**UNIVERSIDADE DO ESTADO DA BAHIA**
**DEPARTAMENTO DE CIÊNCIAS EXATAS E DA TERRA**
**COLEGIADO DE SISTEMAS DE INFORMAÇÃO**

# ISOUNDFX: BIBLIOTECA DE EFEITOS SONOROS *OPEN SOURCE* PARA PLATAFORMA IOS

Pedro Matos Motta

Salvador
2012

**UNIVERSIDADE DO ESTADO DA BAHIA**
**DEPARTAMENTO DE CIÊNCIAS EXATAS E DA TERRA**
**COLEGIADO DE SISTEMAS DE INFORMAÇÃO**

# ISOUNDFX: BIBLIOTECA DE EFEITOS SONOROS *OPEN SOURCE* PARA PLATAFORMA IOS

Pedro Matos Motta

Monografia apresentada à Banca Examinadora como exigência parcial para obtenção do título de bacharel em Sistemas de Informação pela Universidade do Estado da Bahia.

Orientador:
Eduardo Manuel de Freitas Jorge

Salvador
2012

**PEDRO MATOS MOTTA**

# iSOUNDFX: Biblioteca de Efeitos Sonoros *Open Source* para Plataforma iOS

Monografia apresentada à Banca Examinadora como exigência parcial para obtenção do título de bacharel em Sistemas de Informação pela Universidade do Estado da Bahia.
Orientador. Eduardo Manuel de Freitas Jorge

BANCA EXAMINADORA

---

**Prof. Eduardo Manuel de Freitas Jorge**
Doutor em Difusão do Conhecimento
Universidade do Estado da Bahia

---

**Prof. Alexandre Rafael Lenz**
Doutorando em Ciência da Computação
Universidade do Estado da Bahia

---

**Prof. Uedson Santos Reis**
Mestre em Modelagem Computacional
Serviço Nacional de Aprendizagem Industrial

# *Agradecimentos*

Agradeço aos meus pais pela força e apoio ao longo da vida. Especialmente a minha mãe, por ter batalhado pelas minhas vitórias, e pelo exemplo de pessoa que ela foi e sempre será.

Agradeço também a minha namorada Mariana, por ter me incentivado e me proporcionado muitos momentos de alegria.

Agradeço aos meus amigos que me ajudaram de alguma forma em minha formação: Marcos, Felipe, Raul, Shankar, Leonardo, Osias, Nicole, Luciana, Matheus, Jussi, Fabíola, Weslley, Vitor e meus colegas de sala.

Agradeço aos meus colegas do trabalho também: Ronaldo, por ter corrigido inúmeras vezes o meu trabalho, Vanessa, Flávio, Igor, Gustavo e Eduardo Jorge por ter sido chefe, professor e ter orientado o meu trabalho ao mesmo tempo.

# *Resumo*

Os dispositivos móveis mais atuais possibilitaram o processamento de efeitos sonoros em tempo real. Através de um *smartphone* é possível ligar uma guitarra e ouvir a sonoridade semelhante aos efeitos produzidos pelas pedaleiras e amplificadores. Porém, para a criação de programas nesse contexto, há pouco reuso de componentes gerando uma menor produtividade para concepção de aplicações de efeitos sonoros. Este trabalho propõe a criação de uma biblioteca de efeitos sonoros *Open Source* denominada de iSoundFx para plataforma iOS. Esta biblioteca reúne algoritmos de efeitos sonoros e o uso de recursos de *hardware*, permitindo que aplicações sejam desenvolvidas para *smartphones* e *tablets* em um menor espaço de tempo e por desenvolvedores que não necessitam de conhecimentos profundos na área de efeitos sonoros na plataforma iOS. Para a validação deste trabalho foi construída uma aplicação de exemplo em que os efeitos criados foram testados e analisados.

**Palavras-chave**: Efeitos Sonoros, Dispositivos Móveis, iOS, *Open Source*

# *Abstract*

Modern handheld or mobile devices have overcome the technological limitations which prevented them from processing sound effects in real time. Through smartphone applications, for example, it is possible to plug an electric guitar to a cell phone enabling their user to listen to sound effects similar to the ones produced by pedalboards or amplifiers. However, in order to design programs in this context, there is little reuse of components generating less productivity for conceiving sound effect applications. The present work proposes the creation of an Open Source sound effect library called iSoundFx for iOS platform. This works aims at gathering at this library sound algorithms combined with the use of hardware resources enabling the development of applications for smartphones and tablets in a shorter time lapse which may be conducted by developers not requiring from them deep knowledge in the field. For the validation of this work was built on a sample application that created effects were tested and analyzed.

**Keywords**: Sound Effects, Mobile Devices, iOS, Open Source

# *Lista de Figuras*

# *Lista de Algoritmos*

# *Lista de Tabelas*

# *Sumário*

# 1 Introdução

O mercado de aplicações para dispositivos móveis dispõe de um número significativo de programas para fins musicais. Há uma busca crescente por ferramentas que auxiliem na composição musical, principalmente pela praticidade de carregar um *smartphone* ou *tablet* ao invés de caixas de som, pedais de efeitos, *racks*[1], dentre outros equipamentos de som. Não é mais necessário estar em um estúdio de gravação para ensaiar ou compor uma melodia (KRUEGER, 2008).

A crescente demanda por aplicativos musicais tem incentivado os desenvolvedores. Porém, ao criar programas de efeitos sonoros para dispositivos móveis, existe uma dificuldade inerente ao tema, já que, é necessário conhecimento em algoritmos de processamento de áudio. Além disso, no ambiente de dispositivos móveis deve haver uma preocupação maior que nos computadores convencionais, pois existem limitações quanto ao poder de processamento e capacidade de memória, dificultando ainda mais a construção desses aplicativos.

Diante desse contexto situam-se dois principais grupos de aplicações musicais definidos pelo tipo de algoritmo utilizado: os modificadores de áudio e os sintetizadores. As aplicações denominadas sintetizadoras são responsáveis por produzir um som artificial. Como por exemplo, os *softwares* que simulam instrumentos virtuais, tais como: bateria, flauta e violão. Já as modificadoras de áudio, ou efeitos sonoros, promovem uma alteração do som original, captado de fontes externas[2], ou de um arquivo. Têm-se os pedais de guitarra virtuais e os modificadores de tonalidade de voz como exemplo (APPLE, 2012b).

Segundo os dados da Apple (2012a), as aplicações que fazem parte do grupo de efeitos sonoros possuem o código fonte fechado, ou seja, dificulta que outros programadores possam utilizar os algoritmos para a construção de outros programas, aumentando assim o esforço e o tempo de desenvolvimento. Portanto a existência de código *Open Source* ajudaria que desenvolvedores, com pouca experiência, desenvolvessem aplicações neste domínio. Até o presente

[1] *Rack* é um equipamento de som semelhante a uma estante.

[2] Fontes externas podem ser microfones, guitarras, baixos entre outros.

momento, não foram encontradas bibliotecas de efeitos sonoros para plataforma iOS[3], apenas a especulação de um futuro lançamento do *framework The Amazing Audio Engine*[4].

Diante disso, o presente trabalho tem como objetivo criar uma biblioteca de efeitos sonoros *Open Source* para plataforma iOS, denominada iSoundFX. Essa biblioteca tem como principal requisito o funcionamento em tempo real (as amostras de áudio são capturadas, processadas pelos efeitos, e, em seguida, executadas). Os efeitos embarcados no iSoundFX são: *Delay*, Distorção, *Fuzz*, *Overdrive* e *Pitch*. Um módulo de captura de áudio também foi desenvolvido. Para validar a biblioteca foi construído um aplicativo de exemplo e posteriormente testados os efeitos através da análise dos espectros de áudio.

A relevância esperada com o iSoundFX é o reaproveitamento de código, já que contempla rotinas de utilização da API[5] nativa de áudio, que estaria repetido nas diversas aplicações; e a redução da complexidade, visto que o desenvolvedor precisa se preocupar apenas com a lógica da sua aplicação.

O trabalho está organizado da seguinte forma: no capítulo 2 serão abordados os fundamentos do som, as técnicas de processamento de sinais e os efeitos sonoros. Ainda neste capítulo serão apresentados alguns conceitos da plataforma iOS, bem como a arquitetura, o funcionamento do *framework*[6] de áudio e os conectores para dispositivos iOS. No capítulo 3 será especificada a biblioteca iSoundFX, demonstrado o desenvolvimento do aplicativo de exemplo, e ainda, os resultados dos testes. Por fim, no capítulo 4, serão apresentadas as considerações finais e os trabalhos futuros.

[3] *iPhone Operational System* é o Sistema Operacional para dispositivos móveis da *Apple*.

[4] *The Amazing Audio Engine* é um *framework* para manipulação de áudio no iOS. A data de lançamento pode ser vista no *site* <http://theamazingaudioengine.com/>.

[5] *Application Programming Interface* é um conjunto de funções de um aplicativo, onde os detalhes de implementação são abstraídos.

[6] *Framework* é um conjunto de classes que colaboram para realizar uma responsabilidade para um domínio de um subsistema da aplicação.

# 2 *Aspectos dos Efeitos Sonoros e Plataforma iOS*

Este capítulo está dividido em duas partes principais: os efeitos sonoros e a plataforma iOS. Primeiramente, serão abordados exemplos de efeitos sonoros, mas, para melhor compreensão, é necessário permear por duas áreas de conhecimento: som e processamento de sinais. Os conceitos e as propriedades do som serão vistos na seção 2.1. Em seguida, na seção 2.2 serão demonstradas as técnicas utilizadas no processamento de sinais. E, para finalizar a primeira parte, será explicado o funcionamento dos efeitos sonoros.

Na seção 2.4 está a segunda parte do capítulo, onde serão apresentadas a plataforma iOS e as suas especificidades, tais como: a arquitetura do sistema operacional, o funcionamento de áudio e os conectores para os dispositivos iOS.

## 2.1 Propriedades do Som

O som pode ser definido como uma forma de propagação em meios sólidos, líquidos ou gasosos de uma compressão mecânica ou onda longitudinal (FERREIRA, 2006). Quando há uma massa de ar deslocada com velocidade suficiente, esta produzirá um som. Fisicamente o som possui as seguintes propriedades: duração, intensidade, altura e timbre.

A duração compreende o tempo durante o qual é produzido (LACERDA, 1966). Na Figura 2.1 pode se observar que as ondas A, B e C possuem, quando comparadas, a duração curta, média e longa respectivamente.

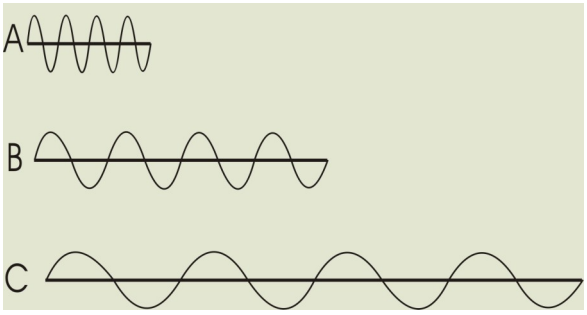


Figura 2.1: Duração das ondas sonoras (OSCROVE, 2012)

Segundo Ferreira (2006), a intensidade, também chamada de amplitude, é definida como a distância em formato de onda da origem até a parte mais elevada na direção vertical. A unidade utilizada para medir a intensidade é denominada decibel (dB). Geralmente a intensidade é erroneamente confundida com o conceito de altura. Por exemplo, quando alguém se refere a "aumentar o som", a intenção do autor é deixar um som com uma maior intensidade, mas em termos de propriedades de som, a frase significa alterar a altura relativa à frequência do som. A Figura 2.2 ilustra diferentes amplitudes de onda, sendo: "A"a onda mais fraca, "B"a onda de média intensidade e "C"a onda mais forte.

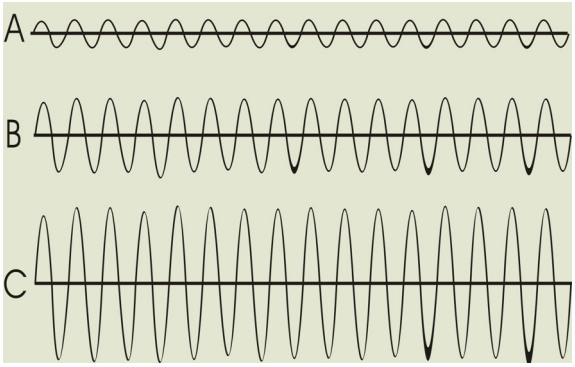


Figura 2.2: Intensidade do som. Adaptada de Oscrove (2012)

Define-se por altura a propriedade do som de ser mais grave ou mais agudo. Este conceito está diretamente relacionado com a frequência, que é o número de oscilações que ocorrem por unidade de tempo (FERREIRA, 2006). Na Figura 2.3 pode ser visto o diferente número de oscilações entre as ondas "A", "B"e "C"para um mesmo período.

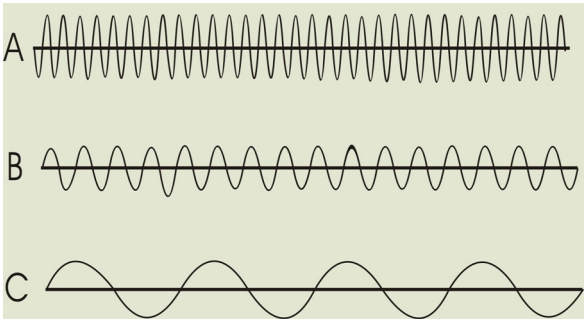


Figura 2.3: Frequência das ondas sonoras (OSCROVE, 2012)

Quando o período da onda é de um segundo, então cada ciclo possui um *Hertz* (Hz). As frequências audíveis para o ouvido humano variam, aproximadamente, da faixa de 20Hz até 20kHz, ou, de 20 à 20.000 ciclos por segundo (VALLE, 2006). As frequências podem ser classificadas em três grupos:

- Graves — entre 20Hz até 200Hz;
- Médios — entre 200Hz até 6kHz;
- Agudos — entre 6khz até 20kHz;

O timbre é tido como a qualidade ou identidade do som. A partir dele é possível reconhecer o instrumento que deu origem ao som (LACERDA, 1966). Por exemplo, um violão e um piano podem alcançar as mesmas frequências, porém o timbre será o que vai diferenciar um som característico de cada instrumento. Na Tabela 2.1 podem ser vistos os diferentes timbres dos instrumentos.

Tabela 2.1: Exemplo de formato de onda (KRUEGER, 2008)

| **Forma de onda** | **Timbre** | **Instrumento** |
|---|---|---|
| Senoidal | Suave, doce | Flauta, assovio |
| Dente de serra | Claro, brilhante | Violino, trompete |
| Retangular | Simples, "quente" | Clarinete, oboé |

## 2.2 Processamento de Sinais

Processamento Digital de Sinais (do inglês *Digital Signal Processing* - DSP) é o conjunto de técnicas utilizadas para análise ou modificação de sinais. Em grande parte dos casos, esses

sinais se originam de dados sensoriais do mundo real: vibrações sísmicas, imagens, ondas sonoras, etc.

