August 2023].

JUCE, n.d. *AudioChannelSet Class Reference.* [Online]
Available at: https://docs.juce.com/master/classAudioChannelSet.html
[Accessed 12 August 2023].

JUCE, n.d. *AudioProcessorValueTreeState Class Reference.* [Online]
Available at: https://docs.juce.com/master/classAudioProcessorValueTreeState.html
[Accessed 17 August 2023].

JUCE, n.d. *dsp::IIR::Coefficients< NumericType > Struct Template Reference.* [Online]
Available at:
https://docs.juce.com/master/structdsp_1_1IIR_1_1Coefficients.html#a0d0a69cb3a91b4cee355c7fd2c0e9785
[Accessed 23 August 2023].

JUCE, n.d. *IIRCoefficients Class Reference.* [Online]
Available at:
https://docs.juce.com/master/classIIRCoefficients.html#a789e5863ae22b5a1c817c7156e406874
[Accessed 28 August 2023].

JUCE, n.d. *Tutorial: Configuring the right bus layouts for your plugins.* [Online]
Available at: https://docs.juce.com/master/tutorial_audio_bus_layouts.html
[Accessed 10 August 2023].

Justin Guitar, n.d. *DELAY: WHAT, HOW & HISTORY.* [Online]
Available at: https://www.justinguitar.com/guitar-lessons/echo-tape-delay-analog-delay-and-digital-delay-fx-301#:~:text=The%20history%20of%20Delay%20starts,composers%20such%20as%20Pierre%20Schaeffer.
[Accessed 15 September 2023].

Justin Paterson, G. L., 2019. Producing 3-D audio. In: *Producing Music.* New York: Routledge, Taylor and Francis.

Kelly, J., Woszczyk, W. & King, R., 2020. *Are you there? : A Literature Review of Presence for Immersive Music Production.* Online, Audio Engineering Society.

KIM, S., 2018. Height Channels. In: P. G. Agnieszka Roginska, ed. *Immersive Sound, The Art and Science of Binaural and Multi-Channel Audio.* New York: Routledge, pp. 221-241.

Kronlachner, M., 2015. *Plug-in Suite Mastering the Production and Playback in Surround Sound and Ambisonics,* Hamburg: Audio Engineering Society.

Laboratories Dolby, 2018. *Dolby Atmos Renderer: guide.* [Online]
Available at: https://professional.dolby.com/siteassets/content-creation/dolby-atmos/dolby_atmos_renderer_guide.pdf
[Accessed 2 August 2023].

Leonard, B., 2018. Applications of Extended Multichannel Techniques. In: P. G. Agnieszka Roginska, ed. *Immersive Sound, The Art and Science of Binaural and Multi-Channel Audio.* New York: Routledge, pp. 333-354.

Liquid Sonics, n.d. *Cinematic rooms.* [Online]
Available at: https://www.liquidsonics.com/software/cinematic-rooms/?gclid=Cj0KCQjwpompBhDZARIsAFD_Fp9Hts3VieS_rq8c9AiWM3QjkXsNBvbt8eB8GWwTsTwS3qrx7TmE-fAaAhYXEALw_wcB
[Accessed 30 September 2023].

Magazine, A., 2016. *The Cargo Cult Slapper.* [Online]
Available at: https://www.attackmagazine.com/reviews/gear-software/the-cargo-cult-slapper-st/
[Accessed 4 June 2023].

Martin, A. T. a. B., 2015. *The vertical precedence effect:,* New York: Audio Engineering Society.

Nicol, R., 2018. Sound Field. In: P. G. Agnieszka Roginska, ed. *Immersive Sound, The Art and Science of Binaural and Multi-Channel Audio.* New York: Routledge, pp. 276-300.

Pirkle, W. C., 2019. *Designing audio effects in C++.* 2nd ed. s.l.:Routledge.

Roginska, A., 2018. Binaural Audio Through Headphones. In: P. G. Agnieszka Roginska, ed. *Immersive Sound, The Art and Science of Binaural and Multi-Channel Audio.* New York: Routledge, pp. 88-117.

Rumsey, F., 2016. Immersive audio: Objects, mixing, and rendering. *Journal of the Audio Engineering Society.*

Rumsey, F., 2018. Surround Sound. In: P. G. Agnieszka Roginska, ed. *Immersive Sound, The Art and Science of Binaural and Multi-Channel Audio.* NewYork: Routledge, pp. 180-217.

Sebastian Kraf, U. Z., 2015. *STEREO SIGNAL SEPARATION AND UPMIXING BY MID-SIDE DECOMPOSITION IN.* Hamburg, Conference on Digital Audio Effects.

Sonics, L., n.d. *Lustrous Plates.* [Online]
Available at: https://www.liquidsonics.com/software/lustrous-plates/?gclid=Cj0KCQjwpompBhDZARIsAFD_Fp_voeirCg8-NhwNOQrGDGONq9g3OqHCZLI0gTzFNJNIyal5yEfIX88aAkUREALw_wcB
[Accessed 22 September 2023].

Sun, X., 2021. *Immersive audio, capture, transport and rendering: a review,* Cambridge: Cambridge University.

Tomás Oramus, P. N., 2020. *COMPARISON OF PERCEPTION OF SPATIAL LOCALIZATION.* s.l., Audio Engineering Society.

Tsingos, N., 2018. Object-Based Audio. In: P. G. Agnieszka Roginska, ed. *Immersive Sound, The Art and Science of Binaural and Multi-Channel Audio.* New York: Routledge, pp. 244-272.

Wagner, M. G., 2022. *HoRNet SAMP: the missing Dolby Atmos master bus.* [Online]
Available at: https://www.youtube.com/watch?v=JgvNWysH67o
[Accessed 11 June 2023].

Waves, n.d. *Spherix Immersive Compressor & Limiter.* [Online]
Available at: https://www.waves.com/plugins/spherix-immersive-compressor-limiter
[Accessed 21 September 2023].

Wikipedia, n.d. *Delay.* [Online]
Available at: https://it.wikipedia.org/wiki/Delay
[Accessed 13 September 2023].

YongGuk Kim, H. K. K., 2009. *Real-Time Conversion of Stereo Audio to 5.1 Channel Audio for Providing Realistic Sounds,* Gwangju: n International Journal of Signal Processing Image Processing and Pattern Recognition.

