UNIVERSIDADE ESTADUAL PAULISTA "JÚLIO DE MESQUITA FILHO"
FACULDADE DE CIÊNCIAS - CAMPUS BAURU
DEPARTAMENTO DE COMPUTAÇÃO
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

LUCAS BASTOS DE FREITAS

EFEITOS SONOROS PARAMETRIZÁVEIS PARA GUITARRA ATRAVÉS DE PROCESSAMENTO DE SINAIS DIGITAIS

BAURU
2019

LUCAS BASTOS DE FREITAS

EFEITOS SONOROS PARAMETRIZÁVEIS PARA GUITARRA ATRAVÉS DE PROCESSAMENTO DE SINAIS DIGITAIS

Trabalho de Conclusão de Curso de Bacharelado em Ciência da Computação da Universidade Estadual Paulista "Júlio de Mesquita Filho", Faculdade de Ciências, Campus Bauru.
Orientador: Prof. Dr. Kleber Rocha de Oliveira

BAURU
2019

Lucas Bastos de Freitas EFEITOS SONOROS PARAMETRIZÁVEIS PARA GUITARRA ATRAVÉS DE PROCESSAMENTO DE SINAIS DIGITAIS/ Lucas Bastos de Freitas. – Bauru, 2019-
68 p. : il. (algumas color.) ; 30 cm.
Orientador: Prof. Dr. Kleber Rocha de Oliveira
Trabalho de Conclusão de Curso – Universidade Estadual Paulista "Júlio de Mesquita Filho"
Faculdade de Ciências
Bacharelado em Ciência da Computação, 2019.
1. Efeitos sonoros 2. Guitarra 3. Processamento de sinais digitais 4. Home Studio 5. Direct Input 6. JUCE

Lucas Bastos de Freitas

EFEITOS SONOROS PARAMETRIZÁVEIS PARA GUITARRA ATRAVÉS DE PROCESSAMENTO DE SINAIS DIGITAIS

Trabalho de Conclusão de Curso de Bacharelado em Ciência da Computação da Universidade Estadual Paulista "Júlio de Mesquita Filho", Faculdade de Ciências, Campus Bauru.

Banca Examinadora

Prof. Dr. Kleber Rocha de Oliveira
Orientador
Universidade Estadual Paulista "Júlio de Mesquita Filho"
Faculdade de Ciências
Departamento de Computação

Profa. Dra. Simone Domingues Prado
Universidade Estadual Paulista "Júlio de Mesquita Filho"
Faculdade de Ciências
Departamento de Computação

Profa. Dra. Andrea Carla Gonçalves Vianna
Universidade Estadual Paulista "Júlio de Mesquita Filho"
Faculdade de Ciências
Departamento de Computação

Bauru, 11 de Novembro de 2019.

Agradecimentos

Agradeço aos meus pais, meus avós, meus amigos, à república Alabama, e todos os outros que botaram fé em mim.

When the power of love overcomes the love of power, the world will know peace.
Jimi Hendrix

Resumo

Devido à revolução digital que se iniciou na década de 60, foi possível usar técnicas de processamento de sinais digitais para suprir as necessidades que surgiam dos consumidores, ocasionando a evolução e barateamento de tecnologias usadas em estúdios de gravação. Por este motivo, se tornou cada vez mais frequente o uso de estúdios de gravação caseiros, tanto entre artistas profissionais quanto entusiastas. Paralelamente a isso, com o surgimento dos computadores digitais pessoais, também sucederam *softwares* voltados para produção musical. Entre estes, destacam-se os *plug-ins* de áudio, que podem ser utilizados, por exemplo, para acrescentar efeitos sonoros à gravações. O presente Trabalho de Conclusão de Curso trata sobre o uso do *framework* JUCE para a criação de *plug-ins* processadores de áudio. Técnicas de processamento de sinais digitais são utilizadas na programação desses *plug-ins* para gerar efeitos sonoros voltados para guitarra, que poderão ser parametrizados pelo usuário final, permitindo que este possa personalizar e encadear efeitos à sua própria maneira.

Palavras-chave: Processamento de sinais digitais. Efeitos sonoros. Guitarra. JUCE. Plug-in.

Abstract

Due to the digital revolution that began in the 60's, it became possible to use digital signal processing techniques to supply consumer necessities, causing the evolution and cheapening of technologies used in recording studios. Because of this, the use of home recording studios became more frequent, by both professional artists and enthusiasts. In parallel, with the rise of digital personal computers, also came music production softwares. Audio plug-ins stand out among these, for they can be used, for instance, to add sound effects to recordings. This Bachelor Final Project is about utilizing the JUCE framework for the development of audio processing plug-ins. Digital signal processing techniques are used in the programming of these plug-ins to generate guitar-based sound effects, which can then be parameterized by the user, allowing them to personalize and chain effects at their own will.

Keywords: Digital signal processing. Sound effects. Electric guitar. JUCE. Plug-in.

Lista de ilustrações

Lista de códigos

Lista de abreviaturas e siglas

ADC	*Conversor Analógico-Digital*
AU	*Audio Unit*
DAW	*Digital Audio Workstation*
DI	*Direct Input*
DSP	*Digital Signal Processing*
IDE	*Integrated Development Environment*
SDK	*Software Development Kit*
VST	*Virtual Studio Technology*

Sumário

1 Introdução

Continuamente a raça humana busca novas maneiras de se expressar: seja através da própria voz, para se comunicar, ou através da manipulação de objetos, para criar instrumentos musicais. Há mais de quarenta mil anos se faz isto (HIGHAM; BASELL; JACOBI, 2012), e por muito tempo haviam disponíveis somente métodos analógicos para se alterar a sonoridade de um instrumento. Porém, graças à revolução digital da década de 60, e estudo de técnicas especializadas em processamento de sinais digitais, hoje em dia quem se interessar pelo assunto possui uma profusa gama de opções quando se trata de criar e gravar músicas.

Com o passar do tempo, equipamentos básicos dos estúdios de gravação tiveram suas formas alteradas pelo tempo, diminuindo em tamanho e em custo, enquanto aumentando em potência. Isso facilitou a tarefa, até então árdua, de se gravar músicas, podendo ser feita até mesmo em casa. Este é o conceito de *home studio*: um lugar da casa onde é possível gravar e produzir músicas, com a qualidade de um estúdio profissional. Este termo é normalmente associado ao seu baixo custo: apenas um computador moderno e uma interface de áudio são necessários para gravação e produção de músicas.

Quando uma interface de áudio é conectada a um computador, atua como uma placa de áudio externa deste, podendo ser usada para gravar um sinal de áudio digital correspondente a um sinal analógico de entrada. Isso permite que o som de muitos instrumentos musicais seja gravado de maneira prática, podendo ser manipulado com minúcia através de *softwares*. Instrumentos acústicos (i.e. que movimentam o ar para produzir som, tais como voz, saxofone, bateria) ainda requerem pelo menos um microfone para a captação do seu som. No entanto, um instrumento elétrico (cuja saída é um sinal elétrico de áudio, tal como a guitarra) pode ser conectado, através de um cabo comum, em uma interface de áudio. Isso permite que o som do instrumento seja gravado diretamente, sem o uso de amplificadores ou microfones. Daí o nome desta técnica de gravação: *Direct Input* (DI), ou seja, entrada direta.

Como mencionado anteriormente, é necessário que seja instalado no computador um *software* capaz de manipular sinais digitais de áudio. Um *software* deste tipo se chama *Digital Audio Workstation* (DAW), e nas últimas décadas, se estabeleceu como a mais predominante ferramenta de produção musical nos estúdios modernos - tanto profissionais quanto caseiros (KIRBY, 2015). Uma DAW é essencialmente um estúdio virtual. Com ela, é possível realizar toda espécie de ajustes em áudios digitais - tanto na gravação quanto na produção de músicas. Imagine vastos consoles de áudio, em uma enorme sala de controle, porém, isso tudo num grande *software*. Sua popularidade está relacionada com a grande praticidade que traz, e alto nível de atenção a detalhes: existem até mesmo DAWs *open-source* que podem ser usadas em cenários profissionais (STUDIO, 2019).

Frequentemente o uso de DAWs está relacionado ao uso de *plug-ins* de áudio - são pequenos aplicativos que podem ser carregados nas DAWs para acrescentar funcionalidades. Em verdade, um *plug-in* é uma biblioteca dinâmica, e geralmente é usado para processar um único efeito sonoro, replicando processadores de *hardware*. Desta maneira, é possível que um usuário carregue diversos *plug-ins* dentro de uma DAW, agregando flexibilidade e versatilidade na produção de músicas. Além disso, se torna possível experimentar com diferentes ordenações de *plug-ins*, pois é sabido que a ordem do encadeamento dos efeitos interfere no som resultante - conceito intitulado *cadeia de sinal*, e que será explicado posteriormente neste trabalho -, aumentando mais ainda as possibilidades para personalização sonora.

Plug-ins de áudio geralmente são programados com o auxílio de uma área de pesquisa da computação chamada Processamento de Sinais Digitais - *Digital Signal Processing* (DSP). Suas raízes são mais profundas, porém o verdadeiro descobrimento do seu potencial se deu no início nos anos 60, junto com o interesse em viagens espaciais (que trouxeram a necessidade de melhorar as imagens fotografadas por satélites), e os primeiros computadores digitais (que trouxeram a necessidade de mais eficiência em processamento). Em pouco tempo o DSP se espalhou por diversos outros campos, marcando grandes conquistas para a ciência, auxiliando em campos como medicina, telecomunicação e militar (SMITH, 1997).

Porém, boa parte de sua fama é devida à sua aplicabilidade quando se trata de processamento de sinais digitais de áudio e imagem. Redução de ruídos e compressão de dados são apenas duas das técnicas mais conhecidas que são abrangidas pelo processamento de sinais digitais nessas áreas (e em muitas outras também). Mais especificamente, tratando-se de *plug-ins* de áudio, a evolução dos *softwares* processadores de efeitos permitiu que fosse possível gerar e manipular sons digitalmente, e que são considerados de alta fidelidade. De fato, existe hoje uma infinidade de *plug-ins* no mercado, com diferentes propósitos, aplicações e custos, sendo usados por grandes compositores e artistas contemporâneos.

Este Trabalho de Conclusão de Curso propõe a criação de alguns *plug-ins* processadores de efeitos sonoros, promovendo a versatilidade desse tipo de ferramenta, permitindo que um músico possa personalizar seu som com facilidade e praticidade. Estes *plug-ins* serão *open-source* e utilizarão técnicas de processamento de sinais digitais. O usuário poderá controlar estes efeitos de maneira visual, através de uma interface gráfica.

A estrutura deste documento se dá na seguinte forma: no capítulo 2 está fundamentada a teoria por trás dos efeitos sonoros, *plug-ins* de áudio e processamento de sinais, bem como um pouco da história e evolução de cada um. No capítulo 3 são apresentadas as ferramentas usadas na implementação deste Trabalho de Conclusão de Curso. O processo da gravação sonora da guitarra é visto no capítulo 4, e a programação e desenvolvimento dos *plug-ins* se dá no capítulo 5. Por fim, no capítulo 6 é feita uma revisão dos resultados obtidos, bem como algumas considerações finais.

1.1 Problema

Com a evolução da tecnologia de processamento de sinais digitais, a produção de música passou a ser possível sem sair de casa. Com pouco investimento inicial, é possível gravar e produzir músicas com facilidade e qualidade profissional, motivo pelo qual se está investindo cada vez mais em estúdios de gravação caseiros (*home studios*). Em 1997, Paul Théberge - compositor e professor de música canadense - estabeleceu uma conexão entre produção musical e consumismo de tecnologia em seu livro vencedor de prêmios, *Any Sound You Can Imagine* (THÉBERGE, 1997).

Juntamente com o aumento de números de *home studios*, cresce o uso de DAWs. Isto é devido ao seu grande custo-benefício: ao invés de utilizar amplificadores e microfones para gerar e captar sons reais (aparelhagens caras, que ocupam espaço e geram barulho, qualidades indesejáveis quando se trabalha em uma casa ou apartamento), é necessário adquirir apenas um computador moderno e uma interface de áudio - equipamentos pequenos e baratos, especialmente quando comparados com antigos consoles analógicos de mixagem. Em verdade, é argumentado que o surgimento das DAWs remodelou a indústria dos estúdios de gravação, se estabelecendo como a mais predominante ferramenta no contexto de produção musical atual (KIRBY, 2015).

De fato, a tecnologia para a gravação de áudio se tornou muito mais acessível nas últimas décadas. Porém, ainda é comum se deparar com alguns obstáculos para adquirir certos equipamentos eletrônicos quando se está montando um *home studio* no Brasil. Já foi visto anteriormente que é cada vez mais comum optar pela opção de gravar instrumentos elétricos com DI. Para isto, é necessária uma interface de áudio - geralmente um produto importado, cujo valor oscila naturalmente devido ao valor do câmbio e volume de importação; somado com isso, a legislação aduaneira é complexa e severa em suas taxas tributárias, acarretando num preço final elevado para o consumidor (SOUZA; LIMA; SOUZA, 2015).

É comum buscar auxílio em *plug-ins* para incrementar o som de seus instrumentos gravados (seja em um estúdio caseiro ou profissional). Essa necessidade geralmente é maior quando se usa DI, pois o som recebido por si só pode frustrar expectativas. Logo, ainda que seja sobreposta a dificuldade em se adquirir uma boa interface de áudio, tem-se um obstáculo adicional quando se grava com uma DAW: obter *plug-ins* de áudio de boa qualidade. Existem *plug-ins* processadores de um único efeito sonoro que podem chegar a custar centenas de dólares, o que acaba por derrotar seu propósito, que é fornecer praticidade de maneira acessível.

Por este motivo, *plug-ins* gratuitos são muito procurados pela comunidade de artistas e compositores. *Plug-ins open-source* são especiais neste quesito, pois permitem que programadores de todo o mundo possam analisar e ajudar no desenvolvimento do código, em prol da criatividade e eficiência no processamento de sinais.

Este trabalho pretende promover a fácil personalização sonora para guitarra usando DI

e uma DAW, por meio da programação de *plug-ins* processadores de efeitos sonoros. Estes *plug-ins* serão *open-source* e utilizarão técnicas de processamento de sinais digitais, podendo ser controlados de maneira visual, através de de uma interface gráfica.

1.2 Objetivos

1.2.1 Objetivos gerais

Permitir que o usuário possa colorir um sinal digital de áudio, de maneira prática e personalizada, através de efeitos sonoros programáveis e encadeáveis em forma de *plug-ins* de áudio. Este sinal será referente ao som de uma guitarra, e deverá ser gravado utilizando uma técnica chamada DI.

1.2.2 Objetivos específicos

a) Programar diferentes *plug-ins* de efeitos sonoros;

b) Gravar o som de uma guitarra usando DI, gerando um sinal de áudio digital;

c) Colorir este sinal de áudio usando os *plug-ins* programados;

d) Permitir que o usuário veja e altere os parâmetros dos efeitos em tempo real.

2 Fundamentação teórica

Neste capítulo será exposta a teoria necessária para o entendimento deste Trabalho de Conclusão de Curso. Primeiramente será vista a evolução dos efeitos sonoros, na seção 2.1. Então serão explicadas duas importantes propriedades dos efeitos sonoros que são relevantes: parâmetros (na seção 2.1.2), e sua categoria (na seção 2.1.3). Na seção 2.2 serão abordados conhecimentos sobre o funcionamento de um *plug-in* de áudio, bem como os fundamentos de processamento de sinais digitais por trás destes, na seção 2.3.

2.1 Efeitos sonoros

É fato conhecido que a raça humana sempre busca novos meios para se expressar. Hoje em dia, é normal associar a ideia de efeitos sonoros ao reino digital dos computadores, mas efeitos sonoros são utilizados há milhares de anos. Em peças teatrais na China e Índia, maquinários eram construídos para atingir maior realismo cênico durante as apresentações; na Grécia, (ASSOCIATION, 2017). A definição de efeito sonoro, portanto é: um som artificialmente criado para enfatizar uma obra artística.

A ideia foi posta em prática muito antes de sequer haver uma noção como o som funciona: a primeira teoria sobre propagação de ondas de som seria proposta cerca de três mil anos depois, por Vitrúvio (KILGOUR, 1963), e serviria de base para o conhecimento moderno sobre o *design* acústicos de teatros - Vitrúvio foi um arquiteto italiano notavelmente conhecido por estabelecer um padrão arquitetônico para teatros acusticamente eficientes, e que são usados até hoje. Desde então, o conceito de acústica e propagação de som foi aprimorado por filósofos e matemáticos como Boécio, Galileu e Marin Mersenne (BERG, 2019).

2.1.1 História

Antigamente, era comum o uso de câmaras de eco (tais como grandes catedrais) para conferir uma sensação suplementar de espaço ao som, devido às reflexões das ondas sonoras causadas pela própria arquitetura do ambiente - efeito conhecido como reverberação. O uso de câmaras de eco foi intensivamente explorado por diversos estúdios de gravação em meados do século 20, que ficaram popularmente conhecidos pela característica reverberação de suas câmaras de eco. Um exemplo pode ser visto na Figura 1, com uma câmara de eco (na Inglaterra) que ficou popular entre gravações de vocais de grandes bandas da década de 60, como os *Beatles* e *Pink Floyd*.

Figura 1 – Câmara de eco do *Studio Two* do *Abbey Road*.

Fonte: reverb.com (FUMO, 2019).

Como é possível ver, uma caixa de som e um microfone são posicionados dentro de uma câmara de eco. A gravação original é tocada e re-gravada com o acréscimo da reverberação natural do ambiente. Este é possivelmente o efeito sonoro mais antigo (WEIR, 2012), devido à sua simplicidade: trata-se de um método completamente analógico, baseado em questões puramente mecânicas. Além disso, cada ambiente possui uma característica sonora distinta, o que torna impossível replicar o som de uma determinada câmara de eco.

Na mesma época, aparelhagens elétricas capazes de reproduzir efeitos sonoros começaram a ser desenvolvidas. Mais notavelmente, surgiram os primeiros pedais de efeito para guitarra. Tratam-se de pequenos dispositivos que carregam o sinal proveniente da guitarra e o incrementam com determinados tipos de efeito. Em 1946 foi lançado o primeiro pedal de guitarra da história, o *Tremolo Control* da DeArmond (vide Figura 2). Para efeito de comparação, o primeiro transistor foi produzido somente um ano depois, em 1947.