DSP utiliza a matemática, algoritmos e técnicas para manipular os sinais após a conversão do sinal analógico para o formato digital. Alguns exemplos de manipulação de sinais: melhoria de imagens, reconhecimento e sintetização de voz, compressão de dados para o armazenamento ou transporte, etc. (SMITH, 2012).

Nesta seção é apresentado o processo de conversão de sinais e algumas das técnicas de DSP comumente utilizadas bem como: convolução, filtros digitais, modulação e a transformada rápida de Fourier (FFT).

### 2.2.1 Conversão Digital de Sinal e Teorema de Nyquist

No processo de digitalização denominado amostragem de sinal, ou *Analog to Digital Converter* (ADC), os dados analógicos de um sinal contínuo são codificados em um sinal discreto. Cada valor desse sinal discreto, ou digital, corresponde a uma amostra em função do tempo. Define-se como taxa de amostragem a quantidade de amostras por unidade de tempo, e, cada valor dessa amostra corresponde à amplitude do sinal (PARK, 2010). Dois exemplos de sinais podem ser observados na Figura 2.4, onde, a figura da esquerda apresenta um sinal de onda contínuo e a da direita um sinal discreto.

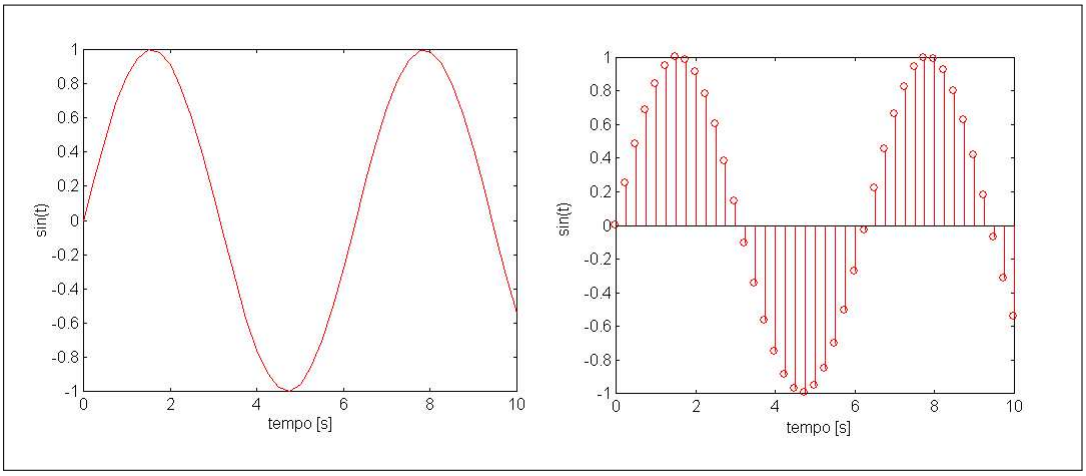


Figura 2.4: Sinais Contínuo e Discreto. Adaptada de Lemos (2012)

Nos CDs, as faixas de áudio possuem uma taxa de amostragem de 44.100Hz, ou seja, 44100 amostras de por segundo com a quantização de 16 bits cada uma. O valor da taxa de amostragem foi calculado com base no Teorema de Nyquist:

$$f_s = 2 * f_{max} \tag{2.1}$$

Sendo,

$$f_s = taxa\ de\ amostragem \tag{2.2}$$

$$f_{max} = frequência\ máxima \tag{2.3}$$

De acordo com as equações anteriores e a frequência máxima audível para os seres humanos, aproximadamente 20kHz, a taxa de amostragem deveria ser de 40kHz, porém por convenção das gravadoras, estipulou-se o valor de 44.100Hz.

### 2.2.2 Convolução

Convolução pode ser definida como uma simples operação matemática de adição ou integração. Por exemplo, quando é feita uma operação de adição, dois números são somados produzindo como resultado um terceiro número. Já no processo de convolução, ao invés de dois números, dois sinais são somados produzindo um terceiro sinal. A operação de convolução, além de ser utilizada em DSP, é comum em áreas como probabilidade e estatística (SMITH, 2012).

### 2.2.3 Transformada Rápida de Fourier

A representação do sinal após o processo de conversão ADC normalmente é feita no domínio do tempo: os dados estão dispostos em função tempo x amplitude. Existe outra forma de representatividade do sinal chamada domínio da frequência, onde os dados estão dispostos nos valores frequência x amplitude. O domínio da frequência, como o próprio nome sugere, permite que os dados sejam analisados sob o aspecto da frequência (SMITH, 2012).

A Transformada Discreta de Fourier, do inglês *Discrete Fourier Transform* (DFT), é uma técnica utilizada para a conversão do sinal discreto do domínio tempo para o domínio da frequência. Uma propriedade da DFT é poder fazer a conversão desse sinal no sentido inverso.

Na computação um algoritmo utilizado para realizar a técnica matemática da DFT é a Transformada Rápida de Fourier (*Fast Fourier Transformer* - FFT). Através desse algoritmo é possível realizar a conversão entre domínios com um menor custo computacional (PARK, 2010). O processo de convolução também é aplicável em conjunto com a FFT, produzindo um algoritmo mais rápido denominado convolução de alta velocidade (SMITH, 2012).

### 2.2.4 Filtros Digitais

Um sinal de áudio muitas vezes apresenta algum ruído indesejado, o som está grave, ou, o som está muito agudo. O processo de filtragem de sinais digitais surge com o intuito de eliminar as frequências indesejadas, chamado separação do sinal, e amplificar uma determinada frequência, denominado restauração do sinal (SMITH, 2012). Os filtros digitais podem ser classificados de acordo com os objetivos:

- Passa-baixas: filtro utilizado para eliminar as frequências que estão acima de uma frequência determinada permitindo apenas a passagem das frequências baixas. Este valor é denominado como frequência de corte;
- Passa-altas: o objetivo desse filtro é fazer o contrário do filtro de passa-baixa: atenuar as frequências que estão abaixo da frequência de corte;
- Passa-faixa: duas frequências de corte, uma alta e uma baixa, são definidas permitindo apenas a passagem das frequências que estão entre elas;
- Rejeita-faixa: é o oposto do filtro passa-faixa, duas frequências de corte são definidas e os valores que estão entre elas são atenuados.

Na Figura 2.5 é exibida graficamente a filtragem dos sinais no domínio da frequência, as faixas selecionadas representam as frequências que farão parte do sinal após o filtro e as que estão fora dela são as frequências que serão atenuadas.

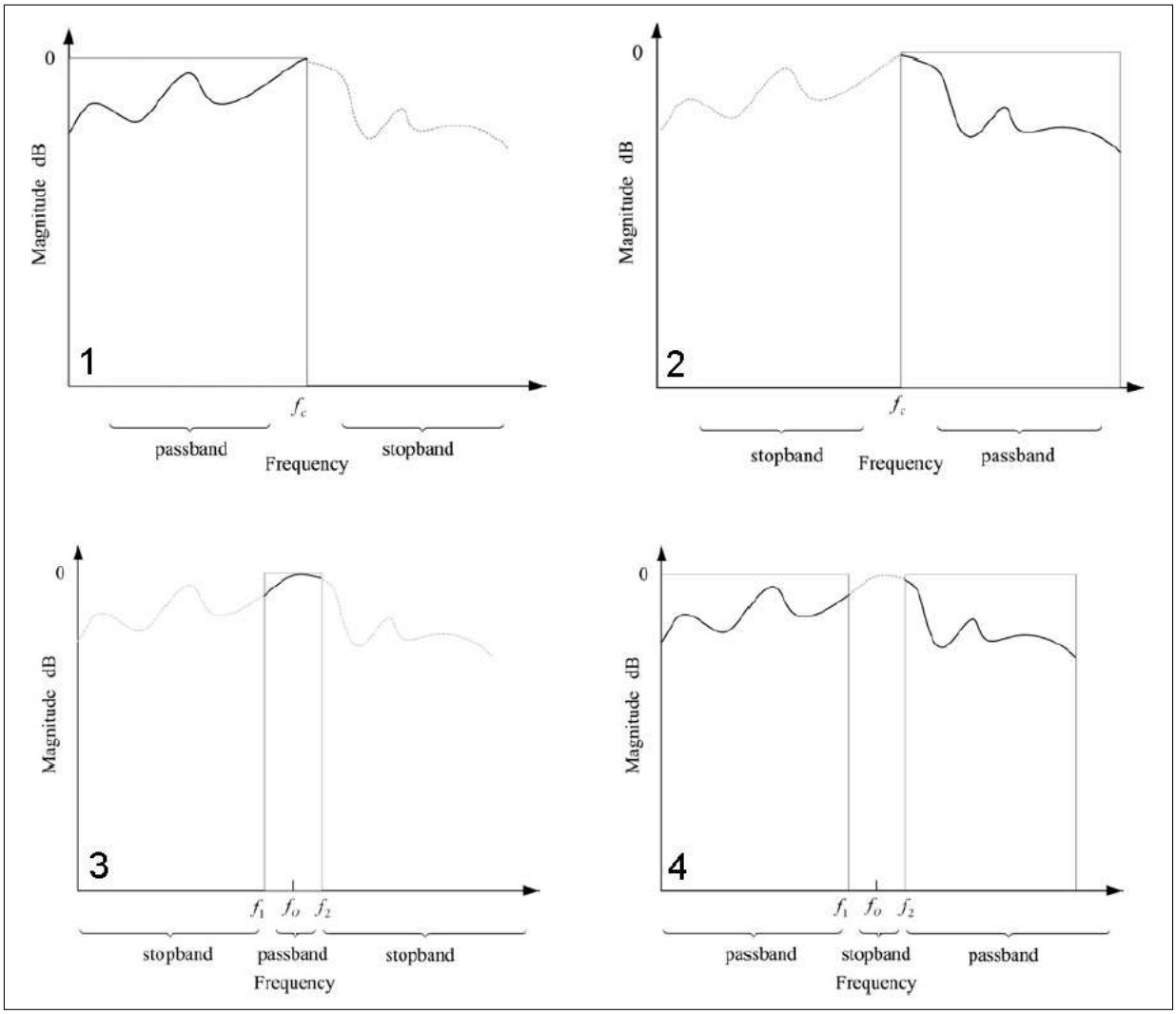


Figura 2.5: Filtros digitais: (1) Passa-baixa; (2) Passa-alta; (3) Passa-faixa; (4) Rejeita-faixa. Adaptado de Park (2010)

### 2.2.5 Modulação

Uma técnica largamente utilizada no processamento digital de sinal é a modulação. Segundo Park (2010), modulação é definida como uma técnica de alteração da amplitude, frequência ou da fase de um sinal.

No processo de alteração de amplitude é aplicado um sinal de baixa frequência denominado oscilador de baixa frequência (*Low Frequency Oscillator* - LFO) sobre outro sinal resultando em um sinal com amplitude modificada. O LFO é importante para a obtenção de alguns efeitos sonoros.

## 2.3 Efeitos Sonoros Digitais

Os efeitos digitais são assim denominados por serem criados a partir dos conceitos aplicados das técnicas de DSP. O sinal de áudio pode sofrer o processo de modificação da altura, do timbre, da intensidade ou até mesmo da duração, ou seja, os efeitos são responsáveis por alterar as propriedades do som, vistas na seção 2.1, trazendo como resultado um sinal de áudio diferente do original.

Estes efeitos são popularmente conhecidos através dos acessórios indispensáveis para guitarristas e baixistas profissionais: os pedais e as pedaleiras de efeitos. No caso das pedaleiras, há um conjunto de efeitos que podem ser utilizados individualmente ou simultaneamente, enquanto os pedais geralmente possuem apenas um único efeito. Mas assim como as pedaleiras é possível que os pedais funcionem conjunto, ligando os pedais em série, formando uma cadeia de efeitos.

A ordem em que estão dispostos os pedais de efeito nesta cadeia pode alterar a resultado final de áudio. Segundo Boss (2012) não há uma ordem correta ou errada da posição dos pedais, mas sugere que eles estejam dispostos de acordo com a Figura 2.6.

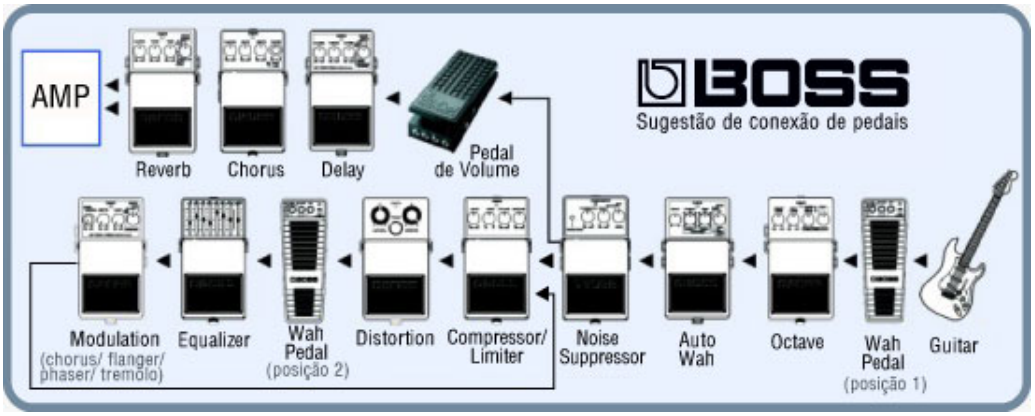


Figura 2.6: A ordem dos pedais (BOSS, 2012)

Segundo Reiman (2012), os efeitos podem ser categorizados em:

- Variação de Tempo — *Delay*, *Reverb*, *Flanger*, *Phaser* e *Chorus*;
- Alteração do Formato de Onda — Distorção;
- Modulação de Amplitude — *Tremolo*, Compressor e *Noise Gate*;
- Variação de Frequência — *Pitch*, *Vibrato*, Equalizador;

Ainda nesta seção será apresentado o princípio de funcionamento de alguns dos efeitos encontrado nas categorias previamente listadas.

O *Delay*, também conhecido como *Echo*, é um efeito de atraso do som original. É a simulação dos choques das ondas sonoras contra um corpo rígido, produzindo a repetição com atraso do som que o originou. A intensidade do som vai se perdendo a cada repetição. O efeito de *Delay* é a base do funcionamento de outros efeitos como *Flanger*, *Phaser*, *Chorus*, *Reverb* conforme a Figura 2.7 (MENDES; GOULART, 2009).