APPENDIX

PluginProcessor.cpp

```
/*

==============================================================================

    This file contains the basic framework code for a JUCE plugin processor.

==============================================================================
*/

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

using namespace juce;
using namespace std;

//==============================================================================
Atmos3DDelayAudioProcessor::Atmos3DDelayAudioProcessor()
#ifndef JucePlugin_PreferredChannelConfigurations
    : AudioProcessor (BusesProperties()
                     #if ! JucePlugin_IsMidiEffect
                      #if ! JucePlugin_IsSynth
                       .withInput  ("Input",  juce::AudioChannelSet::stereo(), true)
                      #endif
```

```
                    .withOutput ("Output", juce::AudioChannelSet::create7point1point2(), true)
                #endif
                  ), parameters(*this, nullptr, "Parameter", createParameters()),
                    lowPassFilter(dsp::IIR::Coefficients<float>::makeLowPass(48000,
20000.0f, 0.8f)),
                    highPassFilter(dsp::IIR::Coefficients<float>::makeHighPass(48000,
20000.0f, 0.8f))

#endif
{
}

Atmos3DDelayAudioProcessor::~Atmos3DDelayAudioProcessor()
{
}

//==============================================================================
const juce::String Atmos3DDelayAudioProcessor::getName() const
{
   return JucePlugin_Name;
}

bool Atmos3DDelayAudioProcessor::acceptsMidi() const
{
  #if JucePlugin_WantsMidiInput
   return true;
  #else
   return false;
  #endif
}
```

```
bool Atmos3DDelayAudioProcessor::producesMidi() const
{
  #if JucePlugin_ProducesMidiOutput
   return true;
  #else
   return false;
  #endif
}

bool Atmos3DDelayAudioProcessor::isMidiEffect() const
{
  #if JucePlugin_IsMidiEffect
   return true;
  #else
   return false;
  #endif
}

double Atmos3DDelayAudioProcessor::getTailLengthSeconds() const
{
   return 0.0;
}

int Atmos3DDelayAudioProcessor::getNumPrograms()
{
   return 1;   // NB: some hosts don't cope very well if you tell them there are 0 programs,
               // so this should be at least 1, even if you're not really implementing programs.
}
```

```
int Atmos3DDelayAudioProcessor::getCurrentProgram()
{
    return 0;
}

void Atmos3DDelayAudioProcessor::setCurrentProgram (int index)
{
}

const juce::String Atmos3DDelayAudioProcessor::getProgramName (int index)
{
    return {};
}

void Atmos3DDelayAudioProcessor::changeProgramName (int index, const juce::String&
newName)
{
}

//==============================================================================

void Atmos3DDelayAudioProcessor::prepareToPlay (double sampleRate, int
samplesPerBlock)
{
    // Reset Delay Buffer information
    float maxDelayTime = parameters.getParameterRange("delayTime").end;
    delayBufferChannels = getTotalNumInputChannels();
    delayBufferSamples = (int)(maxDelayTime * (float)sampleRate) + 1;
    delayBuffer0Channels = getTotalNumInputChannels();
    delayBuffer0Samples = (int)(maxDelayTime * (float)sampleRate) + 2;
```

```
    if (delayBufferSamples < 1) { delayBufferSamples = 1; }

    delayBuffer.setSize(delayBufferChannels, delayBufferSamples);
    delayBuffer0.setSize(delayBuffer0Channels, delayBuffer0Samples);
    delayBuffer.clear();
    delayBuffer0.clear();
    delayWritePosition = 0;

    //Pre-processing for LOW, HIGH, BAND PASS FILTERS
    dsp::ProcessSpec spec;
    spec.sampleRate = sampleRate;
    spec.maximumBlockSize = samplesPerBlock;
    spec.numChannels = getTotalNumOutputChannels();

    lowPassFilter.prepare(spec);
    lowPassFilter.reset();
    highPassFilter.prepare(spec);
    highPassFilter.reset();
}

void Atmos3DDelayAudioProcessor::releaseResources()
{
    // When playback stops, you can use this as an opportunity to free up any
    // spare memory, etc.
}

#ifndef JucePlugin_PreferredChannelConfigurations
bool Atmos3DDelayAudioProcessor::isBusesLayoutSupported(const BusesLayout& layouts)
const
{
#if JucePlugin_IsMidiEffect
```

```
    juce::ignoreUnused(layouts);
    return true;
#else
    // This is the place where you check if the layout is supported.
    // In this template code we only support mono or stereo.
    // Some plugin hosts, such as certain GarageBand versions, will only
    // load plugins that support stereo bus layouts.
        if (layouts.getMainOutputChannelSet() ==
juce::AudioChannelSet::create7point1point2()){ return true; }
        else { return false; }
#endif
}
#endif


void Atmos3DDelayAudioProcessor::updateLowpassFilter()
{
   auto lowpassFreq = parameters.getParameterAsValue("lowpass");
       float currentLowCutOff = lowpassFreq.getValue();
      *lowPassFilter.state = *dsp::IIR::Coefficients<float>::makeLowPass(lastSampleRate,
currentLowCutOff, 0.8f);
}
void Atmos3DDelayAudioProcessor::updateHighpassFilter()
{
     auto highpassFreq = parameters.getParameterAsValue("highpass");
    float currentHighCutOff = highpassFreq.getValue();
   *highPassFilter.state = *dsp::IIR::Coefficients<float>::makeHighPass(lastSampleRate,
currentHighCutOff, 0.8f);
}


void Atmos3DDelayAudioProcessor::processBlock (juce::AudioBuffer<float>& buffer,
juce::MidiBuffer& midiMessages)
{
```

```
//========== Variables ============================================//
ScopedNoDenormals noDenormals;

auto dTime      = parameters.getRawParameterValue("delayTime");
auto mix        = parameters.getRawParameterValue("mix");
auto feedback   = parameters.getRawParameterValue("feedback");
auto balance    = parameters.getRawParameterValue("balance");
auto choice     = parameters.getRawParameterValue("delay_option");
auto offset     = parameters.getRawParameterValue("offset");

currentDelayTime = (dTime->load()) * (float)getSampleRate();
currentMix          = (mix->load());
currentFeedback     = feedback->load();
currentBalance      = balance->load();
currentChoice       = choice->load();
currentOffset       = (offset->load()) * (float)getSampleRate();

int localWritePosition = delayWritePosition;

//========== Processing ===========================================//

// Gain control of input signal
inputGainControl(buffer);

// Perform DSP below
if (currentChoice == 0)
    PingPongDelay(buffer, localWritePosition);
else if (currentChoice == 1)
   SlapBackDelay(buffer, localWritePosition);
else if (currentChoice==2)
```

```
        MidSideDelay(buffer, localWritePosition);



    lpFilter(buffer);
    hpFilter(buffer);

    // Gain control of output signal
    outputGainControl(buffer);

    // This is here to avoid people getting screaming feedback when they first compile a plugin
    for (auto i = getTotalNumOutputChannels(); i < getTotalNumOutputChannels(); ++i) {
buffer.clear(i, 0, buffer.getNumSamples()); }
}

void Atmos3DDelayAudioProcessor::lpFilter(AudioBuffer<float>& inBuffer)
{
    AudioBlock <float> block(inBuffer);
    updateLowpassFilter();
    lowPassFilter.process(ProcessContextReplacing<float>(block));
}

void Atmos3DDelayAudioProcessor::hpFilter(AudioBuffer<float>& inBuffer)
{
    AudioBlock <float> block(inBuffer);