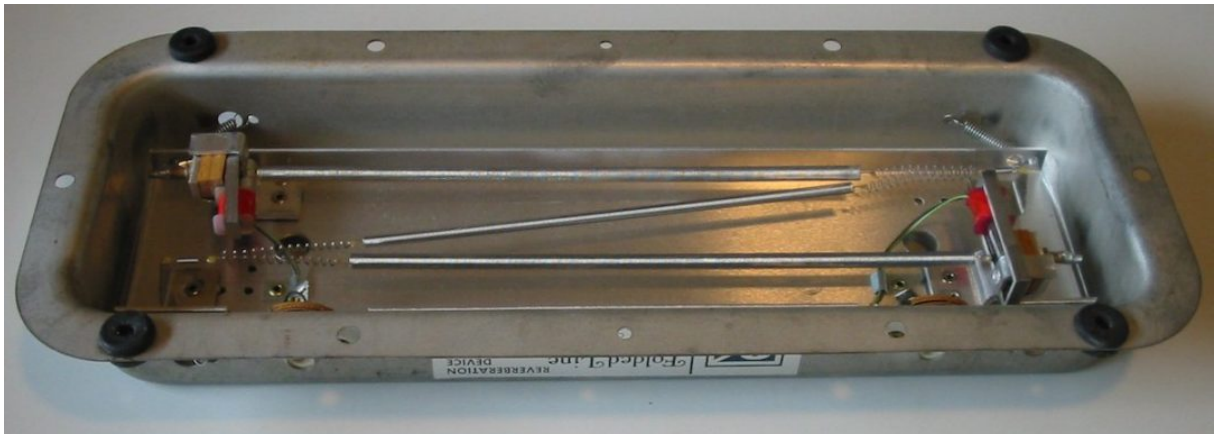
Figura 2 – Pedal de guitarra *DeArmond Tremolo Control*.

Fonte: reverb.com (REVERB, 2019).

O DeArmond Tremolo Control utiliza um mecanismo eletromecânico para modular o volume do som: o sinal da guitarra passa por um compartimento cheio de fluido eletrolítico (ou seja, condutor de corrente elétrica). Esse líquido é agitado por um motor, fazendo com que mais ou menos sinal consiga passar, alterando o volume do sinal de saída - efeito conhecido como *tremolo*. Tanto a intensidade quanto a velocidade do efeito podem ser controladas por botões giratórios na parte dianteira do aparelho - mais detalhes sobre os parâmetros de efeitos sonoros serão explicados na seção 2.1.2.

Menos de uma década depois, em 1954 (SOCIETY, 2014), surgiu o primeiro dispositivo capaz de reproduzir reverberação artificialmente, chamado *spring reverb* (em português, reverberação de mola). Este dispositivo pode ser visto na Figura 3. Seu funcionamento também é eletromecânico: o sinal do som é transmitido eletricamente por fios e molas (daí seu nome) dentro de um tanque metálico fechado - altamente reflexivo, e portanto, reverberante. Ao sair pela outra ponta, o sinal já adquiriu reverberação do ambiente.

Figura 3 – Mecanismo de um *Spring Reverb*.



Fonte: producelikeapro.com (MCALLISTER, 2019).

Este pequeno e simples aparelho popularizou o uso de *reverb* na década de 60, pois logo começou a ser implementado dentro de amplificadores de guitarra, juntamente com o efeito de *tremolo*. Isso ocasionou grandes avanços em gravações, que agora não dependiam mais de estúdios que possuíam câmaras de eco para se obter reverberação.

Nesta época, aproveitando o surgimento dos transistores, o conceito de processamento de sinais digitais começou a ser explorado para produzir novos efeitos sonoros. Isso, juntamente com o aumento de interesse em pedais de efeito para guitarra, levou à criação de clássicos pedais, imortalizados por artistas como Jimi Hendrix. Um exemplo disto é o *Dallas Arbiter Fuzz Face* (vide Figura 4), produzido artesanalmente em 1966 na Inglaterra. Em 1967, Hendrix lançou seu primeiro álbum, *Are You Experienced*, fazendo uso deste pedal (DREGNI, 2012), e consequentemente, popularizando seu uso.

Figura 4 – Pedal de distorção *Dallas Arbiter Fuzz Face*.

Fonte: vintageguitar.com (DREGNI, 2012).

Avançando um pouco, em meados de 1970, gravações digitais com fita magnética começaram a se tornar o padrão em estúdios profissionais. Juntamente com isso, o uso de novas técnicas de DSP possibilitou uma maior variedade de efeitos de modulação, como *phasers* e *pitch-shifters* (harmonizadores). Além disso, os primeiros pedais de *delay*, até então equipamentos bastante grandes que somente gravavam em fita, passaram a ser comercializados utilizando os novos padrões de entrada e saída de áudio da indústria, ou seja, plugues do tipo P10 (como o visto na Figura 2).

Porém, foi na década de 80 que a revolução dos efeitos sonoros digitais teve sua maior propulsão. Além do advento do CD e dos primeiros computadores pessoais, surgiram também os primeiros consoles digitais de mixagem e dispositivos processadores de efeitos, invenções que aperfeiçoaram consideravelmente a qualidade das gravações, devido à baixa interferência de sinais e imunidade a ruídos residuais, comuns nos seus equipamentos analógicos correspondentes (BRITANNICA, 2011). As empresas fabricantes de dispositivos de áudio desta época começaram a investir grandemente nas unidades processadoras de efeitos, chamadas de *rack-mount units*, que podiam ser agrupadas em *racks* (vide Figura 5) e utilizadas tanto em estúdios quanto em *shows* ao vivo.

Figura 5 – *Rack* de estúdio com diversos compressores e equalizadores, entre outros.

Fonte: pro-tools-expert.com (COOPER, 2019).

Surgiram também os primeiros processadores multi-efeitos, que possibilitavam o uso de

diversos efeitos simultaneamente, assim como alterar e salvar suas preferências para fácil uso posterior. Na Figura 6 é possível ver um processador multi-efeitos que obteve bastante sucesso no final dos anos 80, produzido pela Eventide, empresa pioneira no ramo de processamento de sinais digitais para áudio.

Figura 6 – Processador multi-efeitos *Eventide H3000S Ultra-Harmonizer*.

Fonte: worthpoint.com (WORTHPOINT, 2016).

Este dispositivo expandiu consideravelmente a gama de opções para músicos e produtores, e foi muito bem recebido, principalmente pela sua característica sonora, da mais alta qualidade na época. Esta é uma versão lançada pouco tempo depois do original H3000 de 1986, contendo 48 *presets* feitos por Steve Vai, aclamado guitarrista *solo*. Prova da legitimidade sonora deste aparelho é que ele permanece no estúdio do próprio Vai até hoje e é usado em gravações (GUITAR, 2012) - apesar de possuir apenas duas entradas e saídas de áudio.

A partir deste ponto, a tendência deste tipo de tecnologia foi expandir seu poder de processamento, bem como o número de entradas e saídas para conexão com outros dispositivos. Desde então, a Eventide lançou diversos modelos novos desta linha de processadores multi-efeitos, sendo o mais recente o H9000, lançada em Fevereiro deste ano (2019), como pode ser visto na Figura 7.

Figura 7 – Processador multi-efeitos *Eventide H9000*.

Fonte: sweetwater.com (SWEETWATER, 2017).

Pode-se notar que de maneira geral, o design é bem semelhante ao original; no entanto, seu poder de processamento é dezenas de vezes maior. Hoje este dispositivo é considerado estado da arte, podendo servir como o único processador de efeitos de um estúdio de gravação profissional. Possui dezenas de entradas e é capaz de processar até 96 canais de áudio

simultaneamente. Além disso, permite fácil expansão, e acompanha um *software*, através do qual é possível controlá-lo via rede.

Por último, na Figura 8 é possível ver um pedal multi-efeitos para guitarra da mesma empresa. O H9 MAX possui mais de 50 algoritmos distintos de processamento e é considerado o estado da arte da sua categoria. Embora apenas um *preset* possa ser tocado por vez, possui alguns *presets* com mais de um efeito. Foi concebido com a guitarra em mente, mas devido à sua alta fidelidade sonora, pode ser utilizado com quaisquer outros instrumentos, e até mesmo em produções musical, como um dispositivo processador de efeitos.

Figura 8 – Pedal de guitarra harmonizador *Eventide H9 MAX*.

Fonte: eventideaudio.com (AUDIO, 2015).

2.1.2 Parâmetros

Existe uma importante propriedade inerente a todo efeito sonoro: seus parâmetros. Devido a todo o processamento por trás de efeitos sonoros digitais se resumir - em seu núcleo mais interno - a equações matemáticas, é possível parametrizar certos valores que são usados nessas equações, de forma que se possa controlar a sonoridade dos efeitos através de componentes visuais.

Estes parâmetros podem ser abstraídos como propriedades sônicas dos efeitos, e sempre possuem um alcance (isto é, um valor máximo e um valor mínimo), para que deste modo, o usuário possa modificar o efeito ao seu gosto. Cada tipo de efeito possui seus próprios parâmetros, e portanto, produzem sons distintos. A Figura 9 mostra um pedal de guitarra de *delay*, ou seja, que produz repetições rítmicas do seu sinal de entrada.

Figura 9 – Pedal de *delay* para guitarra *TC Electronic Flashback II*.

Fonte: Elaborado pelo autor.

É possível ver neste pedal, que ele possui alguns valores parametrizáveis. Primeiramente, abaixo, se tem um botão de pisar, que serve para ativar ou desativar o aparelho, cuja indicação que está ligado se encontra na luz vermelha logo acima. Este parâmetro (*ligado/desligado*) geralmente é comum entre todos os efeitos, para que o usuário possa ver e controlar quais efeitos estão ativos com facilidade.

Então, na parte de cima, se tem 4 botões giratórios para controle dos parâmetros - e uma chave seletora. O primeiro botão controla o tempo entre repetições do sinal (*delay*); o segundo, o número de repetições (*feedback*); o terceiro (*level*) serve para balancear a quantidade de efeito com a quantidade de sinal original; e por último, um botão para a seleção do tipo de *delay* a ser usado - este pedal conta com 7 algoritmos próprios (inclusive um multi-efeito), e é capaz de guardar em sua memória até 3 configurações personalizadas. Por último, a chave seletora entre os botões *level* e *feedback* define a duração da nota com *delay*.

2.1.3 Tipos de efeitos

Existem alguns tipos básicos de efeitos sonoros, categorizados de acordo com o nível de manipulação que efetuam no sinal. Nesta seção serão explicados quais são esses tipos, bem como o que fazem.

2.1.3.1 Filtros

Tem por objetivo atenuar ou enfatizar frequências específicas, podendo servir como equalizadores, ou filtros de sons indesejados. Normalmente, são usados no início de cadeias de sinal, pois podem modelar um sinal ao agrado.

Por exemplo, um filtro passa-alto (*high-pass*) elimina completamente as frequências abaixo de um ponto de corte especificado, permitindo passar apenas as frequências acima deste ponto. Em contrapartida, um equalizador de banda pode levemente ressaltar certas frequências enquanto suaviza outras, moldando o som conforme a vontade do usuário. É possível ver exemplos de equalizadores na seção 2.2.1, mais especificamente com as Figuras 14 e 15.

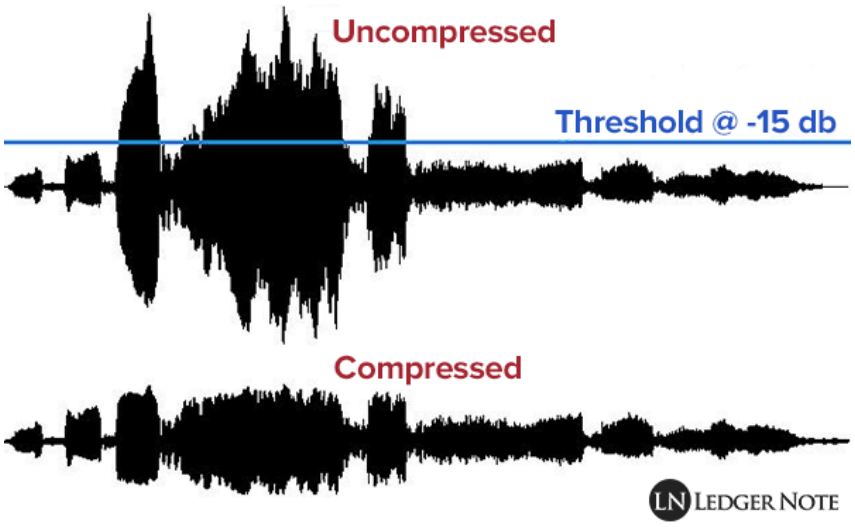
2.1.3.2 Efeitos dinâmicos

São chamados assim pois mudam a amplitude - ou seja, o volume - de um sinal digital de áudio, caso certas condições sejam atendidas. Efetivamente, mudam o alcance dinâmico do sinal, ou seja, a distância entre o ponto com menor amplitude, e o ponto com maior amplitude. Suas aplicações podem ser variadas, mas, em geral, efeitos deste tipo também são utilizados no início de cadeias de sinal, pois geram um sinal com menor ruído e volume mais constante, facilitando o uso de efeitos posteriores.

Por exemplo: portões de ruído (*noise gates*) bloqueiam sinais com volume abaixo do que o definido, efetivamente removendo o ruído nas partes silenciosas do sinal. Compressores, por outro lado, acentuam o som de baixo volume e atenuam o som de alto volume, gerando um sinal com menor variação dinâmica, ou seja, com volume mais constante.

Um exemplo de como um compressor (que apenas atenua sons de alto volume) age sobre um sinal de áudio pode ser visto no gráfico da Figura 10, com uma comparação entre um sinal antes da compressão (acima), e depois (abaixo). Um valor base (*threshold*) é definido, para que se possa comprimir todo sinal acima deste nível de volume (no caso, -15 db). Pode-se notar que o som resultante possui uma forma de onda mais constante em amplitude do que o som de entrada, ou seja, possui menor variação dinâmica.

Figura 10 – Gráfico de compressão de um sinal de áudio digital.

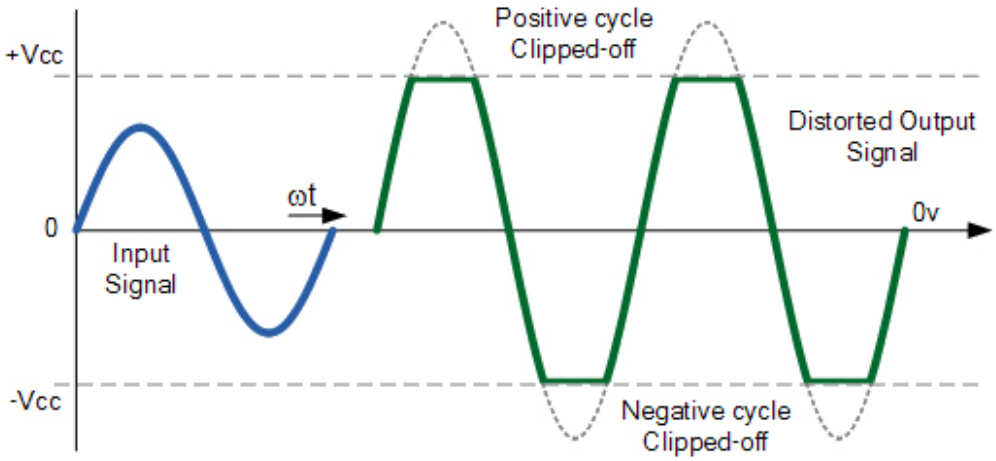


Fonte: ledgernote.com (HOBBS, 2019).

2.1.3.3 Distorção

Efeitos deste tipo tecnicamente podem ser considerados efeitos dinâmicos, pois também alteram o volume de um sinal. No entanto, essa alteração é feita além de um ponto no qual ocorre a distorção da forma de onda do sinal. Este fenômeno - conhecido como *clipping* - ocorre quando o sinal de saída possui uma amplitude maior do que a disponível, e acaba limitando a sua forma de onda - como pode ser visto no gráfico da Figura 11.

Figura 11 – Gráfico de distorção de uma forma de onda digital.



Fonte: electronics-tutorials.ws (TUTORIALS, 2008).

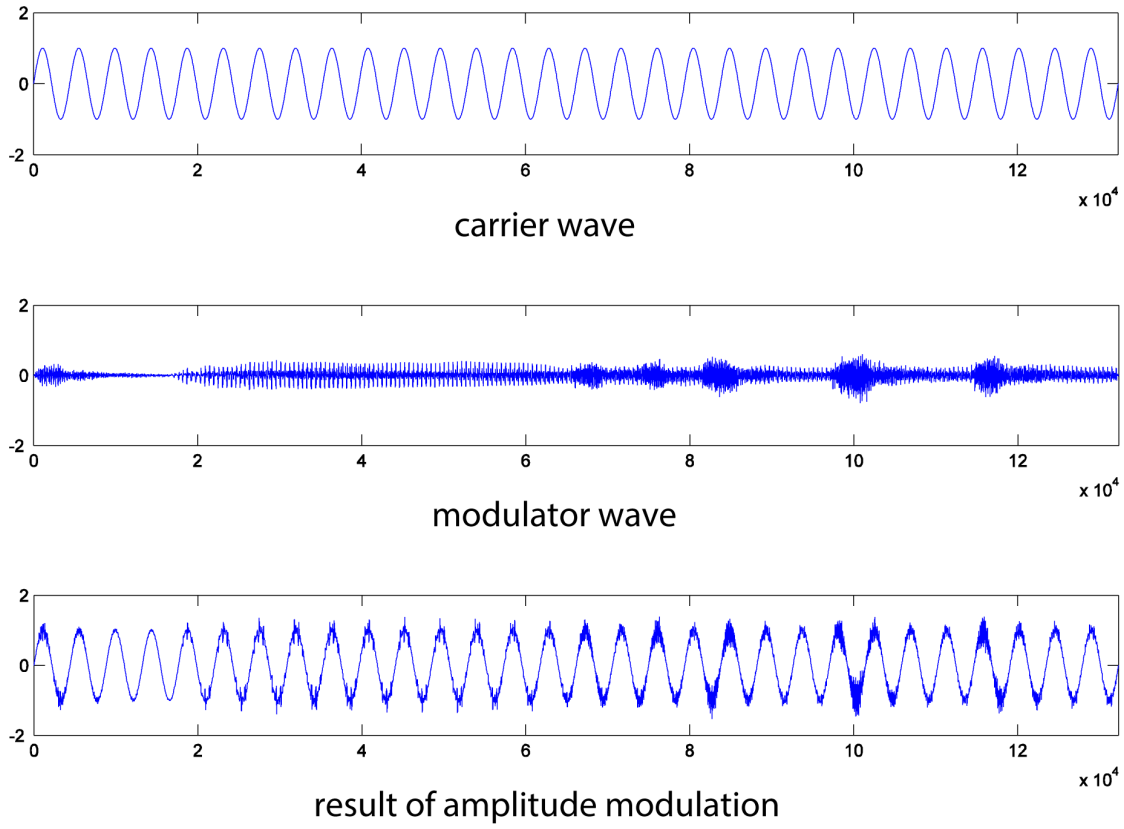
A maneira como a onda é distorcida é o que diferencia os sub-tipos de distorção. No caso da Figura 11, se vê o clássico tipo *distortion* (a onda fica reta e simétrica - *hard clipping*). Existem também os tipos *fuzz* (vide Figura 4), *overdrive* e *booster*, entre outros.

2.1.3.4 Efeitos moduladores

Modular um sinal digital de áudio significa modificá-lo - geralmente acrescentando a ele uma versão levemente atrasada de si mesmo. Com este simples conceito é possível alcançar uma grande variedade de efeitos sonoros, incluindo *pitch-shifters* (harmonizadores - como vistos nas Figuras 6 e 8), *tremolos* (vide Figura 2), e até mesmo *delays*.