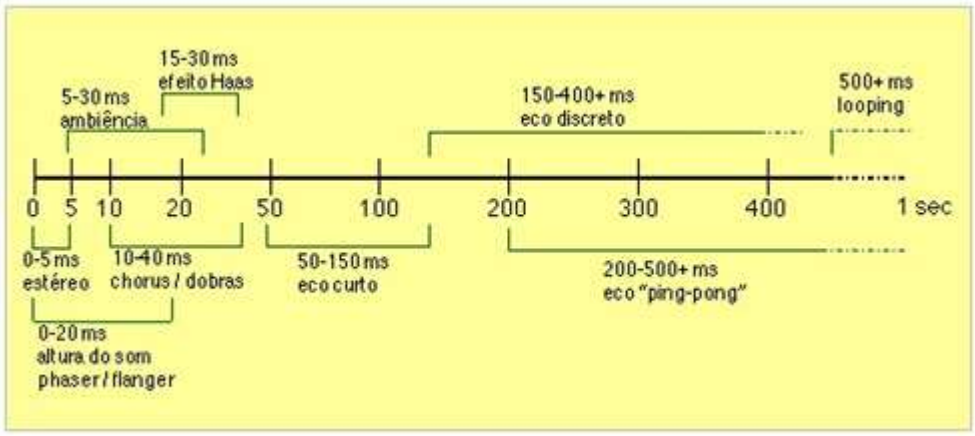


Figura 2.7: Tempos de *Delay* (IAZZETTA, 2012)

Nos pedais as regulagens deste efeito são as de:

- *Feedback* — quantidade de repetições;
- Tempo — tempo entre o som original e as repetições;
- *Level* — intensidade da repetição.

O efeito *Reverb* é semelhante ao efeito *Delay*, o som original reflete em vários corpos rígidos de tamanhos, formatos e densidades diferentes. Como pode ser visto na Figura 2.8: uma fonte sonora, dentro de uma sala ampla, faz com que o áudio seja refletido pelas paredes até chegar ao ouvinte, fazendo com que este escute o som ressoar por mais tempo que em um quarto com dimensões menores (ANHAIA, 2012).

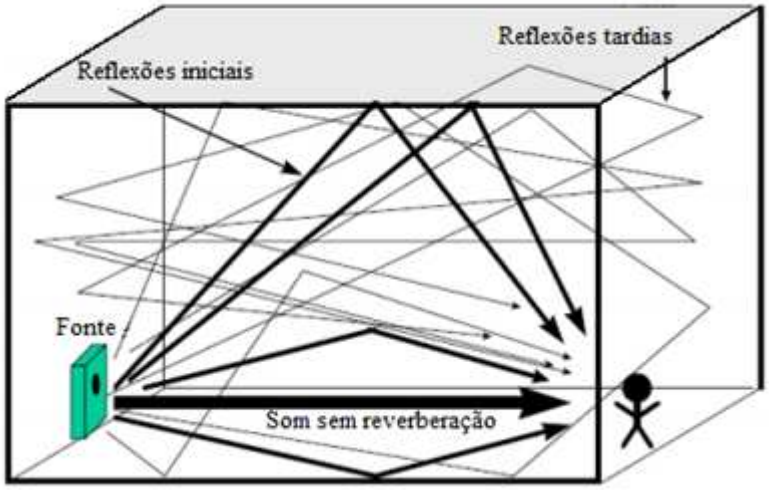


Figura 2.8: Reverberação em ambientes grandes. Traduzido de Mendes e Goulart (2009)

Este é um tipo de efeito que possui um grau de complexidade elevado e um custo alto de processamento, pois é necessária a simulação da reflexão das ondas sonoras em vários corpos e com tempos diferentes. Os parâmetros do *Reverb* são:

- *Size*(tamanho) — controla o tamanho do ambiente;
- *Predelay* — regula tempo em que ocorrem as primeiras reflexões do som original;
- Densidade — define a quantidade de superfícies refletoras no ambiente;
- Difusão — controla o tempo de decaimento da intensidade do som.

*Phaser* é um efeito causado pela soma do som original com o mesmo som de amplitude e frequência idênticas, porém, com variação temporal de 1 a 10ms. O que resulta na interferência construtiva e destrutiva do sinal. As frequências que possuem o período diretamente relacionado ao tempo de atraso são atenuadas devido ao cancelamento de fase (IAZZETTA, 2012). Os controles do *Phaser* são:

- *Rate* — regula a velocidade de varredura do modulador;
- *Range* — controla a faixa varrida pelo modulador.

*Flanger* é um efeito produzido pela junção de dois sinais idênticos, porém um deles possui o atraso variável de 1 a 20ms. No efeito de *Flanger* o sinal é realimentado provocando o cancelamento de algumas frequências criando a sensação de decaimento e aumento em alguma delas. O *Flanger* tem o mesmo princípio de funcionamento do efeito *Phaser*, a diferença é

que no segundo as atenuações e o reforço das frequências ocorrem em intervalos regulares (MENDES; GOULART, 2009).

Os parâmetros do Flanger são:

- *Delay* — controla o tempo de atraso;
- *Feedback* — regula a quantidade do sinal processado que é realimentado no efeito;
- *Rate* — regula a velocidade de variação do modulador;
- *Depth* — define o *delay* mínimo e máximo;

O efeito *Chorus* é semelhante ao *Flanger*, mas com a temporalidade de 20 a 30ms. O *Chorus* dá a sensação de que o som original possui mais fontes sonoras do que o real, chamado de dobra (REIMAN, 2012). Os controles do *Chorus* são:

- *Rate* — controla a velocidade da modulação;
- *Depth* (Profundidade) — controla a intensidade da modulação.

O efeito de distorção pode ser definido como a amplificação do sinal até a sua saturação, dando a sensação de sujeira no som. O processo também é conhecido como *clipping*, onde parte do sinal original é perdido (MENDES; GOULART, 2009). Esse efeito pode ser divido em três subtipos:

- *Overdrive* — possui um baixo nível de saturação, deixando que a sonoridade seja bem limpa em relação às outras distorções;
- Distorção — ela possui um ganho maior que o *Overdrive* e consequentemente um nível de saturação mais elevado;
- *Fuzz* — é o efeito de distorção com maior nível de saturação deixando o som o mais "sujo"possível.

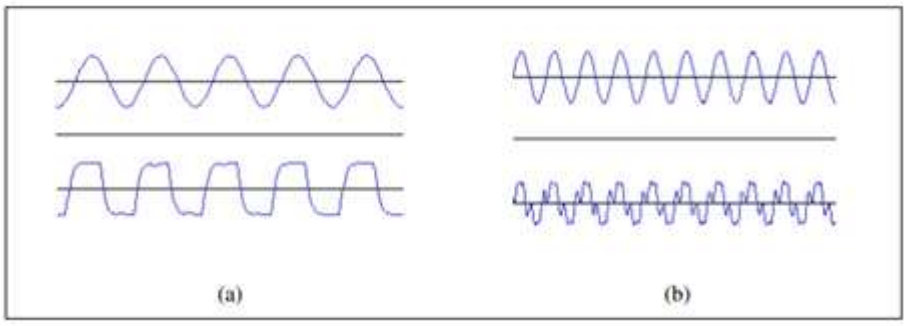


Figura 2.9: Comparativo entre espectros dos efeitos de *Overdrive* e Distorção: (a) Onda sonora original (sup.) e com efeito de *Overdrive* (inf.); (b) Onda sonora original (sup.) e com efeito de Distorção (inf.) (KRUEGER, 2008)

Quanto aos parâmetros de controle, os efeitos de distorção são divididos em:

- *Drive* ou *Gain* — controla a quantidade de saturação.
- Volume — controla a intensidade do sinal;
- *Tone* — regula o timbre do efeito, como o acréscimo de frequências altas.

*Pitch* é o efeito de alteração da tonalidade do som, ou seja, ele altera as frequências de forma que o som fique mais agudo ou mais grave. Normalmente este processo é realizado mudando o tempo do áudio, por exemplo, ao diminuir-se a velocidade ele se torna mais grave, ao se aumentá-la o tom se torna mais agudo. O *Pitch* possui como único parâmetro a transposição do tom (IAZZETTA, 2012).

O efeito *Tremolo* é obtido através da modulação da amplitude do sinal resultado na mudança periódica do volume. (APPLE, 2012c).

Os parâmetros do efeito de *Tremolo* geralmente são:

- *Depth* — controla a profundidade do modulador;
- *Rate* — regula a velocidade do modulador;
- *Wave* — define o formato de onda do modulador.

## 2.4 Plataforma iOS

Lançado em 2007, o iOS é um sistema operacional para dispositivos móveis da *Apple* baseado em *UNIX*. O seu uso é restrito aos produtos *Apple* tais como: *iPad*, *iPhone* e *iPod Touch*.

Inicialmente não era permitido que aplicativos de terceiros fossem executados nos seus dispositivos, só sendo liberado em 2008 após o lançamento do iOS *Software Development Kit* (SDK). Desde o seu lançamento, foram disponibilizadas diferentes versões do Sistema Operacional (SO). Até o presente momento, a versão atual é a 5.1.

As aplicações para o iOS são disponibilizadas na loja de aplicativos *Apple — App Store*. Este serviço possui cerca de quinhentos mil programas e mais de vinte e cinco bilhões de *downloads* no levantamento realizado pela *Apple* no início de 2012. Neste mesmo período, a empresa fechou o ano fiscal com a receita de treze bilhões de dólares, superando os concorrentes e fazendo com que os desenvolvedores fossem atraídos pela visível rentabilidade (FERREIRA, 2006).

A *Apple* disponibiliza um ambiente de desenvolvimento denominado *Xcode*, onde os programadores podem baixá-lo e instalá-lo gratuitamente. Através desta ferramenta é possível desenvolver aplicações tanto para o SO *Mac OS X*, quanto para o iOS, bem como a criação de telas, a simulação dos dispositivos iOS, a realização de testes e a integração com o iOS SDK. As aplicações desenvolvidas no *Xcode* são escritas nas linguagens *Objetive-C* e *C++* (PILONE; PILONE, 2011).

Para a criação de softwares no iOS é necessário entender o funcionamento da sua arquitetura que pode ser dividida em quatro camadas: *Cocoa Touch*, *Media*, *Core Services* e *Core OS*.

### 2.4.1 Arquitetura

A arquitetura do iOS está organizada conforme a Figura 2.10. A última camada representada pelo *Cocoa Touch* possui um nível de abstração mais alto, diferente do *Core OS* que está na camada mais baixa. Entende-se esta abstração como o encapsulamento de trechos de códigos mais complexos, diminuindo assim o esforço do desenvolvedor por reduzir a quantidade de código produzido. O mesmo não acontece na camada mais baixa, onde é necessária uma compreensão maior do funcionamento do sistema (APPLE, 2012b). Cabe ao desenvolvedor optar pelo uso da camada que se enquadra com a necessidade da aplicação:

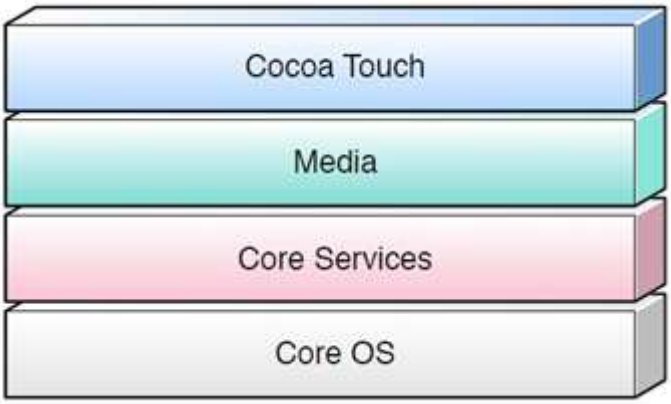


Figura 2.10: Camadas do iOS (APPLE, 2012b)

- *Cocoa Touch* — Provê acesso aos elementos de interface, eventos de toque, uso de recursos de acelerômetro, câmera e sistemas de localização e outros serviços de alto nível;
- *Media* (Serviços de Mídia) — Contém recursos de multimídia, bem como, gravação e mixagem de áudio, núcleo de animações e reprodução de vídeo e áudio;
- *Core Services* (Serviços do sistema) — Dá acesso a arquivos, banco de dados, preferências, livro de endereços entre outros serviços.
- *Core OS* (Núcleo do SO) — Possui recursos como segurança, gerenciamento de energia, entre outros.

### 2.4.2 API de Áudio

Na camada *Media* vista no modelo de arquitetura do iOS, há um módulo de suporte a áudio nativo denominado *Core Audio*. Este *framework*, escrito na linguagem *C*, permite a manipulação, gravação, mixagem e reprodução de áudio. Dentre os benefícios providos pelo *Core Audio* inclui-se o *framework Audio Units*, responsável pelo processamento digital de sinal de áudio.

O *Audio Units* possui uma comunicação direta ao *hardware* (Figura 2.11), provendo acesso simultâneo de E/S (entrada e saída) e com prioridade em tempo real, tornando possível a execução deste trabalho. O sistema operacional para dispositivos móveis da *Google* (*Android*) difere-se do iOS neste ponto, já que não possui suporte para aplicações sonoras de tempo real. Isso ocorre devido a limitações relacionadas à latência de áudio (MULTIMEDIA, 2012).

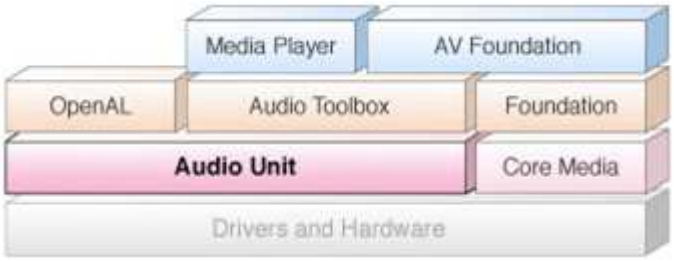


Figura 2.11: *Frameworks* de áudio no iOS (APPLE, 2012b)

### 2.4.3 Conectores para os dispositivos iOS

Existem conectores para os dispositivos móveis da *Apple*, vendidos separadamente, que possibilitam a conexão destes com uma guitarra ou até mesmo um microfone. Dois principais produtos concorrentes deste mercado são o *iRig*, fabricado pela *IK Multimedia*, e o *AmpKit LiNK*, produzido pelo fabricante *Peavey*.

O dispositivo *iRig* possui apenas a função de adaptador, ou seja, apenas conecta o aparelho iOS através de um plugue *P2* (3,5mm) com as entradas *P10* (cabos de guitarra; 6,35mm) e *P2* (para os fones de ouvido ou amplificador; 3,5mm). Já o dispositivo *AmpKit LiNK*, além de ter a mesma função do *iRig*, possui um *hardware* compensador de sinal, o que melhora sensivelmente a captura de áudio (KARASINSKI, 2012). Na Figura 2.12 pode ser observado o conector *iRig* à esquerda e o conector *AmpKit LiNK* à direita.


