



UNIVERSIDADE  
ESTADUAL DE LONDRINA

Centro de Tecnologia e Urbanismo  
Departamento de Engenharia Elétrica

**Matheus Raphael Elero**

**Desenvolvimento de software aplicado  
ao ensino de Engenharia Elétrica  
utilizando Unity 3D**

Monografia apresentada ao curso de Engenharia Elétrica da Universidade Estadual de Londrina, como parte dos requisitos necessários para a conclusão do curso de Engenharia Elétrica.

Londrina, PR  
2018



UNIVERSIDADE  
ESTADUAL DE LONDRINA

Matheus Raphael Elero

**Desenvolvimento de software aplicado  
ao ensino de Engenharia Elétrica  
utilizando Unity 3D**

Monografia apresentada ao curso de Engenharia Elétrica da Universidade Estadual de Londrina, como parte dos requisitos necessários para a conclusão do curso de Engenharia Elétrica.

Área: Circuitos Elétricos / Processamento de Sinais

Orientador:

Prof. Dr. Ernesto Fernando Ferreyra Ramirez

Londrina, PR  
2018

## **Ficha Catalográfica**

Raphael Elero, Matheus

Desenvolvimento de software aplicado ao ensino de Engenharia Elétrica utilizando Unity 3D. Londrina, PR, 2018. 122 p.

Monografia (Trabalho de Conclusão de Curso) – Universidade Estadual de Londrina, PR. Departamento de Engenharia Elétrica

- .
1. Séries de Fourier 2. Efeitos de Áudio 3. Aplicativo 4. Jogo Digital 5. Matlab.

**Matheus Raphael Elero**

# **Desenvolvimento de software aplicado ao ensino de Engenharia Elétrica utilizando Unity 3D**

Monografia apresentada ao curso de Engenharia Elétrica da Universidade Estadual de Londrina, como parte dos requisitos necessários para a conclusão do curso de Engenharia Elétrica.

Área: Circuitos Elétricos / Processamento de Sinais

## **Comissão Examinadora**

---

Prof. Dr. Ernesto Fernando Ferreyra  
Ramirez  
Depto. de Engenharia Elétrica  
Universidade Estadual de Londrina  
Orientador

---

Prof. Dr. Márcio Roberto Covacic  
Depto. de Engenharia Elétrica  
Universidade Estadual de Londrina

---

Profa. Dra. Maria Bernadete de Moraes  
França  
Depto. de Engenharia Elétrica  
Universidade Estadual de Londrina

*“Do. Or do not. There is no try.”* Yoda.

# Agradecimentos

Primeiramente, agradeço aos meus pais Enivaldo e Regina por todo apoio, confiança, carinho e paciência em toda esta jornada. Às minhas irmãs Natalí e Gabriela que também contribuíram com todo apoio.

Ao professor Ernesto pela excelente orientação durante o projeto, que contribuiu muito para o trabalho e meu aprendizado. Cada conversa realizada ao longo deste período foi extremamente proveitosa.

À 3E-UEL, e todas as pessoas com o qual tive o prazer de conhecer e trabalhar durante os 3 anos de empresa. A 3E foi um marco na minha graduação, que permitiu uma experiência incrível, mudando muito a minha percepção da graduação e vida profissional. Além disso, fiz grandes amigos e vivi momentos marcantes. Guardo cada instante, e cada pessoa deste período com muito carinho. Obrigado a todos.

Aos colegas de sala por toda contribuição neste período, tanto nos estudos quanto na amizade. Sem eles tudo isso não seria possível.

Por fim, a minha namorada Bárbara pela companhia, apoio, e paciência. Com ela, até os momentos de dificuldade da graduação foram mais felizes.

# Resumo

Alguns conceitos teóricos de áreas da Engenharia Elétrica podem ser bastante abstratos e de difícil compreensão para alunos dos primeiros anos do curso, que é o caso, por exemplo, das séries de Fourier. Com o objetivo de contribuir para a solução desse problema, este trabalho mostra o desenvolvimento de um *software* de caráter educativo, que apresenta uma interação lúdica e visual acerca do tema, juntamente com uma aplicação prática na área de áudio. O aplicativo foi construído utilizando o *software* Unity 3D e programação em C#, que é uma ferramenta gratuita para projetos sem fins lucrativos, e destinada para desenvolvimento de jogos digitais. O *software* foi testado na disciplina de Circuitos Elétricos no curso de Engenharia Elétrica da Universidade Estadual de Londrina, quando foi verificado que a aplicação do *software* facilita o entendimento do assunto pelos alunos.

**Palavras-chave:** Séries de Fourier; Efeitos de Áudio; Aplicativo; Jogo Digital; Matlab.

# Abstract

Some theoretical concepts in the areas of Electrical Engineering can be quite abstract and difficult to understand for students in the first years of the course, which is the case, for example, of the Fourier series. In order to contribute to solve this problem, this work shows the development of an educational software, which presents a playful and visual interaction about the subject along with a practical application in the audio area. The app was built using Unity 3D software and programming in C#, which is a free tool for nonprofit projects, and is designed to development of digital games. The software was tested in the discipline of Electrical Circuits in the Electrical Engineering course of the State University of Londrina, when it was verified that the use of the software facilitates the understanding of the subject by the students.

**Key-Words:** Fourier Series; Audio Effects; App; Digital Games ; MatLab.

# Sumário

## **Lista de Figuras**

## **Lista de Tabelas**

<b>1</b>	<b>Introdução</b>	<b>1</b>
1.1	Objetivos . . . . .	2
1.1.1	Objetivos Gerais . . . . .	2
1.1.2	Objetivos Específicos . . . . .	2
<b>2</b>	<b>Revisão da Literatura</b>	<b>4</b>
2.1	Jogos Digitais na Educação . . . . .	4
2.2	Séries Trigonométricas de Fourier . . . . .	6
2.3	Áudios e seus efeitos . . . . .	8
2.3.1	Processamento Digital de Sinais de Áudio . . . . .	9
2.3.2	Espectro de Frequências . . . . .	10
2.3.3	Filtros . . . . .	12
2.3.4	Distorção . . . . .	13
2.3.5	Tremolo . . . . .	16
2.3.6	Chorus . . . . .	17
2.4	Reaper . . . . .	18
2.5	Unity 3D . . . . .	19
2.5.1	Ambiente de Desenvolvimento . . . . .	20
2.5.2	<i>Vector3</i> . . . . .	21
2.5.3	<i>Void Start</i> . . . . .	22
2.5.4	<i>Void Update</i> . . . . .	22

2.5.5	<i>Button</i>	22
2.6	C# e programação orientada a objetos	23
<b>3</b>	<b>Metodologia</b>	<b>25</b>
3.1	Séries de Fourier no Matlab	25
3.2	Gráficos Unity 3D	26
3.3	Jogo Séries de Fourier	28
3.3.1	Estrutura do Jogo	28
3.3.2	Desenvolvimento do jogo	30
3.4	Simulações de Efeitos de Áudio no Matlab	32
3.4.1	Tremolo	32
3.4.2	Distortion	33
3.4.3	Chorus	34
3.5	Componentes de Áudio Unity 3D	37
3.5.1	<i>Audio Source</i> e <i>Audio Clip</i>	37
3.5.2	Sinal no Tempo	37
3.5.3	Espectro do Sinal	38
3.5.4	<i>Distortion Filter</i>	39
3.5.5	Filtragem	40
3.5.6	Chorus	40
3.5.7	Tremolo	42
3.6	Aplicativos de Efeitos de Áudio Unity 3D	43
3.7	Validação do <i>Software</i>	47
<b>4</b>	<b>Resultados e Discussões</b>	<b>48</b>
4.1	Séries de Fourier Matlab	48
4.2	Gráficos Unity 3D	50
4.3	Jogo Séries de Fourier	52
4.4	Simulações de Efeitos de Áudio Matlab	56

4.4.1	Tremolo . . . . .	56
4.4.2	<i>Distortion</i> . . . . .	56
4.4.3	Chorus . . . . .	58
4.5	Componentes de Áudio Unity 3D . . . . .	60
4.5.1	Plot de Sinal no Tempo . . . . .	60
4.5.2	Plot do Espectro do Sinal . . . . .	61
4.5.3	<i>Distortion Filter</i> . . . . .	63
4.5.4	Filtragem . . . . .	65
4.5.5	Chorus . . . . .	66
4.5.6	Tremolo . . . . .	68
4.6	Aplicativo de Efeitos de Áudio Unity 3D . . . . .	69
4.7	Validação . . . . .	73
<b>5</b>	<b>Conclusões e Sugestões de Trabalhos Futuros</b>	<b>75</b>
5.1	Conclusões . . . . .	75
5.2	Sugestões de Trabalhos Futuros . . . . .	76
<b>Referências</b>		<b>78</b>
<b>Apêndice A – Teste Aplicado para Validação</b>		<b>82</b>
A.1	Teste . . . . .	82
A.2	Resolução . . . . .	83
A.2.1	Questão 1 . . . . .	83
A.2.2	Questão 2 . . . . .	83
A.2.3	Questão 3 . . . . .	84
A.2.4	Questão 4 . . . . .	85
<b>Apêndice B – Script completos das simulações de Séries de Fourier no Matlab</b>		<b>86</b>
B.1	Onda Quadrada . . . . .	86

B.2 Onda Triangular . . . . .	87
<b>Apêndice C – Scripts de Desenvolvimento do jogo</b>	<b>89</b>
C.1 Controle de Sinal $S_r$ . . . . .	89
C.2 Game Controller . . . . .	91
C.3 Plot de Sinal Esperado . . . . .	99
<b>Apêndice D – Script do aplicativo de áudio completo</b>	<b>101</b>
<b>Apêndice E – Artigo apresentado no COBENGE 2017</b>	<b>111</b>

# Listas de Figuras

2.1	Tela do jogo Ruckingenur II.	5
2.2	Exemplo de sinal periódico triangular.	6
2.3	Exemplo de sinal periódico quadrangular.	7
2.4	Forma de onda típica senoidal.	9
2.5	Diagrama de Blocos básico de um sistema de processamento digital de sinais.	9
2.6	Exemplo de amostragem, processamento e reconstrução de um sinal analógico.	10
2.7	Exemplo de espectro de frequências de um sinal analógico qualquer.	11
2.8	Exemplo de Espectro de um sinal cossenoidal.	12
2.9	Função de Transferência Filtro Passa-Baixas.	12
2.10	Função de Transferência Filtro Passa-Altas.	13
2.11	Curva característica do <i>Distortion</i> .	14
2.12	Efeito da distorção aplicado em um sinal senoidal de frequência 1 <i>kHz</i> e decaimento de amplitude.	15
2.13	Espectro de um sinal senoidal de frequência 1 <i>kHz</i> e decaimento de amplitude com efeito de distorção.	15
2.14	Diagrama de Blocos Tremolo.	16
2.15	Exemplo de Tremolo com $f_{LFO} = 20 \text{ Hz}$ .	17
2.16	Diagrama de Blocos Chorus.	18
2.17	Tela de áudios Reaper.	19
2.18	Exemplo de <i>Game Object</i> e suas componentes.	21
2.19	Cena com a prévia do jogo desenvolvido.	21
2.20	Esquemático da variável “c” declarada em C# como um objeto Conta.	24

3.1	Etapas da Metodologia do trabalho. . . . .	25
3.2	Exemplo de desenho do gráfico da função $f(x) = x^2$ em MatLab. .	26
3.3	Diagrama de Blocos Genérico do Jogo. . . . .	29
3.4	Máquina de estados com cada nível do jogo. . . . .	30
3.5	Exemplo de Diagrama de Blocos realizado no Google Desenhos com uso da fonte Press Start. . . . .	31
3.6	Captura de tela do ambiente de desenvolvimento do jogo. . . . .	32
3.7	Comparativo da função Normalizada e Não Normalizada. . . . .	33
3.8	Chorus simulado via Simulink. . . . .	35
3.9	Configurações dos blocos para simulação do Chorus no Simulink. .	36
3.10	Diagrama de Blocos do aplicativo de áudios. . . . .	43
3.11	Lógica de aplicação dos efeitos sonoros. . . . .	45
4.1	Representações em Séries de Fourier de uma Onda Quadrada, de amplitude 2 e período 4. . . . .	49
4.2	Representações em Séries de Fourier de uma Onda Triangular, de amplitude 2 e período 4. . . . .	49
4.3	Gráfico $f(x) = x$ plotado na Unity. . . . .	50
4.4	Gráficos plotados na Unity. . . . .	51
4.5	Gráficos plotados na Unity com alteração de parâmetros em tempo real. . . . .	52
4.6	Telas principais de seleção do jogo. . . . .	53
4.7	Tela de início do nível 1.6 do Jogo. . . . .	53
4.8	Tela de jogo do nível 1.6. . . . .	54
4.9	Diagrama de Blocos sendo exibido no nível 1.6. . . . .	54
4.10	Telas de acerto e erro exibida em cada nível. . . . .	55
4.11	Efeito do Tremolo simulado com o uso do Matlab. . . . .	56
4.12	Áudios no tempo com efeito de Distorção desenvolvido no Matlab.	57
4.13	Espectros dos áudios com efeito de Distorção desenvolvido no Matlab.	58

4.14 Simulações via Simulink do Chorus, com Delay = 441, Depth = 100, e Rate variável. . . . .	59
4.15 Simulações via Simulink do Chorus, com Delay = 441, Depth = 10, e Rate variável. . . . .	60
4.16 Plot de sinal tonal na Unity, com frequência igual a 440 $Hz$ , e 512 pontos. . . . .	60
4.17 Plot de sinal tonal na Unity, com frequência igual a 10 $kHz$ , e 512 pontos. . . . .	61
4.18 Espectro de sinais tonais na Unity, com frequência igual a 10 $kHz$ e 440 $Hz$ , e 512 pontos. . . . .	62
4.19 Espectro do áudio de onda quadrada na Unity, com frequência igual a 440 $Hz$ , e 512 pontos. . . . .	62
4.20 Comparativo do efeito da Distorção da Unity com simulação no Matlab. . . . .	63
4.21 Comparativo do efeito da Distorção da Unity com simulação no Matlab. . . . .	64
4.22 Comparativo do efeito da Distorção da Unity com simulação no Matlab. . . . .	64
4.23 Comparativo de som proveniente de um trecho de guitarra com e sem filtragem passa-baixa. . . . .	65
4.24 Comparativo de som proveniente de um trecho de guitarra com e sem filtragem passa-alta. . . . .	66
4.25 Comparativo do espectro com Chorus extraído do Simulink e Unity, aplicado a um sinal tonal de 440 $Hz$ , $Delay = 441$ , $Depth = 100$ , $Rate = 1 Hz$ . . . . .	67
4.26 Comparativo do espectro com Chorus extraído do Simulink e Unity, aplicado a um sinal tonal de 440 $Hz$ , $Delay = 441$ , $Depth = 100$ , $Rate = 5 Hz$ . . . . .	67
4.27 Comparativo do espectro com Chorus extraído do Simulink e Unity, aplicado a um sinal tonal de 440 $Hz$ , $Delay = 441$ , $Depth = 100$ , $Rate = 10 Hz$ . . . . .	68
4.28 Comparativo do Tremolo desenvolvido na Unity com a simulação em Matlab. . . . .	69

4.29 Interface principal do aplicativo de áudio. . . . .	69
4.30 Áudio tonal de frequência igual a $440\ Hz$ sendo tocado, com es- pectro exibido. . . . .	70
4.31 Áudio tonal de frequência igual a $440\ Hz$ sendo tocado, com sinal no tempo exibido. . . . .	70
4.32 Seleção de áudios, com o um arquivo de onda quadrada e frequência de $440\ Hz$ . . . . .	71
4.33 Efeito da distorção aplicado ha um sinal tonal de frequência $440\ Hz$ .	71
4.34 Audio tonal de frequência igual a $440\ Hz$ sendo tocado, com di- torção e FPB ativados. . . . .	72
4.35 Comparativo de quantidade de acertos por questão das Turmas com e sem o uso do <i>software</i> . . . . .	74

# **Lista de Tabelas**

4.1 Desempenho da Turma Sem <i>Software</i> . . . . .	73
4.2 Desempenho da Turma Com <i>Software</i> . . . . .	73

# 1 Introdução

O mercado de jogos digitais vem crescendo cada vez mais, com um rendimento mundial aproximadamente igual a 101,1 bilhões de dólares no ano de 2016 (MCDONALD, 2017). Dessa forma, os jogos digitais se mostram cada vez mais presentes no dia a dia, atraindo um público de todas as idades. Por toda essa popularidade, e suas características que permitem aos usuários resolverem problemas, divertir-se e interagir, a utilização de jogos na educação pode ser muito benéfica.

Existem alguns componentes básicos que são comumente encontrados em um jogo digital, como papel ou personagem, regras, metas e objetivos, quebra-cabeças, problemas ou desafios, história, interações do jogador, estratégias e feedbacks (SAVI R.; ULRICH, 2008). Cada um deles possui sua importância de acordo com os objetivos do projeto. Por exemplo, há jogos que possuem um desafio elevado, porém sem nenhuma história. Em compensação, existem outros que possuem um nível de dificuldade praticamente igual a zero, mas a história contada pelo jogo é mais importante.

Sabendo disso, os jogos proporcionam alguns benefícios educacionais, como o desenvolvimento de habilidades cognitivas, aprendizado por descoberta, experiência de novas identidades, socialização, coordenação motora, facilitador de aprendizado e motivacional (SAVI R.; ULRICH, 2008).

O tópico Séries de Fourier é lecionado no curso de Engenharia Elétrica da Universidade Estadual de Londrina (UEL) na disciplina de Circuitos Elétricos 1, oferecida no segundo ano do curso. Depois, este conceito é aprofundado em outras disciplinas ao longo da graduação. É indiscutível a relevância deste assunto para a formação do engenheiro eletricista, já que é aplicado nas áreas de Telecomunicações, Processamento de Sinais, Circuitos Eletrônicos, Controle e Automação, entre outras.

O ensino unificado da prática com a teoria é o principal fator de formação presente nas escolas de engenharia. Sabendo que o profissional deve ser capaz de

aplicar a ciência, para solução de problemas e criação, as grandes instituições de ensino promovem a produção científica e tecnológica (AGUIAR et al., 2016). Desse modo, é importante que o assunto Séries de Fourier também seja acompanhado de uma aplicação prática.

Assim, este trabalho relata a confecção de um jogo digital para usufruir das suas vantagens na educação. Além disso, o projeto também apresenta um aplicativo de áudio, que simula efeitos de som, em conjunto com desenhos no tempo e frequência. Este último realiza a conexão da prática com a teoria, abordando um tema que está presente no dia a dia de todos, no qual existe muita engenharia envolvida.

Além deste primeiro capítulo, o trabalho está dividido em outros 4, mais os apêndices. O segundo capítulo, Revisão Bibliográfica, apresenta os principais temas pesquisados, e que foram fundamentais para o desenvolvimento do projeto. O terceiro, Metodologia, mostra os principais recursos da Unity 3D que foram aplicados no projeto, e como foram utilizados, além de abordar as simulações em Matlab. O capítulo 4 apresenta os principais resultados coletados com o projeto. Por fim, o último, mostra as conclusões e sugestões para trabalhos futuros.

## 1.1 Objetivos

### 1.1.1 Objetivos Gerais

O objetivo é propiciar uma interface simples e visual, para que o aluno resolva, de forma lúdica, problemas relacionados a Séries de Fourier, assunto relevante para a formação do engenheiro eletricista. E também explore os efeitos sonoros, como filtragem, Distorção, Chorus e Tremolo.