No gráfico da Figura 12 é possível ver uma simples aplicação de modulação em um sinal. Acima, se tem o sinal original, em sua forma de onda senoide. Então, um segundo sinal (ao meio) é acrescentado a este, resultando em uma forma de onda que representa a soma entre essas duas ondas. Esta onda resultante pode ser vista na parte de baixo do gráfico.

Figura 12 – Gráfico de modulação de uma forma de onda digital.



Fonte: digitalsoundandmusic.com (BURG; ROMNEY; SCHWARTZ, 2019).

2.1.3.5 Efeitos baseados em tempo

Tecnicamente podem ser enquadrados como efeitos moduladores, porém sua distinção surge do seu propósito: criar espaçamento ao som, baseado na repetição rítmica do sinal de entrada. Efeitos desta categoria incluem reverberação (e.g. como na Figura 1 - feita de maneira analógica) e *delay* (eco) - como visto na Figura 9.

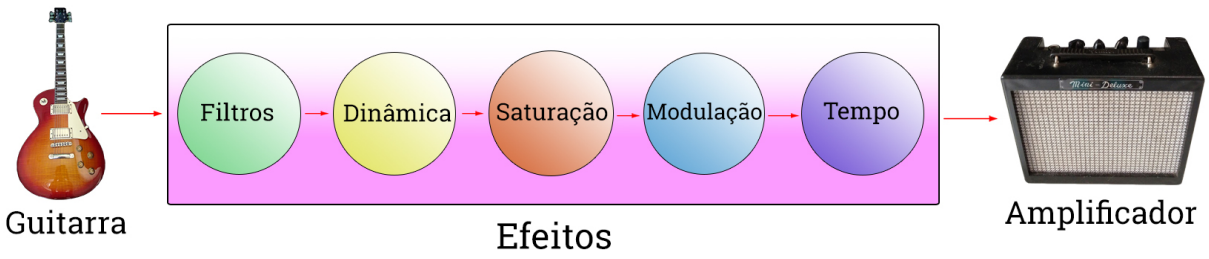
2.1.4 Cadeia de sinal

É inegável a presença de um caráter subjetivo no som que ouvimos. Cada pessoa tem para si uma definição do que soa bem, e o que soa mal: não existe uma fórmula matemática que descreva um som universalmente agradável, apesar da música como um todo ser regida por leis matemáticas. Graças a essa dubiedade, músicos foram capazes de alcançar sons até então inexplorados, realizando simples experimentações com seus equipamentos.

O conceito de cadeia de sinal (ou *signal chain*) é muito simples: se trata do caminho que o sinal de um som percorre, de maneira sequencial, desde o instrumento que o gera, passando por múltiplos efeitos, até finalmente ser reproduzido. Pode se aplicar em diferentes situações (como pedais de efeitos para guitarra, ou *plug-ins* usados em uma DAW). A lógica é que a ordem dos componentes usados - ou seja, seu encadeamento - altera o som resultante, podendo passar de prazeroso a detestável com apenas um clique.

Por definição, não é possível determinar uma cadeia de sinal como "certa"ou "errada": apenas como "agradável"ou "desagradável", devido à subjetividade musical inerente de cada pessoa. É por isso que é recomendado experimentar com diferentes encadeamentos de efeitos, até achar um que seja mais preferível do que os demais: o juiz final deve sempre ser o ouvido. Dito isso, existe um padrão que é comumente aceito pela comunidade de guitarristas, como se pode ver na Figura 13.

Figura 13 – Popular cadeia de sinal de efeitos sonoros.



Fonte: Elaborado pelo autor.

Ao sair da guitarra, o sinal passa primeiramente por filtros (como um *noise gate*), para eliminar ruídos e outros sons indesejados. Depois, passa por efeitos de dinâmica para se obter um sinal com menor alcance dinâmico, ou seja, com volume mais constante. Então é adicionada a saturação (ou distorção), alterando seu timbre. Após isso são adicionados efeitos moduladores, que podem alterar ou complementar o sinal de diversas maneiras. Por último, são usados efeitos baseados em tempo para dar uma sensação de profundidade e preenchimento do som, com reverberações e ecos. É importante salientar que não existe um único jeito correto de construir uma cadeia de sinal, e é sempre recomendado experimentar com diferentes combinações até encontrar uma que seja mais agradável que as demais - a cadeia de sinal apresentada na Figura 13 pode servir como um guia inicial.

2.2 *Plug-ins* de áudio

Na sua definição mais crua, um *plug-in* é um aplicativo que pode ser incorporado em um programa maior, para adicionar funcionalidades. Mais especificamente sobre áudio, trata-se de uma biblioteca de rotinas processadoras de sinais, que pode ser carregada e usada por uma DAW. A utilização de *plug-ins* de áudio possibilita uma fácil expansão de ferramentas, e atualmente é um assunto de grande importância no ramo de produção musical, sendo utilizados por numerosos artistas ao redor do mundo e até produtores de filmes, devido à sua grande praticidade.

A quantidade de efeitos sonoros que um *plug-in* é capaz de processar simultaneamente varia de acordo com o seu propósito. No entanto, independentemente disso, todo *plug-in* fornece uma interface gráfica para o usuário controlar seus efeitos com facilidade. Este controle é feito de maneira análoga a como era feito nos pedais para guitarra - através de botões que controlam os parâmetros dos efeitos.

2.2.1 Visão geral

Paralelamente à evolução dos computadores pessoais na década de 80, foram surgindo as primeiras DAWs. Em 1992, uma empresa israelense chamada Waves Audio, pioneira no ramo de processamento de sinais digitais, desenvolveu o primeiro *plug-in* de áudio da história. Trata-se de um equalizador paramétrico de 10 bandas chamado Q10 Paragraphic Equalizer. Até hoje permanece em lugar de destaque entre vários produtores de música, tendo recebido em 2010 uma indicação para o *TECnology Hall of Fame*, prêmio prestigiado de inovações musicais (AUDIO, 2011). Em 2017, em homenagem ao seu aniversário de 25 anos, foi lançada uma edição comemorativa do plug-in, contendo uma reformulação da velha interface gráfica, como pode-se ver na Figura 14.

Figura 14 – Edição comemorativa de 25 anos do Waves Q10 EQ.

Fonte: waves.com (AUDIO, 2017).

Não é incomum *plug-ins* buscarem recriar os aspectos sonoros (e por vezes, visuais) de equipamentos consagrados do passado, entre eles, pedais de guitarra e processadores de efeitos de *rack*. A Figura 15 mostra uma comparação entre o primeiro equalizador ativo da história (o Pultec EQP-1, visto acima) e a sua versão em *plug-in*, desenvolvida pela Waves Audio (abaixo).

Figura 15 – Comparação das versão reais e virtual do equalizador EQP-1A.

Fonte: musictech.net (PICKFORD, 2014) (acima) e waves.com (AUDIO, 2019) (abaixo).

Porém, nem sempre um *plug-in* possui apenas um efeito - existem também os *plug-ins all-in-one* para guitarra, ou seja, que servem como o *plug-in* utilizado dentro de uma DAW. Estes plug-ins dispõem de múltiplos efeitos e simuladores de amplificadores, podendo personalizar uma ou mais cadeia de efeitos com grande integração e facilidade. Porém, um *software* deste nível de complexidade exige extenso esforço programático, motivo pelo qual é comum ser produzido apenas um efeito por *plug-in*.

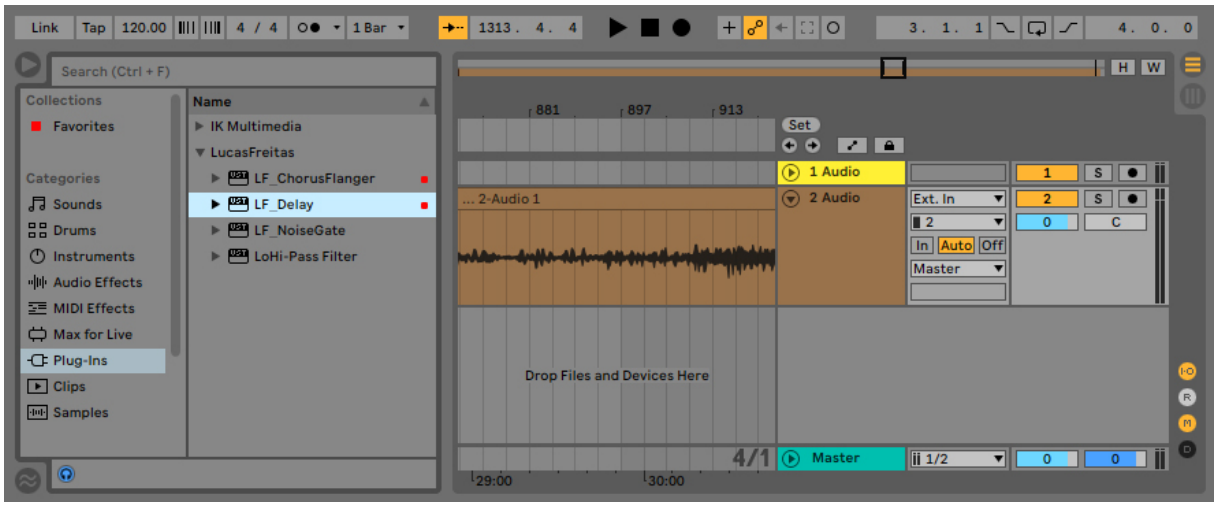
Finalizando, existem também os *plug-ins* que servem ao propósito de produção musical como um todo, não sendo limitados a guitarra. Um exemplo interessante disto é o iZOTOPE Ozone 9, um *plug-in* de masterização, ou seja, o estágio final de produção de uma música. Uma característica singular deste *plug-in* é que ele utiliza Inteligência Artificial (IA) para analisar uma música de referência, e sugere configurações semelhantes de equalização, compressão, e outros efeitos - que podem ser aplicadas na música desejada.

2.2.2 Utilização

Como mencionado previamente, um plug-in é como um aplicativo que pode ser carregado por uma aplicação hospedeira (no caso, uma DAW) para acrescentar funcionalidades. Em verdade, um plug-in se trata de um único arquivo, que é colocado em um diretório visível pela DAW. Uma vez que um plug-in é carregado, é possível - por intermédio da DAW - configurá-lo através de sua interface gráfica.

Na Figura 16 é demonstrada a seleção de *plug-ins* para uso. Basta selecionar um *plug-in* através do menu à esquerda, e arrastá-lo até a faixa de áudio desejada (à direita). Também é possível marcar *plug-ins* como "Favoritos", proporcionando um fácil acesso.

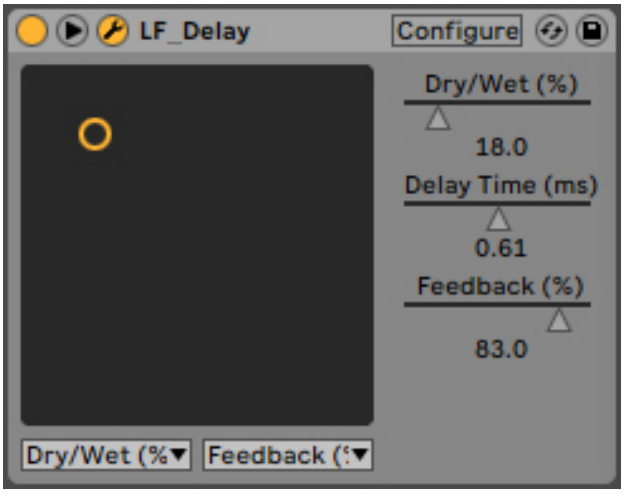
Figura 16 – Seleção de um *plug-in* em uma DAW.



Fonte: Elaborado pelo autor.

Uma vez carregado, é possível configurar o plug-in. Além da interface gráfica do próprio *plug-in*, a DAW oferece uma interface rudimentar para controle dos parâmetros, como pode ser vista na Figura 17.

Figura 17 – Visualização de um *plug-in* pela DAW.

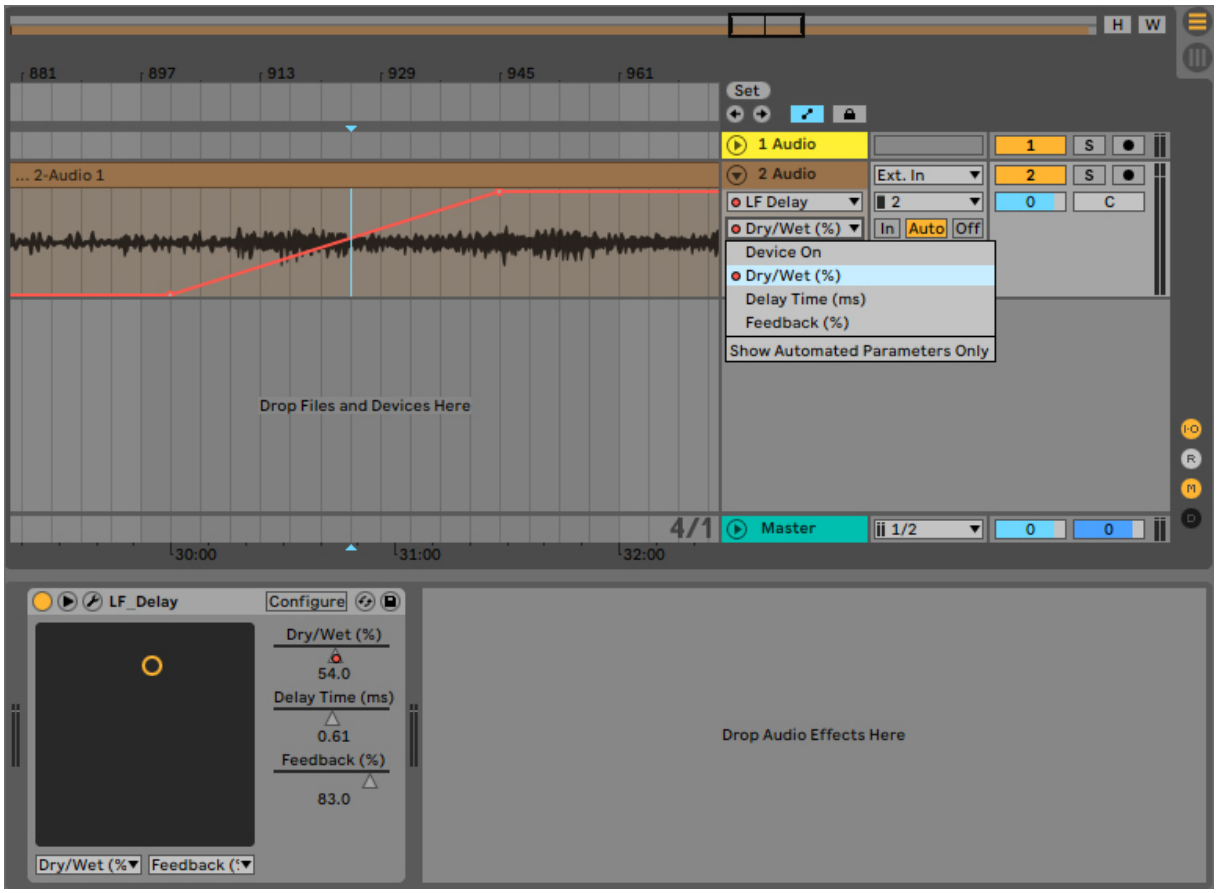


Fonte: Elaborado pelo autor.

Também é possível programar - pela DAW - a mudança de valores de parâmetros dos *plug-ins* de acordo com o tempo (a isto se dá o nome de automação). Um exemplo de automação é visto na Figura 18, com o parâmetro **Dry/Wet** do *plug-in* de *delay* desenvolvido

neste trabalho. Pode-se notar que o valor deste parâmetro é alterado com o tempo (no caso, sobe de 0% a 100%). Um pequeno círculo vermelho é usado para indicar que este parâmetro foi automatizado.

Figura 18 – Automação de parâmetros pela DAW.



Fonte: Elaborado pelo autor.

2.3 Processamento de sinais digitais

2.3.1 Visão geral

DSP é uma área de estudo que abrange vários campos. Não se trata sobre a transmissão de sinais (como por telefone ou ondas de rádio), mas sim sobre a manipulação desses sinais para melhorar a transmissão ou uso desses dados. Isto é feito através de algoritmos matemáticos, e possuem as mais diversas aplicações. Dentre os processos estudados por esta área encontram-se codificação, detecção, compressão, análise, sintetização, gravação e reprodução de sinais.

Os benefícios que o DSP trouxe para a tecnologia são muitos para se listar; porém, suas principais áreas de foco (ou pelo menos, de fama) são análogas a dois sentidos humanos: visão e audição. O processamento de sinais digitais de imagem se deu nos primórdios do DSP, com a necessidade de melhoria de imagens recebidas por satélites nos anos 60; já o processamento

de sinais digitais de áudio começou ainda antes, no começo da década de 40, para codificar transmissões de vozes durante a Segunda Guerra Mundial (PAUL, 2006).

Como a representação digital de um sinal se dá através de valores numéricos discretos, qualquer tipo de sinal digital pode ser processado da mesma maneira. Porém, sinais provenientes do mundo real geralmente são fornecidos de maneira analógica e contínua, sendo necessário realizar a conversão do sinal para o âmbito digital para que este possa ser processado. Isto é feito através do uso de conversores analógico-digital (ADC) - antigamente se encontravam em dispositivos dedicados, mas hoje em dia é comum existir um ADC embutido em outros aparelhos que executam DSP.

Na seção 2.3.2 serão estudados os pontos mais importantes da trajetória do processamento de sinais, desde sua origem até a contemporaneidade. Então, na seção 2.3.3, serão explicados alguns conceitos que são necessários para se entender como é feita a conversão de um sinal analógico em digital.

2.3.2 Contextualização

Enquanto a verdadeira revolução do DSP veio nos anos 60, com o advento dos primeiros computadores digitais, as raízes do campo de processamento de sinais se encontram no século 19, com um matemático chamado Joseph Fourier. Ele propôs um método matemático que substitui a descrição em tempo de um sinal, por sua descrição em frequência - método conhecido hoje como Transformada de Fourier (*Fourier Transform*), e é a base para processamento de sinais digitais de áudio da era moderna (FLANDRIN, 2018).

Mais de um século depois, em 1965, este conceito seria incrementado, gerando o *Fast Fourier Transform*, que permitiu que a Transformada de Fourier pudesse ser implementada com agilidade. Porém, devido às limitações computacionais desta época, seu uso só se daria cerca de quinze anos depois, com o surgimento dos primeiros *chips* de DSP. Durante a década de 60, o processamento de sinais digitais não tinha muitas aplicações ainda (comercialmente falando), e portanto, foi muito usado de acordo com o interesse do governo: principalmente no setor militar, com a leitura de sensores e sonares, além da melhoria de imagens recebidas por satélites (tendo em vista a corrida espacial entre Rússia e Estados Unidos) (SMITH, 1997).