Figura 2.12: Conectores *iRig* e *AmpKit LiNK*. Adaptado de Karasinski (2012)

# 3 *Especificação, Desenvolvimento e Validação do iSoundFx*

A demanda por aplicações para dispositivos móveis tem crescido nos últimos tempos. E, para atendê-la, é necessário que o processo de desenvolvimento dos aplicativos não exceda o *time to market*[7]. Porém, é comum que ocorram atrasos neste processo, principalmente quando as aplicações envolvem áreas de conhecimento fora da esfera da equipe de desenvolvimento. Nesse contexto o reuso de código surge como uma alternativa para otimizar a construção desses aplicativos, evitando a utilização de trechos de códigos repetidos e diminuindo o tempo gasto com rotinas já criadas por outros programadores.

No domínio de aplicações musicais para a plataforma iOS, o reuso de código ainda é algo pouco utilizado devido a carências de fontes. Segundo a Apple (2012a), o *framework Audio Units* possibilita que os desenvolvedores utilizem funções que facilitam a manipulação de áudio na plataforma, mas também salienta que por ter um grau de abstração baixo, a quantidade de código necessário para realizar operações triviais ainda é alto.

Portanto, esse trabalho tem como objetivo desenvolver uma biblioteca de efeitos sonoros *Open Source* para a plataforma iOS com o intuito de diminuir os esforços dos desenvolvedores e agilizar o tempo de programação. Essa biblioteca foi denominada iSoundFX junção dos nomes (iOS + *Sound* + *Effects*).

Com o uso dessa biblioteca, os programadores podem desenvolver aplicativos que auxiliam os músicos na composição musical, jogos que envolvam o uso de efeitos sonoros, além de outros aplicativos na categoria musical ou de entretenimento.

Para que isso seja possível, o usuário terá que utilizar o microfone do dispositivo ou conectar um instrumento musical a um adaptador que seja compatível com a interface do mesmo. Feito isto, o áudio capturado é processado pelos algoritmos de efeitos do iSoundFX, que estará presente na aplicação, e em seguida executado na *interface* de *hardware* responsável por

[7] *Time to market é o tempo de lançamento de um produto no mercado.*

transmitir o áudio a um fone de ouvido, ou caixas de som plugadas no aparelho.

A Figura 3.1 demonstra o percurso que o fluxo de áudio segue do momento da captura até ser executado nos fones de ouvido: o som é produzido pela guitarra, passa por uma adaptador que conecta a guitarra a um dispositivo iOS, representado na imagem como fluxo de entrada. Após o processamento dos efeitos o som passa pelo adaptador e é escutado pelos fones de ouvido, simbolizado na figura como fluxo de saída.


Figura 3.1: Exemplo de funcionamento do ISoundFX

Nas próximas seções serão apresentados exemplos de aplicações de efeitos sonoros na loja *Apple Store*, e duas aplicações *Open Source* do domínio musical relevantes no desenvolvimento do trabalho. A partir das informações obtidas dos aplicativos analisados foi possível conduzir a elaboração dos requisitos funcionais e não funcionais utilizados na concepção deste trabalho.

Ainda neste capítulo serão abordados o diagrama de arquitetura do projeto, o diagrama de classes e os principais métodos da biblioteca. Logo após será demonstrado como foram desenvolvidos os efeitos sonoros. Para finalizar serão apresentados a aplicação de exemplo utilizando o iSoundFX e os resultados alcançados.

## 3.1 Aplicativos Disponíveis na *Apple Store*

Na *Apple Store* as aplicações são categorizadas de acordo com o seu grau de afinidade. As categorias podem ser: Jogos, Educação, Entreternimento, Livros, Músicas e outras. As

funcionalidades pertinentes a especificação do iSoundFX foram obtidas através da análise, na categoria musical, dos aplicativos que utilizam recursos de efeitos de áudio. Na Figura 3.2 podem ser observados alguns dos efeitos vistos nas aplicações *Amplitube* e *GarageBand*.


Figura 3.2: Aplicações *Amplitube* e *GarageBand*

A partir do estudo das cinco aplicações mais populares foi possível levantar informações tais como: os efeitos utilizados, a simultaneidade (quantidade de efeitos em conjunto) e o valor cobrado por aplicativo, conforme pode ser visto na Tabela 3.1. Pode-se observar que alguns efeitos vistos nesta tabela não foram apresentados na seção 2.3, pois estes não tiveram relevância no desenvolvimento deste trabalho.

Foram escolhidos os seguintes efeitos para fazer parte do iSoundFX: Distorção, *Overdrive*, *Fuzz*, *Tremolo*, *Delay* e *Pitch Shifter*. Constatou-se que estes efeitos fazem parte da maioria dos aplicativos avaliados.

Ainda com base nos dados da Tabela 3.1, observou-se também a quantidade máxima de efeitos que podem ser usados simultaneamente que, na média, obteve-se o valor 4,4. Acerca desse dado, foi definido que o iSoundFX deve suportar até quatro efeitos simultâneos.

Outro ponto observado nesta análise é que em quase todas as aplicações vistas, a ordem dos efeitos pode ser alterada, assim como a ligação em série dos pedais, vistos na seção 2.3. Este recurso também foi incluído na biblioteca.

Tabela 3.1: Principais efeitos das aplicações pagas da *Apple Store*

| | *GarageBand* | *AmpKit* | *AmpliTube* | *PocketAmp* | *Stompbox* | Total |
|---|---|---|---|---|---|---|
| *Chorus* | X | X | X | X | X | 5 |
| *Equalizer* | X | X | X | X | X | 5 |
| *Flanger* | X | X | X | X | X | 5 |
| *Delay / Echo* | X | X | X | X | X | 5 |
| *Overdrive* | X | X | X | X | | 4 |
| Distorção | X | X | | X | X | 4 |
| *Fuzz* | X | X | X | | X | 4 |
| *Noise Filter* | | X | X | X | X | 4 |
| *Tremolo / Vibrato* | X | X | | X | X | 4 |
| *Reverb* | X | X | | X | X | 4 |
| *Phaser* | | X | X | | X | 3 |
| *Compressor* | X | X | | | X | 3 |
| *Wah* | | X | X | | | 2 |
| *Envelope Filter* | | X | X | | | 2 |
| *Octave* | | X | X | | | 2 |
| *Pitch Shifter* | | | | | X | 1 |
| Efeitos Simultâneos | 4 | 6 | 3 | 3 | 6 | 4,4 |
| Preço | $4,99 | $19,99 | $19,99 | $4,99 | $19,99 | $13,99 |

## 3.2 Aplicações *SkypeFX* e *audioGraph*

A análise das aplicações *Open Source* foi fundamental no processo de desenvolvimento dos efeitos, levantamento dos requisitos, arquitetura do projeto e diagrama de classes (o *SkypeFX* e o *audioGraph*). Esses programas foram escolhidos, principalmente, devido aos recursos encontrados e pela falta de opções dada a quantidade limitada de *softwares* nesta área.

A aplicação *SkypeFX* é um projeto em *C#* que permite o uso de efeitos sonoros em tempo real para uma conversação em chat do *Skype*[8] (HEATH, 2012). Essa aplicação permitiu que o efeito de *Tremolo* fosse adicionado no iSoundFX realizando as alterações necessárias. A tela principal do programa pode ser visualizada na Figura 3.3.

[8] *Skype* é um programa para conversação pela *internet*.

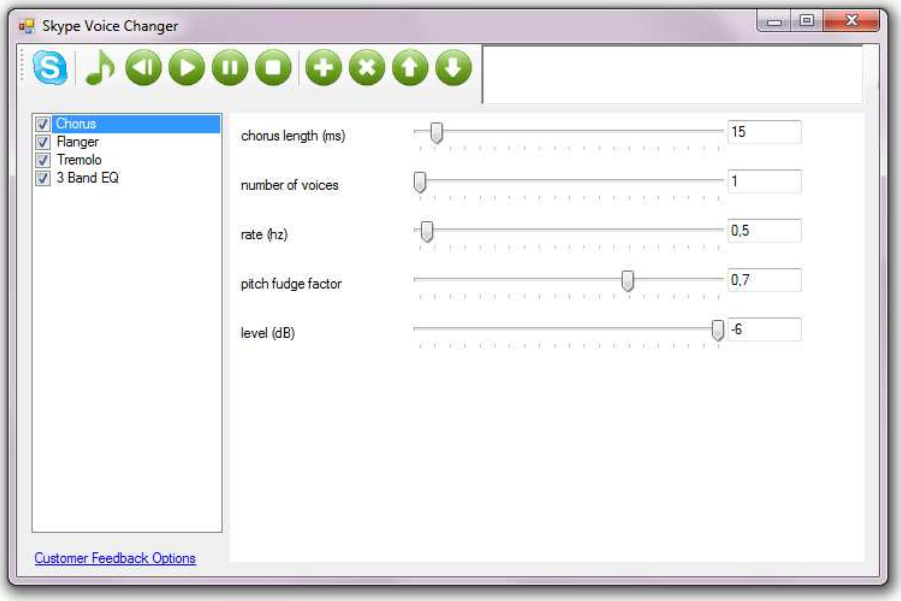


Figura 3.3: Programa *SkypeFX*

O *audioGraph* é um aplicativo de exemplo de utilização dos recursos de áudio na plataforma iOS (ZICARELLI, 2012). Entre outros recursos, essa aplicação inclui o uso de efeitos sonoros em tempo real e a captura e execução simultânea de áudio, os quais foram objetos de estudo para o processo de desenvolvimento do iSoundFX. O aplicativo pode ser visto na Figura 3.4.

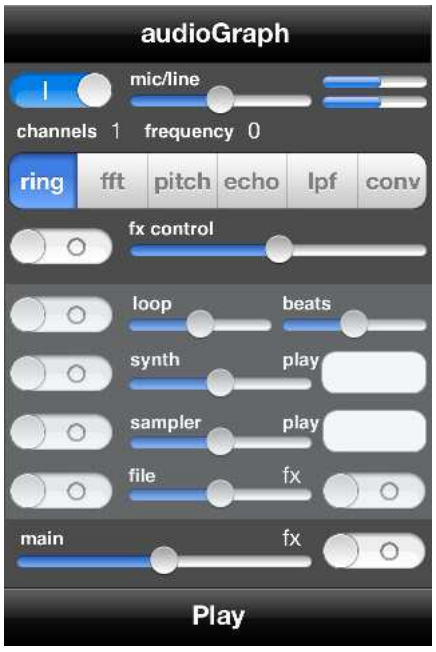


Figura 3.4: Aplicativo *audioGraph*

Uma das falhas encontradas no *SkypeFX* foi o acoplamento dos efeitos com a interface de usuário, com isso muitas vezes trechos de código se misturavam com as telas da aplicação,

diminuindo a legibilidade do código. Outro ponto considerado negativo foi a utilização de herança de classe em pontos não relevantes, como por exemplo, classe de funções matemáticas sendo herdadas por classes de efeitos musicais.

No portal onde está hospedado o projeto *SkypeFX*, não foram encontradas documentações acerca da arquitetura, requisitos ou diagrama de classes, dificultando o processo de identificação do funcionamento dos efeitos e da aplicação como um todo.

Apesar da documentação, o que facilitou bastante o entendimento do seu funcionamento, a aplicação *audioGraph* falhou por criar um acoplamento de código através da junção das funcionalidades em um único arquivo, ocasionando um desconforto visual na leitura do código.

Fazendo um comparativo entre este trabalho e as aplicações *SkypeFX* e *audioGraph* obteve-se a Tabela 3.2. Ressaltando que acoplamento de código e interface são aspectos negativos.

Tabela 3.2: Comparação entre as aplicações *SkypeFX*, *audioGraph* e a biblioteca iSoundFX

| Recurso | *SkypeFX* | *audioGraph* | iSoundFX |
|---|---|---|---|
| *Chorus* | X | | |
| *Compressor* | X | | |
| *Delay* | X | X | X |
| Distorção | | | X |
| *Equalizer* | X | | |
| *Flanger* | X | | |
| *Fuzz* | | | X |
| *Overdrive* | | | X |
| *Pitch* | X | X | X |
| *Ringer* | | X | |
| *Tremolo* | X | | X |
| Efeitos Simultâneos | X | | X |
| Código *Open Source* | X | X | X |
| Documentação | | X | X |
| **Acoplamento de código** | | X | |
| **Acoplamento de interface** | X | | |

## 3.3 Projeto Arquitetural e Classes Relevantes

Como visto na seção anterior, a análise das aplicações do mercado e *Open Source* contribuiram para o levantamento dos requisitos do ISoundFX. Para o desenvolvimento do projeto foram definidos os requisitos funcionais (RF) e não funcionais (RNF), que podem ser vistos na Tabela 3.3.

Tabela 3.3: Requisitos do iSoundFX

| Requisito | Descrição |
|---|---|
| RF1 | A biblioteca deve conter os efeitos de Distorção, *Overdrive*, *Fuzz*, *Tremolo*, *Delay* e *Pitch Shifter* |
| RF2 | Cada efeito deve ter os parâmetros de funcionamento ajustados individualmente |
| RF3 | A biblioteca deve dar suporte ao funcionamento de até quatro efeitos simultâneos |
| RF4 | Os efeitos, quando em conjunto, devem suportar a mudança da ordem de execução |
| RF5 | Os efeitos podem ser desativados, removidos ou adicionados à cadeia de efeitos |
| RF6 | Um módulo de captura e execução de áudio também deve ser incluso na biblioteca |
| RNF1 | A biblioteca deve funcionar em aparelhos com o Sistema iOS com versão 5.1 ou superior |
| RNF2 | O tempo de execução dos efeitos deve ser em tempo real |
| RNF3 | A arquitetura da biblioteca deve suportar à inclusão de novos efeitos |

O projeto arquitetural deste trabalho é ilustrado pela Figura 3.5: na camada (a) está a aplicação hospedeira, pelo fato de hospedar a biblioteca, que realizará os ajustes dos efeitos individualmente de acordo com o requisito RF2. A aplicação também controlará quando será iniciado o fluxo de captura de áudio e quando será interrompido.