    updateHighpassFilter();
    highPassFilter.process(ProcessContextReplacing<float>(block));
}



void Atmos3DDelayAudioProcessor::inputGainControl(AudioBuffer<float>& buffer)
{
```

```
    auto gain = parameters.getRawParameterValue("inGain");
    float gainValue = gain->load();
    if (gainValue == startGain)
    {
        buffer.applyGain(gainValue);
    }
    else
    {
        buffer.applyGainRamp(0, buffer.getNumSamples(), startGain, gainValue);
        startGain = gainValue;
    }
}

void Atmos3DDelayAudioProcessor::outputGainControl(AudioBuffer<float>& buffer)
{
    auto gain = parameters.getRawParameterValue("outGain");
    float gainValue = gain->load();
    if (gainValue == finalGain)
    {
        buffer.applyGain(gainValue);
    }
    else
    {
        buffer.applyGainRamp(0, buffer.getNumSamples(), finalGain, gainValue);
        finalGain = gainValue;
    }
}

//==============================================================================

bool Atmos3DDelayAudioProcessor::hasEditor() const
```

```
{
    return true; // (change this to false if you choose to not supply an editor)
}



void Atmos3DDelayAudioProcessor::MidSideDelay(AudioBuffer<float>& buffer, int
localWritePosition)
{
    float* leftchannelData = buffer.getWritePointer(0);
    float* rightchannelData = buffer.getWritePointer(1);
    float* centerchannelData = buffer.getWritePointer(2);
    float* surroundleftchannelData = buffer.getWritePointer(6);
    float* surroundrightchannelData = buffer.getWritePointer(7);
    float* rearleftchannelData = buffer.getWritePointer(4);
    float* rearrightchannelData = buffer.getWritePointer(5);
    float* topleftchannelData = buffer.getWritePointer(8);
    float* toprightchannelData = buffer.getWritePointer(9);

    float* leftdelayData = delayBuffer.getWritePointer(0);
    float* rightdelayData = delayBuffer.getWritePointer(1);
    float* centerdelayData = delayBuffer.getWritePointer(2);
    float* surroundleftdelayData = delayBuffer.getWritePointer(6);
    float* surroundrightdelayData = delayBuffer.getWritePointer(7);
    float* rearleftdelayData = delayBuffer.getWritePointer(4);
    float* rearrightdelayData = delayBuffer.getWritePointer(5);
    float* topleftdelayData = delayBuffer.getWritePointer(8);
    float* toprightdelayData = delayBuffer.getWritePointer(9);

    for (int sample = 0; sample < buffer.getNumSamples(); ++sample)
    {
        // Input samples for each channel
```

```
const float sampleInput = (leftchannelData[sample] + rightchannelData[sample]) / 2;

// Output samples for each channel
float leftsampleOutput = 0.0f;
float rightsampleOutput = 0.0f;
float centersampleOutput = 0.0f;
float surroundleftsampleOutput = 0.0f;
float surroundrightsampleOutput = 0.0f;
float rearleftsampleOutput = 0.0f;
float rearrightsampleOutput = 0.0f;
float topleftsampleOutput = 0.0f;
float toprightsampleOutput = 0.0f;

// Obtain the position to read and write from and to the buffer
float centerReadPosition = fmodf((float)localWritePosition - currentDelayTime +
(float)delayBufferSamples, delayBufferSamples);
float rightReadPosition = fmodf((float)localWritePosition - (currentDelayTime -
currentOffset) + (float)delayBufferSamples, delayBufferSamples);
float leftReadPosition = fmodf((float)localWritePosition - (currentDelayTime +
currentOffset) + (float)delayBufferSamples, delayBufferSamples);


int centerLocalReadPosition = floorf(centerReadPosition);
int rightLocalReadPosition = floorf(rightReadPosition);
int leftLocalReadPosition = floorf(leftReadPosition);

if (centerLocalReadPosition != localWritePosition)
{
  //================================PROCESSING
DELAY=======================================//
  float centerFraction = centerReadPosition - (float)centerLocalReadPosition;
  float rightFraction = rightReadPosition - (float)rightLocalReadPosition;
```

```
        float leftFraction = leftReadPosition - (float)leftLocalReadPosition;


        float delayed1L = leftdelayData[(leftLocalReadPosition + 0)];
        float delayed1R = rightdelayData[(rightLocalReadPosition + 0)];
        float delayed1C = centerdelayData[(centerLocalReadPosition + 0)];


        float delayed2L = leftdelayData[(leftLocalReadPosition + 1) % delayBufferSamples];
        float delayed2R = rightdelayData[(rightLocalReadPosition + 1) %
delayBufferSamples];
        float delayed2C = centerdelayData[(centerLocalReadPosition + 1) %
delayBufferSamples];


        float delayed3SL = surroundleftdelayData[(leftLocalReadPosition + 0)];
        float delayed3SR = surroundrightdelayData[(rightLocalReadPosition + 0)];
        float delayed4SL = surroundleftdelayData[(leftLocalReadPosition + 1) %
delayBufferSamples];
        float delayed4SR = surroundrightdelayData[(rightLocalReadPosition + 1) %
delayBufferSamples];


        float delayed5RL = rearleftdelayData[(leftLocalReadPosition + 0)];
        float delayed5RR = rearrightdelayData[(rightLocalReadPosition + 0)];
        float delayed5TL = topleftdelayData[(centerLocalReadPosition + 0)];
        float delayed5TR = toprightdelayData[(centerLocalReadPosition + 0)];


        float delayed6RL = rearleftdelayData[(leftLocalReadPosition + 1) %
delayBufferSamples];
        float delayed6RR = rearrightdelayData[(rightLocalReadPosition + 1) %
delayBufferSamples];
        float delayed6TL = topleftdelayData[(centerLocalReadPosition + 1) %
delayBufferSamples];
        float delayed6TR = toprightdelayData[(centerLocalReadPosition + 1) %
delayBufferSamples];
```

```
        leftsampleOutput = delayed1L + leftFraction * (delayed2L - delayed1L);
        rightsampleOutput = delayed1R + rightFraction * (delayed2R - delayed1R);
        centersampleOutput = delayed1C + centerFraction * (delayed2C - delayed1C);
        surroundleftsampleOutput = delayed3SL + leftFraction * (delayed4SL - delayed3SL);
        surroundrightsampleOutput = delayed3SR + rightFraction * (delayed4SR - delayed3SR);
        rearleftsampleOutput = delayed5RL + leftFraction * (delayed6RL - delayed5RL);
        rearrightsampleOutput = delayed5RR + rightFraction * (delayed6RR - delayed5RR);
        topleftsampleOutput = delayed5TL + centerFraction * (delayed6TL - delayed5TL);
        toprightsampleOutput = delayed5TL + centerFraction * (delayed6TL - delayed5TL);

        //=========================MIX AND OUTPUT FOR CURRENT SAMPLE================================//
        leftchannelData[sample] = sampleInput * (1 - currentMix) + currentMix * (leftsampleOutput - sampleInput);
        rightchannelData[sample] = sampleInput * (1 - currentMix) + currentMix * (rightsampleOutput - sampleInput);
        centerchannelData[sample] = currentMix * (centersampleOutput);
        surroundleftchannelData[sample] = sampleInput + currentMix * (surroundleftsampleOutput - sampleInput);
        surroundrightchannelData[sample] = sampleInput + currentMix * (surroundrightsampleOutput - sampleInput);
        rearleftchannelData[sample] = sampleInput + currentMix * (rearleftsampleOutput - sampleInput);
        rearrightchannelData[sample] = sampleInput + currentMix * (rearrightsampleOutput - sampleInput);