Conforme a tecnologia foi se desenvolvendo, também foram feitos avanços na área de imagiologia médica (diagnóstico por imagem), e na leitura de sensores sísmicos. Em 1979 surgiram os primeiros *chips* processadores de sinais digitais (MUSEUM, 2007), capazes de manipular sinais de áudio. O surgimento desses *chips* possibilitou sua integração em bens de consumo eletrônicos (tais como brinquedos e celulares), já que podiam analisar e sintetizar voz.

Com a tecnologia que estava disponível a partir deste ponto, foi possível aumentar o poder de processamento de *chips* de DSP enquanto seu diminuir seu tamanho, o que acabou

expandindo mais ainda o seu uso pelo público. Foi então, na década de 80, que ocorreu a popularização do DSP, pois o desenvolvimento de suas tecnologias passou a ser motivado pelo mercado de consumo, ao invés do governo (SMITH, 1997). Vale lembrar que foi nesta época que surgiram os CDs, e a tecnologia digital estava passando por uma revolução.

Desde então, a interdisciplinaridade do DSP cresceu consideravelmente, e uma infinidade de dispositivos eletrônicos do cotidiano possuem *chips* processadores de sinais digitais dentro de si: telefones, celulares, computadores, satélites, câmeras, rádios digitais, e até carros (e.g. em sistemas de freio anti-trava ABS). Atualmente, um entendimento básico de DSP é requisitado em diversas áreas da ciência e engenharia, e possui aplicação em muitas outras.

2.3.3 Conversão analógico-digital

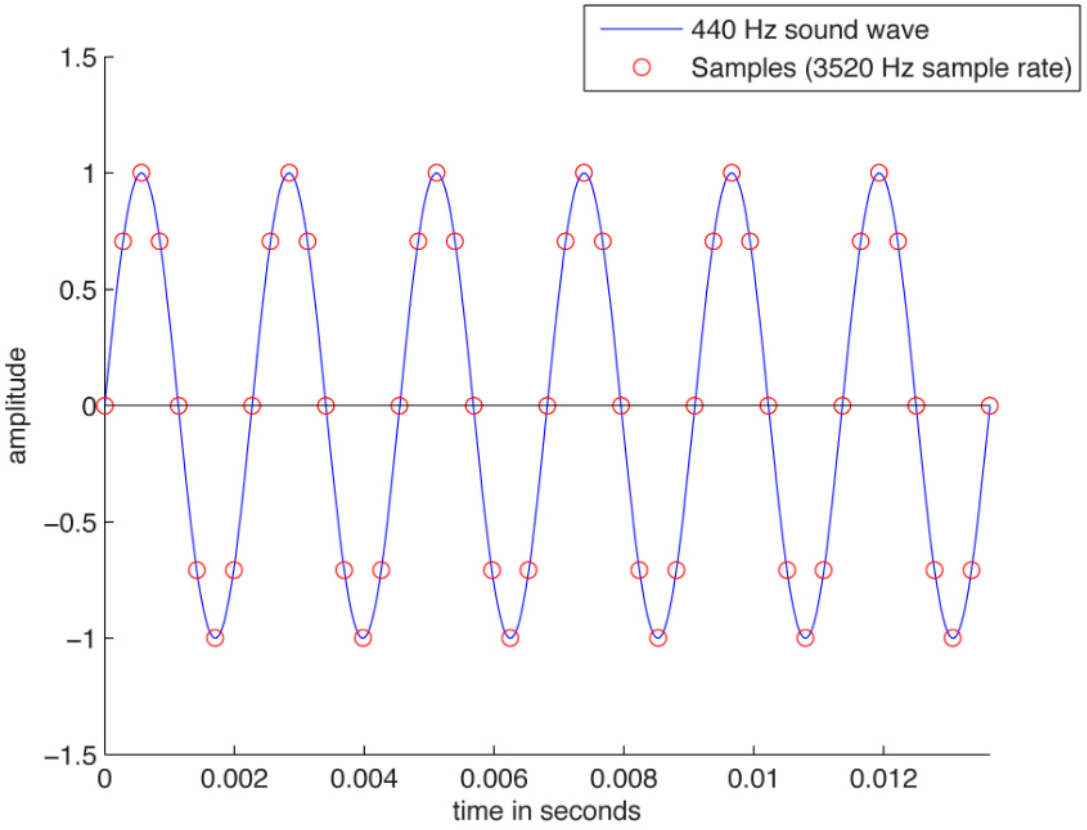
Um sinal da vida real é analógico e contínuo. Já um sinal digital é discreto - uma representação matemática (numérica). A conversão de um sinal analógico permite que sinais de origem analógica possam ser processados uniformemente, o que efetivamente faz com que seja possível analisar o mundo real de diversas maneiras diferentes (no entanto, convém notar que nem todo sinal é gerado analogicamente - nesses casos não é necessária a conversão).

O processo de conversão pode ser referido como digitalização, que por sua vez, é dividido em duas etapas: amostragem (digitalização no tempo) e quantização (digitalização na amplitude).

A amostragem consiste em mensurar um sinal em intervalos periódicos de tempo. A taxa de amostragem de uma conversão é definida em Hz (*amostras/s*), e portanto, representa quantas amostras do sinal de entrada são tomadas em um único segundo. Na prática, este valor depende da fidelidade sonora requerida para a aplicação como um todo, pois, se escolhida uma taxa de amostragem baixa demais, parte do sinal não será amostrado, e portanto, sua reconstrução não poderá ser feita de maneira fiel. De fato, existe um teorema (Teorema de Nyquist-Shannon), que propõe que, para um sinal poder ser amostrado sem perdas, a taxa de amostragem usada na conversão do sinal deve ser no mínimo duas vezes maior que a frequência mais alta contida neste sinal (OLSHAUSEN, 2000).

Na Figura 19, é possível ver o processo de amostragem de um sinal - mais especificamente, de uma onda senoide contínua de 440 Hz, mostrada em azul. Como a taxa de amostragem deste caso é de 3520 Hz, são tiradas 8 amostras de cada onda, representadas por circunferências vermelhas. Como a taxa de amostragem usada é maior que o dobro da frequência mais alta contida neste sinal (ou seja, é maior que 880 Hz), a onda pode ser reconstruída perfeitamente, após sua conversão.

Figura 19 – Processo de amostragem de um sinal.



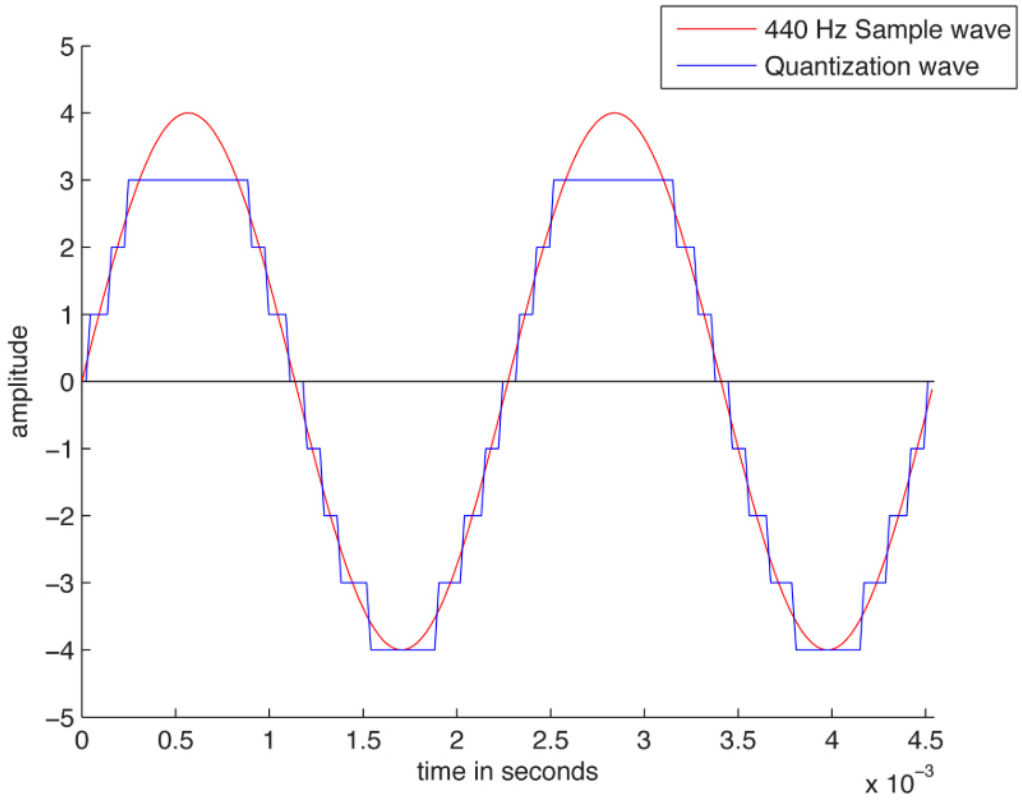
Fonte: digitalsoundandmusic.com (BURG; ROMNEY; SCHWARTZ, 2019).

Quando uma amostra de um sinal é tirada, o valor de sua amplitude (ou seja, seu volume) deve ser mensurado em números discretos. A este fenômeno se dá o nome de quantização, e constitui a segunda parte do processo de digitalização pelo qual um sinal passa durante a sua conversão.

O alcance mínimo e máximo dos valores que poderão ser utilizados para mensurar a amplitude de uma amostra, durante o processo de quantização, é calculado através da resolução de *bits* do ADC. Em outras palavras, o valor de cada amostra deve ser arredondado para caber em uma representação binária legível pelo ADC. É sabido que o maior número decimal que pode ser representado por um número binário de n *bits*, é $2^n - 1$. Assim, é fácil perceber que, se usada uma baixa resolução de *bits* para se captar um sinal, é provável que mais valores tenham que ser arredondados, causando grande perda de qualidade.

Um exemplo de uma quantização feita com baixa resolução de bits se encontra na Figura 20. É possível ver que, se utilizando 3 *bits* de resolução, pode-se ter no máximo 8 valores ($2^3 = 8$), sendo que estes estão distribuídos convencionalmente entre positivos e negativos, de modo que o maior valor positivo que pode ser usado é 3. Por este motivo, todos os valores de amplitude das amostras que passarem de 3, deverão ser arredondados pra baixo, causando uma deformação na onda resultante.

Figura 20 – Processo de quantização de um sinal com 3 *bits* de resolução.



Fonte: digitalsoundandmusic.com (BURG; ROMNEY; SCHWARTZ, 2019).

Até algumas décadas atrás, os conversores analógico-digital eram fabricados como dispositivos separados (juntamente com os conversores digital-analógico). No entanto, devido à evolução da tecnologia, atualmente é comum encontrar placas conversoras analógico-digital e digital-analógico em aparelhos como interfaces de áudio, celulares e *home theaters*.

Para finalizar o capítulo, é interessante notar que por muito tempo o padrão para gravações de músicas foi usar ADCs com 16 *bits* de resolução - a primeira gravação em 16 *bits* foi feita em 1976; gravações de 24 bits seriam feitas duas décadas depois (SOCIETY, 2014). Um ADC com resolução de 24 *bits* possui 16.7 milhões de valores inteiros à sua disposição para representar níveis de amplitude de um sinal, enquanto um ADC com resolução de 16 *bits* possui apenas 65 mil. Por este motivo, conversores com 24 *bits* de resolução estão se tornando mais comuns atualmente.

3 Ferramentas

Neste capítulo serão apresentadas as ferramentas usadas no desenvolvimento deste Trabalho de Conclusão de Curso. Para a programação dos *plug-ins*, será utilizado um framework chamado JUCE em conjunto com o Visual Studio 2017 Community. Uma DAW será usada para execução dos *plug-ins*, assim como para a gravação do som da guitarra.

3.1 JUCE

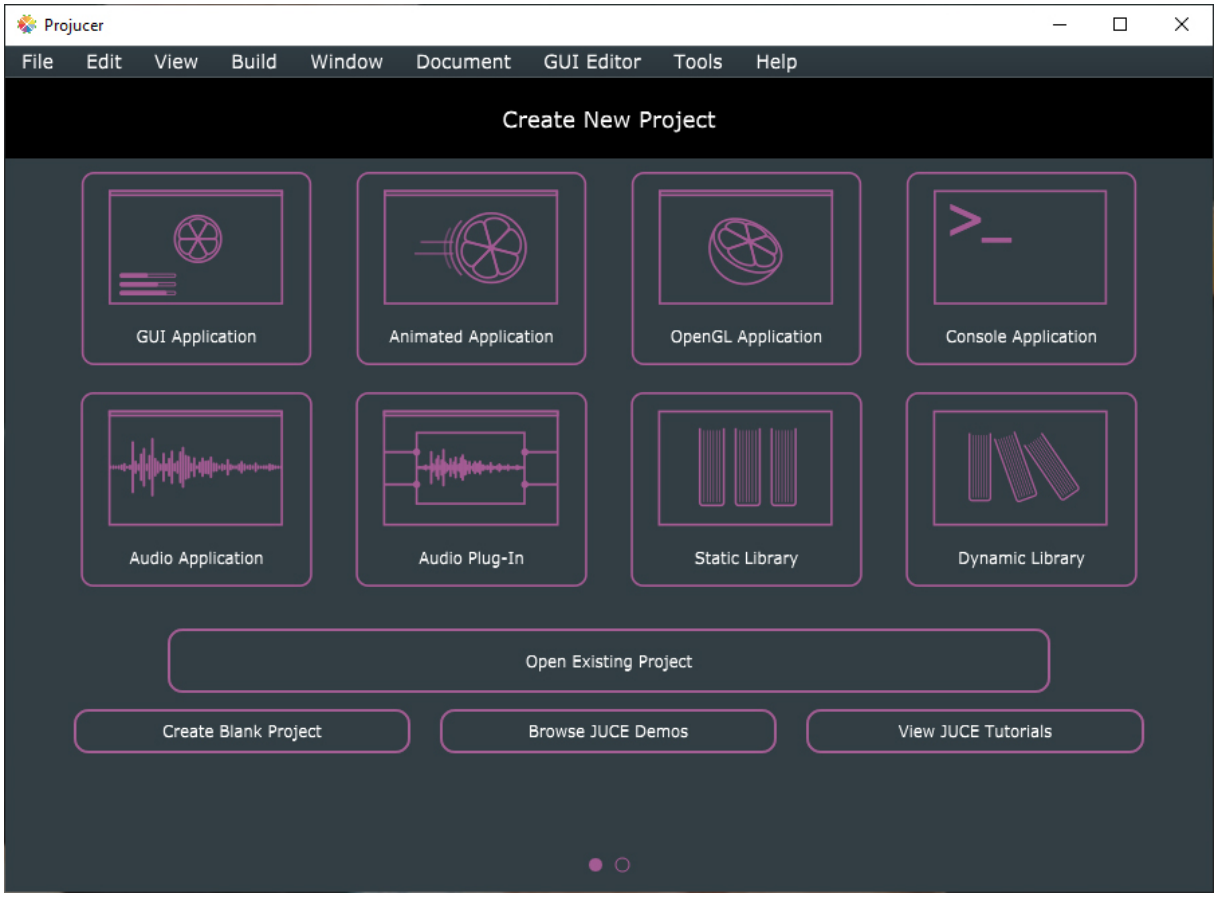
A ferramenta principal usada neste Trabalho de Conclusão de Curso se trata de um *framework* em C++ capaz de auxiliar no desenvolvimento de diversas aplicações de áudio. Seu nome vem do seu criador, Jules, que o lançou em 2004: *Jules' Utility Class Extensions*. Dez anos depois foi adquirido pela empresa britânica ROLI (BUTCHER, 2014), pioneira no ramo de tecnologia musical, certamente devido ao enorme potencial técnico e criativo que oferece. Jules hoje faz parte do time da ROLI, e permanece como editor-chefe e líder do time de desenvolvimento do JUCE, tendo obtido bastante sucesso em estabelecer um padrão no ramo de desenvolvimento de *softwares* processadores de áudio.

Além do fato que com o JUCE é possível desenvolver todo tipo de aplicação de áudio - desde bibliotecas estáticas até aplicações independentes com interface gráfica -, é possível também realizar a exportação de uma aplicação para diferentes plataformas (tais como Windows, MacOS e celulares), utilizando o mesmo código-fonte.

Para realizar corretamente a instalação do JUCE no Windows 10, sistema operacional usado neste trabalho, é necessário primeiramente instalar a IDE (*Integrated Development Environment*) do Visual Studio, pois o JUCE faz uso de SDKs (*Software Development Kits*) e pacotes presentes no mesmo (mais detalhes sobre a instalação e uso dessa ferramenta se encontram na seção 3.3). Uma vez que a IDE foi instalada, é possível usá-la para abrir o projeto do JUCE, disponível no seu repositório do GitHub (WEAREROLI, 2007) e o compilar, gerando assim um arquivo executável chamado Projucer.

O Projucer é o gerenciador de projetos do JUCE. Com ele é possível criar e configurar projetos, além de fornecer editores de interface gráfica, e códigos-fonte. Ao executá-lo, são apresentados diversos *templates* iniciais para se criar um projeto, como é possível ver na Figura 21.

Figura 21 – Tela de abertura do Projucer.

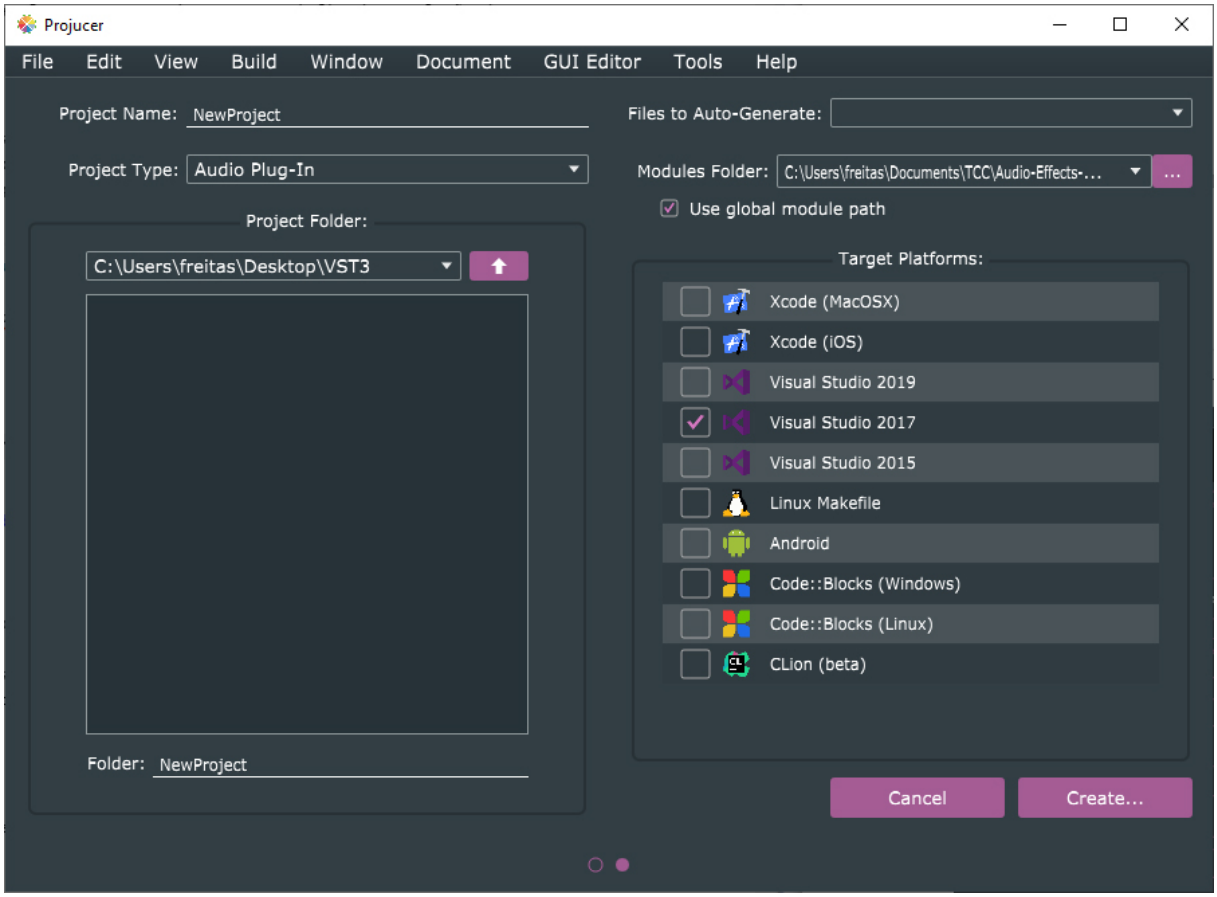


Fonte: Elaborado pelo autor.