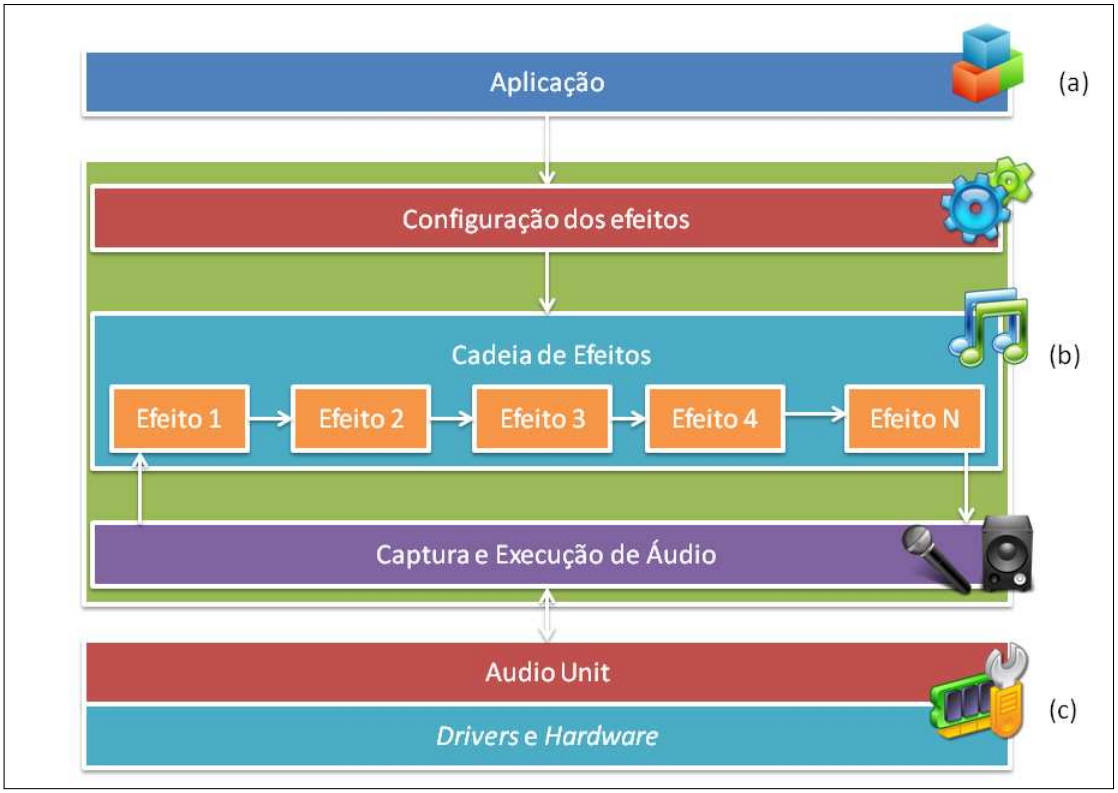


Figura 3.5: Arquitetura do iSoundFX: (a) Aplicação Hospedeira; (b) Biblioteca iSoundFX; (c) API Nativa.

Na camada inferior (c) está o *Audio Unit framework*, responsável por gerenciar os *Drivers*, *Hardware* de áudio no iOS, entre outras funcionalidades. A principal função desta camada é prover as amostras de áudio capturadas para a biblioteca e executar essas mesmas amostras após o processamento de áudio contemplando o requisito RF6.

Por último está a camada intermediária indicada por (b) que é a biblioteca iSoundFX, podendo ser subdividida em três módulos:

- Configurações dos efeitos — é por este módulo que a aplicação realizará os ajustes do sistema bem como adicionar, remover, desativar e ordenar os efeitos, contemplando os requisitos RF2, RF4 e RF5;
- Captura e Execução de Áudio — é por esse pacote que a biblioteca irá capturar as amostras de áudio, passar essas amostras para a cadeia de efeitos para ser processado e depois retornará a ela para ser executado conforme o requisito RF6;
- Cadeia de efeitos — os efeitos criados funcionarão em série, compartilhando as amostras de áudio entre si de forma unidirecional. Apesar do requisito RF3 indicar que apenas

quatro efeitos serão executados simultaneamente, o projeto também poderá dar suporte a N efeitos.

Após a definição dos requisitos e do projeto arquitetural foi possível obter um diagrama de classes que pudessem contemplar as funções e rotinas da biblioteca. O estudo dos códigos fonte dos projetos *Open Source* auxiliou na identificação das principais classes envolvidas no projeto.

A fim de reduzir os impactos na criação e adição de novos efeitos no iSoundFX optou-se por utilizar o padrão de projeto chamado *Strategy*. Esse padrão permitiu que os efeitos fossem adicionados ao projeto reduzindo o esforço do desenvolvedor.

A partir do diagrama de classes da Figura 3.6 serão apresentadas as principais classes envolvidas na elaboração deste trabalho.

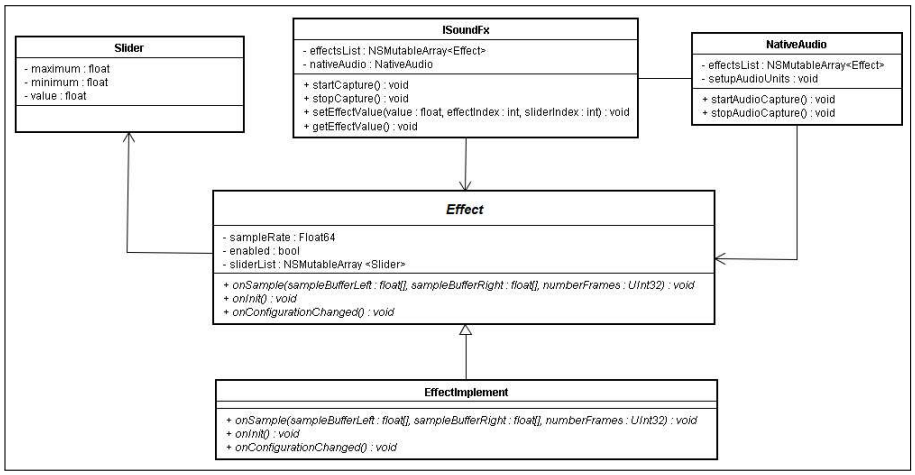


Figura 3.6: Diagrama de Classes do iSoundFX

***Effect*** — É a classe abstrata comum a todos os efeitos. Ela tem como principais atributos a quantidade de amostras (*sampleRate*), o estado de ativação e os parâmetros do efeito. Seus principais métodos são:

- *onSample*: que receberá as amostras de áudio;
- *onInit*: será chamado quando o efeito for iniciado;
- *onConfigurationChanged*: será chamado sempre que houver uma modificação nos parâmetros dos efeitos;

***EffectImplement*** — É a implementação da classe abstrata *Effect* (o nome da classe foi usado apenas ilustrar). Nessa classe apenas os métodos abstratos deverão ser sobescritos. As

alocações de memória e inicialização de atributos do efeito, bem como a inserção dos parâmetros deverão ser realizados no método *onInit*, e não no método nativo *init*, o qual não pode ser sobescrito. O método *onSample* será chamado sempre que houver novas amostras para serem processadas do áudio, recebendo por parâmetro a quantidade de amostras, *numberFrames*, e os dados de áudio esquerdo e direito *sampleBufferLeft* e *sampleBufferRight*.

***Slider*** — Responsável por armazenar as configurações dos efeitos bem como o valor máximo, mínimo e corrente: *maximum*, *minimum* e *current* respectivamente). Os métodos são apenas de *get* e *set*.

***NativeAudio*** — Essa é a classe que fará as configurações e inicialização do *Audio Units* para a captura e execução simultânea das amostras de áudio. Uma referência da cadeia de efeitos será passada para ela com o intuito de serem processados, nesse caso o *Strategy* proporcionou que não fosse necessário realizar modificações nessa classe ao incluir novos efeitos ao projeto, já que o único método utilizado por ela é o abstrato.

***ISoundFX*** — É a classe de acesso à biblioteca pela aplicação. Através dela será possível manipular as principais funções da biblioteca, como configurar, adicionar, remover, desativar e alterar a ordem dos efeitos. Também será possível controlar quando será iniciado e parado o processo de captura de áudio. Os principais métodos de utilização dela podem ser vistos na Tabela 3.4.

A linguagem utilizada no projeto, a *Objective-C*, possui algumas limitações em relação às outras linguagens de paradigma de Orientação a Objetos (OO), portanto algumas modificações foram realizadas para adaptá-la a necessidade do projeto.

Como o conceito de métodos abstratos não existe para essa linguagem, optou-se por solucionar o problema lançando uma exceção em tempo de execução caso o método não seja sobrescrito. Foi o caso dos métodos abstratos da classe *Effect* exemplificados no Algoritmo 3.1.

A falta de métodos do tipo *final*, ou seja, métodos que não podem ser sobrescritos pelas classes filhas foi uma limitação que não teve uma alternativa viável. Se o programador quiser sobrescrever os métodos ele não será impedido, mas fazendo isso ele comprometerá o funcionamento da classe. É o caso do método *init* na classe *EffectImplement*.

Tabela 3.4: Principais Métodos do iSoundFX

| Método | Descrição | Tipo de retorno |
|---|---|---|
| *startAudioCapture* | Inicia a captura de áudio | *void* |
| *stopAudioCapture* | Para a captura de áudio | *void* |
| *addEffect* | Adiciona o efeito na cadeia | *void* |
| *removeEffect* | Remove o efeito na cadeia | *void* |
| *exchangeEffectAtIndex* | Troca a ordem do efeito na cadeia | *void* |
| *enableEffect* | Habilita o efeito | *void* |
| *changeEffect* | Configura os parâmetros do efeito | *void* |
| *count* | Obtém a quantidade de efeitos na cadeia | *int* |
| *isEffectEnabled* | Verifica se o efeito está habilitado | *bool* |
| *getEffectMaxValue* | Obtém o valor máximo do parâmetro do efeito | *float* |
| *getEffectMinValue* | Obtém o valor mínimo do parâmetro do efeito | *float* |
| *getEffectValue* | Obtém o valor atual do parâmetro do efeito | *float* |

```
1  #pragma mark - Métodos Abstratos
2   // Chamado ao iniciar o efeito . Não implementar o método nativo init !
3  -(void)onInit{
4      [NSException raise:NSInternalInconsistencyException format:@"Você deve sobescrever o método
           onInit na classe filha !"];
5  }
6   // Chamado ao aplicar as configurações no efeito
7  -(void)onConfigurationChanged{
8      [NSException raise:NSInternalInconsistencyException format:@"Você deve sobescrever o método
           onConfigurationChanged na classe filha !"];
9  }
10  // Chamado ao receber novas amostras de áudio .
11 -(void)onStream: (UInt32) inNumberFrames sampleBufferLeft: (float*) sampleBufferLeft
       sampleBufferRight: (float*) sampleBufferRight{
12     [NSException raise:NSInternalInconsistencyException format:@"Você deve sobescrever o método
           onStream na classe filha !"];
13 }
```

Algoritmo 3.1: Adaptação de métodos abstratos na linguagem *Objective-C*

## 3.4 Desenvolvimento dos Efeitos

Neste seção será demonstrado os detalhes no desenvolvimento dos efeitos: Distorção, *Overdrive*, *Fuzz*, *Tremolo*, *Delay* e *Pitch Shifter*.

O primeiro passo para a criação de um efeito sonoro utilizando o iSoundFx é criar uma classe que implemente a classe abstrata *Effect*. Desta forma, o método *onSample* receberá por parâmetro as amostras de áudio dos canais esquerdo e direito e a quantidade dessas amostras. A taxa de amostragem foi quantizada em 16 bits e a 44.100Hz. Por padrão, as amostras são capturadas no tipo inteiro, mas, para facilitar as operações dos algoritmos realizou-se uma conversão para o tipo ponto flutuante com variação entre 0 e 1.

### 3.4.1 *Delay*

O efeito de Eco ou *Delay* como explicado na Seção 2.3 pode ser brevemente descrito como a repetição do sinal original até o decaimento total do som. Uma estrutura de dados que é utilizada para a construção deste efeito é a fila circular (Figura 3.7). Ela funciona da seguinte forma: os dados são inseridos na fila e são removidos de acordo com a ordem de inserção. Para o caso do efeito de *Delay* as amostras de áudio são inseridas na fila e conforme o tempo pré-definido as amostras são executadas e removidas da fila.

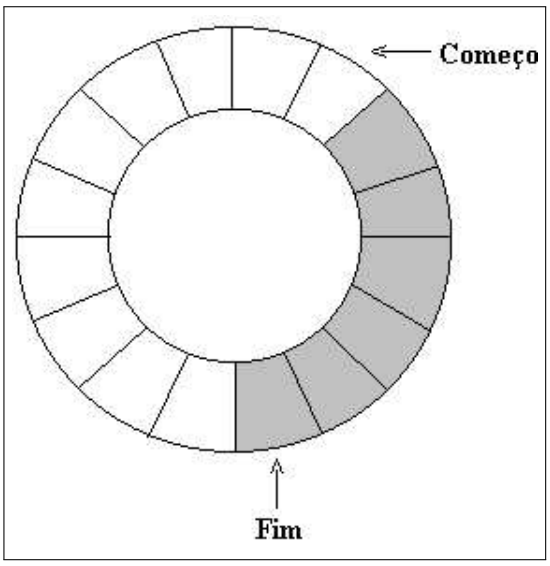


Figura 3.7: Fila Circular (PIMENTEL; CRISTINA, 2012)

Os parâmetros de configuração do efeito são:

- *Level* — Expressa a intensidade das repetições. Os valores estão entre 0 e 0,5;

- *Feedback* — É a quantidade de repetições. Os valores vão de 1 à 10;
- *Delay* — É o tempo entre as repetições. Os valores estão entre 0 até 5 segundos.

Para o cálculo do decaimento do *delay* é necessário o uso das através das fórmulas:

$$S = (feedback + 1) * feedback/2 \tag{3.1}$$

$$levelFactor = level/S \tag{3.2}$$

Sendo o *levelFactor* a variável que será multiplicada pela posição da amostra de áudio na fila circular, para obter o level de cada repetição do sinal. No Algoritmo 3.2 pode ser observado esse procedimento no último laço do "for".

```
1       // Iteração entre as amostras de áudio
2     for (int index = 0; index < inNumberFrames; index++) {
3         bufferLeft[bufferHead] = sampleBufferLeft[index];
4         bufferRight[bufferHead] = sampleBufferRight[index];
5         bufferHead++;
6         if(bufferHead >= BUFFER_LENGTH - 1){
7             bufferHead = 0;
8         }
9         sampleBufferRight[index] *=.5;
10        sampleBufferLeft[index] *=.5;
11    }
12
13      // Iteração da repetição do sinal
14    for(int i = 1, j = feedback; i < feedback; i++, j--){
15        bufferTail = (bufferHead - (delay + inNumberFrames) * i);
16        if( bufferTail < 0){
17            bufferTail = (BUFFER_LENGTH - 1) - abs(bufferTail);
18        }
19
20          // Iteração para o cálculo da intensidade do sinal .
21        for (int index = 0; index < inNumberFrames; index++) {
22            // Level da repetição do sinal
23            sampleBufferLeft[index] += bufferLeft[ bufferTail ] * (level * j);
24            sampleBufferRight[index] += bufferRight[ bufferTail ] * (level * j);
25            bufferTail ++;
26            if( bufferTail >= BUFFER_LENGTH - 1){
27                bufferTail = 0;
```

```
28                  }
29              }
30          }
```

Algoritmo 3.2: Código do efeito de *Delay*

### 3.4.2 *Overdrive*

O efeito de *Overdrive* foi proveniente de Chang (2011), o algoritmo foi adaptado para o funcionamento no iSoundFx a partir das fórmulas:

$$K = \frac{2 * f_{ganho}}{1 - f_{ganho}} \tag{3.3}$$

$$X = \frac{((1+K) * X)}{1 + K * abs(X)} \tag{3.4}$$

Sendo que $f_{ganho}$ corresponde ao nível de distorção e $X$ à amostra de áudio. O Algoritmo 3.3 expressa as fórmulas de distorção aplicadas.

```
1  (void)onStream : (UInt32) inNumberFrames sampleBufferLeft: ( float *) sampleBufferLeft
               sampleBufferRight: ( float *) sampleBufferRight{
2       float *x1, *x2;
3       x1 = sampleBufferLeft;
4       x2 = sampleBufferRight;
5       for( int i = 0; i < inNumberFrames; i++){
6               // Fórmula de ganho :
7           x1[i] = ((1 + k) * x1[i]) / ( 1 + k * abs(x1[i]) );
8               // Fórmula de intensidade :
9           x1[i] = x1[i] * level ;
10          x2[i] = x1[i];
11      }
12 }
```

Algoritmo 3.3: Código do efeito de *Overdrive*

Os parâmetros do efeito foram definidos como:

- *Gain* — Corresponde a quantidade saturada. Os valores variam de -1 à 1;
- *Level* — É a intensidade do sinal. Os valores podem ser ajustados de 0 à 1.