### 1.1.2 Objetivos Específicos

Os objetivos específicos são apresentados a seguir:

- Revisão Bibliográfica a respeito de jogos digitais na educação, linguagem de programação C#, ferramenta Unity 3D, Séries de Fourier e efeitos de áudio.
- Simulação, em Matlab, de representação de sinais periódicos em Séries de Fourier.

- Plotagem de gráficos na Unity 3D.
- Alteração do gráfico em tempo real na Unity 3D.
- Simulação dos efeitos Chorus, distorção e Tremolo em Matlab.
- Estudo e implementação das componentes de áudio presentes na Unity 3D.
- Validação do *software* para turma de Circuitos Elétricos 1.

## 2 Revisão da Literatura

Neste capítulo serão apresentadas as revisões bibliográficas, que foram cruciais para o desenvolvimento do trabalho. Dentro dos tópicos abordados aqui, estão jogos digitais utilizados na educação, principalmente em engenharia, a teoria acerca das Séries Trigonométricas de Fourier, áudios e seus efeitos, o *software* Unity 3D, e a programação em C#.

### 2.1 Jogos Digitais na Educação

Os jogos possuem uma enorme importância cultural para a população, dessa forma muitos desenvolvedores e pesquisadores estão voltando seus esforços para utilizar destas atividades em prol da educação.

No Congresso Brasileiro de Educação em Engenharia(COBENGE) são mostrados alguns trabalhos a respeito de jogos educativos na Engenharia. Foram encontrados alguns artigos sobre o assunto nas edições 2014 e 2011 do evento, tanto com jogos analógicos quanto digitais.

Publicado no COBENGE como “O Jogo do Barco”, é um jogo analógico que simula a linha de produção de um barco (PACCOLA et al., 2014). Outro trabalho apresentado, foi a respeito do desenvolvimento de um jogo para plataforma mobile, que tem o objetivo de educar jovens do ensino médio a respeito de ecossistema e ciência no geral (RODRIGUES et al., 2014). Além desses jogos, foram encontrados outros para aprendizagem de tratamento de água (SCHEGOCHESKI et al., 2014), gestão de projetos (NAVARRO M.P.; MARQUES, 2014), engenharia de software(CHAVES et al., 2011), e mecânica vetorial (MORSCH I.B.; ROCHA, 2011).

A *engenharia elétrica* parece um tema pouco recorrente no mercado de jogos. Existem trabalhos desenvolvidos para o aprendizado de engenharia civil (HEADUP GAMES, 2017), engenharia aero-espacial (SQUAD, 2017), lógica e matemática (NINTENDO, 2017), entre outros.

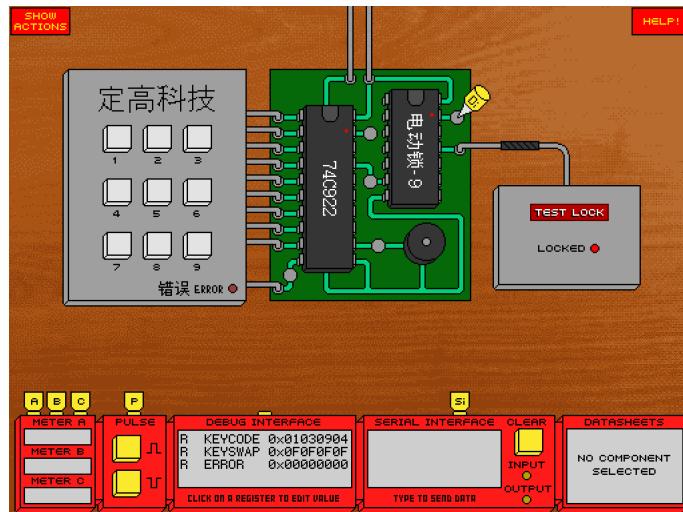
Porém, ainda assim existem jogos que propõem um aprendizado em Engenharia Elétrica. A empresa *Oniria Software* desenvolve o simmaq 3D, um aplicativo didático para simulação e capacitação em Controladores Lógicos Programáveis (CLP's) (ONIRIA, 2016).

Outra produtora, a Zachtronics, possui um portfólio com jogos educativos, que incluem temáticas de eletrônica e programação. Um exemplo é o TIS-100, com a jogabilidade de quebra-cabeças, tem o objetivo de educar os usuários a programar em *Assembly* (ZACHTRONICS, 2015).

A mesma produtora também disponibiliza alguns jogos gratuitos com sobre Engenharia Elétrica, que é o caso do jogo KOHCTPYKTOP, que consiste em implementar circuitos digitais para cumprimento dos objetivos, tendo que construir as dopagens P e N de cada componente (ZACHTRONICS, 2009).

Outro jogo gratuito é o Ruckingenur II, que aborda a engenharia reversa. O jogador deve analisar algumas placas de circuitos e tomar ações para cumprimento do objetivo. O jogo apresenta uma interface bastante simples, e deixa a disposição ferramentas para que as atividades sejam cumpridas, como multímetros, *datasheets* entre outros (ZACHTRONICS, 2008). A Figura 2.1 mostra uma tela do jogo.

**Figura 2.1:** Tela do jogo Ruckingenur II.



**Fonte:** (ZACHTRONICS, 2008)

Por fim, o jogo Lâmpadas tem como objetivo auxiliar no aprendizado de circuitos elétricos. Com uma interface bem simples, disponibiliza alguns componentes como gerador e fusível, para que o jogador ascenda as lâmpadas com conhecimento de circuitos (LIMA, 2015).

## 2.2 Séries Trigonométricas de Fourier

A Série de Fourier é uma forma de representar um sinal periódico não senoidal, na soma de senos e cossenos (NILSSON J. W.; RIEDEL, 2009). Caso uma série formada por este somatório seja convergente, é denominada de Série Trigonométrica de Fourier (LOTUFO, 2014). A Equação 2.1 mostra a forma de uma Série Trigonométrica de Fourier aplicada a uma função  $f(x)$ ,

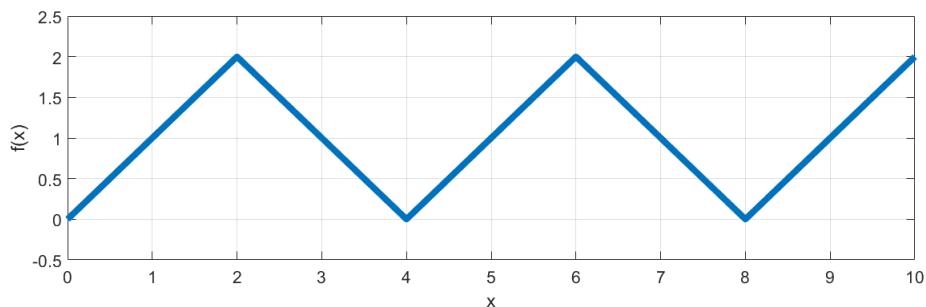
$$f(x) = \frac{a_0}{2} + \sum_{m=1}^{\infty} \left[ a_m \cdot \cos\left(\frac{m\pi x}{L}\right) + b_m \cdot \sin\left(\frac{m\pi x}{L}\right) \right]. \quad (2.1)$$

Os termos  $a_0$ ,  $a_m$  e  $b_m$  são denominados de coeficientes de Fourier, e são característicos da função  $f(x)$  representada (NILSSON J. W.; RIEDEL, 2009). Sabendo que  $T$  é o período da função, as senoides e cossenoides da Equação 2.1, para qualquer  $m \in \mathbb{N}^*$ , possuem período fundamental equivalente a  $T = \frac{2L}{m}$ . Ou seja, para  $m = 1$ , o parâmetro  $L$  pode ser calculado como sendo (LOTUFO, 2014):

$$L = \frac{T}{2}. \quad (2.2)$$

Na Engenharia Elétrica, geralmente há a necessidade de processar sinais elétricos periódicos, sendo a aplicação de Séries de Fourier primordial para estes trabalhos. Basicamente, este importante conceito é utilizado para sintetizar e facilitar a resposta em frequência de funções periódicas, tais como ondas quadradas, triangulares e dentes de serra. A Figura 2.2 mostra um exemplo de função periódica triangular.

**Figura 2.2:** Exemplo de sinal periódico triangular.



**Fonte:** Autoria Própria

Para que um função  $f(x)$  possa ser representada por uma série de potências, ela deve ser infinitamente derivável, e a fórmula de Taylor deve possuir resto tendendo para zero. Assim, para uma série trigonométrica, também deve-se analisar sua convergência (LOTUFO, 2014).

Dessa forma, para representar um sinal periódico ( $f(x) = f(x+T)$ ) através de uma Série de Fourier, deve-se calcular os coeficientes  $a_m$  e  $b_m$  vistos na Equação 2.1, os quais são calculados utilizando as relações:

$$a_m = \frac{1}{L} \cdot \int_{-L}^L f(x) \cos\left(\frac{m\pi x}{L}\right) dx \quad (2.3)$$

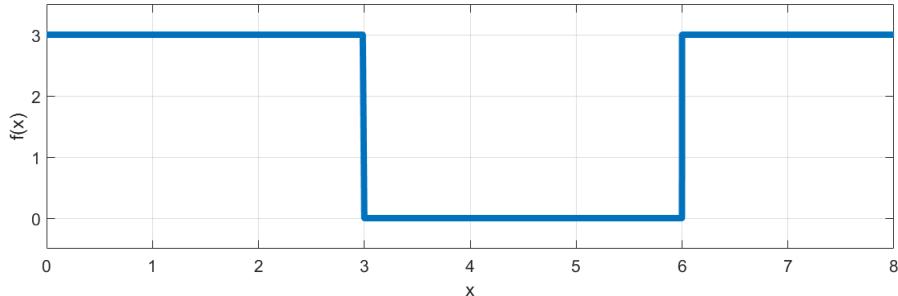
e

$$b_m = \frac{1}{L} \cdot \int_{-L}^L f(x) \sin\left(\frac{m\pi x}{L}\right) dx, \quad (2.4)$$

onde  $m = 1, 2, 3, \dots$

Como exemplo, será descrito o processo para encontrar a Série de Fourier correspondente a uma quadrada de período 6 e amplitude 3, mostrada na Figura 2.3.

**Figura 2.3:** Exemplo de sinal periódico quadrangular.



**Fonte:** Autoria Própria

Primeiramente, deve-se equacionar o sinal periódico em uma  $f(x) = f(x+T)$ . Sabendo que o período é igual a 6, a função pode ser definida como:

$$f(x) = \begin{cases} 0 & se -3 \leq x \leq 0 \\ 3 & se 0 \leq x \leq 3 \end{cases} \quad (2.5)$$

Neste caso, o parâmetro  $L$  corresponde à metade do período, ou seja  $L = 3$ . Assim, os parâmetros são calculados utilizando as Equações 2.3 e 2.4,

$$a_0 = \frac{1}{3} \cdot \int_{-3}^3 f(x) \cdot \cos(0) dx = 3, \quad (2.6)$$

$$a_m = \frac{1}{3} \cdot \int_{-3}^3 f(x) \cdot \cos\left(\frac{m \cdot \pi \cdot x}{3}\right) dx = 0 \quad (2.7)$$

e

$$b_m = \frac{1}{3} \cdot \int_{-3}^3 f(x) \cdot \sin\left(\frac{m \cdot \pi \cdot x}{3}\right) dx = \frac{3}{m \cdot \pi} \cdot [1 - \cos(m \cdot \pi)]. \quad (2.8)$$

Disto, os parâmetros calculados são,

$$a_0 = 3, \quad (2.9)$$

$$a_m = 0, \quad m = 1, 2, 3, \dots \quad (2.10)$$

e

$$b_m = \begin{cases} 0, & m \text{ par} \\ \frac{6}{m \cdot \pi}, & m \text{ ímpar.} \end{cases} \quad (2.11)$$

Portanto, a função  $f(x)$  da Figura 2.3 e Equação 2.5 pode ser representada pela série:

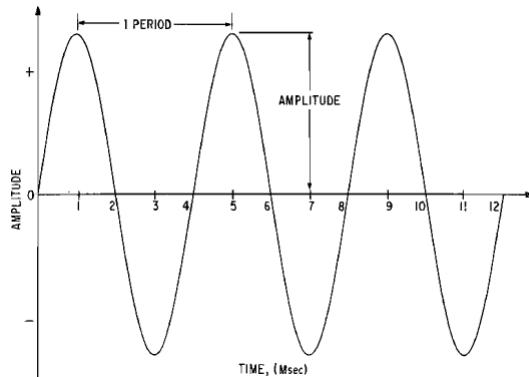
$$f(x) = 1,5 + \frac{6}{\pi} \cdot [\sin\left(\frac{\pi \cdot x}{3}\right) + \frac{1}{3} \cdot \sin\left(\frac{3 \cdot \pi \cdot x}{3}\right) + \frac{1}{5} \cdot \sin\left(\frac{5 \cdot \pi \cdot x}{3}\right) + \dots]. \quad (2.12)$$

Observe que existe uma lei de formação do sinal periódico de onda quadrada em Série de Fourier, onde este sinal é composto pelo somatório apenas de senoides, com decréscimo de amplitude e acréscimo de frequência apenas em termos ímpares. Ou seja, a soma possui seu índice  $m$  assumindo apenas valores ímpares.

## 2.3 Áudios e seus efeitos

A música está presente no dia a dia do homem desde os primórdios. A evolução tecnológica que envolve o som é nítida, se no começo os instrumentos não passavam de paus e pedras. Atualmente existem aparelhos complexos, bem como acessórios que aprimoraram a qualidade sonora (CHAMBERLIN, 1987).

O som é composto por 2 parâmetros principais, amplitude e frequência. O primeiro representa o volume, enquanto o segundo a tonalidade (PATSKO, 2015). A Figura 2.4 mostra os parâmetros em uma forma de onda típica, senoidal.

**Figura 2.4:** Forma de onda típica senoidal.

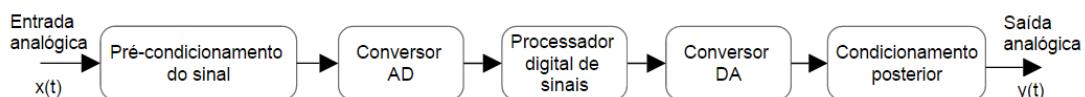
**Fonte:** (CHAMBERLIN, 1987)

O ouvido humano é capaz de ouvir sons com frequências entre  $20\ Hz$  e  $20\ kHz$ . Quando um determinado som é escutado, sua frequência caracteriza um tom. No entanto, ambos os parâmetros são diferentes, a frequência representa uma propriedade física, o tom tem origem subjetiva, que existe apenas na mente de quem escuta (CHAMBERLIN, 1987).

Como o som é a mudança de pressão de ar em uma determinada frequência, ao ser recebido pelo ouvido, ocasiona vibrações na membrana do tímpano, que em seguida é convertido em impulsos neurais (SMITH, 1999). A amplitude é a mudança de pressão percorrida no ar, enquanto que em um circuito eletrônico, este parâmetro é representado por tensão ou corrente (CHAMBERLIN, 1987).

### 2.3.1 Processamento Digital de Sinais de Áudio

As etapas básicas de um processamento digital de sinais são o condicionamento prévio, a conversão AD (Analógico - Digital), o processamento, a conversão DA (Digital - Analógico), e por fim o condicionamento posterior (Figura 2.5).

**Figura 2.5:** Diagrama de Blocos básico de um sistema de processamento digital de sinais.

**Fonte:** (PATSKO, 2015)

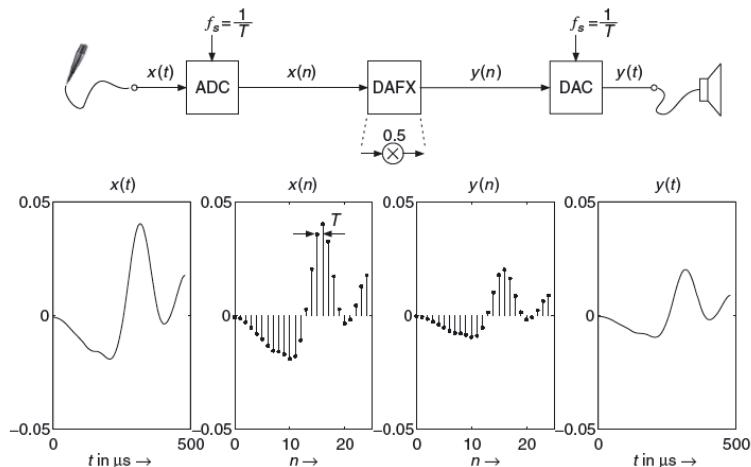
O condicionamento do sinal, que é realizado tanto no início, quanto no fim do sistema, são basicamente operações de filtragem, amplificação, entre outras. Com objetivo de melhorar a qualidade do sinal, para ser processado ou emitido (NATIONAL INSTRUMENTS, 2012).

A conversão AD deve seguir uma frequência de amostragem ( $F_s$ ) de acordo com a Taxa de Nyquist (Equação 2.13). Seja um sinal de entrada analógico com uma largura de banda  $F_B$ , a taxa para amostrar o sinal, e convertê-lo digitalmente, deve ser maior que o dobro de  $F_B$ . Assim, a informação do sinal não corre o risco de ser perdida (ZÖLZER, 1999). Como o ouvido humano consegue ouvir frequências até  $20\text{ kHz}$ , arquivos de áudio, geralmente possuem  $F_s = 44.100\text{ Hz}$ .

$$F_s > 2F_B \quad (2.13)$$

Já o processamento do sinal pode ser um efeito aplicado, uma função de atenuação, leitura de dados, entre outros. A Figura 2.6 mostra um exemplo básico de processamento, no caso o sinal é amostrado e digitalizado, atenuado pela metade, e convertido para analógico novamente.

**Figura 2.6:** Exemplo de amostragem, processamento e reconstrução de um sinal analógico.

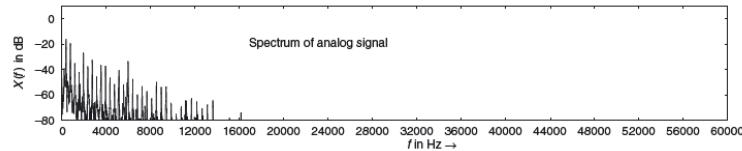


**Fonte:** (ZÖLZER, 2011)

### 2.3.2 Espectro de Frequências

A distribuição de energia, conforme a faixa de frequências de um sinal, é denominada de espectro (ZÖLZER, 2011). A Figura 2.7 mostra um exemplo proveniente de um sinal analógico. Note que a potência está distribuída em uma faixa de frequências entre 0 e  $16\text{ kHz}$ .