        topleftchannelData[sample] = currentMix * (topleftsampleOutput);
        toprightchannelData[sample] = currentMix * (toprightsampleOutput);

        //leftdelayData[localWritePosition]        = centersampleInput      + rightsampleOutput       * currentFeedback;
        //rightdelayData[localWritePosition]       = centersampleInput      + leftsampleOutput * currentFeedback;
        centerdelayData[localWritePosition] = sampleInput;
```

```
        surroundleftdelayData[localWritePosition] =
(sampleInput)+surroundleftsampleOutput * currentFeedback;

        surroundrightdelayData[localWritePosition] =
(sampleInput)+surroundrightsampleOutput * currentFeedback;

        rearleftdelayData[localWritePosition] = (sampleInput)+rearleftsampleOutput *
currentFeedback;

        rearrightdelayData[localWritePosition] = (sampleInput)+rearrightsampleOutput *
currentFeedback;

        topleftdelayData[localWritePosition] = sampleInput + topleftsampleOutput *
currentFeedback;

        toprightdelayData[localWritePosition] = sampleInput + toprightsampleOutput *
currentFeedback;

    }


    if (++localWritePosition >= delayBufferSamples) { localWritePosition -=
delayBufferSamples; }


  }


  delayWritePosition = localWritePosition;


  for (int channel = getTotalNumInputChannels(); channel < getTotalNumOutputChannels();
++channel)

    buffer.clear(channel, 0, buffer.getNumSamples());

}



void Atmos3DDelayAudioProcessor::PingPongDelay(AudioBuffer<float>& buffer, int
localWritePosition)

{

  float* leftchannelData = buffer.getWritePointer(0);

  float* rightchannelData = buffer.getWritePointer(1);

  float* centerchannelData = buffer.getWritePointer(2);

  float* surroundleftchannelData = buffer.getWritePointer(6);
```

```
    float* surroundrightchannelData = buffer.getWritePointer(7);
    float* rearleftchannelData = buffer.getWritePointer(4);
    float* rearrightchannelData = buffer.getWritePointer(5);
    float* topleftchannelData = buffer.getWritePointer(8);

    float* toprightchannelData = buffer.getWritePointer(9);

    float* leftdelayData = delayBuffer.getWritePointer(0);
    float* rightdelayData = delayBuffer.getWritePointer(1);
    float* centerdelayData = delayBuffer.getWritePointer(2);
    float* surroundleftdelayData = delayBuffer.getWritePointer(6);
    float* surroundrightdelayData = delayBuffer.getWritePointer(7);
    float* rearleftdelayData = delayBuffer.getWritePointer(4);
    float* rearrightdelayData = delayBuffer.getWritePointer(5);
    float* topleftdelayData = delayBuffer.getWritePointer(8);
    float* toprightdelayData = delayBuffer.getWritePointer(9);

    for (int sample = 0; sample < buffer.getNumSamples(); ++sample)
    {
        // Input samples for each channel
        const float leftsampleInput = (1.0f - currentBalance) * leftchannelData[sample];
        const float rightsampleInput = currentBalance * rightchannelData[sample];
        const float centersampleInput = (leftchannelData[sample] + rightchannelData[sample]) /
2;

        // Output samples for each channel
        float leftsampleOutput = 0.0f;
        float rightsampleOutput = 0.0f;
        float centersampleOutput = 0.0f;
        float surroundleftsampleOutput = 0.0f;
        float surroundrightsampleOutput = 0.0f;
```

```
        float rearleftsampleOutput = 0.0f;
        float rearrightsampleOutput = 0.0f;
        float topleftsampleOutput = 0.0f;
        float toprightsampleOutput = 0.0f;

        // Obtain the position to read and write from and to the buffer
        float frontReadPosition = fmodf((float)localWritePosition - currentDelayTime +
(float)delayBufferSamples, delayBufferSamples);
        float midReadPosition  = fmodf((float)localWritePosition - (currentDelayTime -
currentOffset) + (float)delayBufferSamples, delayBufferSamples);
        float rearReadPosition = fmodf((float)localWritePosition - (currentDelayTime +
currentOffset) + (float)delayBufferSamples, delayBufferSamples);



        int frontLocalReadPosition = floorf(frontReadPosition);
        int midLocalReadPosition = floorf(midReadPosition);
        int rearLocalReadPosition = floorf(rearReadPosition);

        if (frontLocalReadPosition != localWritePosition)
        {
            //================================PROCESSING
DELAY=========================================//
            float frontFraction = frontReadPosition - (float)frontLocalReadPosition;
            float midFraction   = midReadPosition - (float)midLocalReadPosition;
            float rearFraction = rearReadPosition - (float)rearLocalReadPosition;

            float delayed1L = leftdelayData[(frontLocalReadPosition + 0)];
            float delayed1R = rightdelayData[(frontLocalReadPosition + 0)];
            float delayed1C = centerdelayData[(frontLocalReadPosition + 0)];

            float delayed2L = leftdelayData[(frontLocalReadPosition + 1) %
delayBufferSamples];
```

```
        float delayed2R = rightdelayData[(frontLocalReadPosition + 1) %
delayBufferSamples];
        float delayed2C = centerdelayData[(frontLocalReadPosition + 1) %
delayBufferSamples];


        float delayed3SL = surroundleftdelayData[(midLocalReadPosition + 0)];
        float delayed3SR = surroundrightdelayData[(midLocalReadPosition + 0)];
        float delayed4SL = surroundleftdelayData[(midLocalReadPosition + 1) %
delayBufferSamples];
        float delayed4SR = surroundrightdelayData[(midLocalReadPosition + 1) %
delayBufferSamples];


        float delayed5RL = rearleftdelayData[(rearLocalReadPosition + 0)];
        float delayed5RR = rearrightdelayData[(rearLocalReadPosition + 0)];
        float delayed5TL = topleftdelayData[(frontLocalReadPosition + 0)];
        float delayed5TR = toprightdelayData[(frontLocalReadPosition + 0)];


        float delayed6RL = rearleftdelayData[(rearLocalReadPosition + 1) %
delayBufferSamples];
        float delayed6RR = rearrightdelayData[(rearLocalReadPosition + 1) %
delayBufferSamples];
        float delayed6TL = topleftdelayData[(frontLocalReadPosition + 1) %
delayBufferSamples];
        float delayed6TR = toprightdelayData[(frontLocalReadPosition + 1) %
delayBufferSamples];


        leftsampleOutput          = delayed1L     + frontFraction * (delayed2L - delayed1L);
        rightsampleOutput         = delayed1R     + frontFraction * (delayed2R - delayed1R);
        centersampleOutput        = delayed1C     + frontFraction * (delayed2C - delayed1C);
        surroundleftsampleOutput  = delayed3SL    + midFraction   * (delayed4SL -
delayed3SL);
        surroundrightsampleOutput = delayed3SR    + midFraction   * (delayed4SR -
delayed3SR);
```

```
        rearleftsampleOutput        = delayed5RL    + rearFraction * (delayed6RL -
delayed5RL);

        rearrightsampleOutput       = delayed5RR    + rearFraction * (delayed6RR -
delayed5RR);

        topleftsampleOutput         = delayed5TL    + frontFraction * (delayed6TL -
delayed5TL);

        toprightsampleOutput        = delayed5TL    + frontFraction * (delayed6TL -
delayed5TL);


        //=========================MIX AND OUTPUT FOR CURRENT
SAMPLE================================//

        leftchannelData[sample]           = leftsampleInput   *   (1 - currentMix)     +