### 3.4.3 Distorção

A construção do efeito de Distorção foi baseada no trabalho de Krueger (2008). A equação 3.3 foi reutilizada, e a equação 3.4 teve uma alteração:

$$X = \frac{((f_{ganho2} + K) * X)}{f_{ganho2} + K * abs(X)} \tag{3.5}$$

Sendo que $f_{ganho2}$ corresponde a segunda saturação. O algoritmo de funcionamento ficou semelhante ao algoritmo de *Overdrive*, como pode ser visto no Algoritmo 3.4.

```
1  (void)onStream : (UInt32) inNumberFrames sampleBufferLeft: ( float *) sampleBufferLeft
           sampleBufferRight: ( float *) sampleBufferRight{
2      float *x1, *x2;
3      x1 = sampleBufferLeft;
4      x2 = sampleBufferRight;
5      for( int i = 0; i < inNumberFrames; i++){
6          // Fórmula de ganho .
7          x1[i] = ((boost + k) * x1[i]) / ( boost + k * abs(x1[i]) );
8          // Fórmula de intensidade .
9          x1[i] = x1[i] * level ;
10         x2[i] = x1[i];
11     }
12 }
```

Algoritmo 3.4: Código do efeito de Distorção

Os parâmetros do efeito foram definidos abaixo:

- *Boost* — É a saturação do sinal no primeiro nível. Os valores variam de -1 à 1;
- *Gain* — É a saturação do sinal no segundo nível. Os valores podem ser ajustados de 0 à 1;
- *Level* — É a intensidade do sinal. Os valores podem ser ajustados de 0 à 1.

### 3.4.4 *Fuzz*

O efeito de *Fuzz*, foi originado de Rio et al. (2012). Sua implementação foi mais simples que o efeito de *Overdrive*: definiu-se o limiar de clipagem do sinal e depois o amplificou. As fórmulas utilizadas podem ser expressas pelas equações:

$$X = \begin{cases} lim, & \text{se } X > lim \\ -lim, & \text{se } X < -lim \end{cases}$$

$$X = X * ganho \tag{3.6}$$

Sendo *lim* o limiar de clipagem, *ganho* o nível de saturação e $X$ o sinal. O Algoritmo 3.5 demonstra a aplicação das fórmulas para obter o efeito.

```
1  (void)onStream : (UInt32) inNumberFrames sampleBufferLeft: ( float *) sampleBufferLeft
               sampleBufferRight: ( float *) sampleBufferRight{
2       float *x1, *x2;
3       x1 = sampleBufferLeft;
4       x2 = sampleBufferRight;
5       for( int i = 0; i < inNumberFrames; i++){
6             // Clipagem do sinal
7           if(x1[i] > lim){
8               x1[i] = lim;
9           }else if(x1[i] < -lim){
10              x1[i] = -lim;
11          }
12
13            // Intensidade do sinal
14          x1[i] = x1[i] * level;
15          x2[i] = x1[i];
16      }
17 }
```

Algoritmo 3.5: Código do efeito de *Fuzz*

Os parâmetros do efeito de *Fuzz* são:

- *Gain* — Corresponde ao nível de clipagem. Os valores variam de 0 à 1;
- *Level* — É a intensidade da distorção. Os valores podem ser ajustados de 0 à 1.

### 3.4.5 *Tremolo*

O efeito de *Tremolo* foi derivado da aplicação *SkypeFX*. Para a construção desse efeito foi utilizado um oscilador de baixa frequência cossenoidal. A fórmula simplificada que deu a

origem ao efeito foi:

$$X = X * (\cos(pos) + 1) * amp \tag{3.7}$$

Sendo que: $X$ corresponde ao sinal de entrada, *pos* é a posição do oscilador e *amp* a intensidade do oscilador. O código pode ser visto no Algoritmo 3.6.

```
1 -(void) onStream:(UInt32)inNumberFrames sampleBufferLeft:(float *)sampleBufferLeft
          sampleBufferRight:(float *)sampleBufferRight{
2     for (int index = 0; index < inNumberFrames ; index++ ) {
3         sampleBufferLeft[index] = sampleBufferLeft[index] * ((cos(pos) + 1) * sc + amount);
4         sampleBufferRight[index] = sampleBufferRight[index] * ((cos(pos + sep) + 1) * sc + amount
                  );
5         pos += adv;
6     }
7 }
```

Algoritmo 3.6: Código do efeito de *Tremolo*

Os parâmetros do efeito são:

- *Rate* — É a frequência ou velocidade do oscilador. Os valores variam de 1 à 30;
- *Depth* — É a intensidade do oscilador ou profundidade. A variação é entre 0 à 1;
- *Ping-Pong* — Define o nível de separação do sinal entre os canais esquerdo e direito. Os valores variam igual ao *Rate*.

### 3.4.6 *Pitch*

O efeito *Pitch* foi retirado da aplicação *audioGraph*. Não foi necessário realizar muitas adaptações para o funcionamento no iSoundFX, pois o algoritmo original já trabalhava com sinais do tipo flutuante. Portanto essa seção não demonstrará o desenvolvimento do efeito. O parâmetro de configuração do efeito é apenas a mudança do tom do sinal, podendo ser mais grave ou mais agudo.

## 3.5 Aplicação de Exemplo

Nesta seção será demonstrado o uso do iSoundFX através da construção de uma aplicação de exemplo. Será abordado também o processo de elaboração da interface desse programa.

A criação da aplicação foi elaborada em duas fases de desenvolvimento: na primeira etapa foram criados protótipos de tela que explorassem as funcionalidades da biblioteca; na segunda etapa as telas foram implementadas e os métodos de acesso do iSoundFX foram ligados as ações dos componentes de interface.

O processo de concepção dos protótipos de interface foi realizado através do recurso de *Storyboard* presente no ambiente de desenvolvimento *Xcode*. Dentre outras facilidades, este recurso permite que os componentes sejam arrastados na tela dispensando o uso de outros programas para este fim.

Na tela principal da Figura 3.8 pode-se observar os elementos de interface: a barra superior comporta os botões de captura de áudio representado por (1). Do lado oposto, representado por (2), estão os botões de mover para esquerda da cadeia, adicionar um novo efeito, remover o efeito atual e mover para a direita da cadeia respectivamente.

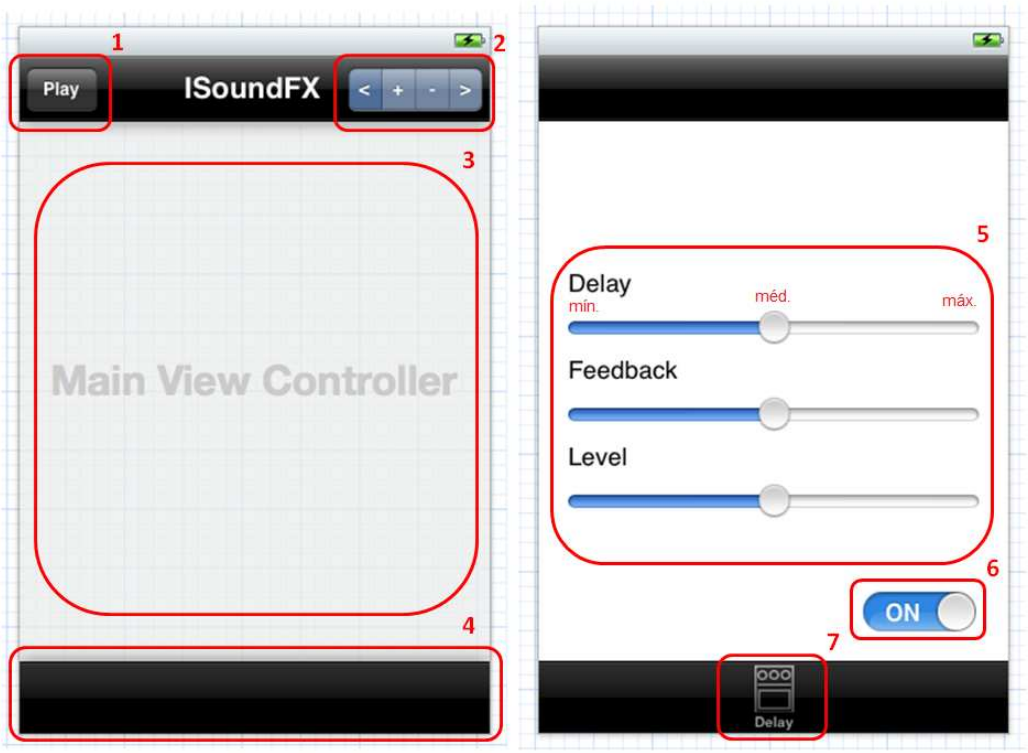


Figura 3.8: Tela principal (a esq.) e tela do efeito de *Delay* (a dir.)

A parte central da tela, representado por (3), é a área onde estarão dispostos os componentes de interface responsáveis por configurar os parâmetros dos efeitos. E, na barra inferior, representado por (4), é onde estarão as abas dos efeitos adicionados. As configurações dos efeitos serão apresentadas na região central da tela sempre que o usuário alternar entre as abas.

Na tela de *Delay*, na mesma figura, estão: em (5) os componentes de interface responsáveis pelas configurações do efeito, para este fim optou-se por utilizar controles deslizantes; em (6) está o botão de estado, onde é possível ativar ou desativar o efeito; e em (7) a aba do efeito selecionado.

A partir das telas da Figura 3.8, chegou-se ao resultado final ilustrado na Figura 3.9. Essa figura foi capturada da tela do *iPod* enquanto a aplicação estava sendo executada. Observa-se que a aba selecionada está com um destaque no ícone com a cor azul. Nota-se também que o botão ativar/desativar efeito está desativado na tela de *Delay* e ativado na tela de *Overdrive*.

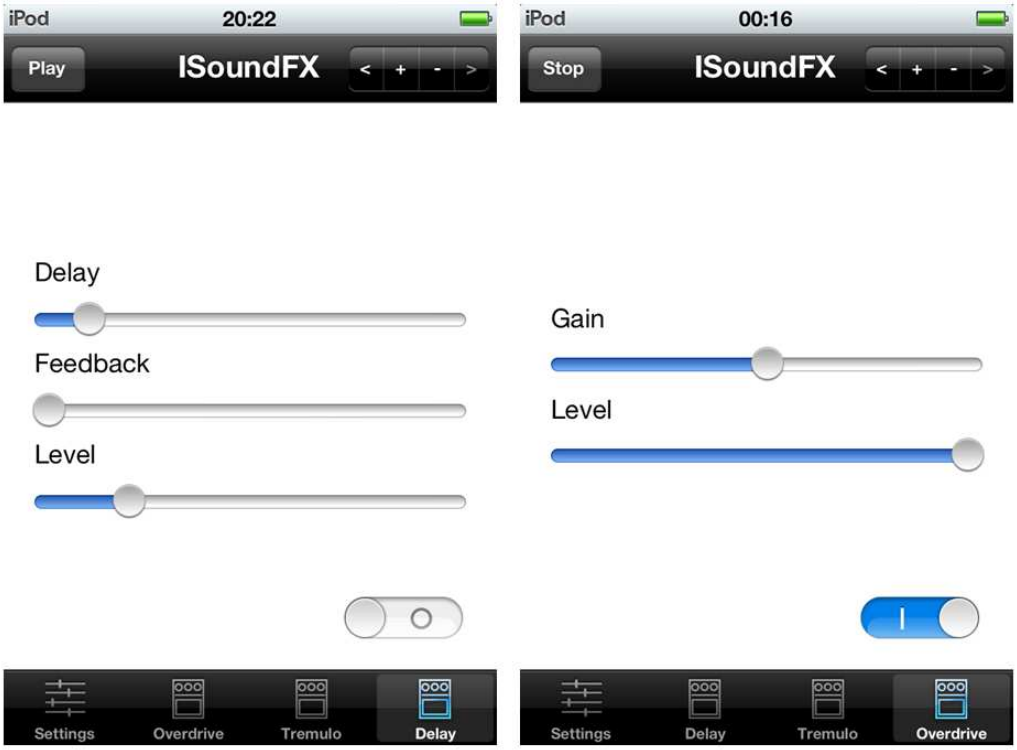


Figura 3.9: Tela do efeito de *Delay* e *Overdrive* (da esq. para a dir.)