**Figura 2.7:** Exemplo de espectro de frequências de um sinal analógico qualquer.



**Fonte:** (ZÖLZER, 2011)

Como já apresentado, um sinal periódico é representado por um somatório de senoides e cossenoides, definido pelas Séries de Fourier. Esta representação, quando atribuída em forma de uma integral, é definida como Transformada de Fourier (OPPENHEIM A. V.; SCHAFER, 2010). Seja um sinal  $x(t)$  contínuo, sua Transformada de Fourier ( $X(j\omega)$ ) é representada por:

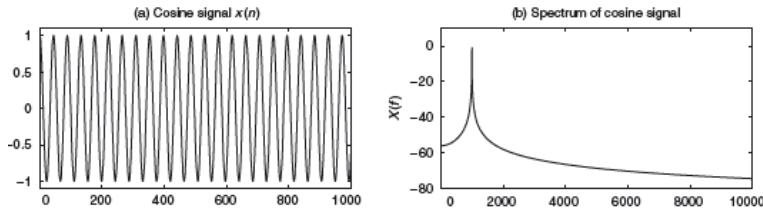
$$X(j\omega) = \int_{-\infty}^{\infty} x(t)e^{-j\omega t} dt. \quad (2.14)$$

Para sinais periódicos, a representação ocorre de forma que as exponenciais complexas possuam amplitudes relacionadas aos coeficientes  $a_m$  e  $b_m$  da Série de Fourier, o que resulta em um conjunto discreto, dispostos nas frequências das harmônicas ( $m = 0, 1, 2, \dots$ ). Dessa forma, o módulo Transformada de Fourier é denominada como espectro de  $x(t)$ , pois descreve o sinal  $x(t)$  em sinais senoidais de diferentes frequências (OPPENHEIM A. V.; SCHAFER, 2010).

Para sinais digitais, a representação é dada pela transformada Discreta de Fourier. Diferentemente do tempo contínuo, esta é escrita como um somatório e não como uma integral (ZÖLZER, 2011). Seja um sinal digital ( $x[n]$ ), a transformada é dada por (OPPENHEIM A. V.; SCHAFER, 2010):

$$x(e^{j\omega}) = \sum_{n=-\infty}^{\infty} x[n]e^{-j\omega n}. \quad (2.15)$$

Veja o exemplo da Figura 2.8, onde mostra um sinal cossenoidal amostrado, e seu espectro de frequências. Como para um sinal do tipo não existem harmônicas, o espectro é definido apenas com um pico em sua frequência.

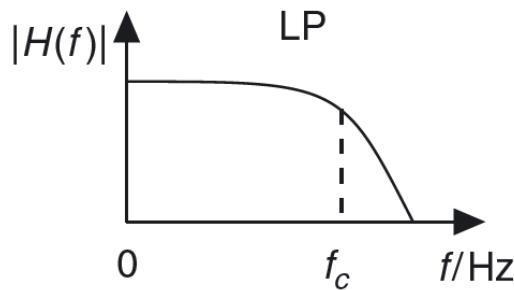
**Figura 2.8:** Exemplo de Espectro de um sinal cossenoidal.

**Fonte:** (ZÖLZER, 2011)

### 2.3.3 Filtros

De maneira geral, filtragem significa que a partir de um conjunto grande de elementos, deseja-se selecionar apenas alguns com alguma determinada característica. Sabendo que um sinal tem 2 parâmetros principais, frequência ( $f$ ) e amplitude ( $A$ ), o filtro irá rejeitar uma específica faixa de frequências, e selecionar apenas o desejado. De outra forma, ocorre a atenuação das amplitudes nas frequências em que se deseja eliminar (ZÖLZER, 2011).

Neste trabalho foram abordados apenas 2 filtros mais básicos, Passa-Baixas (FPB) e Passa-Altas (FPA). O primeiro rejeita altas frequências a partir de uma frequência de corte ( $f_c$ ), selecionando apenas abaixo deste valor (ZÖLZER, 2011). O gráfico da função de transferência ( $|H(f)|$ ) do filtro em relação à frequência é mostrada pela Figura 2.9.

**Figura 2.9:** Função de Transferência Filtro Passa-Baixas.

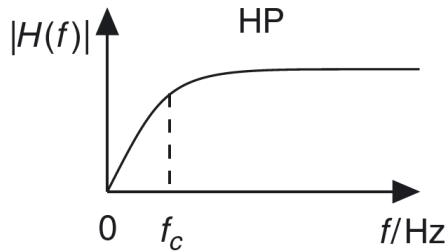
**Fonte:** (ZÖLZER, 2011)

De modo geral, um sistema que apresenta uma função de transferência conforme a Equação 2.16, onde  $\omega_c = 2\pi f_c$  e  $s = j\omega$ , possui um papel de Filtro Passa-Baixas (NILSSON J. W.; RIEDEL, 2009).

$$H_{FPB}(s) = \frac{\omega_c}{s + \omega_c} \quad (2.16)$$

O Filtro Passa-Altas tem função inversa que o FPB, ou seja, rejeita as baixas frequências, e seleciona apenas as altas, a partir de uma frequência de corte ( $f_c$ ) (ZÖLZER, 2011). O gráfico da Função de Transferência deste filtro no domínio da frequência esta disposta na Figura 2.10.

**Figura 2.10:** Função de Transferência Filtro Passa-Altas.



**Fonte:** (ZÖLZER, 2011)

O comportamento de um FPA pode ser escrito conforme a relação:

$$H_{FPA}(s) = \frac{s}{s + \omega_c}, \quad (2.17)$$

que representa sua função de transferência.

Existem diversos outros tipos de filtro, como Passa-Faixa, Rejeita-Faixa, entre outros. Além de que, para o projeto do sistema, pode-se levar em consideração várias topologias, ordem e outras características importantes. No entanto, para este projeto foram considerados apenas os filtros básicos, sem a preocupação com as considerações específicas de projeto.

### 2.3.4 Distorção

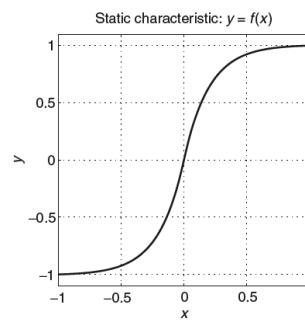
A distorção é um efeito de áudio Não-Linear, que consiste na criação de harmônicas que não estão presentes no sinal original (ZÖLZER, 2011). Existem basicamente 3 tipos deste efeito: o *Overdrive* que opera na região linear para baixas amplitudes, e não-linear para altas, cujo resultado é um som mais suave. O *Distortion* ou Distorção, que trabalha principalmente na região não-linear, resultando em um som pesado, muito utilizado em músicas de Metal e Grunge (ZÖLZER, 2011). Por fim o *Fuzz*, que consiste em uma forte clipagem do sinal, mais especificamente em suas extremidades, resultando em um som sujo e ruidoso (FALCÃO, 2015).

Neste trabalho foi abordado apenas o *Distortion*, cuja curva característica é dada pela função:

$$y = f(x) = \operatorname{sgn}(x) \cdot (1 - e^{-|x|}). \quad (2.18)$$

A Figura 2.11 mostra o gráfico da respectiva curva característica. Note que, o valor de  $x$  é o sinal de entrada, e  $y$  distorcido. Para pequenas amplitudes, o efeito é quase mínimo, resultando em um sinal muito próximo ao de entrada, já para valores maiores, a distorção ocorre de forma mais nítida. Para grandes valores de entrada, acontece uma saturação na distorção.

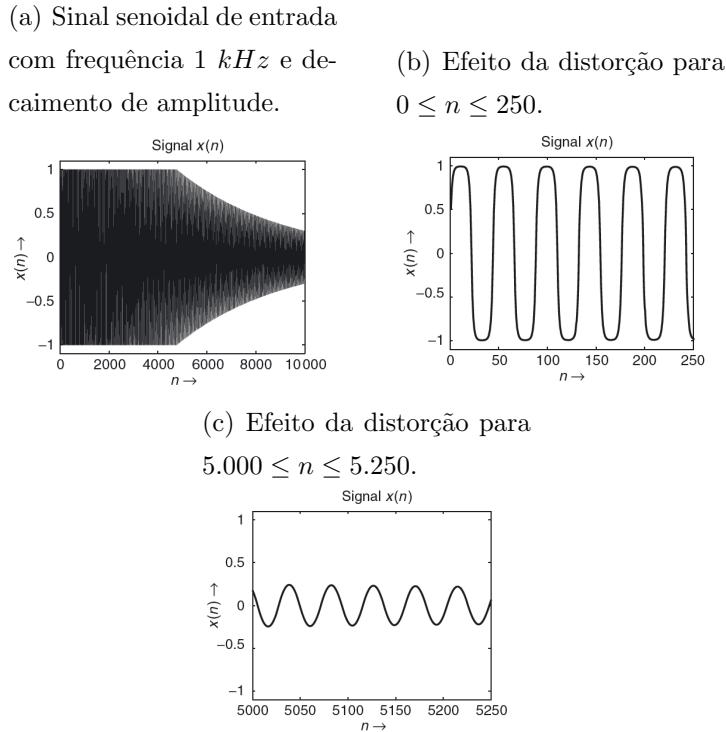
**Figura 2.11:** Curva característica do *Distortion*.



**Fonte:** (ZöLZER, 2011)

Com isso, o nível de distorção pode ser trabalhado com um Ganho ( $G$ ), que multiplica  $x$  e entra na função. Conforme o aumento de  $G$ , maior será o efeito, até a saturação. Observe o exemplo proposto por Zölzer (2011), em que o efeito é aplicado a um sinal senoidal com decaimento de amplitude (Figura 2.12).

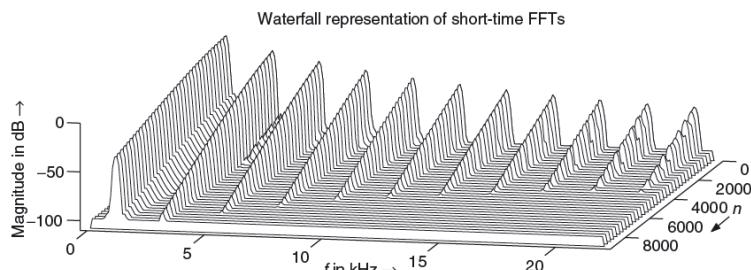
**Figura 2.12:** Efeito da distorção aplicado em um sinal senoidal de frequência 1 kHz e decaimento de amplitude.



**Fonte:** (ZÖLZER, 2011)

Para as primeiras amostras, onde a amplitude do sinal é alta, o efeito acaba sendo mais forte, já quando a amplitude é baixa ( $5.000 \leq n \leq 5.250$ ) o efeito é muito pequeno. Na Figura 2.12 b), o sinal possui uma deformação, que tende para uma onda quadrada em caso de saturação. Isso, faz com que em seu espectro harmônicas apareçam com decaimento de amplitude, localizadas nas frequências de produto ímpar com a fundamental. Isso pode ser visto pelo espectro da Figura 2.13.

**Figura 2.13:** Espectro de um sinal senoidal de frequência 1 kHz e decaimento de amplitude com efeito de distorção.

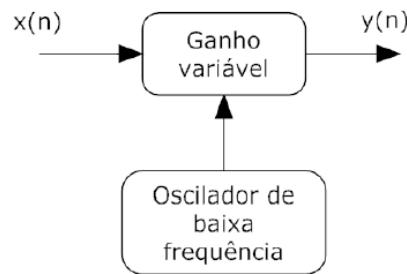


**Fonte:** (ZÖLZER, 2011)

### 2.3.5 Tremolo

Seja um sinal de entrada  $x[n]$ , o efeito do Tremolo consiste no produto deste com um sinal oscilador de baixa frequência (*Low Frequency Oscilator* ou LFO). É o mesmo que uma modulação por amplitude (AM) (PATSKO, 2015). Esta operação resulta em uma variação de amplitude do áudio de entrada, conforme a oscilação do LFO, gerando um efeito sonoro pulsante. A Figura 2.14 mostra o diagrama do efeito.

**Figura 2.14:** Diagrama de Blocos Tremolo

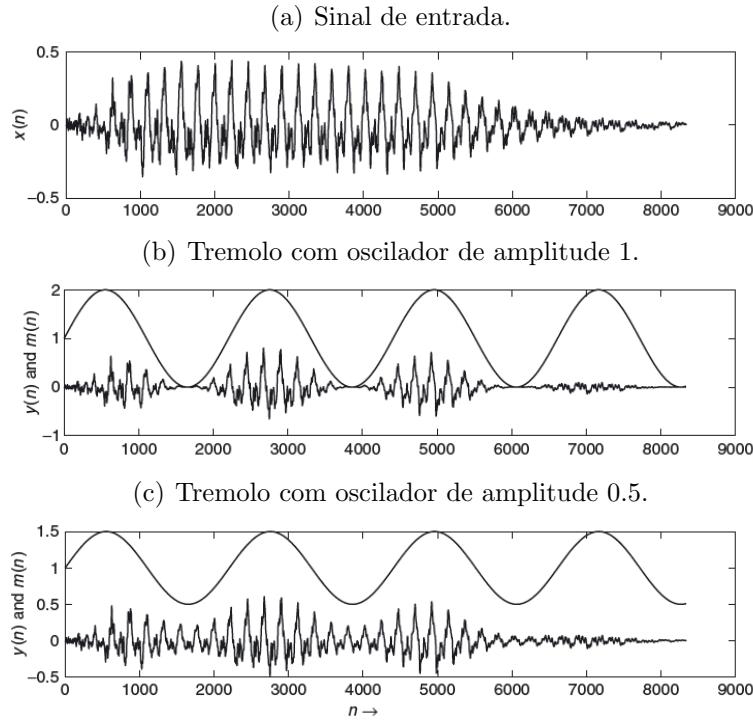


**Fonte:** (PATSKO, 2015)

Sabendo que o LFO, geralmente é uma onda senoidal, o efeito do Tremolo pode ser definido como a Equação 2.19. O sinal oscilador pode ser representado com outras funções além de uma senoide, como por exemplo, alterações no *offset* do LFO (PATSKO, 2015).

$$y[n] = x[n] \cdot \text{sen}(2 \cdot \pi \cdot n \cdot f_{LFO}/f_s) \quad (2.19)$$

Com a frequência de oscilação ( $f_{LFO}$ ) variando de 0 até 20 Hz, o efeito no domínio do tempo pode ser percebido, com mudanças de amplitude conforme a oscilação. Porém, com o aumento da frequência ocorrerá o aparecimento de outras componentes espectrais, o que altera o comportamento sonoro (ZÖLZER, 2011).

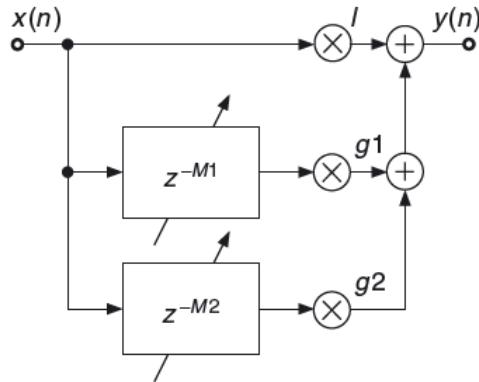
**Figura 2.15:** Exemplo de Tremolo com  $f_{LFO} = 20 \text{ Hz}$ .

**Fonte:** (ZÖLZER, 2011)

A Figura 2.15 a) representa um sinal sonoro de entrada. Já as Figuras 2.15 b) e c) mostram o sinal com efeitos de tremolo diferentes, com variações na amplitude do LFO. Note como a amplitude do sinal de entrada é alterada de acordo com o LFO. Na Figura 2.15 b), o sinal de entrada tem sua amplitude variada até 0, em uma frequência de  $20 \text{ Hz}$ . Na última (Figura 2.15 c)), esta variação é um pouco mais sutil, em que o sinal não possui sua amplitude nula em nenhum momento, contendo apenas uma pequena atenuação em uma frequência de  $20\text{Hz}$ .

### 2.3.6 Chorus

Como o próprio nome já diz, o Chorus simula um coro, de forma em que sejam adicionados em um som, outras cópias deste sinal com linhas de atraso variável. Isso gera um efeito de profundidade no áudio (PATSKO, 2015). O Diagrama de Blocos do efeito pode ser visto na Figura 2.16.

**Figura 2.16:** Diagrama de Blocos Chorus.

**Fonte:** (ZÖLZER, 2011)

A linha de atraso é composta de um parâmetro “*Delay*”, somado com um oscilador, que possui amplitude denominada como “*Depth*”, e frequência “*Rate*”. Ou seja, o Chorus é a soma de cópias ao sinal original, onde cada uma delas possui um atraso com variações em uma certa frequência (ZÖLZER, 2011). A alteração de fase para um sinal amostrado é dada por:

$$M1 = Delay + Depth \cdot \sin(2 \cdot \pi \cdot n \cdot Rate/f_s). \quad (2.20)$$

## 2.4 Reaper

Reaper é um *software* destinado para edição, mixagem, gravação, amostragem, e arranjo de áudios, entre outras funções. O sistema permite também que interfaces de *hardware* sejam conectadas, para uma melhor qualidade de trabalho (REAPER, 2017).

O manuseio do *software* é simples, basta arrastar um arquivo de áudio, ou realizar a gravação, para poder utilizar de seus recursos. São aceitos diversos tipos de arquivo, entre eles .wav e .mp3 (REAPER, 2017). A Figura 2.17 mostra uma tela de uso do *software*.

**Figura 2.17:** Tela de áudios Reaper.



Fonte: (REAPER, 2017)

Com os arquivos de áudio dispostos no *software*, é possível aplicar alguns *plugins*, que auxiliam no trabalho. Existem vários, como osciloscópio, efeitos, entre outros. O mais utilizado no projeto foi o “ReaFIR” para análise em frequência, que é um equalizador dinâmico baseado no espectro (REAPER, 2017).

## 2.5 Unity 3D

Unity é um *software* voltado para desenvolvimento de jogos, criado pela empresa de mesmo nome. Também conhecido como Unity 3D, a ferramenta permite criar tantos jogos em 3D quanto em 2D, além de poder desenvolver-los para inúmeras plataformas, como celulares, tablets, computadores, web e videogames (UNITY, 2017s).

Essa ferramenta suporta 2 linguagens de programação, C# e JavaScript. O desenvolvimento de algum projeto utilizando a ferramenta se baseia na manipulação de objetos, ou seja, cada cena do jogo é composta por objetos que são dos mais variados tipos, como áudios, imagens, textos, botões, entre outros. Cada objeto é composto por componentes, que dão a cara ao objeto, por exemplo, um objeto de jogo que toca uma música, só consegue executar tal funcionalidade pois nele existe um componente específico para executar esta função. Além disso, também é possível anexar Scripts programados pelo desenvolvedor, para executar funções específicas, manipular objetos e componentes (UNITY, 2017l).

Todas as funções disponíveis para programação pertencem ao conjunto de funcionalidades denominado de MonoBehaviour (UNITY, 2017s). O site da empresa contém documentações, manuais e tutoriais a respeito de todas as funcionalidades do *software*.

A seguir, serão apresentados os principais conceitos utilizados para o desenvolvimento do projeto.

### 2.5.1 Ambiente de Desenvolvimento

O projeto de um jogo na Unity é subdividido em cenas, sendo que cada uma delas pode representar uma fase, um menu, entre outros. Inicialmente, cada cena inicia com uma câmera e luz (UNITY, 2017t). O primeiro é primordial para posicionar o campo de visão do jogador.