No caso deste Trabalho de Conclusão de Curso, serão criados projetos a partir do template *Audio Plug-In*.

Na tela seguinte é possível definir o nome do projeto e selecionar as plataformas-alvo desejadas para exportação (vide Figura 22), incluindo MacOSX, Linux, e até plataformas *mobile* (iOS e Android). Porém, é necessário a instalação da ferramenta respectiva à plataforma-alvo selecionada (por exemplo, a IDE Xcode é necessária para a exportação para o ambiente iOS).

Figura 22 – Plataformas-alvo para exportação no Projucer.

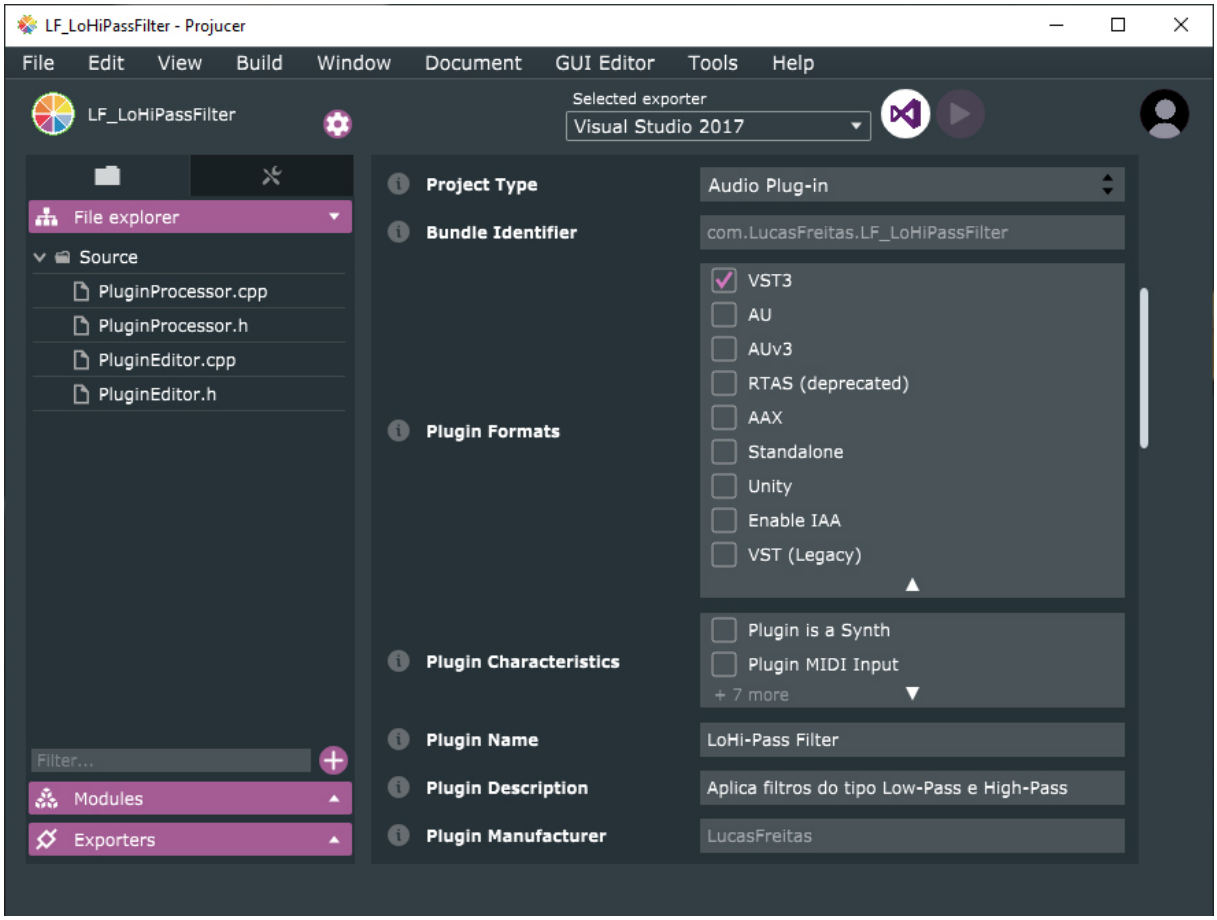


Fonte: Elaborado pelo autor.

É interessante notar que o Projucer gera um arquivo-solução do projeto para cada um dos exportadores selecionados, possibilitando a abertura do projeto pela IDE respectiva. Para os *plug-ins* desenvolvidos neste trabalho, a exportação será feita somente para Windows, e portanto, foi selecionado apenas o Visual Studio 2017.

Uma vez criado o projeto, é possível configurá-lo, alterando opções como nome, descrição e formatos de exportação do plug-in, tal como mostra a Figura 23.

Figura 23 – Configurações de um projeto no Projucer.



Fonte: Elaborado pelo autor.

Note que também é possível escolher a opção "Standalone", a qual gera um arquivo executável (no caso do Windows, de formato .exe), dispensando o uso de uma DAW. No entanto, foi escolhido o uso do formato VST3 (*Virtual Studio Technology*, versão 3), juntamente com uma DAW, pelos seguintes motivos:

a) o uso de uma DAW deixa todo o processo substancialmente mais profissional, oferecendo maior flexibilidade e numerosas opções;

b) a DAW usada neste Trabalho de Conclusão de Curso (vide seção 3.2) suporta apenas os formatos VST3 e *Audio Unit* (AU) (ABLETON, 2012);

c) além do VST3 ser o formato mais aceito pelas DAWs no mercado atualmente, é suportado tanto por ambientes Windows quanto MacOS, enquanto o AU é destinado apenas a sistemas MacOS.

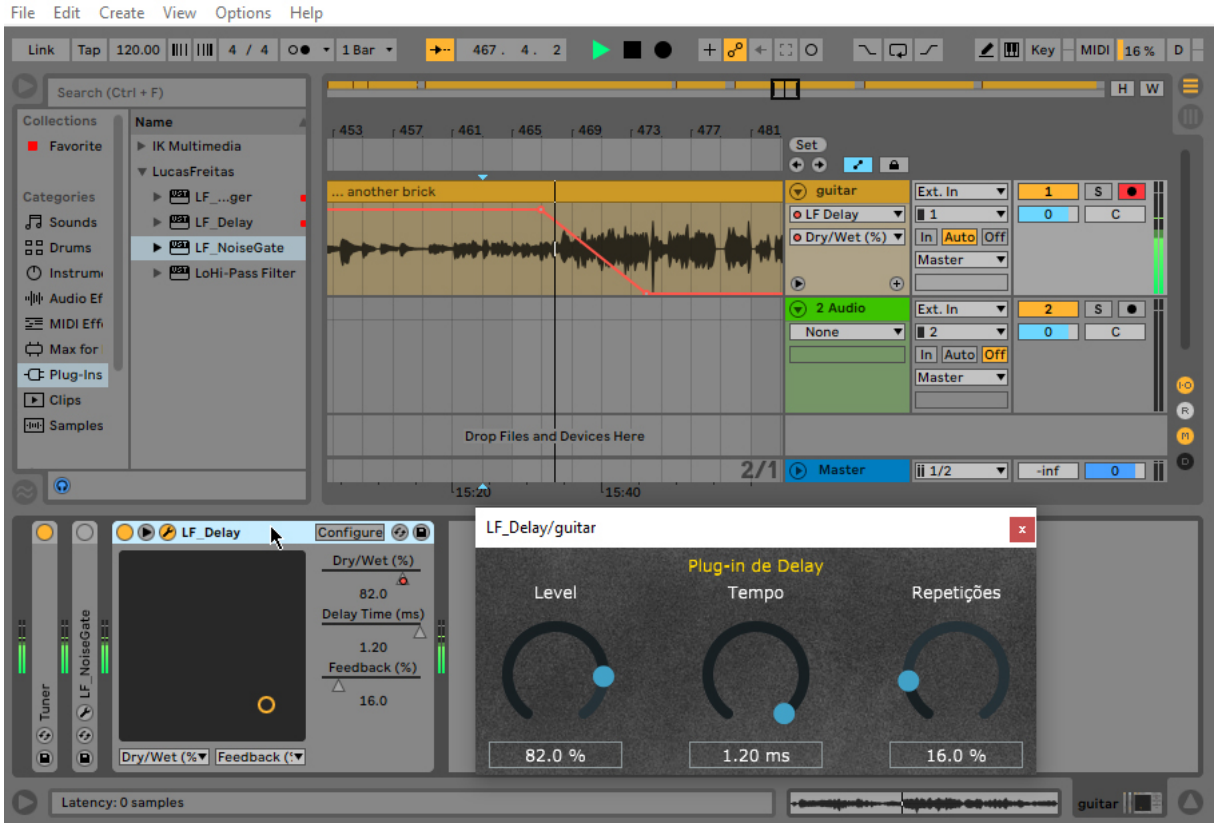
Vale a pena também notar que existe, na parte superior da tela, uma lista com os exportadores selecionados na criação do projeto, e um botão ao lado direito, que abre o projeto neste exportador; no caso, o Visual Studio 2017.

3.2 Ableton Live 10

O uso de uma *Digital Audio Workstation* para edição de áudio vem com grandes benefícios, com o principal sendo a vasta quantidade de ferramentas disponíveis. Isso, somado com a alta fidelidade do som digital, e o baixo custo do *software* em relação a consoles de áudio, são motivos pelos quais estúdios de gravação estão adotando cada vez mais o uso de DAWs, sendo usadas por praticamente todos os estúdios da atualidade (KIRBY, 2015). A escolha da DAW deve ser baseada principalmente na fluida interação do usuário com a interface. Já foi provado que pouca coloração do som ocorre pelas DAWs modernas - geralmente só quando é feita uma conversão da taxa de amostragem (WAVE, 2019). Ainda assim, essa diferença em qualidade dificilmente é perceptível ao ouvido comum, de modo que a única diferença realmente notável entre diferentes DAWs é a facilidade de se usar.

A DAW escolhida para realizar a gravação da guitarra e testar os *plug-ins* chama-se Ableton Live 10, e pode ser vista na Figura 24. Neste programa se tem acesso a todas as faixas de som gravadas, bem os *plug-ins* usados em cada uma. Além disso, com esta DAW é possível automatizar parâmetros - isto é, programar a mudança dos seu valores de acordo com o tempo, como também é possível ver nesta figura, com o parâmetro **Dry/Wet** (Level).

Figura 24 – Automação de parâmetro com o Ableton Live 10.



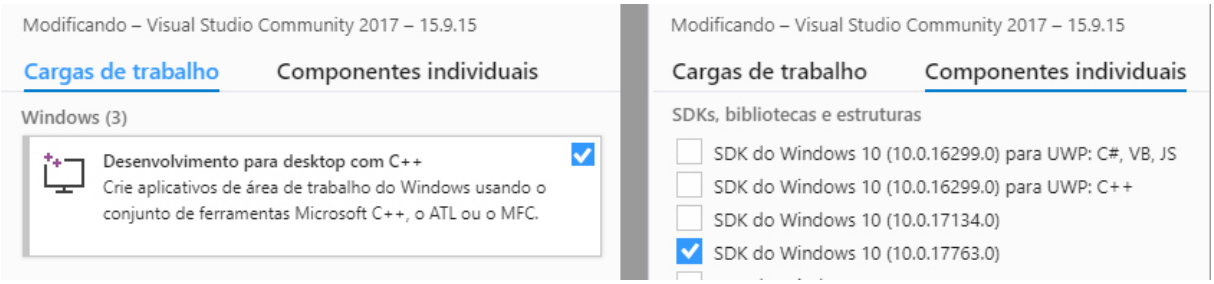
Fonte: Elaborado pelo autor.

3.3 Visual Studio 2017 Community

Caso se esteja usando Windows para a programação, o JUCE necessita da plataforma Visual Studio (versões de 2015 a 2019) para poder realizar a exportação de qualquer aplicação. A versão Community é disponibilizada gratuitamente pela Microsoft e permite desenvolver programas *open-source*.

A instalação do Visual Studio 2017 Community deve ser customizada, de maneira a abrangir certos pacotes de desenvolvimento que possibilitam o uso do JUCE. Na Figura 25 é possível ver os pacotes instalados que são mais críticos para o funcionamento do *framework*.

Figura 25 – Instalação do Visual Studio 2017 Community.



Fonte: Elaborado pelo autor.

Uma vez que o Visual Studio foi instalado, é possível usá-lo para abrir o projeto do JUCE (WEAREROLI, 2007) e o compilar, gerando assim o arquivo executável do seu gerenciador de projetos, o Projucer. Além disso, é possível utilizar o Visual Studio para a parte de programação em si dos *plug-ins*, pois possui várias funcionalidades que facilitam a programação, como a sub-divisão da janela, para visualizar múltiplos arquivos simultaneamente.

4 Gravação da guitarra

Precedentemente à programação, é necessário ter um arquivo de áudio para poder exercer testes sobre os *plug-ins*. Para tal, é realizada a obtenção do som de uma guitarra, através da gravação da mesma. Isso é feito com uma técnica chamada *Direct Input*, e utiliza apenas uma guitarra, uma interface de áudio e um computador pra salvar o arquivo. Cada um desses elementos será abordado em mais detalhes nas seguintes seções.

4.1 *Direct Input*

A técnica de gravação da guitarra usada neste Trabalho de Conclusão de Curso consiste na captação direta do som, através de um cabo regular de áudio, dispensando o uso de microfones ou amplificadores. Isso é possível graças ao uso de uma interface de áudio, cujo aprofundamento se dá na seção 4.3.

Os benefícios e malefícios de se gravar com DI são:

a) Método acessível e fácil para se gravar instrumentos elétricos;

b) Com *plug-ins*, é possível emular sons de equipamentos clássicos.

c) Gravações podem soar desinteressantes se não combinadas com o uso de *plug-ins*;

d) *Plug-ins* podem custar caro, e introduzem latência na cadeia de sinal.

Devido à praticidade e baixo custo, além do fato que serão desenvolvidos *plug-ins* para complementar o som gravado, este foi o método escolhido para captação do som da guitarra. É possível ver a aplicação desta técnica na Figura 27, pelo cabo de áudio conectado na frente da interface, ao centro da foto.

4.2 Guitarra

A guitarra usada para a gravação foi uma Les Paul de 6 cordas da *Tanglewood*, como pode ser vista na Figura 26. A única modificação feita na guitarra foi a substituição de ambos os captadores originais por um par de captadores *Seymour Duncan*, modelo Trembucker '59.

Figura 26 – Guitarra Tanglewood Les Paul TSB 58 Cherry Sunburst.

Fonte: Elaborado pelo autor.

4.3 Interface de áudio

Se trata de um equipamento imprescindível em um *home studio*. A justificativa pro uso de uma interface se dá pela qualidade de sua placa de áudio intrínseca. Uma placa de som de um computador, mesmo que moderno, não possui qualidade nem poder de processamento suficiente para realizar uma gravação com alta fidelidade. Com uma interface de áudio, porém, é possível se conectar a um computador (geralmente via USB) e se comunicar com ele, possibilitando gravar e reproduzir sons com baixa latência e sem perda de qualidade, agindo efetivamente como uma placa de áudio externa ao computador. Com este único dispositivo, é possível realizar gravações de qualquer instrumento musical, geralmente tendo a opção de trabalhar tanto com Direct Input, quanto com microfone.

Comercialmente falando, as interfaces de áudio são classificadas de acordo com o número de entradas e saídas, ou seja, a quantidade de canais que podem ser gravados simultaneamente (entrada), e a quantidade de canais disponíveis para monitoramento do som (saída). Uma opção popular hoje em dia são interfaces com dois canais de entrada (permitindo gravar dois sons distintos ao mesmo tempo) e dois canais de saída (para as duas caixas de um sistema de som estéreo). Ademais, frequentemente possuem também uma saída para monitoramento num fone de ouvido.

Além disso, possivelmente a maior diferença que se percebe entre diferentes modelos e marcas, diz respeito ao pré-amp de cada uma. O pré-amp se trata de um circuito eletrônico embutido na interface, que amplifica o sinal de entrada até um nível audível e manipulável. O pré-amp influencia grandemente na qualidade final do som, pois pode oferecer variados níveis de ruído, ganho e resposta de frequência, o que acaba colorindo o sinal, por vezes de maneira irreversível. Tendo isso em mente, até mesmo uma interface de áudio relativamente barata pode fornecer alta qualidade de gravação hoje em dia.

A interface usada chama-se Scarlett 2i2, da marca Focusrite (vide Figura 27). Possui 2 canais de entrada de microfones/instrumentos, e 2 de saída. A conexão ao computador é feita através de um cabo USB-C, e a guitarra é conectada à interface através de um cabo regular de guitarra (P10 mono). É possível notar também que existe uma luz indicadora da qualidade do sinal de entrada, além de uma saída para fone de ouvido.

Figura 27 – Interface de áudio Focusrite Scarlett 2i2 3rd gen.

Fonte: Elaborado pelo autor.

4.4 *Software* de gravação

O *software* usado para gravação é o Ableton Live 10. O sinal da gravação é recebido da interface, e é salvo em um arquivo .wav, sem perda de qualidade. Depois esse arquivo (ou faixa) pode ser manipulado no mesmo programa, motivo pelo qual essa DAW também foi a escolhida para a utilização dos plugins. Mais detalhes sobre esta ferramenta são abordados na seção 3.2.

5 Programação dos *plug-ins*

A programação de um plug-in deve ser feita tendo algumas coisas em mente: o seu ciclo de vida, bem como suas chamadas, dependem totalmente da DAW. Além disso, é a DAW que fornece a taxa de amostragem que deverá ser usada, de modo que devem ser adicionados tratamentos para quando são selecionadas taxas de amostragem que não são suportadas.