A próxima etapa do desenvolvimento da aplicação foi instanciar a biblioteca iSoundFX e relacioná-la às ações dos componentes de interface. O método *viewDidLoad* da classe principal da aplicação instancia o *ISoundFX* uma única vez e sua referência é passada para a classe *MainAppDelegate*, para que o acesso a biblioteca seja compartilhado através do padrão *Singleton*. Essa operação pode ser vista no código do Algoritmo 3.7.

```
1  - (void)viewDidLoad
2  {
3      [super viewDidLoad];
4        // Criação da classe
5      iSoundFx = [[ISoundFx alloc] init ];
6        // Obtenção da referência da classe MainAppDelegate
7      MainAppDelegate *appDelegate =(MainAppDelegate *)[[UIApplication sharedApplication] delegate
                ];
8        // Compartilhamento da instância
9      appDelegate.iSoundFx = iSoundFx;
10 }
```

Algoritmo 3.7: Compartilhamento da classe *ISoundFX*

Para vincular controles de interface à biblioteca foram definidos três tipos de métodos: *initControls* — reponsável por iniciar os elementos de interface;e *effectEnabled* — responsável por ativar e desativar o efeito; *effectChanged* — reponsável por alterar o valor do efeito. O código do Algoritmo 3.8 exemplifica a utilização dos métodos para o efeito de *Pitch*.

```
1  -(void) initControls {
2      [ pitchShift setMaximumValue: [iSoundFx getEffectMaxValue:self.indexInParent sliderIndex :
                PITCH_SHIFT]];
3      [ pitchShift setMinimumValue:[iSoundFx getEffectMinValue:self.indexInParent sliderIndex :
                PITCH_SHIFT]];
4      [ pitchShift setValue: [iSoundFx getEffectValue : self.indexInParent sliderIndex :PITCH_SHIFT
                ]];
5      [pitchSwitch setOn:[iSoundFx isEffectEnabled : self.indexInParent ] ];
6  }
7
8  - (IBAction) effectEnabled :( id )sender {
9      UISwitch *switchBtn = (UISwitch*)sender;
10     [iSoundFx enableEffect :switchBtn.isOn effectIndex : self.indexInParent ];
11 }
12
13 - (IBAction)effectChanged :( id )sender {
14     UISlider * slider = (UISlider*)sender;
15     [iSoundFx changeEffect: slider.value effectIndex : self.indexInParent sliderIndex :
           PITCH_SHIFT];
16 }
```

Algoritmo 3.8: Métodos de configuração do efeito de *Pitch*.

## 3.6 Resultados

Nessa seção serão apresentados os resultados obtidos a partir de testes realizados com a aplicação de exemplo da biblioteca iSoundFX. O intuito desses testes é fazer uma análise visual dos efeitos produzidos pela biblioteca através dos espectros dos sinais de áudio.

Para aplicar os testes foi utilizado um programa chamado *Audacity*, através desta ferramenta foi possível visualizar os espectros de áudio no domínio do tempo e no domínio da frequência. Os materiais utilizados para os testes foram:

- Guitarra *Vintage* Modelo *Stratocaster*;
- *iPod Touch* 4G versão 5.1.1;
- Conector *iRig*;
- Computador com *Windows* 7 com o programa *Audacity* versão 1.3 instalado.

A Figura 3.10 exemplifica o esquema de funcionamento dos testes. A guitarra é conectada ao *iPod* pela interface *iRig*, que por sua vez tem sua saída conectada a entrada de microfone do computador. A biblioteca foi alterada para que o canal de áudio esquerdo não realizasse modificações no sinal, enquanto que no canal direito, os efeitos foram aplicados. Dessa forma, os espectros puderam ser comparados a um mesmo sinal de entrada.

Os espectros dos sinais foram capturados pelo *Audacity* ao tempo que se tocava acordes e notas na guitarra. A disposição das imagens obtidas estão na seguinte forma: o eixo horizontal representa o tempo, em segundos, e o eixo vertical a amplitude, na escala de 0 à 1. A imagem superior corresponde ao sinal antes do efeito e a imagem inferior ao sinal com o efeito aplicado.

Na Figura 3.11 é possível observar os resultados do teste realizado com o efeito de *Delay*. Os parâmetros foram ajustados em três repetições, 0,1 segundos de *delay* e 0,8 de *level*. Pela análise do espectro do sinal pode-se constatar que o efeito obteve êxito, já que a intensidade do sinal vai decaindo em cada repetição do sinal, caracterizando o efeito de *Delay* (MENDES; GOULART, 2009).

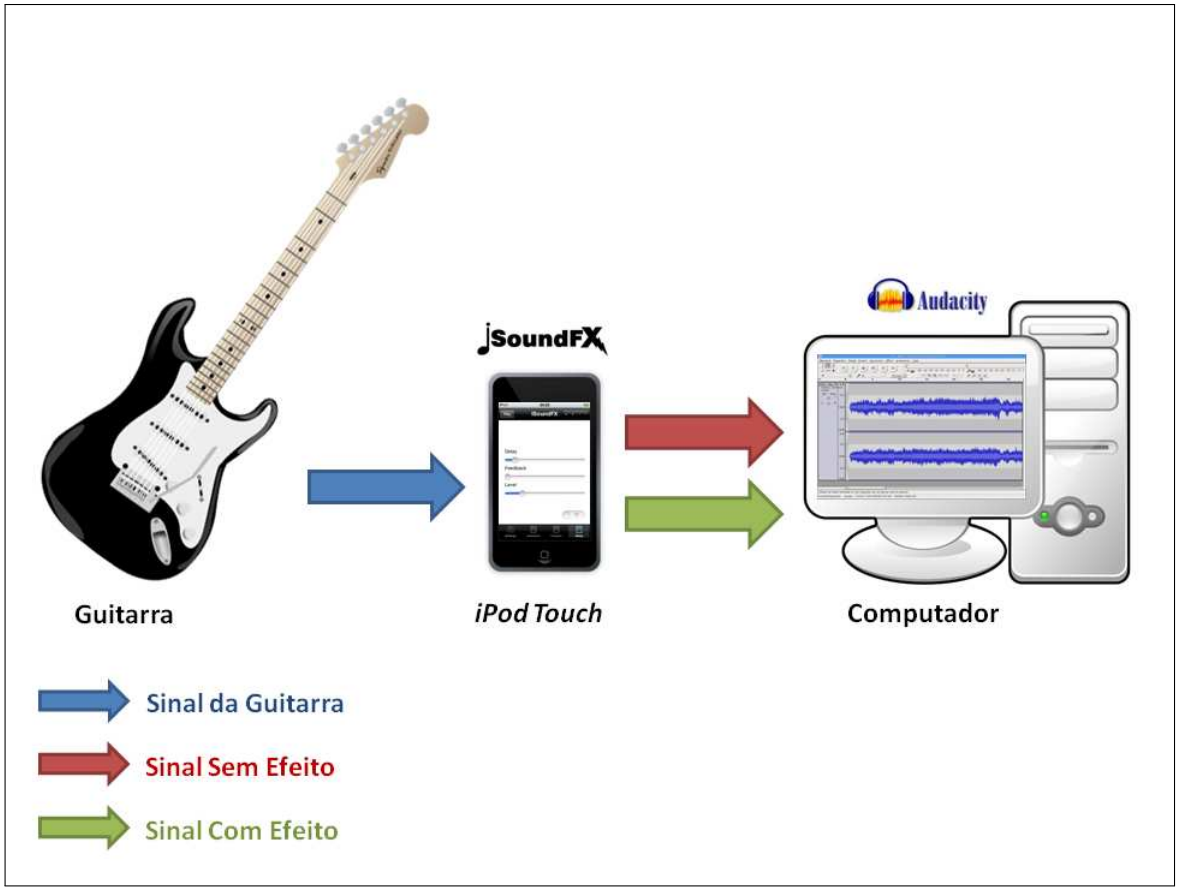


Figura 3.10: Esquema de teste do iSoundFX

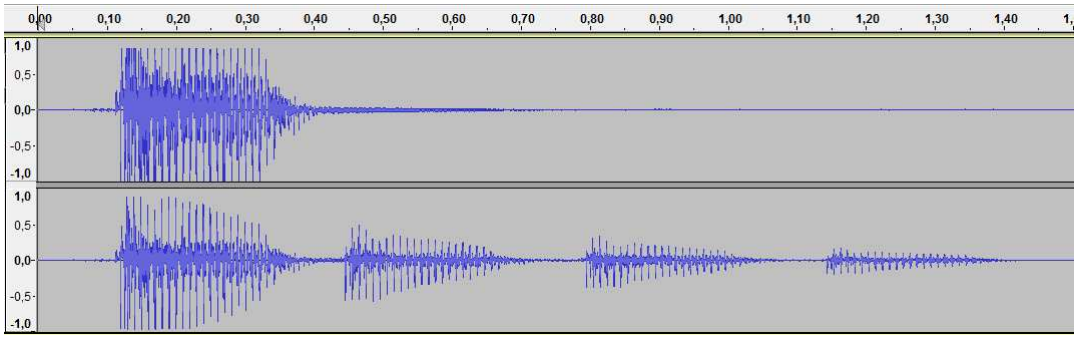


Figura 3.11: Sinal original e com efeito de *Delay*

Os resultados do teste com o efeito de *Overdrive* podem ser vistos na Figura 3.12. Os parâmetros foram ajustados em 0.8 de *gain* e 0.6 de *level*. Pelo sinal obtido é possível afirmar que o efeito em questão está cumprindo as características da sua definição: o sinal está amplificado e saturado em alguns pontos (MENDES; GOULART, 2009). Devido a similaridade entre os sinais dos efeitos de *Overdrive*, *Fuzz* e Distorção, não foram realizados testes nos dois últimos efeitos.

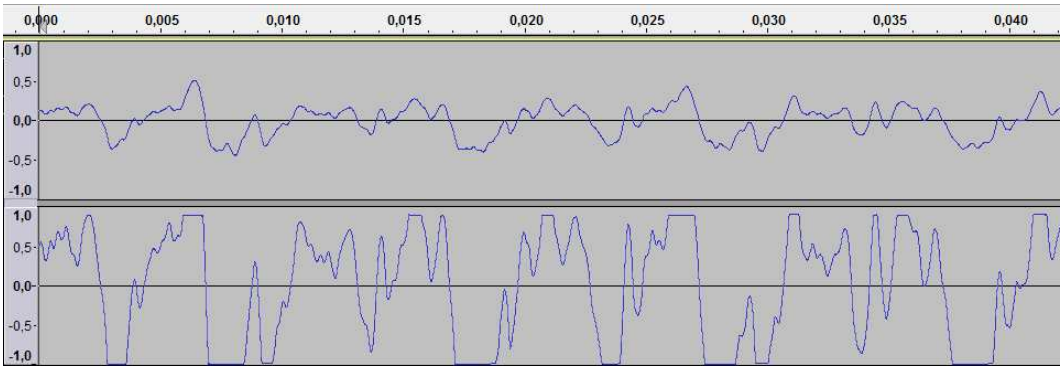


Figura 3.12: Sinal original e com efeito de *Overdrive*

A Figura 3.13 demonstra os resultados do teste com o efeito de *Tremolo*. Os parâmetros foram ajustados em 0.5 de *depth*, 10Hz de *rate* e 0 de *ping-pong*. O efeito em análise cumpriu com as expectativas: o sinal foi atenuado periodicamente.

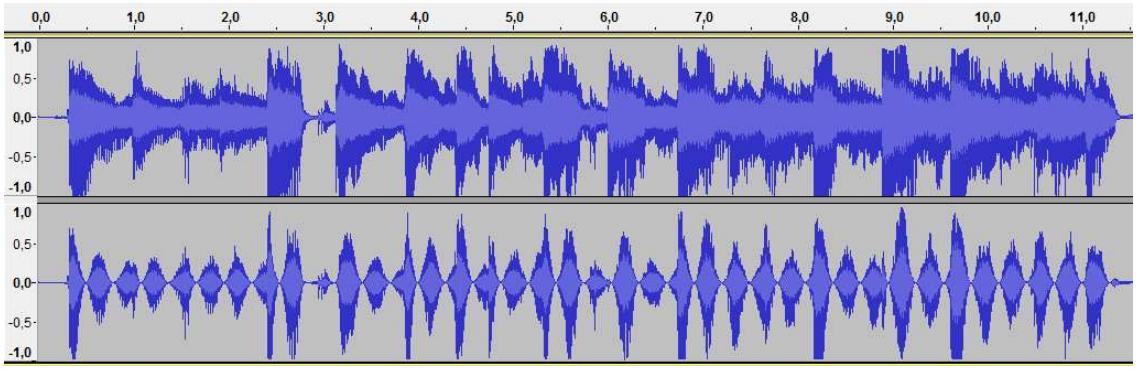


Figura 3.13: Sinal original e com efeito de *Tremolo*

A diferença entre os sinais obtidos no teste do efeito de *Pitch* não foi evidente quando a análise foi realizada no domínio do tempo (amplitude X tempo), por isso utilizou-se um recurso do *Audacity* que demonstra, através do processo de FFT, os sinais no domínio da frequência. Nesta forma de representação do espectro o sinal está sob forma de frequência, em Hz, por amplitude, em dB. A imagem à esquerda corresponde ao sinal original e à direita o sinal após o efeito.

A Figura 3.14 demonstra os resultados do teste com o efeito de *Pitch*. O parâmetro *level* foi ajustado em 1,0. Observa-se, ao analisar os espectros, que o sinal original possui um pico acima de -21dB na frequência de 300Hz, enquanto que o sinal com o efeito aplicado, possui um pico também acima de -21dB na frequência de 600Hz. Isso quer dizer que o efeito obteve sucesso.

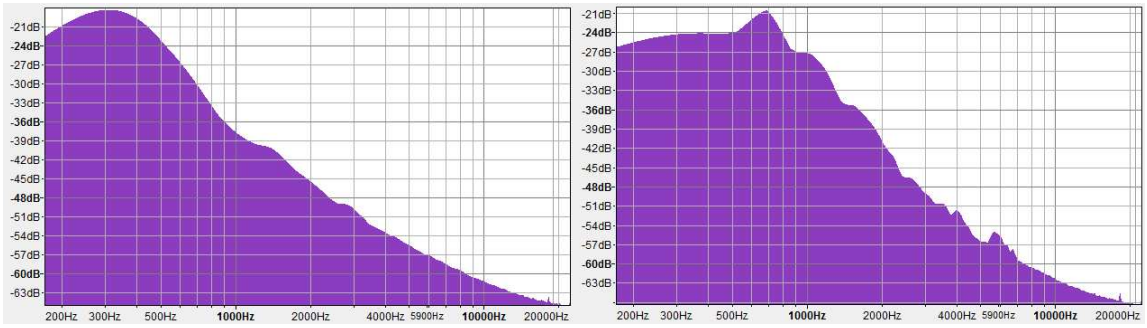


Figura 3.14: Sinal original e com efeito de *Pitch*

A análise visual dos espectros de sinais de áudio constataram que o funcionamento dos efeitos desenvolvidos cumpriram com o esperado. Pode-se afirmar que, auditivamente, o resultado sonoro produzido pelos efeitos do iSoundFX tiveram uma qualidade satisfatória quando comparados com as aplicações *GarageBand* e *Amplitube*.

# 4 Considerações Finais

Este trabalho apresentou o desenvolvimento de uma biblioteca de efeitos sonoros *Open Source* para a plataforma iOS, denominada iSoundFX. O intuito desse trabalho foi oferecer aos desenvolvedores uma alternativa na construção de aplicações musicais, fornecendo uma biblioteca que facilitasse este processo.

Foram integrados à biblioteca os efeitos de *Delay*, Distorção, *Fuzz*, *Overdrive* e *Pitch*. Alguns desses efeitos foram provenientes das aplicações *SkypeFX* e *audioGraph*, e dos trabalhos de Chang (2011), Krueger (2008) e Rio et al. (2012). Foi desenvolvido também um módulo de utilização do *framework* nativo do iOS, com a finalidade de capturar e processar as amostras de áudio.