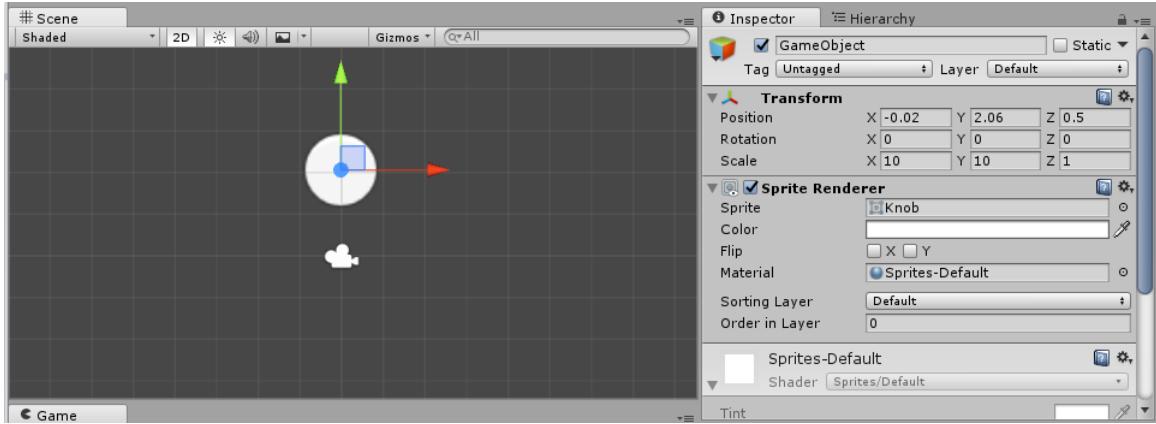
A partir da inicialização de uma nova cena, é possível criar e disponibilizar objetos, denominados de *Game Objects*. Estes podem ser quaisquer coisas no jogo, desde um personagem controlável a uma fonte sonora. Porém, cada um deles, para funcionar da forma desejada, devem ser adicionadas as componentes adequadas (UNITY, 2017m).

Existem diversos tipos de componentes disponíveis na Unity, e cada uma delas possui funções específicas. Ao criar um novo *Game Object*, por exemplo, a componente *Transform* é padrão a todos, que indica as posições x, y e z do plano 3D da cena, rotação e escala (UNITY, 2017n). Além disso, existem outras componentes com funcionalidades de áudio, texto, interação, efeitos especiais, entre outros.

Como já mencionado, os objetos são posicionados em coordenadas no plano da Unity. O projeto apresentado neste trabalho foi desenvolvido em 2D, portanto apenas as coordenadas x e y foram utilizadas.

A Figura 2.18 mostra um objeto posicionado no espaço bidimensional. Este *Game Object* contém componentes de posicionamento, e renderização de *Sprite*. Este último tem caráter gráfico, onde podem ser incluídos imagens, desenhos, entre outros.

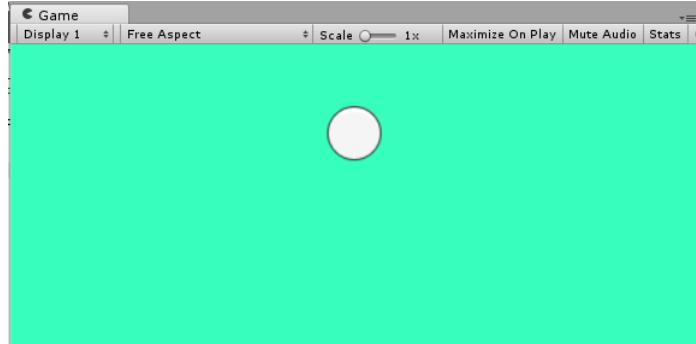
**Figura 2.18:** Exemplo de *Game Object* e suas componentes.



**Fonte:** Autoria Própria

O ambiente de desenvolvimento consta com 2 telas, uma com a cena (Figura 2.18), onde é possível realizar as manipulações, e trabalhar no projeto. E outra com uma prévia do jogo, ou seja, como será visto quando iniciar (Figura 2.19). Para testar o projeto, basta apertar um botão de *play* indicado.

**Figura 2.19:** Cena com a prévia do jogo desenvolvido.



**Fonte:** Autoria Própria

### 2.5.2 *Vector3*

O *Vector3* é um conceito bastante utilizado na Unity. Como o próprio nome já diz, é a representação de pontos e vetores 3D. Ou seja, é uma *struct* aplicada para posicionamento e direção (UNITY, 2017u). Basicamente sua representação é da forma  $\text{Vector3}(x,y,z)$ , e também pode ser declarado como um vetor, ou seja um conjunto com várias coordenadas.

### 2.5.3 *Void Start*

No início de cada cena, quando o determinado *Script* é inicializado, a função *Start* é chamada apenas uma única vez (UNITY, 2017q). Esta rotina é muito utilizada para atribuição de valores em varáveis, passagem de parâmetros, e inicializações.

### 2.5.4 *Void Update*

A função *Update* é chamada a cada *frame* do jogo. De maneira análoga, funciona basicamente como um *loop* embarcado de um microprocessador (UNITY, 2017r). Esta função é importante para o funcionamento de cada cena, pois as constantes atualizações possibilitam a programação de funções que acompanham cada *frame*, e seguem com o andamento do jogo.

### 2.5.5 *Button*

Evidentemente, os botões executam um determinado evento quando acionados. Sua programação segue um padrão, ele deve ser inicializado na função *Start*, onde é atribuída uma sub-rotina ao objeto, que será chamada quando o botão for apertado. Observe o exemplo abaixo retirado de (UNITY, 2017k).

```
public Button yourButton;

void Start()
{
    Button btn = yourButton.GetComponent<Button>();
    btn.onClick.AddListener(TaskOnClick);
}

void TaskOnClick()
{
    Debug.Log("You have clicked the button!");
}
```

## 2.6 C# e programação orientada a objetos

A linguagem C# foi lançada pela Microsoft em 2002, como resultado do projeto COOL (*C-like Object Oriented Languade*). A iniciativa partiu como uma maneira de trabalhar com diferentes linguagens e tecnologias existentes. Dessa forma, o C# compõem o ambiente .NET, desenvolvido por meio do projeto(CAELUM, 2013).

Geralmente, para iniciar um código em C# já existem blocos prontos, que só deve ser alocados para programação. Assim é necessária a inclusão das bibliotecas, provenientes do ambiente .NET. A mais comum é escrever a linha de código *using System*, que contém as funcionalidades mais usuais (MICROSOFT DOCS, 2015).

Os tipos de variáveis, estruturas de controle e repetição, são muito parecidos que a linguagem C, com mudanças muito sutis em algumas escritas. No entanto, o C# possibilita a programação por meio de classes e objetos, que muitas vezes torna o código mais organizado e eficiente.

O exemplo apresentado por Caelum (2013) mostra a criação de um objeto referente a uma conta bancária, sabendo que as informações principais de uma conta são número, titular e saldo. Inicialmente a classe é criada, onde são descritas as principais informações. O trecho de código é mostrado a seguir. Note que as variáveis são definidas como públicas (*Public*), que permite a leitura e escrita da classe.

```
class Conta
{
    // numero, titular e saldo são atributos do objeto
    public int numero;
    public string titular;
    public double saldo;
}
```

Para utilizar a classe deve-se escrever o comando *New*, que reserva memória para armazenagem dos dados. Por exemplo, a declaração de uma variável “c”, é implementada como o código a seguir (CAELUM, 2013).

```
private void button1_Click(object sender, EventArgs e)
{
    Conta c = new Conta();
```

{}

Basicamente, o comando *New* funciona como um ponteiro, que aponta para os espaços reservados na memória (CAELUM, 2013). A Figura 2.20 mostra um esquemático da lógica.

**Figura 2.20:** Esquemático da variável “c” declarada em C# como um objeto Conta.



**Fonte:** (CAELUM, 2013)

Para atribuir valores a qualquer um dos campos de `c`, basta utilizar um ponto seguido do termo, ou seja `c.numero = 100` por exemplo (CAELUM, 2013).

Existem muitos outros conceitos avançados em C#, como encapsulamento, herança, entre outros. Porém para este projeto não foi necessário um aprofundamento no assunto, pois a manipulação de objetos foi suficiente para aplicar as funções da Unity.

# 3 Metodologia

Neste capítulo serão apresentadas as metodologias e atividades desenvolvidas para cumprir os objetivos. Todo o projeto pode ser dividido conforme a Figura 3.1.

**Figura 3.1:** Etapas da Metodologia do trabalho.



**Fonte:** Autoria Própria

As primeiras atividades foram voltadas para estudo do *software* Unity 3D, assim como de conceitos acerca das Séries de Fourier e Efeitos de Áudio. Os estudos foram complementados de simulações via Matlab e Simulink, que possibilitaram o desenvolvimento do projeto como um todo.

## 3.1 Séries de Fourier no Matlab

Como uma série trigonométrica se baseia em um somatório infinito, a simulação computacional é a melhor maneira de verificar seu resultado na prática. Sabendo que o conteúdo em questão é o tema principal do jogo, a elaboração de simulações tornaram-se importantes para o desenvolvimento do projeto.

Para isso foi utilizado o *software* Matlab, onde foram programados 2 *Scripts*, um para simulação da representação de uma onda quadrada, e outro para triangular.

Do primeiro *Script* resulta o comparativo de um onda quadrada, de amplitude 2 e período 4, com o somatório de senoides e cossenoides (Equação 2.1), para diferentes limites de n. Para a função periódica proposta, sua representação em Séries de Fourier pode ser escrita como:

$$f(x) = 1 + \frac{4}{\pi} \cdot \sum_{n=0}^{\infty} \frac{1}{2n+1} \cdot \sin \left( \frac{(n+1) \cdot \pi \cdot x}{2} \right). \quad (3.1)$$

Dessa forma, foram plotados os gráficos de  $f(x)$  para  $2n+1 = 0, 3, 5, 7, 9$  e 1.000. A onda quadrada foi gerada automaticamente com a função *square* do Matlab. Todos os gráficos foram alocados em uma única imagem para comparação.

O mesmo foi realizado no segundo *Script*. Porém neste caso, o objetivo foi simular a representação de uma onda triangular, com amplitude 2 e período 4. Para esta função, a relação matemática de série trigonométrica é dada por:

$$f(x) = 1 + \frac{8}{\pi^2} \cdot \sum_{n=0}^{\infty} \frac{1}{(2n+1)^2} \cdot \cos \left( \frac{(2n+1)^2 \cdot \pi \cdot x}{2} \right). \quad (3.2)$$

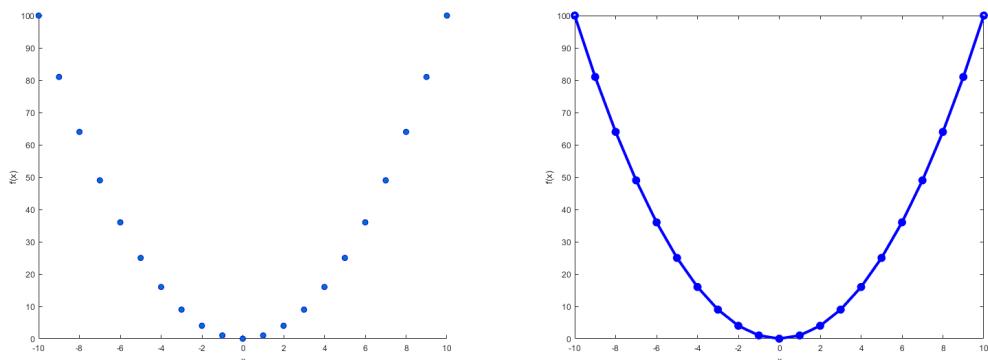
Os gráficos foram plotados com os mesmos valores de  $2n+1$  da primeira simulação. Estes resultados são apresentados na Seção 4.1. Ambos os códigos podem ser vistos no Apêndice B.

## 3.2 Gráficos Unity 3D

Seja uma função real  $f(x) = y$ , o princípio básico para plotar seu gráfico é a disposição de suas coordenadas  $(x, y)$  no plano cartesiano. Assim, estes pontos são conectados por retas, que resulta no gráfico. É importante ressaltar que a qualidade do desenho é diretamente proporcional à quantidade de pontos. A figura 3.2 mostra um exemplo de plotagem para função  $f(x) = x^2$ .

**Figura 3.2:** Exemplo de desenho do gráfico da função  $f(x) = x^2$  em MatLab.

(a) Pontos dispostos no Plano Cartesiano.      (b) Gráfico da função  $f(x) = x^2$ .



**Fonte:** Autoria Própria

Foi partindo deste princípio, que foi elaborado o *Script* para plotar gráficos na Unity 3D. Toda a programação foi feita através da manipulação e uso da componente *Line Renderer*, a qual recebe um vetor de 2 ou mais pontos, e em seguida desenha linhas entre cada um deles (UNITY, 2017o).

Foram declaradas 3 variáveis básicas, uma referente aos valores do eixo das abscissas (“*i*”). Outra para incremento do *Loop* (“*pos*”). E por fim, um *Vector 3* de dimensão 300 (“*positions*”).

O *Script* calcula, através do *Loop For*, os valores de  $f(x) = y$ , e os armazenam no *Vector3(x,y,z)* declarado. É importante notar, que os valore de *x* possuem um passo de 0,1 entre eles. Com os pontos calculados e armazenados, utiliza-se a função *LineRenderer.SetPositions*, que recebe o vetor de pontos, e seta na componente para desenhar as linhas (UNITY, 2017p). Este código foi baseado na própria documentação da Unity citada.

```
private LineRenderer lr;
float i = -7;
int pos = 0;
Vector3[] positions = new Vector3[300];
void Start () {
    lr = GetComponent<LineRenderer>();
    for(pos = 0 ; pos <300;pos++)
    {
        positions [pos] = new Vector3(i , 2* Mathf.Sin( i *
            Mathf.PI ) + 2 , 0.0f );
        i = i + 0.1f ;
    }
    lr . SetPositions ( positions );
}
```

Para atender os objetivos do projeto, o gráfico deve ser alterado em tempo real conforme mudanças de parâmetros da função  $f(x)$ . Assim, esta programação foi implementada de forma não bloqueante, dentro do *loop Void Update*. Isso possibilitou que a relação matemática fosse alterada em tempo real, conforme acesso público de variável, ou ação externa do usuário, como por exemplo, apertar um botão. Observe o código desenvolvido, onde  $f(x) = A * \sin(2\pi i)$ , e sua amplitude *A* é alterada com acesso público pelo *software*.

```
private LineRenderer lr;
float i = -7;
int pos = 0;
public float Amplitude;
float = yvalue
Vector3[] positions = new Vector3[300];
void Start () {
    lr = GetComponent<LineRenderer>();
}

void Update () {
    if (pos <= 200) {
        yvalue = A*Mathf.Sin(i*Mathf.PI)
        positions [pos] = new Vector3 (i, yvalue, 0.0f);
        i = i + 0.1f;
        pos++;
        if (pos == 200) {
            pos=0;
            i = -7;
        }
    }
    lr.SetPositions (positions);
}
```

## 3.3 Jogo Séries de Fourier

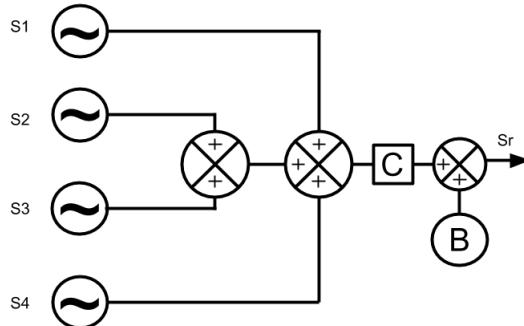
### 3.3.1 Estrutura do Jogo

A matemática muitas vezes pode ser abstrata para todos os alunos por diversos motivos. Um deles é a carência de algum recurso visual, que possibilita transformar toda fórmula e equação em algo mais visual. Foi partindo deste ponto que os autores do trabalho elaboraram as regras, desafios e conceitos do jogo.

De acordo com Schell (2008) um jogo pode ser definido como uma atividade de resolução de problemas, ligado com uma atitude lúdica. Partindo desta definição, pode-se construir o conceito do jogo, já que a formação de qualquer engenheiro está diretamente ligada com a capacidade de resolver problemas. Com isso, o jogo apresentado neste trabalho se propõe a deixar o aprendizado de forma lúdica.

A premissa do jogo apresentado neste trabalho é bastante simples, o jogador tem à sua disposição 4 sinais (Senos ou Cossenos), os quais serão somados, como mostra o diagrama de blocos da Figura 3.3. A partir destes recursos disponibilizados, o jogador pode ajustar a amplitude de cada sinal, e tem como objetivo sintetizar um sinal periódico proposto em um determinado período de tempo, seguindo o conceito de Séries de Fourier.

**Figura 3.3:** Diagrama de Blocos Genérico do Jogo.



**Fonte:** Autoria Própria

Os sinais a serem controlados podem ser representados por:

$$S_i = A_i \cdot \sin(2 \cdot \pi \cdot f_i), \quad i = 1, 2, 3 \text{ e } 4, \quad (3.3)$$

onde a amplitude pode ser ajustada. Considerando a Equação 3.3, a representação matemática do diagrama de blocos da Figura 3.3 será:

$$S_r = \left[ \sum_{i=1}^4 A_i \cdot \sin(2 \cdot \pi \cdot f_i) \right] \cdot C + B. \quad (3.4)$$

Fica evidente que o sinal resultante não é exato em comparação com o requerido. Entretanto, com 4 sinais, ou  $m = 4$ , o resultado é aproximado, e é possível entender o processo como um todo. O jogo plota o gráfico da Equação 3.4 em tempo real conforme as alterações das amplitudes. Basta o jogador apertar um botão para verificar o resultado.

O jogo propõe ao aluno o desafio de transformar os sinais iniciais em um outro requerido, e também permite que ele analise padrões na Série de Fourier, como o obtido pela Equação 2.1. Além disso, possibilita a percepção visual para entender o processo de representação de um sinal periódico.

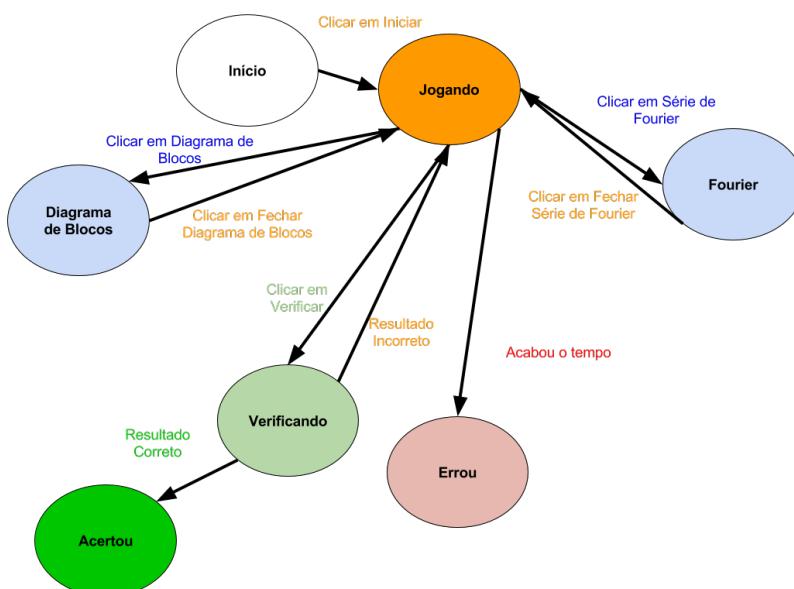
### 3.3.2 Desenvolvimento do jogo

O *Script* para plotar gráficos foi a etapa inicial para o desenvolvimento. O jogo foi dividido em 8 cenas, sendo os diferentes níveis de jogo e um menu geral. As fases iniciais são simples, e não possuem muita ligação com as Séries de Fourier, apenas para que o jogador se acostume e aprenda a jogabilidade.

A interação consiste no ajuste das amplitudes através de barras ajustáveis, denominadas de “*Sliders*”. Além disso, o jogador tem a sua disposição um botão que mostra o diagrama de blocos, e também outro que disponibiliza o teorema de Fourier, para o auxiliar na resolução do problema. Quando o jogador quiser verificar sua resposta, basta clicar em um botão específico e intuitivo.