currentMix   * (leftsampleOutput- leftsampleInput);

        rightchannelData[sample]          = rightsampleInput  *   (1 - currentMix)     +
currentMix   * (rightsampleOutput - rightsampleInput);

        centerchannelData[sample]         =                   currentMix   * (centersampleOutput);

        surroundleftchannelData[sample]     = leftsampleInput   + currentMix   *
(surroundleftsampleOutput-leftsampleInput);

        surroundrightchannelData[sample]    = rightsampleInput  + currentMix   *
(surroundrightsampleOutput - rightsampleInput);

        rearleftchannelData[sample]         = leftsampleInput   + currentMix   *
(rearleftsampleOutput - leftsampleInput);

        rearrightchannelData[sample]        = rightsampleInput  + currentMix   *
(rearrightsampleOutput - rightsampleInput);

        topleftchannelData[sample]          =                   currentMix   * (topleftsampleOutput);

        toprightchannelData[sample]         =                   currentMix   *
(toprightsampleOutput);


        //leftdelayData[localWritePosition]         = leftsampleInput       + rightsampleOutput
* currentFeedback;

        //rightdelayData[localWritePosition]        = rightsampleInput      + leftsampleOutput
* currentFeedback;

        centerdelayData[localWritePosition]       = centersampleInput;

        surroundleftdelayData[localWritePosition]   = (leftsampleInput)     +
surroundrightsampleOutput * currentFeedback;
```

```
        surroundrightdelayData[localWritePosition]  = (rightsampleInput)    +
surroundleftsampleOutput  * currentFeedback;

        rearleftdelayData[localWritePosition]       = (leftsampleInput)     +
rearrightsampleOutput     * currentFeedback;

        rearrightdelayData[localWritePosition]      = (rightsampleInput)    +
rearleftsampleOutput      * currentFeedback;

        topleftdelayData[localWritePosition]        = rightsampleInput      +
toprightsampleOutput      * currentFeedback;

        toprightdelayData[localWritePosition]       = leftsampleInput       +
topleftsampleOutput       * currentFeedback;

      }


      if (++localWritePosition >= delayBufferSamples) { localWritePosition -=
delayBufferSamples; }


   }


   delayWritePosition = localWritePosition;


   for (int channel = getTotalNumInputChannels(); channel < getTotalNumOutputChannels();
++channel)

      buffer.clear(channel, 0, buffer.getNumSamples());

}


void Atmos3DDelayAudioProcessor::SlapBackDelay(AudioBuffer<float>& buffer, int
localWritePosition)

{

   for (int channel = 0; channel < 10; ++channel)

   {

      float* inputData;

      if (channel != 3)

      {

         //Duplicate original stereo channels to multi-channel
```

```
if (channel == 1 || channel == 4 || channel == 7 || channel == 8)
{
   // Left Channel Data
   inputData = buffer.getWritePointer(0);
}
else if (channel == 0 || channel == 5 || channel == 6 || channel == 9)
{
   // Right Channel Data
   inputData = buffer.getWritePointer(1);
}
else
{
   // Center Channel Data
   inputData = buffer.getWritePointer(0);
}

float* channelData = buffer.getWritePointer(channel);
float* delayData = delayBuffer.getWritePointer(channel);
localWritePosition = delayWritePosition;

for (int sample = 0; sample < buffer.getNumSamples(); ++sample)
{
   const float in = (1/sqrt(2))*inputData[sample];
   float out = 0.0f;

   float readPosition = fmodf((float)localWritePosition - (currentDelayTime) +
(float)delayBufferSamples, delayBufferSamples);
   int localReadPosition = floorf(readPosition);

   if (localReadPosition != localWritePosition)
   {
```

```
                float fraction = readPosition - (float)localReadPosition;

                float delayed1 = delayData[(localReadPosition + 0)];
                float delayed2 = delayData[(localReadPosition + 1) % delayBufferSamples];
                out = delayed1 + fraction* (delayed2 - delayed1);

                channelData[sample] = in*(1-currentMix) + (currentMix * (out - in));
                delayData[localWritePosition] = in + out * currentFeedback;
            }

            if (++localWritePosition >= delayBufferSamples)
                localWritePosition -= delayBufferSamples;
        }
    }
  }

  delayWritePosition = localWritePosition;

  for (int channel = getTotalNumInputChannels(); channel < getTotalNumOutputChannels();
++channel)
      buffer.clear(channel, 0, buffer.getNumSamples());
}

juce::AudioProcessorEditor* Atmos3DDelayAudioProcessor::createEditor()
{
  return new Atmos3DDelayAudioProcessorEditor (*this);
}

//==============================================================================
==========

void Atmos3DDelayAudioProcessor::getStateInformation (juce::MemoryBlock& destData)
```

```
{
    auto state = parameters.copyState();
    unique_ptr<XmlElement> xml(state.createXml());
    copyXmlToBinary(*xml, destData);
}

void Atmos3DDelayAudioProcessor::setStateInformation (const void* data, int sizeInBytes)
{
    unique_ptr<XmlElement> xmlState(getXmlFromBinary(data, sizeInBytes));

    if (xmlState.get() != nullptr)
    {
        if (xmlState->hasTagName(parameters.state.getType())) {
parameters.replaceState(ValueTree::fromXml(*xmlState)); }
    }
}

AudioProcessorValueTreeState::ParameterLayout
Atmos3DDelayAudioProcessor::createParameters()
{

    // Parameter Vector
    vector<unique_ptr<RangedAudioParameter>> parameterVector;

    // Input Gain
    parameterVector.push_back(make_unique<AudioParameterFloat>("inGain",
"Input Gain",   0.0f, 2.0f, 1.0f));

    // Delay Time and Mix
    parameterVector.push_back(make_unique<AudioParameterFloat>("delayTime",
"Delay Time",      0.0f, 4.0f, 2.0f));
```

```
    parameterVector.push_back(make_unique<AudioParameterFloat>("mix",
"Mix",          0.0f, 1.0f, 0.5f));

    parameterVector.push_back(make_unique<AudioParameterFloat>("feedback",
"Feedback",     0.0f, 0.9f, 0.5f));

    parameterVector.push_back(make_unique<AudioParameterFloat>("balance",
"Balance",      0.0f, 1.0f, 0.5f));

    parameterVector.push_back(make_unique<AudioParameterFloat>("offset",
"Offset",       -1.0f, 1.0f, 0.0f));


    // Low Pass, high pass, band pass

    parameterVector.push_back(make_unique<AudioParameterFloat>("lowpass",
"Low Pass",     1000.0f, 20000.0f, 5000.0f));

    parameterVector.push_back(make_unique<AudioParameterFloat>("highpass",
"High Pass", 50.0f, 15000.0f, 5000.0f));


    // Output Gain

    parameterVector.push_back(make_unique<AudioParameterFloat>("outGain",
"Output Gain",  0.0f, 2.0f, 1.0f));


    // Delay Options

    // StringArray for Options

    StringArray choices; choices.insert(1, "Ping-Pong"); choices.insert(2, "Normal");
choices.insert(3, "MidSide");

    parameterVector.push_back(make_unique<AudioParameterChoice>("delay_option",
"Delay Options", choices, 1));


    return { parameterVector.begin(), parameterVector.end() };

}


//==============================================================================

// This creates new instances of the plugin..

juce::AudioProcessor* JUCE_CALLTYPE createPluginFilter()

{
```

```
    return new Atmos3DDelayAudioProcessor();
}
```