Um terceiro fator importante a ser considerado no desenvolvimento de um plug-in é sua latência. Este valor está intimamente relacionado com as técnicas de programação implementadas, especialmente em sua função de processamento, pois a latência é representada pelo tempo que o plug-in demora para receber e processar um sinal de entrada e fornecer um sinal de saída. Em outras palavras, é o tempo que o seu uso acrescenta em uma cadeia de sinal. Por este motivo, o uso de plug-ins prevalece mais em estúdios (especialmente em pós-produções, onde tempo de resposta não é uma preocupação) do que, por exemplo, em *shows* ao vivo.

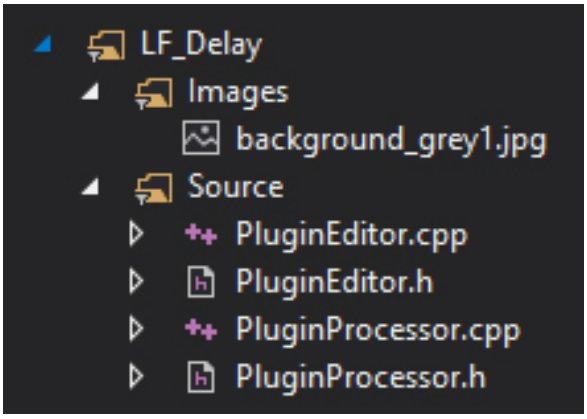
Neste capítulo será explanada a programação dos *plug-ins* de efeito, começando com a apresentação de alguns conceitos fundamentais para a compreensão da lógica por trás da programação. Após isso, cada efeito programado será abordado individualmente, expondo os raciocínios de suas abordagens, bem como trechos relevantes de códigos.

5.1 Conceitos fundamentais

Existem duas classes que são essenciais para o funcionamento de um *plug-in* com o JUCE - **AudioProcessor** (gerencia a parte de processamento) e **AudioProcessorEditor** (gerencia a interface gráfica). Quando um *plug-in* é aberto pela DAW, ou seja, instanciado, estas duas classes distintas também são instanciadas, e rodam de maneira concomitante, permitindo controle ao usuário sobre os parâmetros dos efeitos, enquanto os valores destes são obtidos para o processamento. Estas duas classes serão elucidadas ao longo desta seção, bem como outros conhecimentos relevantes.

Caso surja a necessidade, novas classes ou arquivos podem ser acrescentados ao projeto por variados motivos (e.g. melhor organização). Sabendo-se que para cada classe são gerados dois arquivos (um .cpp e um .h), tem-se quatro arquivos-fonte padrões criados junto com um projeto do tipo *Audio Plug-in*, como é possível ver na Figura 28, dentro da pasta *Source*.

Figura 28 – Estrutura do projeto de *plug-in* de *Delay*.



Fonte: Elaborado pelo autor.

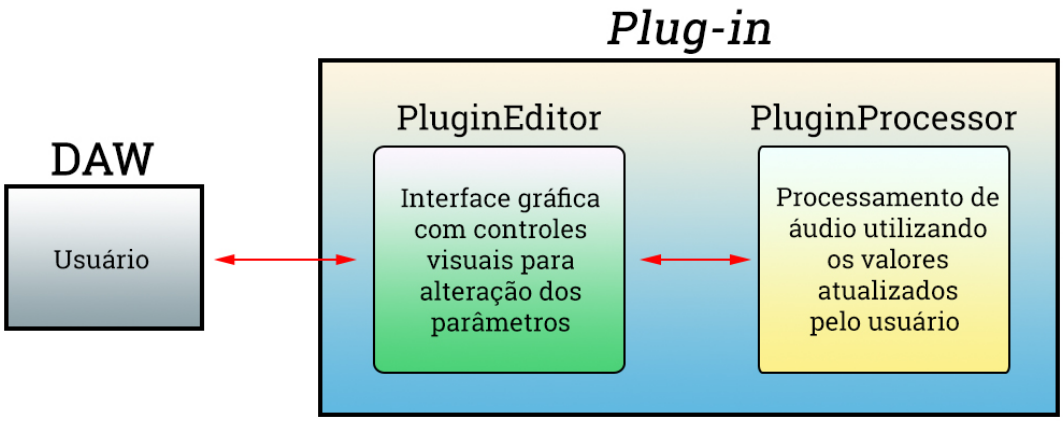
Convém notar que a pasta intitulada *Images* foi acrescentada posteriormente à criação do projeto, e contém apenas a imagem de fundo empregada no *plug-in* (mais detalhes sobre este tópico encontram-se na seção 5.1.4). Na pasta *Source* pode-se ver os quatro arquivos que são gerados automaticamente pelo Projucer:

a) PluginEditor (.cpp e .h): herda da classe *AudioProcessorEditor*. Contém o design dos elementos visuais, tais como exibição de textos e botões para controle dos parâmetros. Análogo ao *front-end*.

b) PluginProcessor (.cpp e .h): herda da classe *AudioProcessor*. Contém a implementação do algoritmo de processamento de sinais, bem como quaisquer outros mecanismos relevantes ao funcionamento interno do *plug-in*. Análogo ao *back-end*;

As classes geradas pelos arquivos PluginEditor.* e PluginProcessor.* devem se comunicar de maneira constante para que a atualização dos valores dos parâmetros, bem como o processamento do efeito possam ocorrer de maneira fluida para o usuário. A Figura 29 mostra como é feita a comunicação entre estas duas classes, bem como a comunicação do *plug-in* com a DAW.

Figura 29 – Diagrama de interface do *plug-in* com a DAW.



Fonte: Elaborado pelo autor.

Cada um destes arquivos será visto em maiores detalhes nas próximas sub-seções, pois há trechos de códigos que são comuns (ou bastante similares) a todos os *plug-ins* feitos neste Trabalho de Conclusão de Curso.

5.1.1 PluginProcessor.h

Neste arquivo *header* são declaradas constantes e variáveis essenciais para o funcionamento do *plug-in* como um todo, principalmente na parte de processamento de sinais.

Já foi visto que geralmente, efeitos sonoros são controlados através de parâmetros. O JUCE fornece classes com as quais é possível armazenar e gerenciar parâmetros com facilidade, definindo valores padrões e de mínimo e máximo. Além disso, é possível atualizar seus valores automaticamente através da interface gráfica.

A conexão de um parâmetro de efeito a um componente visual específico se dá através do uso de textos identificadores únicos, definidos neste arquivo *header*, onde também podem ser declaradas constantes para outras propriedades dos parâmetros (e.g. seus valores máximos). O Código 1 exemplifica tais declarações, utilizando o exemplo do parâmetro do número de repetições do *plug-in* de *Delay*.

Código 1 – Definições das propriedades de um parâmetro no arquivo *PluginProcessor.h*.

```
1 // parâmetro de Feedback (número de repetições)
2 #define FEEDBACK_ID       "feedback"  // identificador interno
3 #define FEEDBACK_NAME     "Feedback"  // nome que aparecerá na DAW
4 #define FEEDBACK_MAX      0.60f       // valor máximo (float)
5 #define FEEDBACK_DEFAULT  0.28f       // valor padrão (float)
```

A criação da maioria dos parâmetros é análoga a esta: um identificador e um nome para exibição são os essenciais. É através dos identificadores internos (como o **FEEDBACK_ID**) que é possível alterar os valores dos parâmetros de um efeito visualmente. O que varia muito entre diferentes parâmetros, é o alcance mínimo e máximo de valores, pois podem possuir diferentes escalas.

Na linha 3 é possível ver o nome do parâmetro que será exibido na DAW. Na linha 4, é definido um valor máximo para o número de repetições. Este valor (*0.60f*) significa que no máximo 60% do sinal original será utilizado para as repetições. Por mim, é definido o valor padrão que será atribuido a este parâmetro na inicialização do *plug-in*.

Além destas definições, existem alguns métodos - e uma variável - cujas declarações são importantes para o funcionamento do processador - como é possível ver no Código 2.

Código 2 – Declarações de objetos notáveis no arquivo *PluginProcessor.h*.

```
1  // classe processadora de um plug-in de Delay
2  class Lf_delayAudioProcessor : public AudioProcessor
3  {
4  public:
5    // variável para gerenciamento dos parâmetros
6    AudioProcessorValueTreeState treeState;
7
8    // função para declaração e configuração dos parâmetros
9    AudioProcessorValueTreeState::ParameterLayout createParameterLayout();
10
11   // funções referentes ao processamento
12   void prepareToPlay (double sampleRate, int samplesPerBlock) override;
13   void processBlock  (AudioBuffer<float>&, MidiBuffer&) override;
14
15   // salvar e recuperar valores dos parâmetros
16   void getStateInformation (MemoryBlock& destData) override;
17   void setStateInformation (const void* data, int sizeInBytes) override;
18
19 private:
20   // funções e variáveis privadas
21   float linearInterpolation(float x0, float x1, float phaseValue);
22   float mDelayTimeSmoothed;
23 }
```

Para o controle visual dos parâmetros e obtenção de seus valores para o processamento, é declarada uma variável chamada *treeState* na linha 6, do tipo **AudioProcessorValueTreeState** - se trata de uma classe com estrutura de árvore, que contém tratamentos próprios para lidar com parâmetros, podendo guardar todos os tipos de dados (de fato, todo o estado de um *AudioProcessor*). Note que esta variável deve ser declarada como pública, pois necessita

ser vista pela classe editora para a afixação do valor de um parâmetro em um componente visual (mais detalhes na seção 5.1.4).

Para a instanciação e definição dos parâmetros em si, é utilizada uma função chamada *createParameterLayout*, pertencente à classe **AudioProcessorValueTreeState**, na linha 9. Nela é possível configurar as propriedades de um parâmetro, tal como seu tipo, texto identificador, e alguns valores padrões. Mais detalhes sobre sua implementação são vistos na seção 5.1.2.

Nas linhas 12 e 13 é possível ver as funções **prepareToPlay** e **processBlock** - fundamentais para o processamento do *plug-in*. A primeira se trata de um evento disparado antes do processamento em si, servindo como um espaço para preparar as variáveis para o uso. Devido ao fato que este evento também é disparado quando a taxa de amostragem é modificada, é recebido por argumento a taxa de amostragem a partir daquele momento.

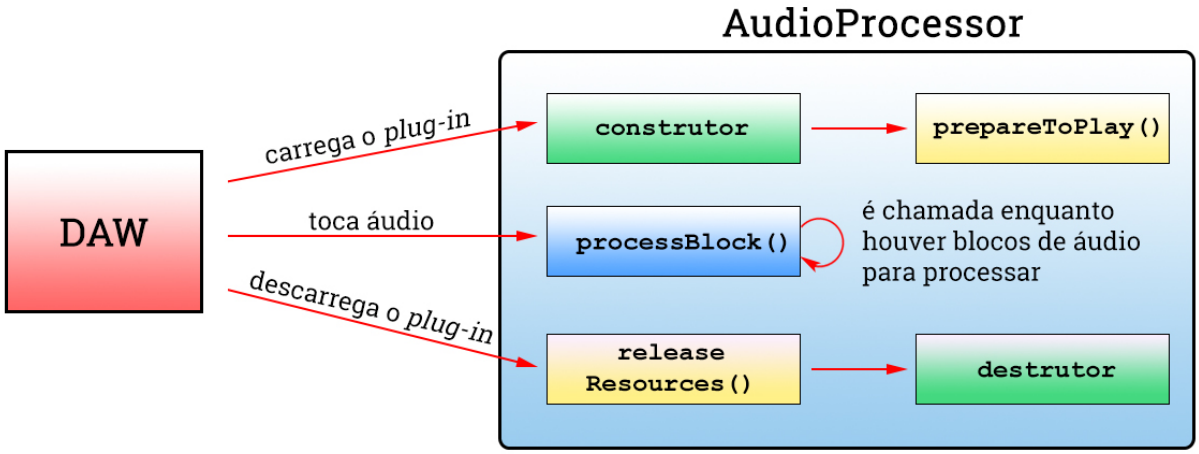
A função **processBlock** é chamada em seguida, e é onde deve ser feito todo o processamento de efeitos. Por argumento é recebido um *AudioBuffer* do tipo *float*, que representa o vetor contendo o bloco de sinais a ser processado naquele instante. Esta função é chamada enquanto houver áudio a ser tocado - motivo pela qual também é referida como função de *callback* (REISS; MCPHERSON, 2014).

Nas linhas 16 e 17 são declaradas duas funções utilizadas para salvar os valores dos parâmetros usados pelo plug-in na memória do programa, de modo que é possível recuperar estes valores quando o plug-in é re-instanciado.

Por último, na seção privada da classe, foram incluídos no exemplo uma função (linha 21) e uma variável (linha 22), que são referentes ao cálculo de interpolação matemática aplicado nos valores recebidos pela interface gráfica. Este cálculo é feito para se obter um som resultante mais suave enquanto o valor deste parâmetro é alterado.

Na Figura 30, é possível ver o ciclo de vida de um objeto do tipo **AudioProcessor**. Como plug-ins são essencialmente bibliotecas dinâmicas, não existe tipo uma função Main. Só existem funções que são chamadas quando o programa hospedeiro (no caso, a DAW) executa certas ações.

Figura 30 – Ciclo de vida de um objeto do tipo AudioProcessor.



Fonte: Elaborado pelo autor.

A DAW pode executar, a grosso modo, três ações: carregar e descarregar um plugin, e tocar áudio. Além disso, quando o usuário muda a taxa de amostragem, o evento **prepareToPlay** é disparado.

5.1.2 PluginProcessor.cpp

É neste arquivo que são aplicados os algoritmos processadores dos efeitos. As funções mais importantes chamam-se *prepareToPlay* e *processBlock*. Porém, primeiramente deverá ser visto o construtor do objeto *AudioProcessor* implementado neste arquivo, para entendimento d. O construtor e destrutor dessa classe são chamados, respectivamente, quando o *plug-in* é carregado e descarregado pela DAW.

Em sua maioria, as classes fornecidas pelo JUCE possuem métodos para liberação automática da memória utilizada pela instância do seu objeto em seus destrutores. Deste modo, geralmente só é necessário fazer modificações no construtor da classe *AudioProcessor*, como pode-se ver no Código 3.

Código 3 – Construtor da classe processadora do *plug-in*.

```
1 // construtor de uma classe de Delay (herda da AudioProcessor)
2 Lf_delayAudioProcessor::Lf_delayAudioProcessor()
3     // configurações dos canais de entrada e saída
4     : AudioProcessor(BusesProperties()
5         .withInput ("Input",  AudioChannelSet::stereo(), true)
6         .withOutput("Output", AudioChannelSet::stereo(), true)
7     ),
8     // instanciar a árvore de valores, chamando seu construtor
9     treeState(
```

```
10          *this,                    // AudioProcessor para se conectar
11          nullptr,                  // gerenciador de ações a ser usado
12          TREESTATE_ID,             // identificador único desta árvore
13          createParameterLayout()   // configurações dos parâmetros
14      )
15  {
16      // resetar valores de variáveis globais
17      mDelayTimeInSamples = 0;
18  }
```

No construtor de um objeto do tipo AudioProcessor, deve-se definir as configurações para canais de entrada e saída - caso contrário, serão criados barramentos de áudio desabilitados por padrão, que deverão ser habilitados por código posteriormente. Isto pode ser visto nas linhas 4 a 7 - são configurados canais estéreo tanto para entrada quanto para saída. Ou seja, o *plug-in* recebe e envia de volta para a DAW sinais de áudio estéreo.

Nas linhas 9 a 14 é possível ver a chamada do construtor da variável *treeState*, que armazena os valores de parâmetros deste *plug-in*. O primeiro argumento deste construtor (na linha 10) recebe um *AudioProcessor*, ao qual é passado o ponteiro referente à instância atual. Segundo a própria documentação do JUCE (JUCE, 2018), um objeto do tipo *AudioProcessorValueTreeState* só deve ser conectado a um único objeto do tipo *AudioProcessor*, e estes devem ter o mesmo ciclo de vida, pois possuem dependências um do outro.

Caso necessário, é possível associar um gerenciador de ações a este objeto, permitindo que o usuário desfaça ações passadas. Porém, não tendo sido verificada a necessidade para uso deste artifício neste trabalho, então é passado um ponteiro nulo (*nullptr*) neste argumento do construtor, como visto na linha 11. Na linha seguinte, é necessário definir um identificador interno em texto para esta instância deste objeto. Este identificador geralmente é usado quando se tem mais de uma variável da classe **AudioProcessorValueTreeState**.

Na linha 13 é passada a função *createParameterLayout*, que por sua vez configura os parâmetros usados na árvore. É possível ver sua definição a seguir, nos Códigos 4 e 5. E por último, a partir da linha 16, é possível executar quaisquer códigos que precisem ser efetuados no início da execução do *plug-in*. Geralmente este espaço é utilizado para zerar as variáveis globais usadas.

Código 4 – Configuração de um parâmetro dentro da função *createParameterLayout*.

```
1 // "feedback" recebe um ponteiro único e do tipo AudioParameterFloat,
    cuja construção segue
2 auto feedback = std::make_unique<AudioParameterFloat>(
3     FEEDBACK_ID,          // identificador interno
4     FEEDBACK_NAME,        // nome de exibição na DAW
5     NormalisableRange<float>(0.0f, FEEDBACK_MAX), // valores mín. e máx.
6     FEEDBACK_DEFAULT      // valor padrão
```

```
7       juce::String("%")     // texto sufixo que aparece na DAW
8   );
```

Na função **createParameterLayout**, a primeira coisa que deve ser feita é configurar os parâmetros. Um exemplo disto pode ser visto na linha 2: a variável **feedback** recebe um ponteiro através da função **make_unique**. O nome deste tipo de ponteiro (chamado ponteiro único), se dá ao fato de que detém propriedade exclusiva sobre um objeto. O benefício que isso traz é que quando sai de escopo, este ponteiro destrói o objeto apontado, e se auto-destrói em seguida (REFERENCE, 2019a). Com ele, não é necessário se preocupar com o ciclo de vida dos objetos que representam os parâmetros - ou seja, liberar a memória utilizada pelos objetos.

Ainda na linha 2, é interessante notar que a palavra-chave **auto**, a partir do C++11, significa que o tipo da variável sendo declarado será definido pelo compilador, não pelo programador (REFERENCE, 2019b). No caso, como o parâmetro pode ser bem representado por um valor decimal contido num determinado intervalo, foi escolhido o uso do tipo **AudioParameterFloat**. Com ele, é possível definir valores mínimo e máximo, além de um valor padrão para o parâmetro, entre outros.

Nas linhas 3 a 7 é possível ver os argumentos utilizados no construtor de um objeto deste tipo. A identificação do parâmetro, se dá por meio de um texto identificador único, intitulado **FEEDBACK_ID**. A declaração deste valor, assim como os demais argumentos utilizados aqui, se encontra no arquivo *PluginProcessor.h* (vide Código 1).