Todo este processo é controlado por uma máquina de estados em um Script específico denominado de “*GameController*”. São 7 estados que contém um conjunto de funções específicas para a cena, são eles, “Início”, “Jogando”, “Acertou”, “Errou”, “Diagrama de Blocos”, “Fourier” e “Verificando”. Cada botão ou determinado evento realiza a transição destes estados. A Figura 3.4 mostra um diagrama da máquina de estado base para cada nível.

**Figura 3.4:** Máquina de estados com cada nível do jogo.

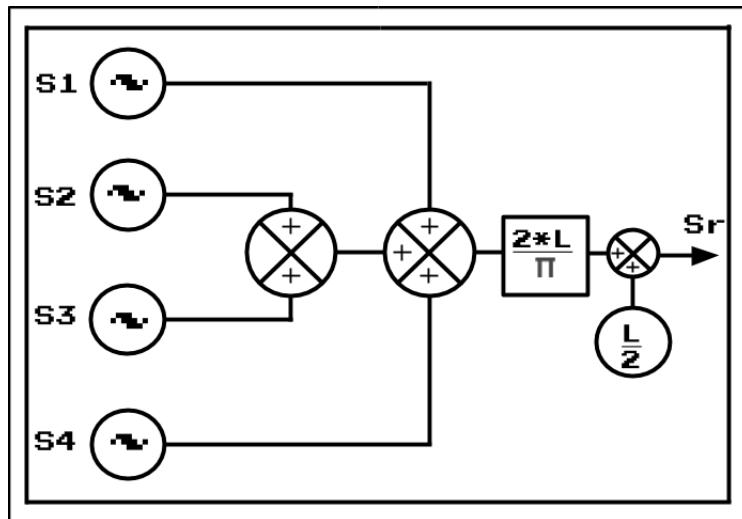


**Fonte:** Autoria Própria

Além do *Script* de controle, existem outros dois para plotar os gráficos. O primeiro deles plota o sinal controlado pelo jogador em tempo real, representado pela equação 3.4. Já o outro, plota o sinal requerido pelo nível do jogo. Ambos desenvolvidos com o método apresentado na Seção 3.2. Portanto, cada cena possui 3 *Scripts*, exceto nos níveis finais, em que o sinal esperado é uma onda quadrada ou triangular. Nestes casos, o gráfico foi desenhado manualmente utilizando a componente *Line Renderer*.

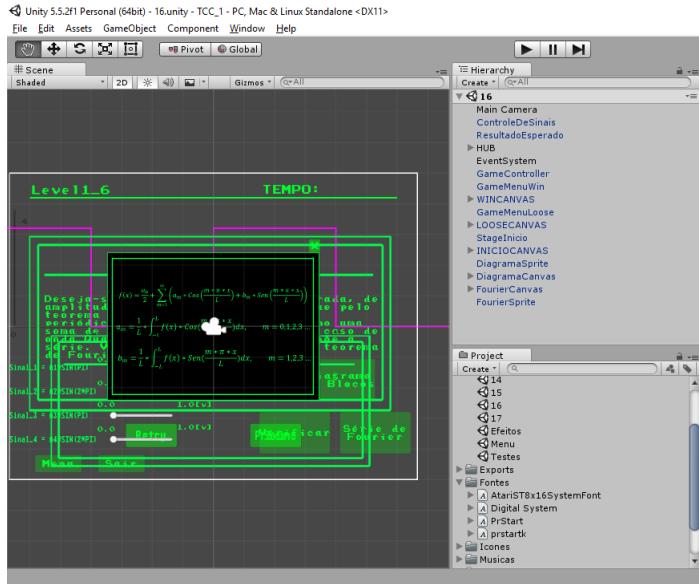
A parte gráfica do jogo, como divisórias, equações e desenhos em geral, foram feitas utilizando o “Google Desenhos”, ferramenta gratuita do “Google Drive” para esta finalidade. Este recurso foi escolhido, pois ele permite salvar imagens de boa qualidade, e de forma bastante simples. A fonte de texto utilizada no jogo, foi a “Press Start”, 100% gratuita (BOISCLAIR, 2001). A Figura 3.5 apresenta um exemplo desenvolvido para o jogo.

**Figura 3.5:** Exemplo de Diagrama de Blocos realizado no Google Desenhos com uso da fonte Press Start.



**Fonte:** Autoria Própria

Assim, todos os blocos gráficos foram dispostos na cena, e sua visibilidade depende dos estados mostrados pela Figura 3.4. A Figura 3.6 exibe uma captura de tela, que representa o ambiente de desenvolvimento de uma das cenas do jogo.

**Figura 3.6:** Captura de tela do ambiente de desenvolvimento do jogo.

**Fonte:** Autoria Própria

Os três scripts base para cada cena podem ser visualizados no Apêndice C.

## 3.4 Simulações de Efeitos de Áudio no Matlab

Como forma de compreender melhor a teoria, e obter um referencial comparativo para análise, foram simulados em Matlab, os efeitos de áudio aplicados no projeto. Os arquivos áudios tonais utilizados tanto no projeto, quanto nas simulações, foram extraídos do site “*Online Tone Generator*” (<http://onlinetonegenerator.com/>). Outros arquivos de som como trecho de guitarra limpo, foi extraído de DeMarco (2009). E um discurso foi baixado em um dos pacotes da Unity.

### 3.4.1 Tremolo

O Tremolo é o efeito mais simples, e foi simulado através de um *Script*. Com a função “*audioread*” do Matlab, leu-se um arquivo .wav, de um sinal tonal com frequência equivalente a 440 Hz. Em uma variável, foram armazenadas os valores de amplitude no tempo, e em outra, a frequência de amostragem, retornada como  $F_s = 44.100 \text{ Hz}$ .

Dessa forma, gerou-se o vetor de tempos ( $t$ ), com dimensão igual a  $F_s$ , que foi usado para calcular o sinal LFO ( $S_{LFO} = \sin(2\pi ft)$ ). As simulações foram realizadas com diferentes frequências e amplitudes. Segue o *Script* elaborado,

com  $f = 10 \text{ Hz}$ .

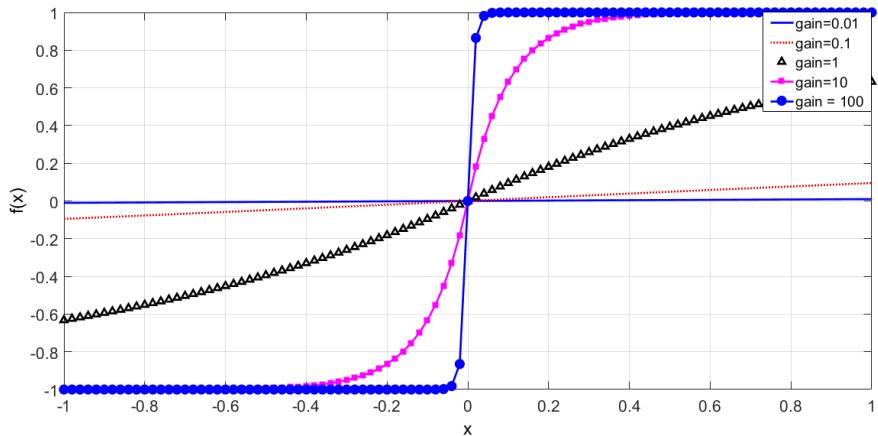
```
[y2, Fs2] = audioread('440Hz.wav');
t = 0:1/Fs2:length(y2)/Fs2;
t(220500) = [];
S1 = sin(10*2*pi*t);
S1 = S1';
sound((1*S1).*y2, Fs2);
```

### 3.4.2 Distortion

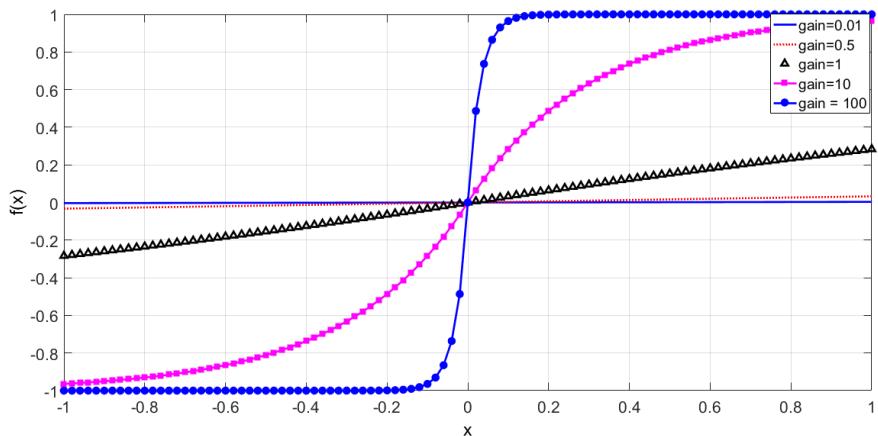
A distorção foi baseada no *Script* apresentado por Zölzer (2011). No caso da simulação apresentada neste trabalho, o sinal de entrada foi normalizado. O uso deste recurso faz com que o sinal sature de forma mais lenta, conforme o aumento do ganho (Figura 3.7).

**Figura 3.7:** Comparativo da função Normalizada e Não Normalizada.

(a) Sem Normalização.



(b) Com Normalização.



**Fonte:** Autoria Própria

Sabendo que o sinal de entrada pode ser representado como  $S_{in}$ , a Equação 3.5 representa o efeito da distorção aplicado na simulação.

$$S_{Distorcido} = \operatorname{sgn}\left(-\frac{\operatorname{Gain} \cdot |S_{in}|}{\operatorname{Max}(|S_{in}|)}\right) \cdot \left(1 - e^{\left(-\frac{\operatorname{Gain} \cdot |S_{in}|}{\operatorname{Max}(|S_{in}|)}\right)}\right) \quad (3.5)$$

Da mesma forma como realizado na simulação do Tremolo (Seção 3.4.1), o *Script* carrega as amostras de um arquivo, e aplica a distorção no sinal através da Equação 3.5. Para esta simulação, também foi utilizado um sinal tonal de 440 Hz.

```
clear all , clc
[y,Fs] = audioread('440HZ.wav');
x = y;
gain = 1;
q = gain*(x/max(abs(x)));
z = sign(-q).* (1-exp(sign(-q).*q));
audiowrite('440_Dist_AEXP_Gain_1.wav',z,Fs)
```

### 3.4.3 Chorus

Diferente dos outros efeitos, o Chorus foi simulado utilizando Simulink. Os blocos utilizados são provenientes da biblioteca DSP do *software*, e organizados conforme a Figura 2.16.

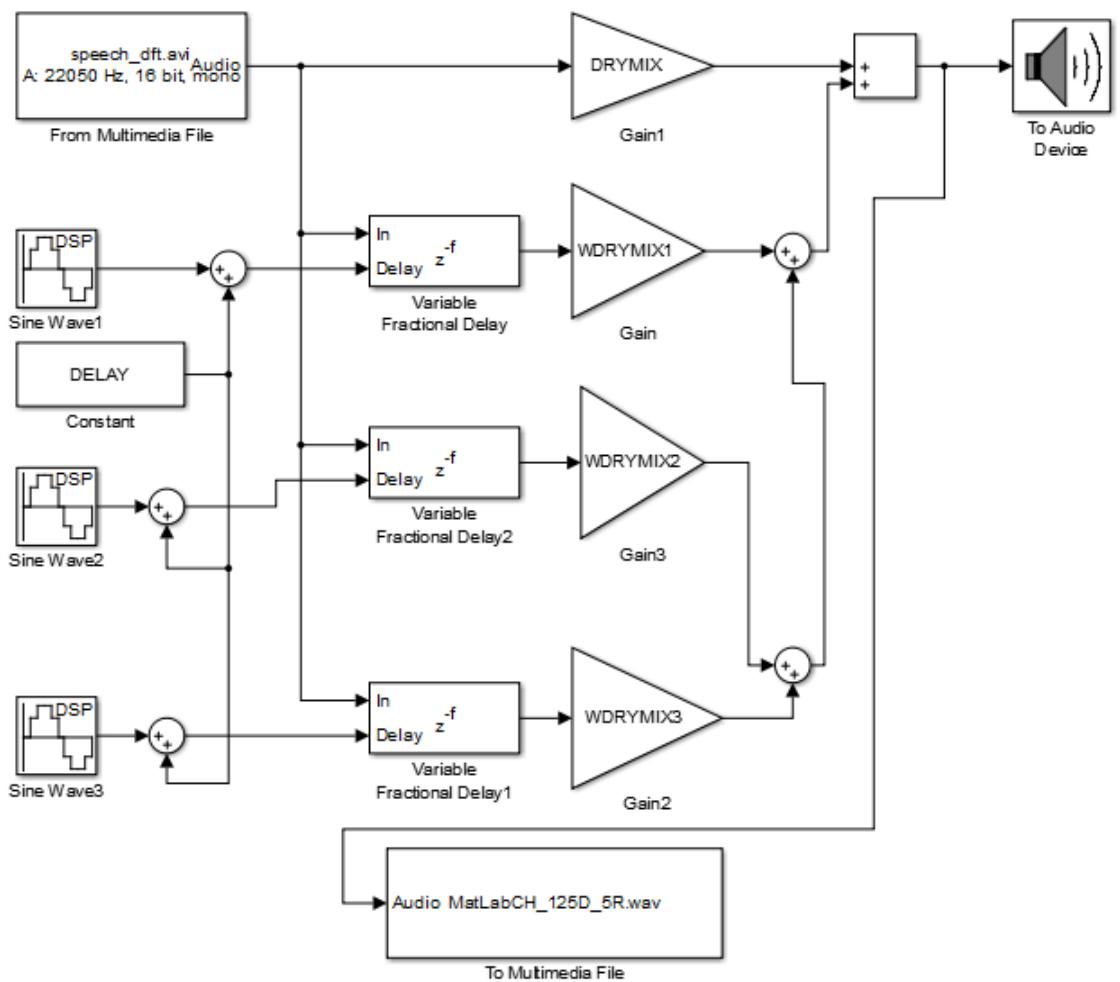
Os blocos aplicados foram “From Multimedia File”, “Sine Wave”, “Constant”, “Variable Fractional Delay”, “Sum”, “Gain”, “To Audio Device” e “To Multimedia File” (CZUBAK; RAHEJA, 2017). Todos os blocos operam com uma taxa de amostragem variável, ou seja, depende do arquivo de áudio alocado no primeiro bloco. A Figura 3.8 mostra a montagem da simulação.

O sistema simplesmente cria três cópias do áudio de entrada. Nestas réplicas são alteradas suas fases, com um parâmetro *Delay* constante, somado a uma senoide de amplitude equivalente a *Depth*, e com frequência igual a *Rate*. Todas as variáveis são setadas no *Matlab*.

Seja um sinal de entrada  $S_{in}(t)$ , a Equação 3.6 representa a relação matemática das cópias de áudio ( $S_1(t)$ ), que compõe o efeito Chorus. A soma de todas as réplicas é o resultado. Vale salientar que as cópias possuem fases diferentes entre si, para isso em uma foi somado  $\pi/2$ , em outra subtraído, e na terceira mantido em 0.

$$S_1(t) = S_{in}(t - (DELAY + DEPTH \cdot \text{sen}(2 \cdot \pi \cdot RATE \cdot t))) \quad (3.6)$$

**Figura 3.8:** Chorus simulado via Simulink.

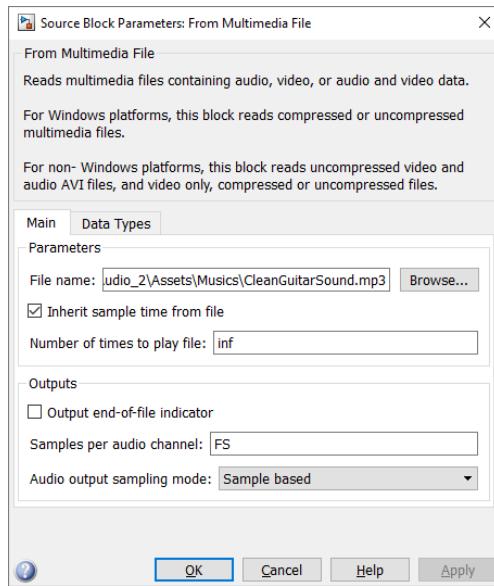


**Fonte:** Autoria Própria

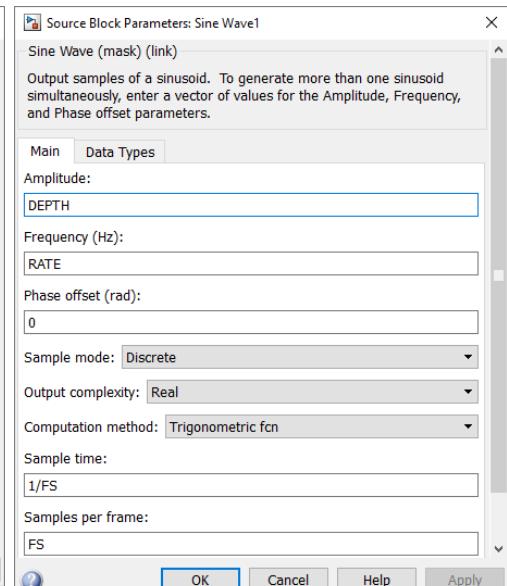
Todas as operações que envolvem tempo, são realizadas na simulação por meio de amostras. Por esse motivo, o parâmetro *Delay* tem seu valor no tempo, multiplicado pela frequência de amostragem. A Figura 3.9 mostra as configurações dos principais blocos.

**Figura 3.9:** Configurações dos blocos para simulação do Chorus no Simulink.

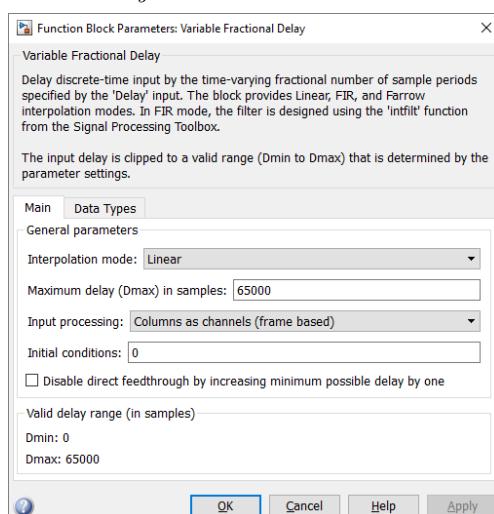
(a) Configuração do Bloco “From Multimedia File”.



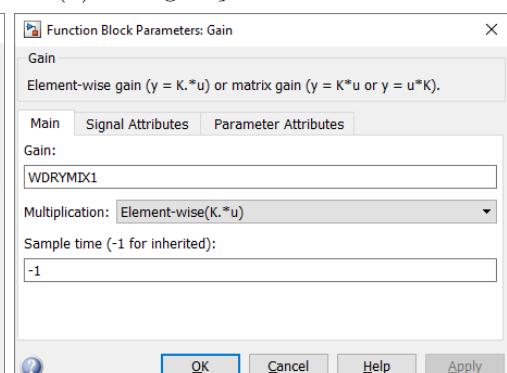
(b) Configuração do Bloco “Sine Wave”.