PluginProcessor.h

```
/*
  ==============================================================================

    This file contains the basic framework code for a JUCE plugin processor.

  ==============================================================================
*/

#pragma once

#include <JuceHeader.h>

using namespace juce;
using namespace std;
using namespace dsp;

//==============================================================================
/**
*/
class Atmos3DDelayAudioProcessor : public juce::AudioProcessor
{
public:
    //==============================================================================
    Atmos3DDelayAudioProcessor();
    ~Atmos3DDelayAudioProcessor() override;

    //==============================================================================
    void prepareToPlay (double sampleRate, int samplesPerBlock) override;
    void releaseResources() override;

   #ifndef JucePlugin_PreferredChannelConfigurations
    bool isBusesLayoutSupported (const BusesLayout& layouts) const override;
   #endif

    //void updateFilter();
    void updateLowpassFilter();
    void updateHighpassFilter();

    void processBlock (juce::AudioBuffer<float>&, juce::MidiBuffer&) override;

    void lpFilter(AudioBuffer<float>& buffer);
    void hpFilter(AudioBuffer<float>& buffer);

    // Functions of Input and Output Gain
    void inputGainControl(AudioBuffer<float>& buffer);
    void outputGainControl(AudioBuffer<float>& buffer);

    //Functions for Delay Processing
    void MidSideDelay(AudioBuffer<float>& buffer, int localWritePosition);
    void PingPongDelay(AudioBuffer<float>& buffer, int localWritePosition);
    void SlapBackDelay(AudioBuffer<float>& buffer, int localWritePosition);

    //==============================================================================
    juce::AudioProcessorEditor* createEditor() override;
    bool hasEditor() const override;

    //==============================================================================
    const juce::String getName() const override;
```

```
    bool acceptsMidi() const override;
    bool producesMidi() const override;
    bool isMidiEffect() const override;
    double getTailLengthSeconds() const override;

    //==============================================================================
    int getNumPrograms() override;
    int getCurrentProgram() override;
    void setCurrentProgram (int index) override;
    const juce::String getProgramName (int index) override;
    void changeProgramName (int index, const juce::String& newName) override;

    //==============================================================================
    void getStateInformation (juce::MemoryBlock& destData) override;
    void setStateInformation (const void* data, int sizeInBytes) override;


    AudioProcessorValueTreeState    parameters;

private:

    //User Variables
    float                       currentDelayTime, currentMix, currentFeedback,
currentBalance, currentChoice, currentOffset;

    // Variables
    float                       startGain, finalGain, lastSampleRate{48000};
    int                         delayBufferSamples, delayBufferChannels,
delayWritePosition;

    AudioSampleBuffer           delayBuffer;


    ProcessorDuplicator<IIR::Filter <float>, IIR::Coefficients <float>> lowPassFilter;
    ProcessorDuplicator<IIR::Filter <float>, IIR::Coefficients <float>>
highPassFilter;

    // Functions
    AudioProcessorValueTreeState::ParameterLayout createParameters();

    //==============================================================================
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (Atmos3DDelayAudioProcessor)
};
```