Código 5 – Retorno dos parâmetros pela função *createParameterLayout*.

```
1  // instancia um vetor de ponteiros do tipo AudioParameterFloat
2  std::vector <std::unique_ptr<AudioParameterFloat>> params;
3
4  // insere os parâmetros no vetor
5  params.push_back(std::move(dryWet));      // volume do efeito
6  params.push_back(std::move(delayTime));   // tempo do delay
7  params.push_back(std::move(feedback));    // número de repetições
8
9  // retorna vetor com parâmetros
10 return { params.begin(), params.end() };
```

Neste caso, existem outros dois parâmetros além do número de repetições: o volume do efeito (*dryWet*) e seu tempo (*delayTime*). A construção destes parâmetros é semelhante à mostrada no Código 4, mudando apenas os identificadores e o possível alcance dos valores, todos declarados no arquivo *PluginProcessor.h*, como pode-se ver no Código 1.

As funções **prepareToPlay** e **processBlock** variam imensamente de acordo com o *plug-in*, portanto, suas implementações serão apresentadas na seção contendo seu *plug-in*

correspondente. A seguir começará a ser vista a classe que gerencia a parte visual.

5.1.3 PluginEditor.h

Neste arquivo ficam as declarações das variáveis e funções utilizadas pela classe gerenciadora da interface gráfica do *plug-in*, como é possível ver no Código 6 (o construtor e destrutor da classe foram omitidos).

Código 6 – Declarações de objetos notáveis no arquivo *PluginEditor.h*.

```
1  // classe da interface gráfica do plug-in de Delay
2  class Lf_delayAudioProcessorEditor : public AudioProcessorEditor
3  {
4  public:
5      // eventos gráficos
6      void paint (Graphics&) override;
7      void resized() override;
8
9  private:
10     // referência à instância da classe processadora
11     Lf_delayAudioProcessor& processor;
12
13     // aqui são declarados os componentes visuais
14     Slider sliderFeedback; // botão para o número de repetições
15     Label  labelFeedback;  // texto para o número de repetições
16     //entre outros...
17
18 public:
19     // anexo do Slider de Feedback
20     std::unique_ptr<AudioProcessorValueTreeState::SliderAttachment>
       feedbackParam;
21     //demais anexos...
22 };
```

A função **paint** (declarada na linha 6) é chamada quando algum componente da janela do *plug-in* necessita ser re-desenhada (e.g. quando o *plug-in* é minimizado, e então restaurado). Aqui se encontram funções para desenhar na janela (e.g. exibir uma imagem de fundo). Já a função **resized** (na linha 7) é chamada quando o tamanho da janela do *plug-in* é alterado pelo usuário, e serve para posicionar corretamente os componentes na tela. No entanto, este recurso de redimensionamento da janela não será utilizado neste Trabalho de Conclusão de Curso, portanto, sua implementação é vazia.

É possível ver na linha 11 que a classe editora possui uma referência à instância correspondente da classe processadora, permitindo que a árvore de valores desta possa ser acessada

pela classe editora. Esta referência permite estabelecer uma conexão entre um parâmetro e um componente visual, e seu uso será exemplificado no Código 7.

Nas linhas 14 e 15 são exemplificadas as declarações de alguns componentes visuais utilizados neste plug-in. Um componente do tipo **Slider** permite que o usuário controle um parâmetro de maneira gráfica (e.g. um botão deslizante). Já uma **Label**, exibe texto.

Na linha 20, pode-se ver a declaração de um ponteiro único do tipo **AudioProcessorValueTreeState::SliderAttachment**. Objetos deste tipo servem para fazer a conexão entre um *Slider* (declarado nesta classe) e um parâmetro de um objeto *AudioProcessorValueTreeState* (declarada na classe processadora). Assim sendo, variáveis do tipo **SliderAttachment** são como objetos anexados aos *Sliders*, e obtém seu valor para atualizá-lo na árvore de valores da classe processadora. Deste modo, o processamento do plug-in pode ocorrer sempre com os valores atualizados pela interface. Esta conexão será elucidada na seção 5.1.4

É interessante notar que estes ponteiros são declarados em uma nova seção de variáveis públicas (feita na linha 18). Isso é feito porque os objetos das classes do JUCE são deletados na ordem reversa em que são declarados (ou seja, de baixo para cima). Segundo a documentação do JUCE (JUCE, 2015), é necessário garantir que estes ponteiros sejam destruídos antes do **Slider** e do **AudioProcessorValueTreeState** aos quais ele se liga.

5.1.4 PluginEditor.cpp

Neste arquivo é implementado todo o código necessário para a visualização e interação gráfica do usuário com o *plug-in*. Primeiramente é mostrado o construtor, juntamente com a instanciação de uma Label e um Slider, bem como seu anexo.

Código 7 – Construtor da classe editora do visual do *plug-in*.

```
1  // construtor da classe gerenciadora da interface gráfica
2  Lf_delayAudioProcessorEditor::Lf_delayAudioProcessorEditor (
3     Lf_delayAudioProcessor& p) : AudioProcessorEditor (&p), processor (p)
4  {
5     // definir tamanho da janela, em pixels
6     setSize (420, 170);
7
8     // configurar Slider de Feedback
9     sliderFeedback.setBounds(300, 45, 120, 120); // posição na janela
10    sliderFeedback.setSliderStyle (Slider::RotaryVerticalDrag);
11    // configurar caixa de texto deste Slider
12    sliderFeedback.setTextBoxStyle(Slider::TextBoxBelow, false, 100, 20);
13    sliderFeedback.setTextValueSuffix(" %"); // texto sufixo da caixa
14
15    // configurar Label referente ao Feedback
16    labelFeedback.setText("Repetições", dontSendNotification);
```

```
17      labelFeedback.attachToComponent(&sliderFeedback, false);
18
19      // adicionar e exibir estes dois componentes
20      addAndMakeVisible(sliderFeedback);
21      addAndMakeVisible(labelFeedback);
22
23      // estabelecer a conexão entre Slider e parâmetro
24      feedbackParam =
25      std::make_unique<AudioProcessorValueTreeState::SliderAttachment>
26      (
27          processor.treeState, // Árvore de valores
28          FEEDBACK_ID,         // identificador interno do parâmetro
29          sliderFeedback       // Slider a ser conectado
30      );
31  }
```

No Código 7, é possível ver a chamada do construtor de uma classe editora de um plug-in do JUCE. Neste construtor, é necessário definir o tamanho da janela (feito na linha 6), bem como as configurações dos componentes visuais que serão exibidos (linhas 9 a 16). Uma vez feita essas configurações, é preciso adicionar os componentes à classe, o que se verifica nas linhas 19 e 20.

Vale a pena notar que o estilo selecionado para o *Slider* em questão é do tipo **Slider::RotaryVerticalDrag**, o que faz com que tenha a aparência de um botão deslizante em círculo. Além disso, foi definido que este *Slider* possuirá uma caixa de texto abaixo de si, que exibirá o seu valor e permitirá que o usuário possa inserir um valor manualmente.

Para a **Label** do número de repetições (*feedback*), foi definido o texto que deverá exibir, na linha 16; na linha seguinte é possível ver que este componente é anexado ao componente **sliderFeedback**, de modo que sua posição na janela não precisa ser determinada (apenas a do *Slider* - feita na linha 9). Tanto o *Slider* quanto a *Label* são efetivamente exibidos com os comandos das linhas 20 e 21.

Como já foi mencionado anteriormente, precisa ser estabelecida uma conexão entre um **Slider** e um parâmetro, para que o valor deste possa ser atualizado e processado de maneira contínua. Esta conexão é feita através do uso ponteiros únicos do tipo **AudioProcessorValueTreeState::SliderAttachment** (um para cada parâmetro). Devido à peculiaridade deste tipo de ponteiro (cuja explicação se dá na seção 5.1.1), estes necessitam ser declarados como variáveis globais, pois caso contrário - se declarados dentro de uma função qualquer -, quando saírem de escopo, a conexão entre um *Slider* e um parâmetro será perdida.

Código 8 – Função *paint* da classe editora do visual do *plug-in*.

```
1  // evento chamado quando a janela é (re-)desenhada
2  void Lf_delayAudioProcessorEditor::paint (Graphics& g)
```

```
3  {
4      // carregar imagem de fundo em uma variável
5      Image backgroundImage = ImageCache::getFromMemory
6          BinaryData::background_grey1_jpg,
7          BinaryData::background_grey1_jpgSize
8      );
9
10     // desenhar (exibir) a imagem de fundo
11     g.drawImageAt(backgroundImage, 0, -50);
12 }
```

Note, no Código 8, que esta função recebe por argumento uma variável chamada **g**, através da qual é possível desenhar componentes na tela. Esta variável é usada apenas para desenhar uma imagem de fundo (na linha 11). Nas linhas 5 a 8 é possível ver o procedimento necessário para carregar um arquivo de imagem programaticamente. O nome deste arquivo (bem como o próprio) podem ser vistos na Figura 28.

Como mencionado previamente, a janela não será redimensionável, portanto, a função **resized** não precisará ser sobrecarregada. Antes de serem mostradas as peculiaridades da programação de cada *plug-in*, será visto um diagrama importante para o entendimento de como é feita a ligação entre as duas classes de um *plug-in*.

5.2 Diagrama de comunicação interclasses

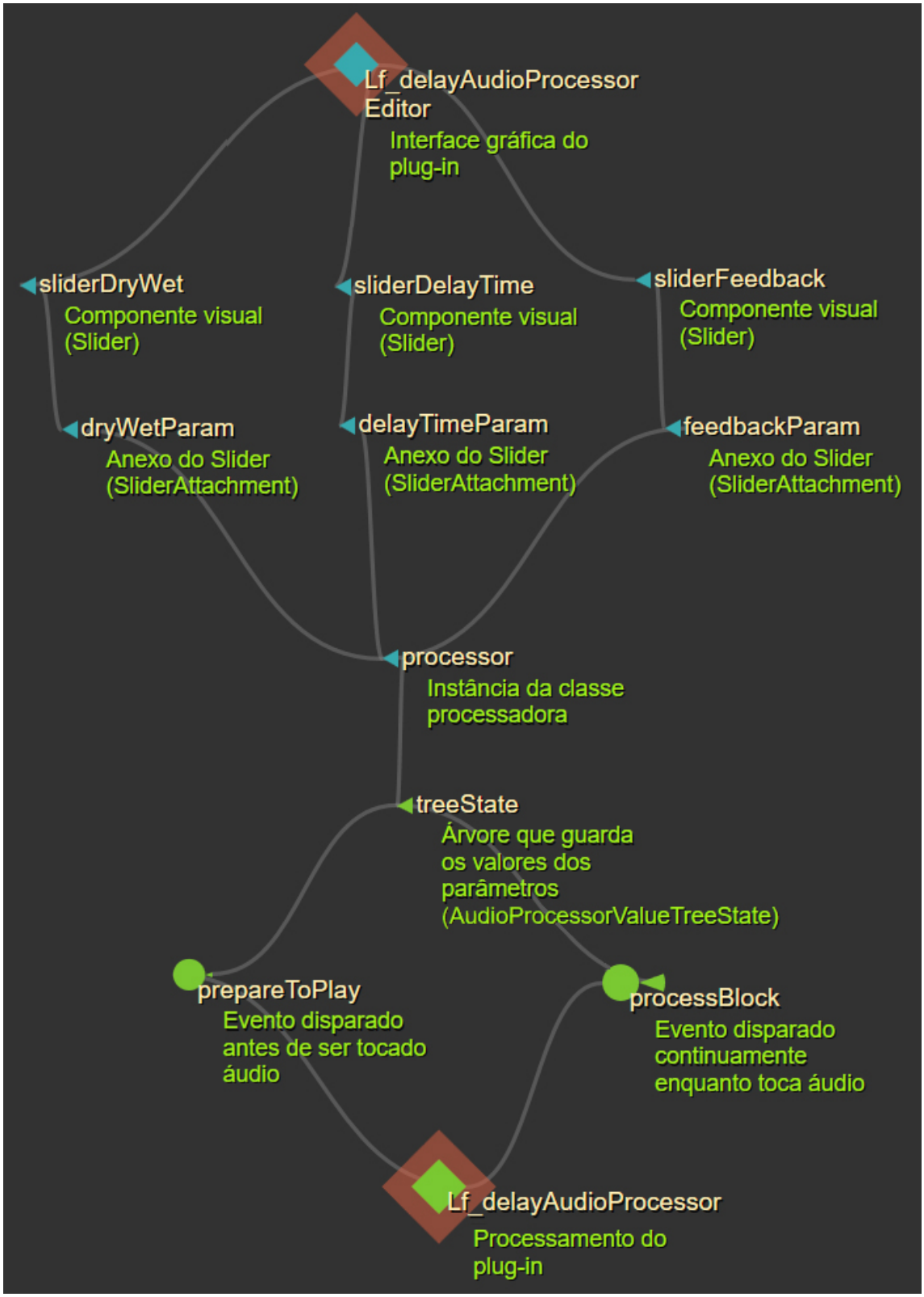
Foi visto que plug-ins de áudio são como bibliotecas que podem ser carregadas dinamicamente por DAWs, e possuem duas classes distintas que executam de maneira concomitante: uma que gere a interface gráfica (do tipo **AudioProcessorEditor**), e uma processadora de sinais (do tipo **AudioProcessor**). A classe processadora possui apenas funções de *callback*, que são chamadas pela DAW em certos eventos (tal como, durante o *playback* de música). Além disso, faz uso de uma estrutura em forma de árvore que gerencia os valores dos parâmetros usados no processamento de sinais.

O usuário pode fazer uso do *plug-in* uma vez que este é carregado pela DAW. Este interfaceamento se dá através da classe editora visual do *plug-in* - mais especificamente, através de componentes visuais chamados **Sliders**. Estes componentes aparecem na tela como botões giratórios, e possibilitam o usuário de alterar certos valores que são usados no processamento do efeito (parâmetros). A conexão entre um parâmetro e um componente visual do tipo *Slider* se dá através de objetos do tipo **SliderAttachment** (anexos).

A Figura 31 evidencia a comunicação entre as duas classes do plug-in; mais especificamente, sobre a atualização dos parâmetros dos efeitos. Acima, em azul, se tem a classe que gerencia a interface gráfica do plug-in, e seus objetos (três Sliders e seus anexos correspondentes, e uma referência à instância da classe processadora). Na parte de baixo do diagrama,

em verde, se tem a classe processadora de sinais do plug-in, bem como sua árvore de valores e dois eventos que são disparados pela DAW.

Figura 31 – Fluxograma de comunicação entre a classe editora visual e a processadora.

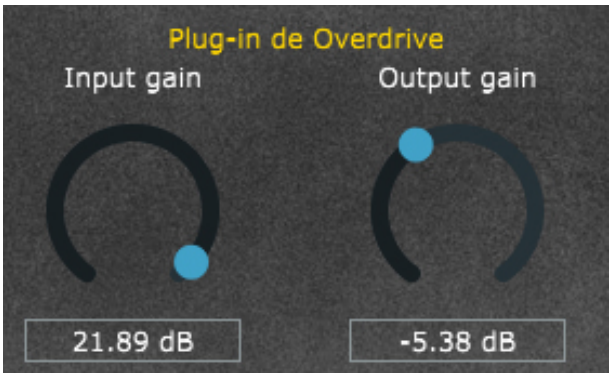


Fonte: Elaborado pelo autor.

5.3 *Plug-in* de Distorção

O *plug-in* programado de Distorção pode ser visto na Figura 32. Contém dois parâmetros: ganho de entrada e ganho de saída, ambos medidos em decibéis.

Figura 32 – Interface gráfica do *plug-in* de Distorção.



Fonte: Elaborado pelo autor.

Foi desenhado de maneira que produz distorção do tipo *hard clipping* (como visto na Figura 11). É possível ver como isto é programado no Código 9 - mais especificamente, nas linhas 21 a 29.

Código 9 – Função *processBlock* do *plug-in* de Distorção.

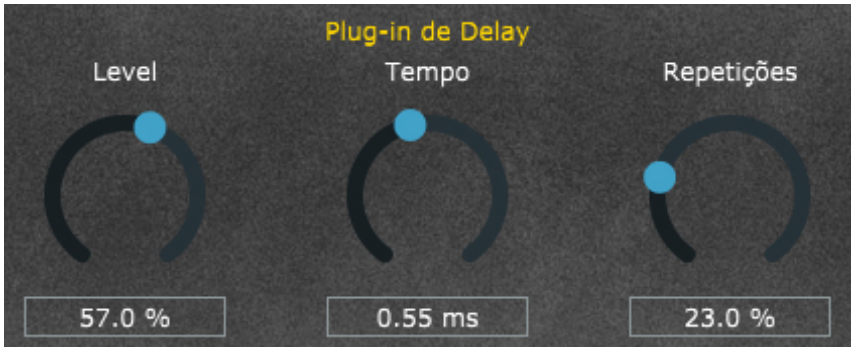
```
1  // percorre todos os canais
2  for (int channel = 0; channel < getTotalNumInputChannels(); ++channel)
3  {
4      // pega o ponteiro de escrita do canal
5      float* channelData = buffer.getWritePointer(channel);
6      float out; // amostra de saída
7
8      // percorre todas as amostras do buffer
9      for (int sample = 0; sample < buffer.getNumSamples(); ++sample)
10     {
11         // recebe valores dos parâmetros
12         auto inputParam  = *tree.getRawParameterValue(INPUTGAIN_ID);
13         auto outputParam = *tree.getRawParameterValue(OUTPUTGAIN_ID);
14
15         // aplica função logarítmica no valor do ganho de entrada
16         auto inputValueLog = powf(10.0f, inputParam * 0.05f);
17         // define amostra de entrada com ganho aplicado
18         const float in = channelData[sample] * inputValueLog;
19
20         // define ponto a partir do qual ocorrerá distorção
21         float threshold = 0.5f;
```

```
22
23          // verifica se a amostra passou deste ponto
24          if (in > threshold)
25              out = threshold;  // valor máximo
26          else if (in < -threshold)
27              out = -threshold; // valor mínimo
28          else
29              out = in;         // não ocorre distorção
30
31          // aplica o filtro na sample
32          float filtered = filters[channel]->processSingleSampleRaw(out);
33          // aplica o ganho de saída
34          const float outputValueLog = powf(10.0f, outputParam * 0.05f);
35
36          // escreve a amostra no buffer
37          channelData[sample] = filtered * outputValueLog;
38      }
39  }
```

5.4 *Plug-in* de *Delay*

Na Figura 33 é possível ver a interface gráfica deste *plug-in*. Foram criados três parâmetros: *Level* (controla a porcentagem de efeito no som resultante); Tempo (quanto tempo terá entre repetições), e número de repetições.