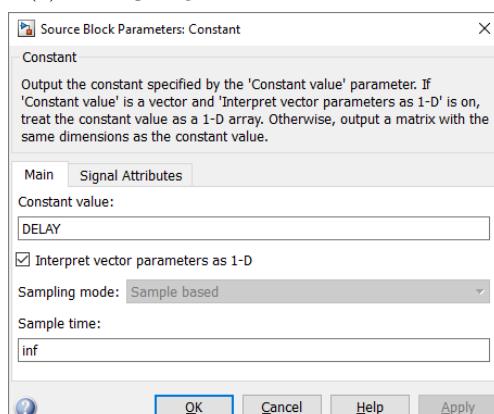
(c) Configuração do Bloco “Variable Fractional Delay”.



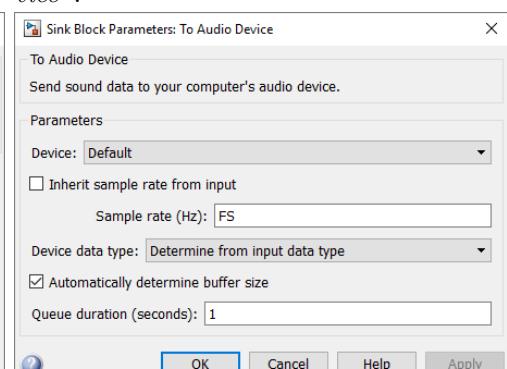
(d) Configuração do Bloco “Gain”.



(e) Configuração do Bloco “Constant”.



(f) Configuração do Bloco “To Audio Device”.



**Fonte:** Autoria Própria

## 3.5 Componentes de Áudio Unity 3D

Após as simulações, foram realizados estudos dos recursos de áudio presentes na Unity. O principal deles é a componente *Audio Source*, que como o próprio nome já diz, carrega um arquivo de som (*Audio Clip*), e o converte em uma fonte sonora dentro da cena. Além disso, existem outras componentes pré-programadas que aplicam efeitos de áudio. E também, existem funções destinadas para programação do processamento de sinais sonoros.

### 3.5.1 *Audio Source* e *Audio Clip*

A componente *Audio Source* aceita arquivos “.wav” e “mp3”, que são os mais comuns, e utilizados neste projeto. O som pode ser tocado em múltiplos canais, de forma espacializada (2D ou 3D), e com outras funcionalidades (UNITY, 2017f). Para este projeto, foi utilizado configuração de áudio em 2D.

Já o *Audio Clip* contém o arquivo de áudio a ser tocado pela componente *Audio Source* (UNITY, 2017a). Através disso, todos os sons stereo foram convertidos para Mono, devido às aplicações do projeto terem sido desenvolvidas para esta característica.

### 3.5.2 Sinal no Tempo

A função “*GetOutputData(float[] samples, int channel)*” aloca em um vetor, amostras do sinal de áudio no tempo (UNITY, 2017j). Este vetor pode ter qualquer dimensão, desde que seja potência de 2, o que influencia diretamente na resolução dos dados colhidos. Vale ressaltar que a dimensão é independente da taxa de amostragem do arquivo usado.

Com esta função, construiu-se uma sub-rotina que plota um sinal no tempo. Para o projeto, foi utilizado 512 como dimensão, o que é equivalente ao número de pontos do gráfico. Acima desta quantidade, o software ficou bastante lento, comprometendo o desempenho dele como um todo.

A componente *Audio Listener* foi utilizada para a função, que nada mais é do que o receptor de som da fonte sonora (*Audio Source*). Após recebida, a música é tocada pelos dispositivos do computador (UNITY, 2017d).

A função *GetOutputData* é aplicada no canal 0, já que o áudio é mono. Em seguida, o mesmo conceito desenvolvido na função de plotar gráficos é usado nesta

etapa (Seção 3.2). O vetor que recebeu as amostras é passado a outro, que neste caso é um *Vector3*, com coordenadas em y equivalente aos valores recebidos, e x os índices do vetor que correspondem ao eixo do tempo.

Estes valores no entanto são multiplicados por alguns parâmetros, que ajustam a escala do gráfico na tela. Por exemplo, o valor em x é multiplicado por 0,1, para comprimir o gráfico na horizontal e encaixar na cena. Já em y é feito o oposto, seu valor é multiplicado por 7, a fim de aumentar a visibilidade na vertical. Além disso, foram alocadas variáveis para ajuste de *offset* em ambos os eixos.

Segue o *Script* desenvolvido para tal aplicação. Esta função é chamada dentro do *loop Update*, de maneira bloqueante. Como é utilizado o *Audio Listener* para colher os dados do som, qualquer alteração feita na fonte como uso de filtros, mudança de volume, e aplicação efeitos, é recebida na sub-rotina.

```
void PlotarAudioNoTempo( float [ ] spectrum , int pos , float
    yvalue , Vector3 [ ] positions , int AjusteX , int AjusteY )
{
    AudioListener . GetOutputData( spectrum , 0 ) ;

    for( pos = 0; pos < spectrum . Length ; pos++ )
    {
        yvalue = spectrum [ pos ] ;
        positions [ pos ] = new Vector3( pos*0.1f + AjusteX , 7*yvalue
            + AjusteY ,0 ) ;
        lr . SetPositions ( positions ) ;
    }
}
```

### 3.5.3 Espectro do Sinal

De forma muito semelhante ao apresentado na Seção anterior (3.5.2), foi desenvolvida a funcionalidade para plotar o espectro do sinal. Neste caso foi utilizado a função “*GetSpectrumData(float[] samples, int channel, FFTWindow window)*”, que basicamente preenche um vetor com o espectro do sinal, selecionando o canal de áudio, e o tipo de janelamento (UNITY, 2017i).

A mesma lógica e parâmetros de ajustes utilizados para plotar o sinal no

tempo, foram usados no espectro. Exceto o fator de escala do eixo das abscissas, o qual foi substituído por uma variável, para facilitar a manipulação.

O canal selecionado foi o 0, pois o arquivo de áudio processado é do tipo mono. A dimensão configurada na função foi 512. Por fim, o tipo de janelamento foi o Retangular, tipo padrão dado na biblioteca.

Segue o *Script* desenvolvido para tal aplicação.

```
void PlotarAudioNaFrequencia( float [ ] spectrum , int pos ,
    float yvalue , Vector3 [ ] positions , int AjusteX , int
    AjusteY , float RangeX )
{
    AudioListener .GetSpectrumData( spectrum , 0 , FFTWindow .
        Rectangular ) ;

    for( pos = 1; pos < spectrum .Length ; pos++ )
    {
        yvalue = spectrum [ pos ];
        positions [ pos ] = new Vector3( pos*RangeX + AjusteX , 7*
            yvalue + AjusteY ,0 );
        AjusteY ,0 );
        lr . SetPositions ( positions );
    }
}
```

### 3.5.4 *Distortion Filter*

O filtro de distorção da Unity é um componente pronto do *software*, que como o próprio nome diz, aplica o dito efeito em um sinal (UNITY, 2017b). Esta componente é bastante simples, e apresenta apenas o nível da distorção como um parâmetro, que consiste no ajuste do ganho descrito pela Equação 3.5.

Para ler e escrever um valor de distorção para o filtro, utilizou a função “*AudioDistortionFilter.distortionLevel*”, que foi igualada ao valor de um *Slider*, ajustado pelo usuário, em uma faixa de 0 a 1.

### 3.5.5 Filtragem

A Unity disponibiliza as componentes tanto do filtro Passa-Baixas (FPB), quanto Passa-Altas (FPA). O FPB por exemplo, possui 2 parâmetros ajustáveis, a frequência de corte, e o coeficiente de ressonância. Assim, o áudio de uma fonte é filtrado de acordo com estes fatores (UNITY, 2017e). O FPA é análogo ao FPB (UNITY, 2017c).

Para ambos os filtros, a frequência de corte foi programada para ser ajustada via interface pelo usuário. Para tal, foi utilizada a função “*AudioLowPassFilter.cutoffFrequency*” para o FPB, e “*AudioHighPassFilter.cutoffFrequency*” para FPA. Os comandos permitem setar ou ler a frequência de corte do filtro. Já o coeficiente de ressonância não foi alterado, mantido em seu padrão e igual a 1.

### 3.5.6 Chorus

Primeiramente foi utilizado o filtro de Chorus, com procedimento semelhante ao aplicado na Distorção e Filtragem (Seções 3.5.4 e 3.5.5). Porém a componente apresentou problemas no parâmetro *Delay*, que não resultava no efeito esperado. Para solucionar tal problema, utilizou a biblioteca de funcionalidades do *Audio Clip*, com o objetivo de programar o Chorus a partir do sinal amostrado.

A função “*AudioClip.GetData*” retorna os valores das amostras de um arquivo de áudio, e consequentemente o armazena em um vetor de dimensão equivalente ao número de amostras do arquivo (UNITY, 2017g). Já a função “*AudioClip.SetData*” escreve um vetor de amostras em um *Audio Clip* (UNITY, 2017h). Este processo permite que o som executado no *software* seja proveniente dos dados escritos, porém o arquivo original não sofre alterações.

A partir disso, possibilitou o processamento do áudio para desenvolvimento do efeito Chorus. Primeiro o sinal é lido e armazenado em um vetor, que possui dimensão igual à quantidade de amostras do arquivo utilizado. Em seguida é aplicado um cálculo para resultar o efeito.

O Chorus foi desenvolvido com 3 cópias do sinal. Todas elas possuem um atraso equivalente a  $D_i$ , com  $i = 1, 2$ , e  $3$ , como segue,

$$D_1 = \left( \text{DELAY} + \text{DEPTH} \cdot \text{sen} \left( \frac{2 \cdot \pi \cdot n \cdot \text{RATE}}{F_S} \right) \right), \quad (3.7)$$

$$D_2 = \left( \text{DELAY} + \text{DEPTH} \cdot \text{sen} \left( \frac{2 \cdot \pi \cdot n \cdot \text{RATE}}{F_S} - \frac{\pi}{2} \right) \right) \quad (3.8)$$

e

$$D_3 = \left( \text{DELAY} + \text{DEPTH} \cdot \text{sen} \left( \frac{2 \cdot \pi \cdot n \cdot \text{RATE}}{F_S} + \frac{\pi}{2} \right) \right). \quad (3.9)$$

Note que cada senoide possui sua amostra (n) dividida pela frequência de amostragem ( $F_S$ ), esse recurso é utilizado para que esta oscilação tenha a mesma taxa de amostragem que o som. Sabendo que  $S_{in}$  é o áudio de entrada, e  $S_{Chorus}$  o som com efeito, a Equação 3.10 foi aplicada no código.

$$S_{Chorus}[n] = G \cdot (S_{in}[n] + S_{in}[n - D_1] + S_{in}[n - D_2] + S_{in}[n - D_3]) \quad (3.10)$$

O parâmetro “DELAY” varia de 0 a 44.100 amostras, “G” de 0 a 1, “DEPTH” de 0 a 100 amostras, e “RATE” de 0 a 10 Hz. Estes intervalos foram determinados de forma empírica, através da audição dos áudios até encontrar uma sonoridade esperada pelo efeito.

No *Script* o cálculo é realizado através de um *loop* lógico, e seu resultado armazenado em um vetor. Por fim, os valores resultantes são escritos no *Audio Clip*, e consequentemente o som com efeito de Chorus aplicado. Todo o processo é realizado de forma bloqueante dentro do *loop Update*. Segue a sub-rotina desenvolvida para aplicação do efeito.

```
void ChorusEffect(float [] SamplesEffect, float []
    SamplesAudio, float DryMixGain, int DELAY, float DEPTH,
    float RATE, int Indice)
{
    float DryMix1;
    float DryMix2;
    float DryMix3;
    while (Indice < SamplesAudio.Length - (DELAY+DEPTH)) {
        DryMix1 = (float) DELAY + DEPTH*Mathf.Sin(2f*Mathf.PI*
            Indice*RATE/audio.clip.frequency);
        DryMix2 = (float) DELAY + DEPTH*Mathf.Sin(2f*Mathf.PI*
            Indice*RATE/audio.clip.frequency - (Mathf.PI/2f));
        DryMix3 = (float) DELAY + DEPTH*Mathf.Sin(2f*Mathf.PI*
            Indice*RATE/audio.clip.frequency + (Mathf.PI/2f));
        SamplesEffect[Indice] = DryMixGain*SamplesAudio[Indice] +
```

```

DryMixGain*SamplesAudio[ Indice + (int) DryMix1 ] +
DryMixGain*SamplesAudio[ Indice + (int) DryMix2 ] +
DryMixGain*SamplesAudio[ Indice + (int) DryMix3 ];
++Indice;
}
audio . clip . SetData( SamplesEffect , 0 );
}

```

### 3.5.7 Tremolo

A implementação do Chorus, a partir da leitura e escrita das amostras do sinal, abriu portas para o desenvolvimento do efeito Tremolo. Na qual foi utilizada uma metodologia muito semelhante à simulação feita no Matlab, Seção 3.4.1.

O procedimento consiste na leitura de um arquivo de áudio, seguido do processamento e escrita via *Audio Clip*. O cálculo matemático para resultar o efeito é dado por:

$$S_{Tremolo}[n] = S_{in}[n] \cdot \cos\left(\frac{2 \cdot \pi \cdot n \cdot F}{F_s}\right). \quad (3.11)$$

A frequência ( $F$ ) pode ser ajustada pelo usuário, a fim de ter a tonalidade do efeito desejado. Da mesma forma como foi realizado no Chorus, o cálculo é feito através de um *loop*, e seu resultado é armazenado em um vetor. Depois os valores calculados são escritos no *Audio Clip*, para execução do som com o Tremolo. Segue o *Script* desenvolvido para esta aplicação.

```

void TremoloEffect( float [ ] SamplesEffect , float [ ]
    SamplesAudio , float LFOFrequency , int Indice )
{
    while ( Indice < SamplesAudio . Length ) {
        SamplesEffect [ Indice ] = SamplesAudio [ Indice ] * Mathf . Cos(2
            f *Mathf . PI *Indice *LFOFrequency / audio . clip . frequency );
        ++Indice ;
    }
    audio . clip . SetData( SamplesEffect , 0 );
}

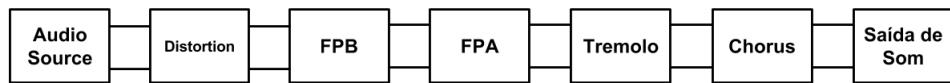
```

## 3.6 Aplicativos de Efeitos de Áudio Unity 3D

O aplicativo desenvolvido é anexado junto ao projeto do jogo, ou seja no menu inicial é possível acessar este recurso. Nesta aplicação o usuário tem a sua disponibilidade uma interface, que permite escolher e executar um trecho de música, exibir este sinal no tempo ou frequência, e aplicar efeitos sobre ele.

Desse modo, o usuário pode escolher um arquivo pré-carregado no projeto, selecionar a forma como o sinal será plotado, e aplicar os efeitos *Distortion*, filtragem (FPB e FPA), Chorus e Tremolo. Os efeitos foram programados em cascata, como o diagrama da Figura 3.10.

**Figura 3.10:** Diagrama de Blocos do aplicativo de áudios.



**Fonte:** Autoria Própria

A habilitação dos efeitos é realizada com o uso de *check box*, que retorna uma variável booleana conforme a seleção. Como já apresentado anteriormente, os parâmetros dos efeitos, como frequência de corte, *Delay*, nível de distorção, entre outros, são manipulados via barras ajustáveis (*Sliders*).

O *plot* do sinal é aplicado utilizando as funções desenvolvidas, e já apresentadas nas Seções 3.5.2 e 3.5.3. Uma máquina de estados, com o uso de *Switch Case*, gerencia a escolha do usuário e aplica a função adequada. Segue o trecho do código responsável por tal funcionalidade.

```

switch (Estado) {
    case StateAudio . Freq :
    {
        PlotarAudioNaFrequencia (spectrum2 , pos , yvalue , positions
            , 10 , 0 , 0.05 f );
        break ;
    }
    case StateAudio . Tempo :
    {
        PlotarAudioNoTempo (spectrum , pos , yvalue , positions , 0 , 0 );
        break ;
    }
}
  
```

{}

Os efeitos que são componentes prontas da Unity, como Distortion e os filtros, foram programados de forma bem simples. Primeiramente eles devem ser alocados ao objeto que contém a fonte de som. Dessa maneira, o *Script* verifica a *Check Box* do efeito, e o ativa ou não com o uso de uma função específica para habilitação da componente. Os valores de seus parâmetros são iguais as leituras de suas barras ajustáveis, que são alocadas na cena. Segue o trecho do código responsável por esta finalidade.

```
if(EnableDistortion.isOn == true){  
    GetComponent<AudioDistortionFilter>().enabled = true;  
}  
else{  
    GetComponent<AudioDistortionFilter>().enabled = false;  
}  
  
if(EnableFPB.isOn == true){  
    GetComponent<AudioLowPassFilter>().enabled = true;  
}  
else{  
    GetComponent<AudioLowPassFilter>().enabled = false;  
}  
  
if(EnableFPA.isOn == true){  
    GetComponent<AudioHighPassFilter>().enabled = true;  
    print("EnableFPA");  
}  
else{  
    GetComponent<AudioHighPassFilter>().enabled = false;  
  
    GetComponent<AudioDistortionFilter>().distortionLevel =  
        LevelDistSlider.value;  
    GetComponent<AudioLowPassFilter>().cutoffFrequency =  
        (float)FPB.value;  
    GetComponent<AudioHighPassFilter>().cutoffFrequency =  
        (float)FPA.value;
```

Já a aplicação dos efeitos Tremolo e Chorus no código são um pouco diferentes. Da forma como foram desenvolvidas, quando os efeitos forem ativados, suas amostras são calculadas de acordo com as funções. Quando desativado deseja-se que o efeito seja removido, e para isso foi necessário o desenvolvimento de uma

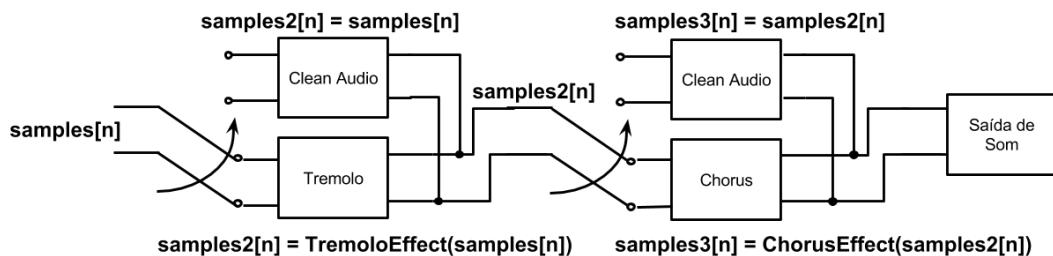
função que limpe o vetor calculado, e retorne seu valor sem o efeito em questão.

Para que o efeito funcione adequadamente, é preciso declarar 2 vetores, um no qual serão atribuídos as amostras calculados com a função de efeito, e outro de *Backup*, para quando se deseja limpar o efeito. Com isso, a função desenvolvida *CleanAudio*, apenas iguala o conjunto de amostras que foi calculado ao vetor com os valores iniciais e limpos. Segue a função programada no *Script*.

```
void CleanAudio( float [ ] SamplesEffect , float [ ]
    SamplesAudio , int Indice )
{
while ( Indice < SamplesAudio . Length ) {
    SamplesEffect [ Indice ] = SamplesAudio [ Indice ] ;
    ++Indice ;
}
}
```

Portanto a programação do Tremolo e Chorus em cascata, exigiu uma lógica diferente. Para entender melhor, observe o Diagrama de Blocos da Figura 3.11.