PluginEditor.cpp

```
/*
  ==============================================================================

    This file contains the basic framework code for a JUCE plugin editor.

  ==============================================================================
*/

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

using namespace juce;
using namespace std;

//==============================================================================
Atmos3DDelayAudioProcessorEditor::Atmos3DDelayAudioProcessorEditor(Atmos3DDelayAudioPr
ocessor& p)
    : AudioProcessorEditor (&p), audioProcessor (p)
{
    // Make sure that before the constructor has finished, you've set the
    // editor's size to whatever you need it to be.
    buildElements();


    setSize(1000, 500);

    startTimerHz(60);
}

Atmos3DDelayAudioProcessorEditor::~Atmos3DDelayAudioProcessorEditor()
{
}

//==============================================================================
void Atmos3DDelayAudioProcessorEditor::paint (juce::Graphics& g)
{
    Image background = ImageCache::getFromMemory(BinaryData::background_png,
BinaryData::background_pngSize);
    g.drawImageAt(background, 0, 0);

    //Draw the semi-transparent rectangle around components
    const Rectangle<float> area(10, 20, 960, 460);
    g.setColour(Colours::ghostwhite);
    g.drawRoundedRectangle(area, 5.0f, 3.0f);

    //Draw background for rectangle
    g.setColour(Colours::black);
    g.setOpacity(0.5f);
    g.fillRoundedRectangle(area, 5.0f);

    //Draw text labels for each component
    g.setColour(Colours::white);
    g.setFont(18.0f);

    // Input Gain
    g.drawText("Input Gain",    0.03, 430,    200, 50, Justification::centred, false);

    //// Delay Time and Mix and Feedback
```

```
    g.drawText("Delay Time",        130, 435,    200, 50, Justification::centred,
false);
    g.drawText("Feedback",          260, 435,    200, 50, Justification::centred,
false);
    g.drawText("Mix",               390, 435,    200, 50, Justification::centred,
false);
    g.drawText("Balance",           150, 265,   200, 50, Justification::centred,
false);
    g.drawText("Offset",            360, 265,   200, 50, Justification::centred,
false);

    //// Low Pass
    g.drawText("High Cut",          720, 430,    200, 50, Justification::centred,
false);
    g.drawText("LOW PASS FILTER",   720, 265, 200, 50, Justification::centred, false);

    //// High Pass
    g.drawText("Low Cut",           720, 230 , 200, 50, Justification::centred,
false);
    g.drawText("HIGH PASS FILTER",  720, 65, 200, 50, Justification::centred, false);


    //// Output Gain
    g.drawText("Output Gain",       550, 430, 200, 50, Justification::centred, false);

    // Title of PlugIn
    g.setFont(35.0f);
    g.drawText("Atmos 3D-Delay",    125, 20, 1160, 75, Justification::centred, false);

    if (delayOptions.getSelectedId() == 1)
    {
        balanceSlider.setVisible(true);
        offsetKnob.setVisible(true);
    }
    else if (delayOptions.getSelectedId() == 2)
    {
        balanceSlider.setVisible(false);
        offsetKnob.setVisible(false);
    }
    else if (delayOptions.getSelectedId() == 3)
    {
        balanceSlider.setVisible(false);
        offsetKnob.setVisible(true);
    }
}

void Atmos3DDelayAudioProcessorEditor::resized()
{
    // This is generally where you'll want to lay out the positions of any
subcomponents in your editor.
    // List of Options
    delayOptions.setBounds      (50, 50, 400, 50);

    // Input Gain Slider
    inputGainSlider.setBounds   (15,    160,    150, 275);

    // Delay Time Knob and Mix Knob
    delayTimeKnob.setBounds     (170, 320, 120, 120);
    mixKnob.setBounds           (430, 320, 120, 120);
    feedbackSlider.setBounds    (300, 320, 120, 120);
    balanceSlider.setBounds     (200, 150, 120, 120);
    offsetKnob.setBounds        (400, 150, 120, 120);
```

```
    // Low Pass Knob
    lowCutSlider.setBounds      (750,  100, 140, 140);
    highCutSlider.setBounds     (750, 300, 140, 140);

    // Output Gain Slider
    outputGainSlider.setBounds  (580, 160, 150, 275);

}

void Atmos3DDelayAudioProcessorEditor::timerCallback()
{

}

void Atmos3DDelayAudioProcessorEditor::buildElements()
{
    delayOptVal =
make_unique<AudioProcessorValueTreeState::ComboBoxAttachment>(audioProcessor.parameter
s, "delay_option", delayOptions);
    delayOptions.setEditableText(false);
    delayOptions.addItem("Ping-Pong", 1);
    delayOptions.addItem("Normal", 2);
    delayOptions.addItem("MidSide", 3);
    delayOptions.setSelectedId(1);
    addAndMakeVisible(&delayOptions);

    //Building the Input Gain
    inputGainVal =
make_unique<AudioProcessorValueTreeState::SliderAttachment>(audioProcessor.parameters,
"inGain", inputGainSlider);
    inputGainSlider.setSliderStyle(Slider::SliderStyle::LinearVertical);
    inputGainSlider.setTextBoxStyle(Slider::TextBoxBelow, false, 100, 20);
    inputGainSlider.setRange(0.0f, 2.0f);
    addAndMakeVisible(&inputGainSlider);

    //Building the Delay Time Knob
    delayTimeVal =
make_unique<AudioProcessorValueTreeState::SliderAttachment>(audioProcessor.parameters,
"delayTime", delayTimeKnob);
    delayTimeKnob.setSliderStyle(Slider::SliderStyle::RotaryHorizontalDrag);
    delayTimeKnob.setTextBoxStyle(Slider::TextBoxBelow, false, 100, 20);
    delayTimeKnob.setTextValueSuffix(" s");
    delayTimeKnob.setRange(0.0f, 4.0f);
    addAndMakeVisible(&delayTimeKnob);

    //Building the Offset Time Knob
    offsetVal =
make_unique<AudioProcessorValueTreeState::SliderAttachment>(audioProcessor.parameters,
"offset", offsetKnob);
    offsetKnob.setSliderStyle(Slider::SliderStyle::RotaryHorizontalDrag);
    offsetKnob.setTextBoxStyle(Slider::TextBoxBelow, false, 100, 20);
    offsetKnob.setTextValueSuffix(" s");
    offsetKnob.setRange(-1.0f, 1.0f);
    addChildComponent(&offsetKnob);
    //addAndMakeVisible(&offsetKnob);

    //Building the Mix Knob
    mixVal =
make_unique<AudioProcessorValueTreeState::SliderAttachment>(audioProcessor.parameters,
"mix", mixKnob);
    mixKnob.setSliderStyle(Slider::SliderStyle::RotaryHorizontalDrag);
```

```
    mixKnob.setTextBoxStyle(Slider::TextBoxBelow, false, 100, 20);
    mixKnob.setRange(0.0f, 1.0f); mixKnob.setTextValueSuffix(" %");
    addAndMakeVisible(&mixKnob);

    //Building the Feedback Knob
    feedbackVal =
make_unique<AudioProcessorValueTreeState::SliderAttachment>(audioProcessor.parameters,
"feedback", feedbackSlider);
    feedbackSlider.setSliderStyle(Slider::SliderStyle::RotaryHorizontalDrag);
    feedbackSlider.setTextBoxStyle(Slider::TextBoxBelow, false, 100, 20);
    feedbackSlider.setRange(0.0f, 1.0f);
    addAndMakeVisible(&feedbackSlider);

    //Building the Balance Knob
    balanceVal =
make_unique<AudioProcessorValueTreeState::SliderAttachment>(audioProcessor.parameters,
"balance", balanceSlider);
    balanceSlider.setSliderStyle(Slider::SliderStyle::RotaryHorizontalDrag);
    balanceSlider.setTextBoxStyle(Slider::TextBoxBelow, false, 100, 20);
    balanceSlider.setRange(0.0f, 1.0f);
    addChildComponent(&balanceSlider);
//    addAndMakeVisible(&balanceSlider);

    //Building the Low Pass Slider
    lowpassVal =
make_unique<AudioProcessorValueTreeState::SliderAttachment>(audioProcessor.parameters,
"lowpass", highCutSlider);
    highCutSlider.setSliderStyle(Slider::SliderStyle::RotaryHorizontalVerticalDrag);
    highCutSlider.setTextBoxStyle(Slider::TextBoxBelow, false, 100, 20);
    highCutSlider.setRange(1000.0f, 20000.0f); highCutSlider.setTextValueSuffix("
Hz");
    addAndMakeVisible(&highCutSlider);

    //Building the High Pass Slider
    highpassVal =
make_unique<AudioProcessorValueTreeState::SliderAttachment>(audioProcessor.parameters,
"highpass", lowCutSlider);
    lowCutSlider.setSliderStyle(Slider::SliderStyle::RotaryHorizontalVerticalDrag);
    lowCutSlider.setTextBoxStyle(Slider::TextBoxBelow, false, 100, 20);
    lowCutSlider.setRange(50.0f, 15000.0f); lowCutSlider.setTextValueSuffix(" Hz");
    addAndMakeVisible(&lowCutSlider);

    //Building the Output Gain
    outputGainVal =
make_unique<AudioProcessorValueTreeState::SliderAttachment>(audioProcessor.parameters,
"outGain", outputGainSlider);
    outputGainSlider.setSliderStyle(Slider::SliderStyle::LinearVertical);
    outputGainSlider.setTextBoxStyle(Slider::TextBoxBelow, false, 100, 20);
    outputGainSlider.setRange(0.0f, 2.0f);
    addAndMakeVisible(&outputGainSlider);
}
```