Verificou-se na especificação do iSoundFX, que a utilização de aplicações *Open Source* é uma estratégia eficaz, em termos de tempo gasto, na construção de *softwares*. A biblioteca iSoundFX possibilita o reuso, a legibilidade, e a facilidade de manutenção do código, já que, premissas como a adoção do padrão de projeto *strategy* e a modularização foram empregadas.

Um aplicativo experimental foi desenvolvido com o objetivo de validar o funcionamento do iSoundFX. Os resultados, obtidos com o aplicativo em questão, foram analisados através do espectro dos sinais utilizando o software de edição de áudio *Audacity*. Observou-se, com esta análise, que os algoritmos criados para o iSoundFX atendem às expectativas de funcionamento.

Como cumprimento de um dos objetivos do trabalho, a biblioteca foi hospedada no *Google Code*[9] sob a licença *Lesser GNU Public License*, viabilizando, desta forma, que os desenvolvedores possam utilizar o iSoundFX na sua aplicação, sem que haja um valor agregado.

Como trabalhos futuros dá-se as sugestões:

- Portar a biblioteca para outras plataformas móveis como *Android* e *Windows Phone* - Considerando que a plataforma dê suporte a captura e execução de áudio em tempo real;

[9] O projeto está disponível em <http://code.google.com/p/isoundfx/>.

- Inclusão de novos efeitos - Visto que existe uma quantidade significativa de efeitos sonoros que poderiam ser adicionados ao iSoundFX;
- Proposta de interface de usuário utilizando a biblioteca desenvolvida - Observou-se que a utilização de componentes de interface de usuário otimizaria o tempo de desenvolvimento das aplicações musicais;
- Suporte às configurações do módulo de captura: taxa de amostragem, latência de áudio e formato das amostras - Os desenvolvedores não se limitariam às configurações utilizadas pelo iSoundFX.

# *Referências Bibliográficas*


ANHAIA, P. *Entendendo os efeitos*. 2012. Disponível em: <http://pauloanhaia.com.br/?page_id=37>. Acesso em: 16 de março de 2012.

APPLE. *Aplicações Musicais*. 2012. Disponível em: <http://itunes.apple.com/us/genre/music/id34>. Acesso em: 27 de fevereiro 2012.

APPLE. *iOS Technology Overview*. 2012. Disponível em: <http://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/iPhoneOSTechOverview.pdf>. Acesso em: 18 de março de 2012.

APPLE. *Tremolo Effect*. 2012. Disponível em: <http://documentation.apple.com/en/logicstudio/effects/index.html>. Acesso em: 20 de março de 2012.

BOSS. *A Famosa Ordem dos Pedais*. 2012. Disponível em: <http://roland.com.br/boss/produtos/315/tips>. Acesso em: 12 de junho de 2012.

CHANG, C.-H. A guitar overdrive/distortion effect of digital signal processing. *Digital Audio Systems*, 2011. Disponível em: <http://ses.library.usyd.edu.au/bitstream/2123/7624/2/DESC9115_DAS_Assign02_310106370.pdf>. Acesso em: 21 de junho de 2012.

FERREIRA, S. *Sistema Especialista para reconhecimento de acordes musicais em tempo real para violão elétrico utilizando técnicas de DSP*. Dissertação (Mestrado) — Universidade Federal da Bahia, Salvador, 2006.

HEATH, M. *Skype Voice Changer*. 2012. Disponível em: <http://skypefx.codeplex.com>. Acesso em: 12 de junho de 2012.

IAZZETTA, F. *Efeitos*. 2012. Disponível em: <http://www.eca.usp.br/prof/iazzetta/tutor/audio/efeitos/effx.html>. Acesso em: 16 de março de 2012.

KARASINSKI, L. *Análise AMPKit x iRig*. 2012. Disponível em: <http://www.tecmundo.com.br/iphone/9481-analise-ampkit-x-irig.htm>. Acesso em: 6 de março de 2012.

KRUEGER, D. *Tratamento de sinais sonoros no computador simulando pedais virtuais*. Itajaí, 2008.

LACERDA, O. *Compêndio de teoria elementar da música*. 9ª. ed. São Paulo: Ricordi, 1966.

LEMOS, R. P. *Teoria de Telecomunicações*. 2012. Disponível em: <http://www.eee.ufg.br/ lemos/Aula2.html>. Acesso em: 19 de junho de 2012.

MENDES, L. F.; GOULART, R. L. *Pedaleira multi-efeitos para guitarra com processamento digital de sinais*. Curitiba, 2009.

MULTIMEDIA, I. *FAQ: AmpliTube iRig for Android?* 2012. Disponível em: <http://www.ikmultimedia.com/forum/viewtopic.php?f=10&t=553>. Acesso em: 26 de fevereiro de 2012.

OSCROVE. *Las cualidades del sonido*. 2012. Disponível em: <http://oscrove.wordpress.com/teoria-musical/el-sonido/las-cualidades-del-sonido>. Acesso em: 10 de março de 2012.

PARK, T. H. *Introduction to Digital Signal Processing*. 1ª. ed. USA: World Scientific, 2010.

PILONE, D.; PILONE, T. *Use a Cabeça! Desenvolvendo para iPhone*. 1ª. ed. Rio de Janeiro: Alta Books, 2011.

PIMENTEL, G.; CRISTINA, M. *Fila Circular*. 2012. Disponível em: <http://www.icmc.usp.br/ sce182/fcirc.html>. Acesso em: 20 de julho de 2012.

REIMAN, D. *Digital Audio Effects Processing with Csound*. 2012. Disponível em: <http://www.donreiman.com//Introductory_Home_Page.htm>. Acesso em: 18 de março de 2012.

RIO, H.; DWISATYO, F.; FAZHA, H.; SURYANEGARA, M.; GUNAWAN, D. Analysis of real time audio effect design using tms320 c6713 dsk. 2012.

SMITH, S. W. *The Scientist and Engineer's Guide to Digital Signal Processing*. [s.n.], 2012. Disponível em: <http://www.dspguide.com>. Acesso em: 17 de março de 2012.

VALLE, S. d. *Manual prático de acústica*. 1ª. ed. Rio de Janeiro: Música & Tecnologia, 2006.

ZICARELLI, T. *audioGraph*. 2012. Disponível em: <http://zerokidz.com/audiograph/docs/audiograph.pdf>. Acesso em: 12 de março de 2012.

# *APÊNDICE A – Método de captura de áudio*

```
1  OSStatus micLineInCallback (void                        *inRefCon,
2                              AudioUnitRenderActionFlags  *ioActionFlags,
3                              const AudioTimeStamp        *inTimeStamp,
4                              UInt32                      inBusNumber,
5                              UInt32                      inNumberFrames,
6                              AudioBufferList             *ioData)
7  {
8      OSStatus renderErr;
9      NativeAudio *THIS = (NativeAudio*) inRefCon;
10     renderErr = AudioUnitRender(THIS.ioUnit, ioActionFlags,
11                                 inTimeStamp, 1, inNumberFrames, ioData);
12
13       // Getting the samples
14     SInt32 *inSamplesLeft  = (SInt32 *) ioData->mBuffers[0].mData;
15     SInt32 *inSamplesRight = (SInt32 *) ioData->mBuffers[1].mData;
16
17     float *sampleBufferLeft = THIS.conversionBufferLeft;
18     float *sampleBufferRight = THIS.conversionBufferRight;
19
20       // Convert SInt32 to float
21     for(int i = 0; i < inNumberFrames; i++){
22         sampleBufferLeft[i] = (inSamplesLeft[i]/(float)0x00FFFFFF);
23         sampleBufferRight[i] = inSamplesRight[i]/(float)0x00FFFFFF;
24     }
25
26       // Effects loop
27     for(id idEffect in THIS.effectsArray) {
28         Effect *effect = (Effect *) idEffect;
29         if(effect.isEnabled){
30             [effect onStream: inNumberFrames sampleBufferLeft:(float*)sampleBufferLeft
                   sampleBufferRight:(float*)sampleBufferRight];
31         }
```

```
32          }
33
34            // Convert       float       to       SInt32
35          for( int  i = 0;  i < inNumberFrames; i++){
36              inSamplesLeft[ i ] = sampleBufferLeft[ i ]*(0x00FFFFFF);
37              inSamplesRight[ i ] = sampleBufferRight[ i ]*(0x00FFFFFF);
38          }
39
40          return  noErr;
41      }
```

# *APÊNDICE B – Método de inicialização do módulo de captura de áudio*

```
1  - (void) setupAudioProcessingGraph
2  {
3      OSStatus result = noErr;
4      result = NewAUGraph (&processingGraph);
5      if (noErr != result ) {
6          NSLog(@"NewAUGraph failed: %ld", result);
7          return ;
8      }
9
10     AudioComponentDescription iOUnitDescription;
11     iOUnitDescription .componentType          = kAudioUnitType_Output;
12     iOUnitDescription .componentSubType       = kAudioUnitSubType_RemoteIO;
13     iOUnitDescription .componentManufacturer  = kAudioUnitManufacturer_Apple;
14     iOUnitDescription .componentFlags         = 0;
15     iOUnitDescription .componentFlagsMask     = 0;
16
17     result =    AUGraphAddNode (
18                                 processingGraph,
19                                 &iOUnitDescription,
20                                 &ioNode);
21
22     if (noErr != result ) {
23         NSLog(@"AUGraphAddNode failed for RemoteIO: %ld", result);
24         return ;
25     }
26
27     AudioComponentDescription mixerUnitDescription;
28     mixerUnitDescription .componentType          = kAudioUnitType_Mixer;
29     mixerUnitDescription .componentSubType       = kAudioUnitSubType_MultiChannelMixer;
30     mixerUnitDescription .componentManufacturer  = kAudioUnitManufacturer_Apple;
```

```
31      mixerUnitDescription.componentFlags        = 0;
32      mixerUnitDescription.componentFlagsMask    = 0;
33
34      result = AUGraphAddNode(
35                                  processingGraph,
36                                  &mixerUnitDescription,
37                                  &mixerNode
38                                  );
39
40      if (noErr != result) {
41          NSLog(@"AUGraphAddNode failed for Mixer: %ld", result);
42          return;
43      }
44
45      result = AUGraphOpen(processingGraph);
46      if (noErr != result) {
47          NSLog(@"AUGraphOpen failed: %ld", result);
48          return;
49      }
50
51      result = AUGraphNodeInfo(processingGraph, ioNode, NULL, &ioUnit);
52      if (noErr != result) {
53          NSLog(@"AUGraphNodeInfo failed: %ld", result);
54          return;
55      }
56
57      if ([session inputIsAvailable]) {
58          UInt32 enableInput = 1;         // Use 0 to disable
59          AudioUnitElement inputBus = 1;      // Use 0 to output bus
60
61          result = AudioUnitSetProperty(ioUnit,
62                                        kAudioOutputUnitProperty_EnableIO,
63                                        kAudioUnitScope_Input,
64                                        inputBus,
65                                        &enableInput,
66                                        sizeof(enableInput));
67
68          if (noErr != result) {
69              NSLog(@"AudioUnitSetProperty for EnableIO failed: %ld", result);
70              return;
71          }
72      }
73
```

```
74      result =    AUGraphNodeInfo (
75                                  processingGraph,
76                                  mixerNode,
77                                  NULL,
78                                  &mixerUnit
79                                  );
80      if (noErr != result ) {
81          NSLog(@"AUGraphNodeInfo for mixerNode failed: %ld", result);
82          return ;
83      }
84
85      if ([ session  inputIsAvailable ]) {
86          UInt16 busNumber = 0;          //    mic    channel    on    mixer
87          AURenderCallbackStruct inputCallbackStruct ;
88
89          inputCallbackStruct .inputProc        = micLineInCallback;    //    8.24    version
90          inputCallbackStruct .inputProcRefCon  = self ;
91          result = AUGraphSetNodeInputCallback (
92                                                processingGraph,
93                                                mixerNode,
94                                                busNumber,
95                                                &inputCallbackStruct
96                                                );
97
98          if (noErr != result ) {
99              NSLog(@"AUGraphSetNodeInputCallback mic/lineIn: %ld", result);
100             return ;
101         }
102
103         memset(&streamDesc, 0, sizeof (streamDesc));
104         int bytesPerSample          = sizeof (AudioUnitSampleType);
105         streamDesc.mFormatID        = kAudioFormatLinearPCM;
106         streamDesc.mFormatFlags     = kAudioFormatFlagsAudioUnitCanonical;
107         streamDesc.mBytesPerPacket  = bytesPerSample;
108         streamDesc.mBytesPerFrame   = bytesPerSample;
109         streamDesc.mFramesPerPacket = 1;
110         streamDesc.mBitsPerChannel  = 8 * (bytesPerSample);
111         streamDesc.mChannelsPerFrame = 2;
112         streamDesc.mSampleRate      = sampleRate;
113         result =    AudioUnitSetProperty (
114                                           ioUnit,
115                                           kAudioUnitProperty_StreamFormat,
116                                           kAudioUnitScope_Output,
```

```
117                                                 1,
118                                                 &streamDesc,
119                                                 sizeof (streamDesc)
120                                                 );
121             if (noErr != result) {
122                 NSLog(@"AudioUnitSetProperty mic/lineIn: %ld", result);
123                 return;
124             }
125         }
126
127         result = AUGraphConnectNodeInput(processingGraph, mixerNode, 0, ioNode, 0);
128         if (noErr != result) {
129             NSLog(@"AUGraphConnectNodeInput for IoNode failed: %ld", result);
130             return;
131         }
132     }
```

# *APÊNDICE C – Classe do efeito de Overdrive*

```
#import "Overdrive.h"

@implementation Overdrive

-(void)onInit{
    Slider *gainSlider = [[Slider alloc] initWithConfiguration:0 min:-1 max:1];
    Slider *levelSlider = [[Slider alloc] initWithConfiguration:1 min:0 max:1];
    [self insertSlider:gainSlider sliderIndex:OVERDRIVE_GAIN];
    [self insertSlider:levelSlider sliderIndex:OVERDRIVE_LEVEL];
}

-(void)onConfigurationChanged{
    float gain = [self getCurrentValue:OVERDRIVE_GAIN];
    level = [self getCurrentValue:OVERDRIVE_LEVEL];
    k = 2*gain/(1-gain);
}

-(void)onStream:(UInt32)inNumberFrames sampleBufferLeft:(float*)sampleBufferLeft
        sampleBufferRight:(float*)sampleBufferRight{
    float *x1;
    x1 = sampleBufferLeft;

    for(int i = 0; i < inNumberFrames; i++){
        x1[i] = ((1+k)*x1[i])/(1+k*abs(x1[i]));
        x1[i] *= level;
        sampleBufferRight[i] = x1[i];
    }
}

@end
```