**Figura 3.11:** Lógica de aplicação dos efeitos sonoros.



**Fonte:** Autoria Própria

O vetor *samples[n]* é o inicial, e contém o áudio proveniente da fonte, mais os blocos prontos de efeitos (Distortion e filtros) conforme ativação. Já *samples2[n]* representa o sinal no ponto após o bloco do Tremolo, que em caso de ativação, o dito efeito é aplicado ao vetor *samples[n]* e atribuído em *samples2[n]*. Caso contrário *samples2[n]* se mantém igual a *samples[n]*, devido à função *CleanAudio*. A mesma lógica ocorre em sequência ao Diagrama no bloco de Chorus, porém nesta etapa são trabalhados com os vetores *samples3[n]*, que indica o ponto após o efeito, e *samples2[n]* depois do Tremolo. O vetor *samples3[n]* é setado no *Audio Clip* para tocar o som. Segue o trecho do *Script* que executa o controle.

```
if(EnableDistortion.isOn == true){  
    GetComponent<AudioDistortionFilter>().enabled = true;  
}  
else{  
    GetComponent<AudioDistortionFilter>().enabled = false;  
}  
  
if(EnableFPB.isOn == true){  
    GetComponent<AudioLowPassFilter>().enabled = true;  
}  
else{  
    GetComponent<AudioLowPassFilter>().enabled = false;  
}  
  
if(EnableFPA.isOn == true){  
    GetComponent<AudioHighPassFilter>().enabled = true;  
    print("EnableFPA");  
}  
else{  
    GetComponent<AudioHighPassFilter>().enabled = false;  
}  
  
if(EnableTremolo.isOn == true){  
    TremoloEffect(samples2,samples,FTremolo.value,indiceAudio)  
    ;  
}  
else{  
    CleanAudio(samples2,samples,indiceAudio);  
}  
  
if(EnableChorus.isOn == true){  
    ChorusEffect(samples3,samples2,DryMix.value,(int)  
        DelaySlider.value,DepthSlider.value,RateSlider.value,  
        indiceAudio);  
}  
else{  
    CleanAudio(samples3,samples2,indiceAudio);  
}  
audio.clip.SetData(samples3, 0);
```

O *Script* completo aplicado ao projeto é apresentado pelo Apêndice D.

## 3.7 Validação do Software

A validação do projeto consistiu na aplicação de um teste para aos alunos da disciplina de Circuitos Elétricos 1, ministrada pelo professor orientador. A turma foi dividida em duas, onde para uma delas houve a apresentação do *software* (Turma Com *Software*), e outra não (Turma Sem *Software*). Este teste teve como objetivo medir quanto o *software* contribuiu no aprendizado de seus usuários, de forma comparativa com o método tradicional.

O teste possui 4 questões, que foram elaboradas conforme as etapas do jogo. Das questões, a primeira (Q1) remete aos níveis iniciais do jogo, que aborda uma temática básica de relações trigonométricas, bem como leitura de um Diagrama de Blocos. A segunda (Q2) e terceira (Q3) perguntam, se referem a representação em Séries de Fourier, com formas de onda quadrada e triangular. A última questão (Q4), propõe ao estudante explorar o aplicativo de efeitos de áudio, e deduzir sua resposta, que aborda os conceitos de Fourier e Filtros. Todas as perguntas foram objetivas e de pesos iguais, e o valor máximo da prova foi de 4. O questionário completo pode ser visto no Apêndice A.

A validação foi realizada em uma aula extra da disciplina. Previamente, o professor ministrou aulas a respeito de resposta em frequência aos alunos. O conteúdo de Séries de Fourier é ministrada na disciplina de Cálculo 2, porém muitos alunos ainda não tinham visto a matéria.

A validação ocorreu nos laboratórios de informática e computação (LIG) do Centro de Tecnologia e Urbanismo (CTU). Dois laboratórios foram utilizados, e a turma foi dividida aleatoriamente em cada sala. A Turma Sem *Software* ficou no LIG 1, onde o orientador ministrou uma aula de Séries de Fourier. Já no LIG 2, o orientando apresentou o projeto a Turma Com *Software*, e monitorou os alunos com o uso. Ambas as apresentações duraram 30 min.

Por fim, foi aplicado o teste para as turmas. Ambas realizaram a prova de maneira individual com consultas. Vale ressaltar que apenas os alunos da Turma Com *Software*, que tiveram o aplicativo apresentado, poderiam usá-lo como consulta e apoio na resolução.

Os desempenhos das turmas foram comparados, com objetivo de medir o impacto que o *software* teve no aprendizado da teoria. Depois de todos terem finalizado as provas, os dois grupos foram juntados, e o aplicativo foi apresentado, juntamente com a correção da prova, e uma apresentação de mixagem de som.

# 4 Resultados e Discussões

No capítulo anterior foram apresentadas as principais metodologias, práticas, e recursos utilizados para desenvolver os aplicativos. Nesta parte serão mostrados os resultados obtidos, assim como as discussões acerca de como os dados impactaram na qualidade do projeto.

Como meio utilizado para simulações, o Matlab também foi aproveitado para extração de alguns dados, como gráficos e extrações de áudio. O Audacity foi usado para gravações de sons a partir da placa de som do computador, com o objetivo de armazenar um áudio produzido pela Unity, já que o *software* não apresenta nenhuma maneira mais prática de extração. Por fim, o Reaper, programa voltado para edições de som, foi empregado para análise de espectro.

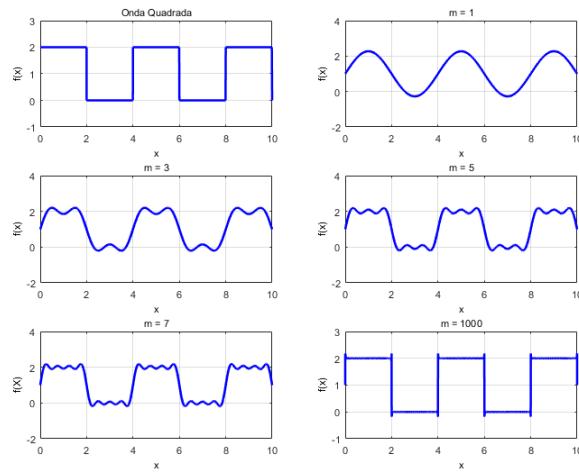
## 4.1 Séries de Fourier Matlab

A primeira função simulada foi:

$$f(x) = \begin{cases} 2 & \text{se } 0 \leq x \leq 2 \\ 0 & \text{se } 2 \leq x \leq 4. \end{cases} \quad (4.1)$$

Dessa maneira as simulações são apresentadas pela Figura 4.1. Note que conforme aumenta-se o número de harmônicas no somatório, melhor é a representação do gráfico. Isso fica evidente na simulação para  $m = 1.000$ , onde o gráfico é quase idêntico a função esperada.

**Figura 4.1:** Representações em Séries de Fourier de uma Onda Quadrada, de amplitude 2 e período 4.



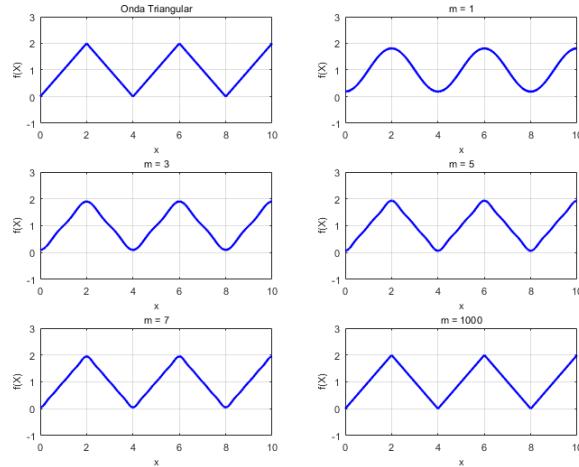
**Fonte:** Autoria Própria

A segunda simulação foi desenvolvida para:

$$f(x) = \begin{cases} x & \text{se } 0 \leq x \leq 2 \\ -x + 4 & \text{se } 2 \leq x \leq 4. \end{cases} \quad (4.2)$$

A Figura 4.2 mostra os resultados obtidos. Da mesma forma como a onda quadrada, fica claro que a representação em Fourier está correta, já que quanto maior o m, mais fiel é o gráfico.

**Figura 4.2:** Representações em Séries de Fourier de uma Onda Triangular, de amplitude 2 e período 4.



**Fonte:** Autoria Própria

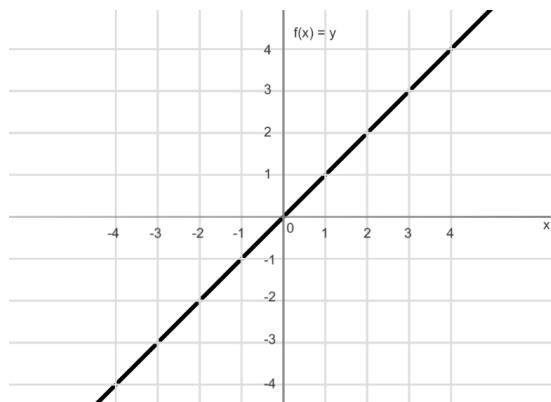
Além de promover um melhor entendimento da teoria, estas simulações foram importantes para o desenvolvimento do projeto. Como já apresentado pela Seção 3.3, as mecânicas do jogo consistem na somatória de 4 sinais, onde o jogador deve ajustar sua amplitude para obter um sinal requerido, dentro dos recursos disponibilizados. Partindo disso, nota-se que em ambas as figuras, para  $m = 5$ , os gráficos, apesar de não serem a representação exata, já são bem próximos do objetivo. Com a interatividade, o aluno pode perceber aos poucos como os sinais somados resultam na representação em Fourier.

## 4.2 Gráficos Unity 3D

Os gráficos apresentados nessa seção foram plotados com 300 pontos, e passo equivalente a 0,1 entre os valores de  $x$ . Para apresentar os resultados, foi feito uma tela de apresentação bastante simples, com fundo branco e desenho das coordenadas.

A primeira função aplicada foi a identidade,  $f(x) = x$  (Figura 4.3). É possível notar pela imagem que o gráfico está condizente com o esperado.

**Figura 4.3:** Gráfico  $f(x) = x$  plotado na Unity.



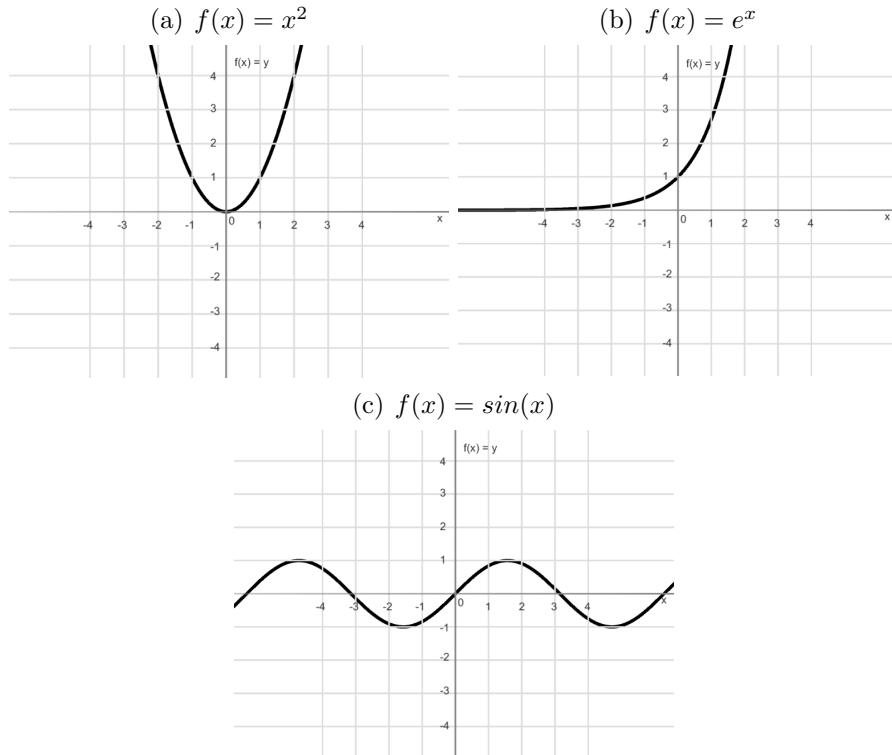
**Fonte:** Autoria Própria

Em seguida foram plotadas funções mais complexas que a anterior. A Figura 4.4 mostra o resultado obtido pelas funções, quadrática, exponencial e senoidal. Observe que todas elas apresentaram desenhos condizentes com o esperado.

Portanto, com os resultados obtidos, foi possível averiguar que os recursos disponibilizados pela programação em C# e pela Unity, possibilitam uma implementação matemática adequada para o projeto, na qual os gráficos possuem boa resolução, existe uma ampla biblioteca de operações matemáticas, e sua imple-

mentação é bastante simples.

**Figura 4.4:** Gráficos plotados na Unity.



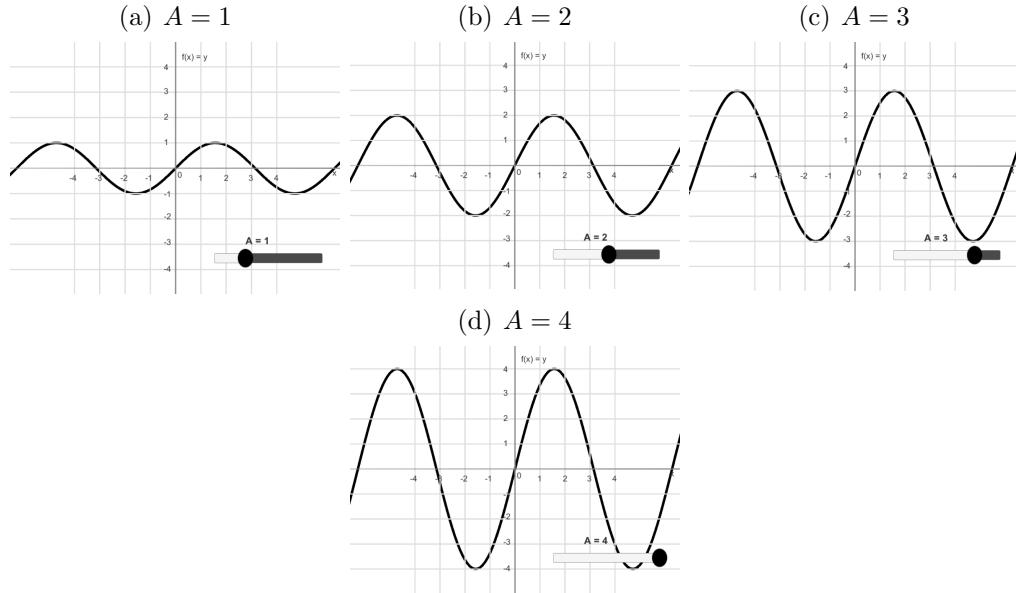
**Fonte:** Autoria Própria

Para alteração em tempo real do gráfico, tem-se como o exemplo a função da Equação 4.3 . Sua amplitude  $A$  é manipulada por uma barra ajustável(*Slider*), em um intervalo de 0 a 4.

$$f(x) = A \cdot \sin(x). \quad (4.3)$$

A Figura 4.5 mostra o gráfico alterando conforme o ajuste. É possível evidenciar o funcionamento do código, já que a amplitude do gráfico muda conforme alterações na barra.

**Figura 4.5:** Gráficos plotados na Unity com alteração de parâmetros em tempo real.



**Fonte:** Autoria Própria

Portanto, os resultados apresentados mostram que as aplicações funcionam adequadamente, atendendo à principal etapa do projeto, com plotagem de diversos gráficos matemáticos, e os alterando em tempo real durante a execução.

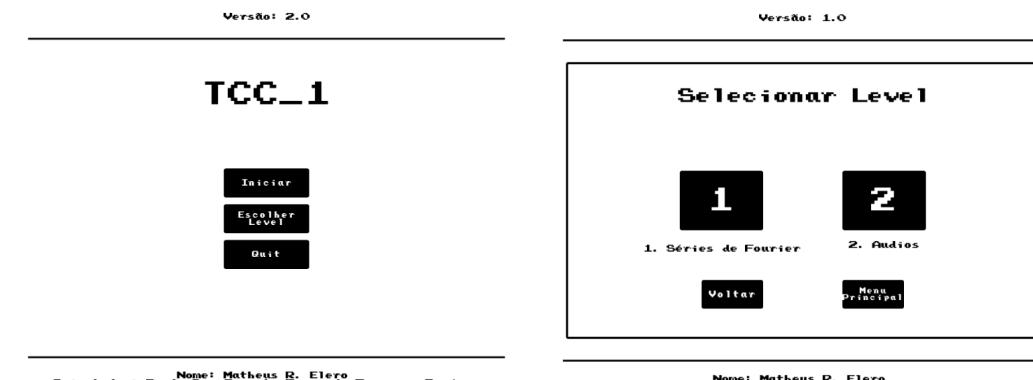
### 4.3 Jogo Séries de Fourier

A seguir serão mostrados os resultados do jogo digital como um todo. Para melhor qualidade da impressão desta monografia, as imagens do aplicativo foram alteradas, de forma que seu fundo ficasse branco, com outros objetos em preto. O fundo escuro, como o original, quando impresso pode prejudicar a resolução da imagem.

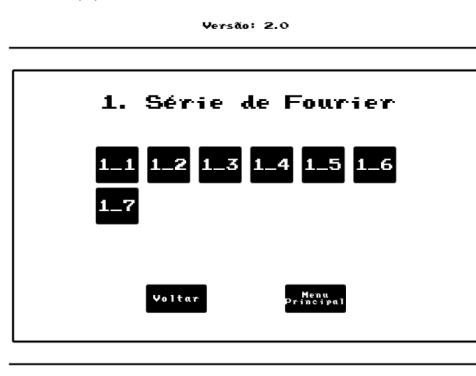
Ao abrir o arquivo um menu é exibido, que dispõe ao jogador opções para escolha de nível, iniciar o jogo, ou acessar o aplicativo de áudio. A Figura 4.6 mostra os estados de controle do menu.

**Figura 4.6:** Telas principais de seleção do jogo.

(a) Menu Principal (b) Menu de escolha de aplicativos.



(c) Menu de escolha de níveis.

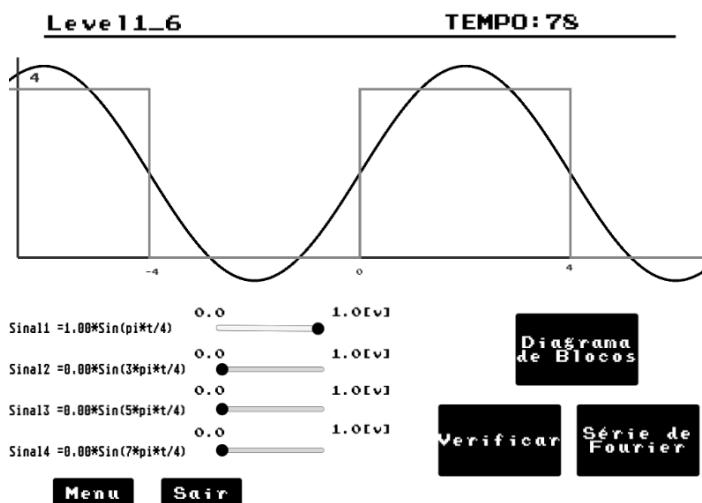
**Fonte:** Autoria Própria

Cada nível se inicia com uma tela, destacando um texto com uma descrição do exercício, o número da fase, e um botão de início. A Figura 4.7 mostra esta etapa para o nível 1.6, que propõe o objetivo de encontrar um sinal de onda quadrada, de amplitude 4 e período 8.