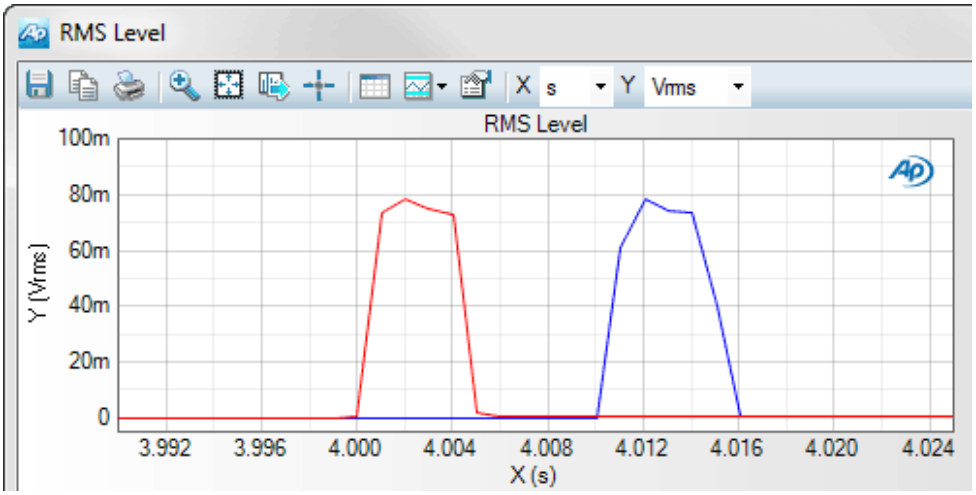
Figura 33 – Interface gráfica do *plug-in* de *Delay*.



Fonte: Elaborado pelo autor.

Uma representação visual deste efeito se encontra na Figura 34. Primeiramente é recebido um sinal (em vermelho). Após um determinado tempo, o sinal é repetido (em azul). É interessante notar que estes sinais não são exatamente iguais - isso é devido ao mecanismo de retroalimentação do efeito, que será elucidado em seguida.

Figura 34 – Representação visual do efeito de *delay*.



Fonte: ap.com (PRECISION, 2013).

Foi necessária a implementação de um *buffer* circular que retém os valores referentes às amostras que já foram processadas. Este *buffer* é retroalimentado com amostras novas, de modo que o parâmetro de *feedback* (repetições) controla a porcentagem das amostras que serão retroalimentadas. Para minimizar o código, foi incluído apenas o código referente ao canal esquerdo, como é possível ver no Código 10, na função *processBlock*.

Código 10 – Função *processBlock* do *plug-in* de *Delay*.

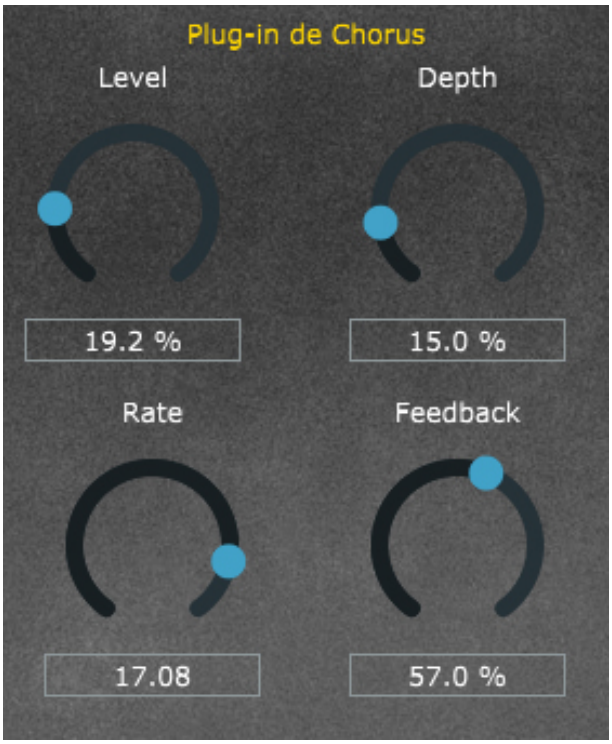
```
// pega o ponteiro de escrita do buffer do canal da esquerda
float* leftChannel = buffer.getWritePointer(0);

// percorre todas as amostras
for (int i = 0; i < buffer.getNumSamples(); i++)
{
    // preenche o buffer do efeito (com feedback retroalimentado)
    circularBufferLeft[writeHead] = leftChannel[i] + mFeedbackLeft;

    // gera a sample do canal da esquerda
    float delaySampleLeft = linearInterpolation(L0, L1, readHeadLeft);

    // pega os valores dos parâmetros
    float levelValue    = *treeState.getRawParameterValue(DRYWET_ID);
    float feedbackValue = *treeState.getRawParameterValue(FEEDBACK_ID);

    // atualiza valor do feedback retroalimentado
    mFeedbackLeft = delaySampleLeft  * feedbackValue;

    // gera amostra com a devida porcentagem da amostra original
    auto sampleLeft = buffer.getSample(0, i) * (1 - levelValue);
```

```
22      // adiciona porcentagem da amostra com efeito
23      sampleLeft += (delaySampleLeft * levelValue)
24
25      // insere a amostra modificada no canal esquerdo
26      buffer.setSample(0, i, sampleLeft);
27  }
```

5.5 *Plug-in* de *Chorus*

A interface gráfica do *plug-in* programado de *Chorus* pode ser vista na Figura 35. Contém quatro parâmetros: volume do efeito (*Level*), profundidade (*Depth*), velocidade (*Rate*) e *Feedback*, que, assim como no *plug-in* de *Delay*, representa a porcentagem do efeito que é retroalimentada em si mesmo.

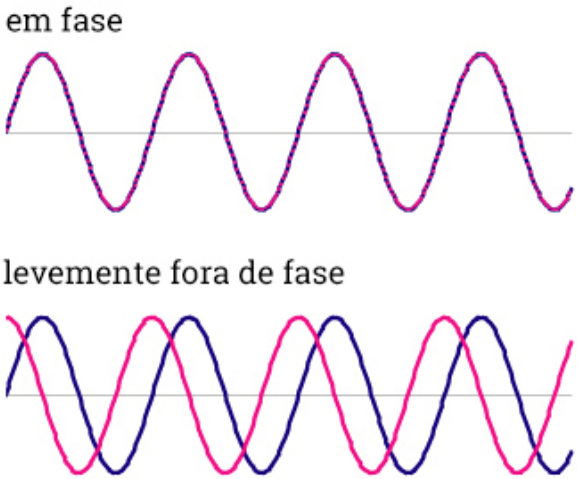
Figura 35 – Interface gráfica do *plug-in* de *Chorus*.



Fonte: Elaborado pelo autor.

A estrutura deste *plug-in* como um todo é muito parecida com a do plug-in de *Delay* (também se faz uso de um *buffer* circular). Porém, também são utilizados dois osciladores de baixa frequência - um para cada canal -, que ficam levemente fora de fase entre si. Este fenômeno gera o efeito de *chorus*, e é possível ver sua representação visual na Figura 36.

Figura 36 – Representação visual da fase entre duas ondas.



Fonte: mcat-review.org (MCAT, 2008).

No Código 11 é possível ver como os parâmetros de profundidade e velocidade são utilizados em conjunto com os osciladores de baixa frequência.

Código 11 – Função *processBlock* do *plug-in* de *Chorus*.

```
// recebe valores dos parâmetros de Velocidade e Profundidade
auto rateValue  = *treeState.getRawParameterValue(RATE_ID);
auto depthValue = *treeState.getRawParameterValue(DEPTH_ID);

// forma de onda do oscilador (esquerdo)
float lfoOutLeft = sin(2 * M_PI * lfoPhaseLeft);

// fase do oscilador (direito) - levemente fora de fase
float lfoPhaseRight = lfoPhaseLeft + 0.05f;
// forma de onda do oscilador (direito)
float lfoOutRight = sin(2 * M_PI * lfoPhaseRight);

// acrescenta profundidade
lfoOutLeft  *= depthValue;
lfoOutRight *= depthValue;

// anda com a fase do oscilador (esquerdo) baseado na velocidade atual
lfoPhaseLeft += rateValue / getSampleRate();
```

Posteriormente as variáveis **lfoOutLeft** e **lfoOutRight** serão utilizadas para calcular a posição do *buffer* circular na qual que deverão ser obtidas as amostras sonoras resultantes.

6 Conclusão

O estudo na área de processamento de sinais digitais engloba dezenas de outros campos, podendo fornecer diversas ferramentas para a solução de problemas da atualidade - que não se limitam à computação. Devido à grande aplicabilidade e interdisciplinaridade desta área, o estudo da mesma requer constante atenção e atualização, visando sempre a máxima eficiência dos algoritmos processadores de sinais.

O processamento de sinais digitais se encontra em grande parte dos eletrônicos atualmente (tais como computadores e celulares). Além disso, graças à acessibilidade de tecnologias de gravação e produção de sons, cada vez mais pessoas aderem à criação de *home studios*, para poderem produzir músicas no conforto do lar. No entanto, o uso das ferramentas de DSP contidas nos equipamentos usados popularmente muitas vezes passa despercebido ao usuário.

Este Trabalho de Conclusão de Curso une conceitos de processamento de sinais digitais com a comodidade de um *home studio* - através da programação de *plug-ins* processadores de efeitos sonoros. A implementação dos efeitos sonoros deste projeto tem como objetivo promover ao usuário comum o uso deste tipo de ferramenta, personalizável e parametrizável. Foi verificado que não é necessário a implementação de algoritmos exageradamente sofisticados para que se obtenha um som satisfatório, e com baixa latência.

Os trabalhos futuros remetem à criação de outros efeitos sonoros popularmente utilizados com guitarra, tais como compressores, harmonizadores, *reverbs* e diferentes tipos de distorção (e.g. *soft clipping* e *fuzz*). O código-fonte dos *plug-ins* desenvolvidos neste trabalho podem fornecer uma boa base para tal, e se encontram no repositório *online* do GitHub do autor (BASTOS, 2019). Em um momento futuro, é interessante re-estruturar o código de maneira que sejam usadas classes para os diferentes objetos usados, como os parâmetros e os osciladores de baixa frequência usados no *plug-in* de *chorus*.

Referências

ABLETON. *Supported Plug-in formats*. 2012. Disponível em: <https://help.ableton-.com/hc/en-us/articles/209769405-Supported-Plug-in-formats>. Acesso em: 01 set 2019.

ASSOCIATION, T. S. D. . C. *HISTORY OF SOUND IN THEATRE*. 2017. Disponível em: <https://tsdca.org/history/>. Acesso em: 01 out 2019.

AUDIO, E. *H9 | Eventide Harmonizer® Effects Pedal*. 2015. Disponível em: <https://www-.eventideaudio.com/products/stompboxes/multi-effect-processor/h9>. Acesso em: 31 out 2019.

AUDIO, W. *Waves Q10 Enters TECnology Hall Of Fame*. 2011. Disponível em: <https://www.waves.com/waves-q10-enters-tecnology-hall-of-fame>. Acesso em: 21 out 2019.

AUDIO, W. *Q10 - 10 Band Paragraphic EQ Plugin*. 2017. Disponível em: <https://www-.waves.com/plugins/q10-equalizer>. Acesso em: 21 out 2019.

AUDIO, W. *PuigTec EQP-1A + MEQ-5 EQ Plugins*. 2019. Disponível em: <https://www-.waves.com/plugins/puigtec-eqs>. Acesso em: 21 out 2019.

BASTOS, L. *Plug-ins de áudio do meu TCC*. 2019. Disponível em: <https://github.com-/baea909/TCC-audio-plugins>. Acesso em: 20 nov 2019.

BERG, R. E. *Acoustics*. 2019. Disponível em: <https://www.britannica.com/science-/acoustics>. Acesso em: 16 out 2019.

BRITANNICA, T. E. of E. *Digital sound recording*. 2011. Disponível em: <https://www-.britannica.com/technology/digital-sound-recording>. Acesso em: 17 out 2019.

BURG, J.; ROMNEY, J.; SCHWARTZ, E. *6.3.3 Type of Synthesis*. 2019. Disponível em: <http://digitalsoundandmusic.com/6-3-3-type-of-synthesis/#noopener/4>. Acesso em: 03 nov 2019.

BUTCHER, M. *Music Hardware Maker ROLI Acquires JUCE, A Key Music Industry Framework*. 2014. Disponível em: <https://techcrunch.com/2014/11/18/music-hardware-maker-roli-acquires-juce-a-key-music-industry-framework/>. Acesso em: 12 out 2019.

COOPER, D. *Buying Guide - Studio Furniture And Outboard Rack Units - We Share Our Experiences With You*. 2019. Disponível em: <https://www.pro-tools-expert.com-/production-expert-1/2019/5/14/what-studio-furniture-desks-and-outboard-rack-units-do-you-own-and-recommend>. Acesso em: 18 out 2019.

DREGNI, M. *Arbiter Fuzz Face*. 2012. Disponível em: <https://www.vintageguitar.com-/16117/arbiter-fuzz-face/>. Acesso em: 31 out 2019.

FLANDRIN, P. *Signal processing: a field at the heart of science and everyday life*. 2018. Disponível em: <http://theconversation.com/signal-processing-a-field-at-the-heart-of-science-and-everyday-life-89267>. Acesso em: 21 out 2019.

FUMO, D. *6 Echo Chambers That Shaped the Sound of Pop Music*. 2019. Disponível em: <https://reverb.com/news/6-echo-chambers-that-shaped-the-sound-of-popular-music>. Acesso em: 16 out 2019.

GUITAR, P. *Studio Tour - Steve Vai's Harmony Hut*. 2012. Disponível em: <https://www-.youtube.com/watch?v=r5D0FfQkDWg>. Acesso em: 21 out 2019.

HIGHAM, T.; BASELL, L.; JACOBI, R. Testing models for the beginnings of the aurignacian and the advent of figurative art and music: The radiocarbon chronology of geißenklösterle. *Journal of Human Evolution*, v. 62, n. 6, p. 664 – 676, 2012. ISSN 0047-2484. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0047248412000425>.

HOBBS, J. *The Best Vocal Compressors for Studio-Quality Audio*. 2019. Disponível em: <https://ledgernote.com/reviews/best-vocal-compressor/>. Acesso em: 03 nov 2019.

JUCE. *AudioProcessorValueTreeState::SliderAttachment Class Reference*. 2015. Disponível em: <https://docs.juce.com/master-/classAudioProcessorValueTreeState_1_1SliderAttachment.html>. Acesso em: 31 out 2019.

JUCE. *AudioProcessorValueTreeState Class Reference*. 2018. Disponível em: <https://docs.juce.com/master/classAudioProcessorValueTreeState-.html#acca3c6aca19e89d2781f0d1314d639e8>. Acesso em: 28 out 2019.

KILGOUR, F. G. Vitruvius and the early history of wave theory. *Technology and Culture*, [The Johns Hopkins University Press, Society for the History of Technology], v. 4, n. 3, p. 282–286, 1963. ISSN 0040165X, 10973729. Disponível em: <http://www.jstor.org/stable/3100857>.

KIRBY, P. R. *The Evolution and Decline of the Traditional Recording Studio*. 2015. Disponível em: <https://livrepository.liverpool.ac.uk/3000867/>. Acesso em: 30 out 2019.

MCALLISTER, M. *What is Spring Reverb?* 2019. Disponível em: <https://producelikeapro-.com/blog/spring-reverb/>. Acesso em: 01 out 2019.

MCAT. *Waves and Periodic Motion*. 2008. Acesso em: 10 nov 2019. Disponível em: <http://mcat-review.org/waves-periodic-motion.php>.

MUSEUM, C. H. *1979: SINGLE CHIP DIGITAL SIGNAL PROCESSOR INTRODUCED*. 2007. Disponível em: <https://www.computerhistory.org/siliconengine/single-chip-digital-signal-processor-introduced/>. Acesso em: 03 nov 2019.

OLSHAUSEN, B. A. Aliasing. 2000. Disponível em: <http://www.rctn.org/bruno/npb261-/aliasing.pdf>. Acesso em: 30 out 2019.

PAUL, J. D. *The Origins Of Dsp And Compression*. 2006. Disponível em: <http://aes-media-.org/historical/pdf/paul_origins-of-dsp.pdf>. Acesso em: 21 out 2019.

PICKFORD, J. *Vintage: Pultec EQP-1A*. 2014. Disponível em: <https://www.musictech-.net/reviews/pultec-eqp-1a/>. Acesso em: 31 out 2019.

PRECISION, A. *APx Waveform Computes Utility*. 2013. Acesso em: 10 nov 2019. Disponível em: <https://www.ap.com/technical-library/apx-waveform-computes-utility/>.

REFERENCE, C. *std::unique_ptr*. 2019a. Disponível em: <https://en.cppreference.com/w-/cpp/memory/unique_ptr>. Acesso em: 29 out 2019.

REFERENCE, C. *placeholder type specifiers (since C++11)*. 2019b. Disponível em: <https://en.cppreference.com/w/cpp/language/auto>. Acesso em: 29 out 2019.

REISS, J. D.; MCPHERSON, A. *Audio Effects: Theory, Implementation and Application*. CRC Press, 2014. Disponível em: <https://books.google.com.br/books?id=z5DaBAAAQBAJ>.

REVERB. *DeArmond Tremolo Control*. 2019. Disponível em: <https://reverb.com/p-/dearmond-tremolo-control>. Acesso em: 16 out 2019.

SMITH, S. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Pub., 1997. ISBN 9780966017632. Disponível em: <https://books.google.com.br-/books?id=rp2VQgAACAAJ>.

SOCIETY, A. E. *An Audio Timeline*. 2014. Disponível em: <http://www.aes.org/aeshc-/docs/audio.history.timeline.html>. Acesso em: 11 out 2019.

SOUZA, R. da S.; LIMA, W. A.; SOUZA, G. da S. Os desafios para as operações de importação no brasil: Um estudo de caso de uma empresa importadora da região do sul de minas gerais. 2015. Disponível em: <https://www.aedb.br/seget/arquivos/artigos15/9122228.pdf>. Acesso em: 13 out 2019.

STUDIO, L. M. *LMMS*. 2019. Disponível em: <https://lmms.io>. Acesso em: 13 out 2019.

SWEETWATER. *Eventide H9000 Multi-effects Processor*. 2017. Disponível em: <https:/-/www.sweetwater.com/store/detail/H9000–eventide-h9000-multi-effects-processor>. Acesso em: 06 out 2019.

THÉBERGE, P. *Any Sound You Can Imagine: Making Music/Consuming Technology*. Wesleyan University Press, 1997. (Any Sound You Can Imagine: Making Music/consuming Technology). ISBN 9780819563095. Disponível em: <https://books.google.com.br-/books?id=jBkT0NXxriIC>.

TUTORIALS, E. *Amplifier Distortion in Transistor Amplifiers*. 2008. Disponível em: <https://www.electronics-tutorials.ws/amplifier/amp_4.html>. Acesso em: 02 nov 2019.

WAVE, I. *SRC Comparisons*. 2019. Disponível em: <http://src.infinitewave.ca>. Acesso em: 12 out 2019.

WEAREROLI. *The JUCE cross-platform C++ framework*. 2007. Disponível em: <https://github.com/WeAreROLI/JUCE>. Acesso em: 31 ago 2019.

WEIR, W. *How Humans Conquered Echo*. 2012. Disponível em: <https://www.theatlantic-.com/entertainment/archive/2012/06/how-humans-conquered-echo/258557/>. Acesso em: 16 out 2019.

WORTHPOINT. *EVENTIDE H3000 ULTRAHARMONIZER. KITCHEN SINK+ UPGRADES REFURB, 777 PRESETS*. 2016. Disponível em: <https://www.worthpoint.com/worthopedia-/eventide-h3000-ultraharmonizer-1814327133>. Acesso em: 06 out 2019.