PluginEditor.h

```
/*
  ==============================================================================

    This file contains the basic framework code for a JUCE plugin editor.

  ==============================================================================
*/

#pragma once

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

using namespace juce;
using namespace std;

//==============================================================================
/**
*/
class Atmos3DDelayAudioProcessorEditor : public juce::AudioProcessorEditor, public
Timer
{
public:
    Atmos3DDelayAudioProcessorEditor(Atmos3DDelayAudioProcessor&);
    ~Atmos3DDelayAudioProcessorEditor() override;

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

    unique_ptr<AudioProcessorValueTreeState::SliderAttachment> inputGainVal;        //
Attachment for Input Gain

    unique_ptr<AudioProcessorValueTreeState::SliderAttachment> delayTimeVal;        //
Attachment for Delay Time
    unique_ptr<AudioProcessorValueTreeState::SliderAttachment> mixVal;              //
Attachment for Delay Mix Value
    unique_ptr<AudioProcessorValueTreeState::SliderAttachment> feedbackVal;         //
Attachment for Feedback Value
    unique_ptr<AudioProcessorValueTreeState::SliderAttachment> balanceVal;         //
Attachment for Balance Value

    unique_ptr<AudioProcessorValueTreeState::SliderAttachment> lowpassVal;          //
Attachment for Low Pass Value
    unique_ptr<AudioProcessorValueTreeState::SliderAttachment> highpassVal;
// Attachment for Low Pass Value

    unique_ptr<AudioProcessorValueTreeState::SliderAttachment> outputGainVal;       //
Attachment for Output Gain
    unique_ptr<AudioProcessorValueTreeState::SliderAttachment> offsetVal;;
    unique_ptr<AudioProcessorValueTreeState::ComboBoxAttachment> delayOptVal;
// Attachment for Delay Option Value

private:
```

```
    // This reference is provided as a quick way for your editor to
    // access the processor object that created it.
    Atmos3DDelayAudioProcessor& audioProcessor;

    Slider      inputGainSlider;        // Slider for Input Gain

    Slider      delayTimeKnob;          // Knob for Delay Time
    Slider      mixKnob;                // Knob for Delay Mix
    Slider      feedbackSlider;         // Slider for Feedback
    Slider      balanceSlider;         // Slider for Balance
    Slider      offsetKnob;             // Slider for Offset

    Slider      lowCutSlider;           // Slider for Low Cut
    Slider      highCutSlider;          // Slider for High Cut

    Slider      outputGainSlider;       // Slider for Output Gain

    ComboBox    delayOptions;       // Options of Delay

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (Atmos3DDelayAudioProcessorEditor)
};
```