**Figura 4.7:** Tela de início do nível 1.6 do Jogo.**Fonte:** Autoria Própria

Em seguida o jogo se inicia, disponibilizando uma tela com os controles e opções (Figura 4.8). O gráfico em cinza representa o resultado esperado, já em preto o controle pelas barras ajustáveis. A plotagem é realizada em tempo real, o que permite ao aluno a percepção visual das equações, permitindo a comparação com o objetivo. Para auxiliar no resultado, ao lado de cada barra, estão disponíveis as equações de cada sinal, e seu valor de amplitude é exibido conforme alteração dos controles. Também existem no início e fim de cada barra, os valores mínimos e máximos que cada uma delas abrange.

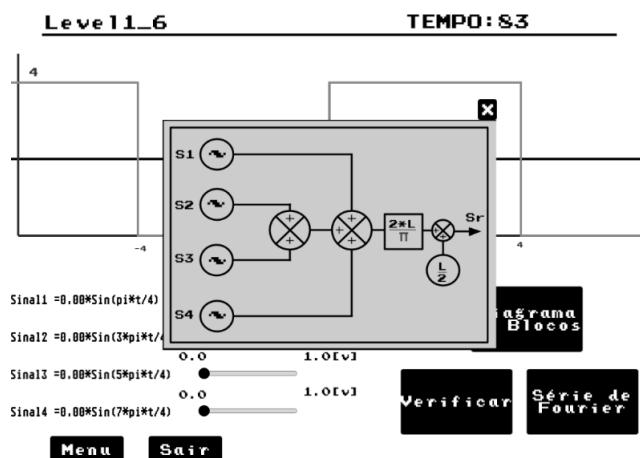
**Figura 4.8:** Tela de jogo do nível 1.6.



**Fonte:** Autoria Própria

Também é possível verificar equações da teoria, assim como o Diagrama de Blocos que opera o nível (Figura 4.9). Estas opções auxiliam o aluno na resolução, além de exercitar o jogador com a leitura das equações e diagrama.

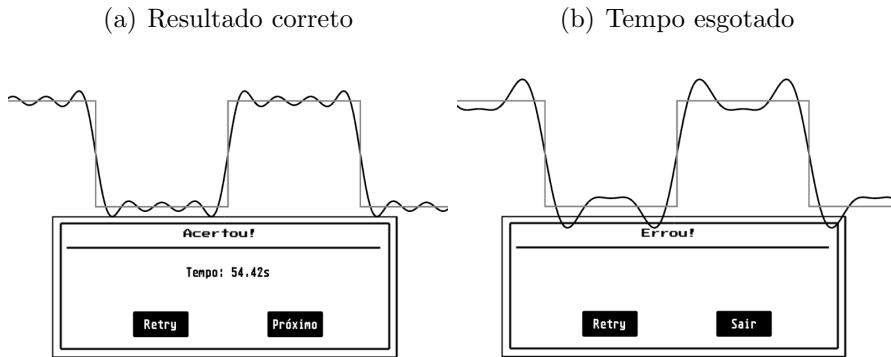
**Figura 4.9:** Diagrama de Blocos sendo exibido no nível 1.6.



**Fonte:** Autoria Própria

O resultado pode ser verificado clicando no botão “Verificar”, caso estiver correto, uma tela se abrirá mostrando o tempo de conclusão, opções de seguir em frente ou sair. Caso o tempo se esgote, e o jogador não tiver apertado o botão com os resultados corretos, uma tela indicando que o exercício está errado será exibida, com alternativas de tentar novamente, ou sair. A Figura 4.10 exibe estes 2 acontecimentos.

**Figura 4.10:** Telas de acerto e erro exibida em cada nível.



**Fonte:** Autoria Própria

Com o objetivo atingido é evidenciado ao jogador o resultado das Séries de Fourier. O comparativo entre os gráficos requerido e controlado facilitam esta percepção, já que este último chega muito próximo ao objetivo. Todo o processo até a finalização é crucial, visto que é nesta etapa que a interatividade, visibilidade e o lúdico aparecem para auxiliar o jogador.

No caso do exercício discutido nesta seção, as amplitudes corretas são  $A_1 = 1 \pm 0,01$ ,  $A_2 = 1/3 \pm 0,01$ ,  $A_3 = 1/5 \pm 0,01$  e  $A_4 = 1/7 \pm 0,01$ , sendo 0,01 a tolerância para acerto. Como já apresentado anteriormente neste trabalho, uma onda quadrada possui sua representação em Séries de Fourier apenas com termos ímpares, ou seja,

$$S_r = 2 + \frac{8}{\pi} \cdot \left[ 1 \cdot \sin\left(\frac{\pi}{4}\right) + \frac{1}{3} \cdot \sin\left(\frac{3 \cdot \pi}{4}\right) + \frac{1}{5} \cdot \sin\left(\frac{5 \cdot \pi}{4}\right) + \frac{1}{7} \cdot \sin\left(\frac{7 \cdot \pi}{4}\right) \right]. \quad (4.4)$$

Os outros níveis do jogo apresentam resultados semelhantes ao exibido nessa Seção. Com diferenças no objetivo final, consequentemente em alguns valores das senoides controláveis, diagrama de blocos, entre outros.

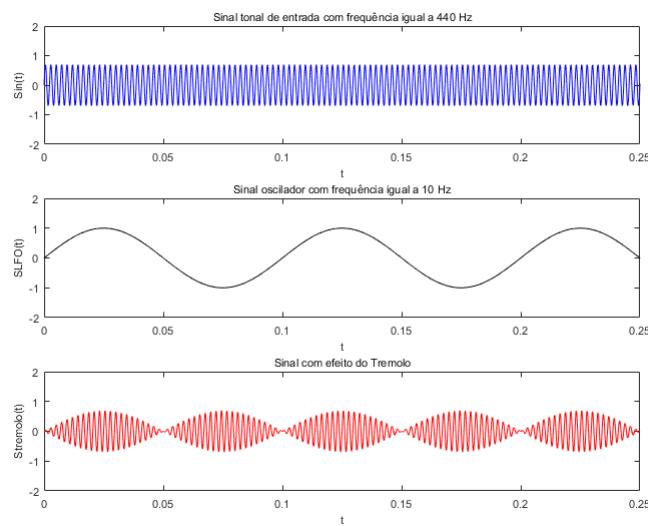
## 4.4 Simulações de Efeitos de Áudio Matlab

### 4.4.1 Tremolo

O efeito do tremolo foi analisado no tempo, devido à facilidade de observação. Com o Matlab foram extraídos os gráficos dos sinais aplicado, oscilador e sob o efeito.

A Figura 4.11 mostra um sinal tonal de frequência igual a  $440\text{ Hz}$ , LFO(*Low Frequency Oscilator*) de  $10\text{ Hz}$ , e o áudio com o efeito em questão. Note que o resultado está condizente com o esperado, já que o sinal com Tremolo possui uma variação de amplitude conforme a frequência de oscilação. Além disso, o som emitido foi equivalente ao esperado por este efeito.

**Figura 4.11:** Efeito do Tremolo simulado com o uso do Matlab.



**Fonte:** Autoria Própria

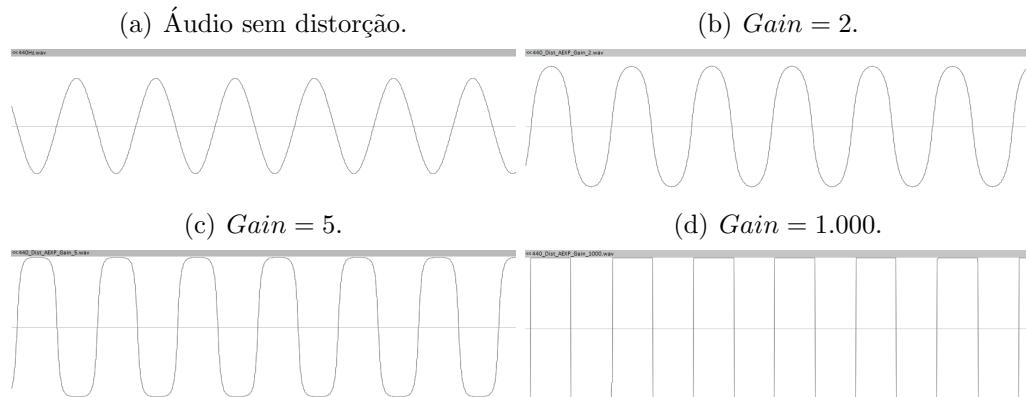
### 4.4.2 Distortion

Da mesma forma como o Tremolo, a distorção foi analisada com um sinal tonal de frequência equivalente a  $440\text{ Hz}$ . A facilidade de visualização do efeito, tanto no tempo, quanto na frequência, foram os motivos pelos quais um sinal tonal foi escolhido.

Como já abordado na Seção 2.3.4, ao aplicar o efeito da distorção com a equação adequada, o sinal sofre uma deformação característica. Com o aumento do ganho, um sinal senoidal tende a se tornar uma onda quadrada (Saturação).

A Figura 4.12 mostra os resultados do sinal no tempo, com o parâmetro *Gain* equivalente a 2, 5 e 1.000. Os gráficos mostram que a função aplicada realmente distorce o sinal como requerido. Note que com o aumento do ganho o sinal foi deformado até a saturação, que indica uma onda quadrada.

**Figura 4.12:** Áudios no tempo com efeito de Distorção desenvolvido no Matlab.



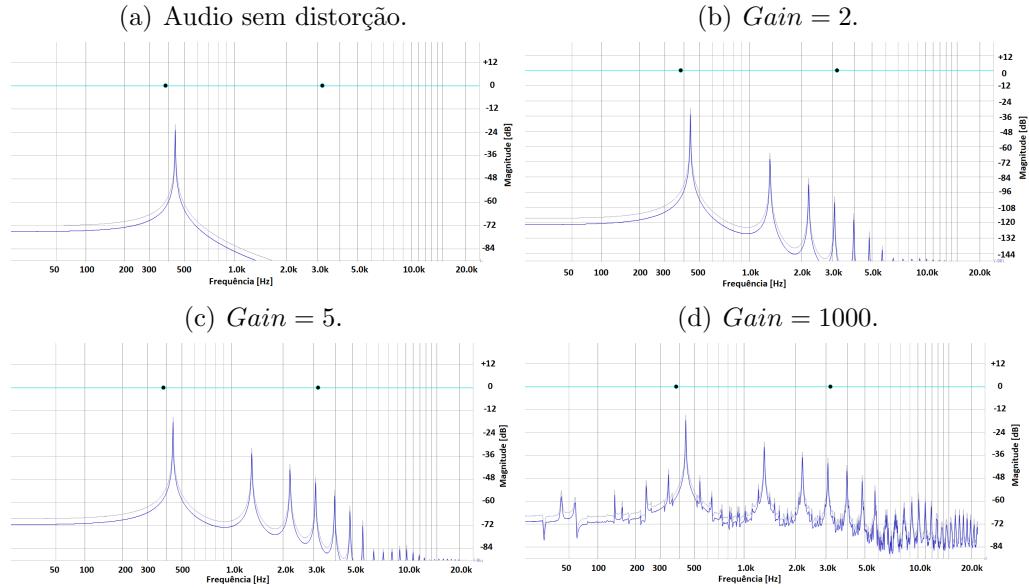
**Fonte:** Autoria Própria

Em consequência disso, no espectro de frequências, conforme o aumento do ganho devem surgir harmônicas nas frequências como:

$$f_h = 440 \cdot n, n = 3, 5, 7 \dots 2n + 1. \quad (4.5)$$

A Figura 4.13 mostra os espectros dos parâmetros atribuídos, onde é possível observar que isso de fato acontece. Para um ganho equivalente a 1.000, também existe o crescimento de ruídos, já que o sinal está saturado e totalmente distorcido.

**Figura 4.13:** Espectros dos áudios com efeito de Distorção desenvolvido no Matlab.



**Fonte:** Autoria Própria

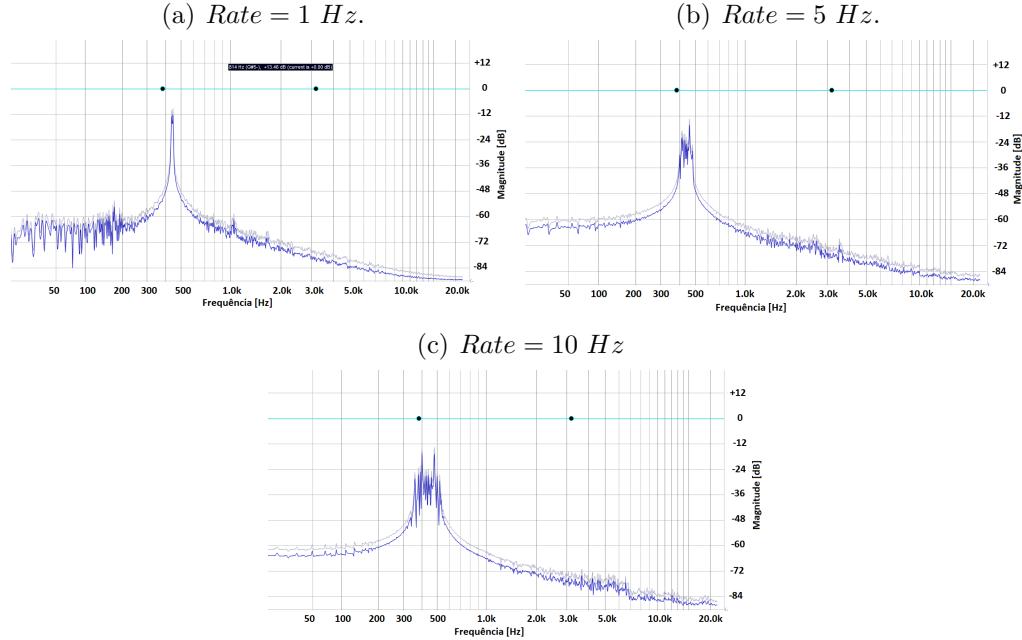
### 4.4.3 Chorus

Um sinal tonal de frequência igual a  $440\text{ Hz}$ , proveniente do mesmo arquivo utilizado no projeto, foi usado para as simulações deste efeito. Configurado com um *Delay* de 441 amostras, *Depth* igual a 100, e o *Rate* foi variado com valores de  $1\text{ Hz}$ ,  $5\text{ Hz}$  e  $10\text{ Hz}$ .

A análise auditiva dos resultados deste efeito foi muito importante para compreendê-los e valida-los. Com *Rate* igual a  $1\text{ Hz}$ , a variação é bastante sutil, onde é possível perceber outras cópias do som, porém de maneira suave. Ao aumentar este parâmetro, o efeito se torna muito mais perceptível.

Este fator pode ser observado pelo espectro da Figura 4.14. Note que com o aumento da frequência, ocorre um alargamento do espectro em torno do sinal fundamental. Isso ocorre, devido a oscilação do atraso presente nas cópias somadas. Por isso, quando utiliza-se o efeito Chorus o som tem uma impressão de profundidade, devido a variação de frequência em torno do sinal. Porém, com estas configurações, o sinal fica bastante distorcido quando aumenta-se o *Rate*.

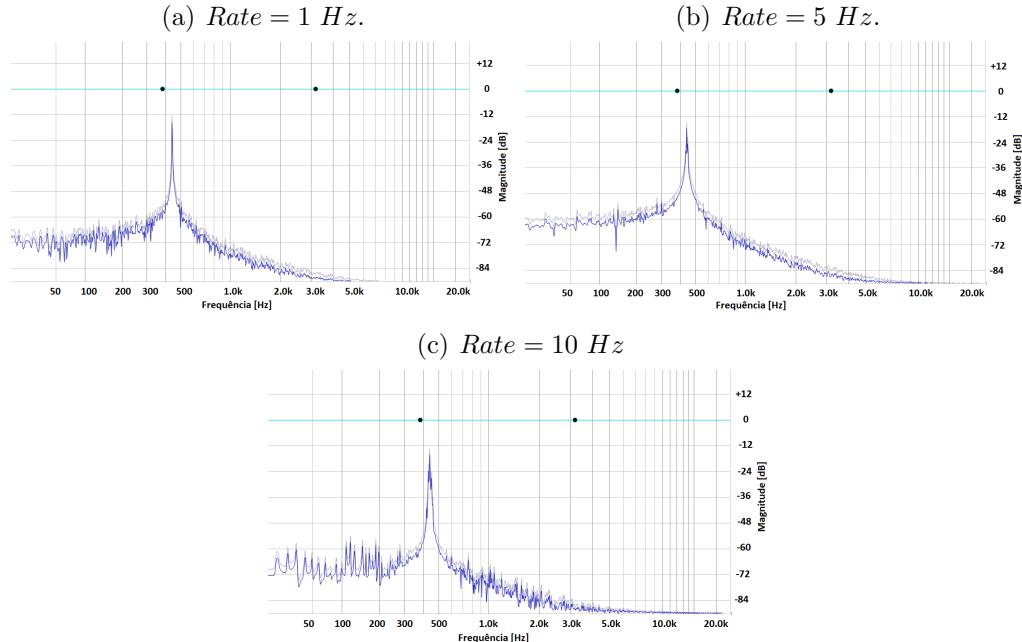
**Figura 4.14:** Simulações via Simulink do Chorus, com Delay = 441, Depth = 100, e Rate variável.



**Fonte:** Autoria Própria

A Figura 4.15 mostra os espectros de outra simulação, com  $Delay = 441$ ,  $Depth = 10$  e Rate variável, equivalente a  $1\text{ Hz}$ ,  $5\text{ Hz}$  e  $10\text{ Hz}$ . Assim é possível perceber que o parâmetro  $Depth$ , por representar os valores máximos e mínimos em que o atraso varia, é crucial para o desempenho do efeito. Com um valor baixo, 10, o Chorus fica bem mais leve, e seu espectro não se expande igual o da Figura 4.14. Este é o caso de um Chorus leve, que é geralmente utilizado, pois não distorce muito o sinal original, e cria o efeito de coro desejado.

**Figura 4.15:** Simulações via Simulink do Chorus, com Delay = 441, Depth = 10, e Rate variável.



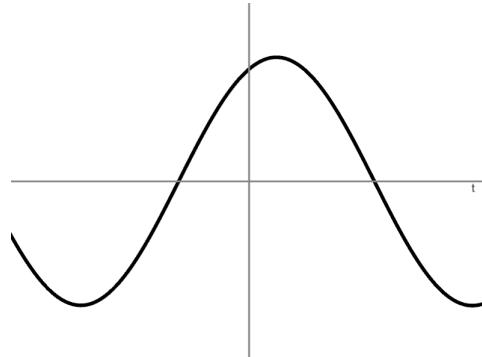
**Fonte:** Autoria Própria

## 4.5 Componentes de Áudio Unity 3D

### 4.5.1 Plot de Sinal no Tempo

Os resultados apresentados nesta seção, são provenientes de sinais tonais, mais especificamente com frequências  $440\text{ Hz}$  e  $10\text{ kHz}$ . A quantidade de pontos utilizada foi 512. A Figura 4.16 mostra o sinal no tempo para frequência de  $440\text{Hz}$ . Como o *plot* do sinal apresentou um objetivo apenas visual para o aplicativo, os valores de escala do gráfico não são apresentados.

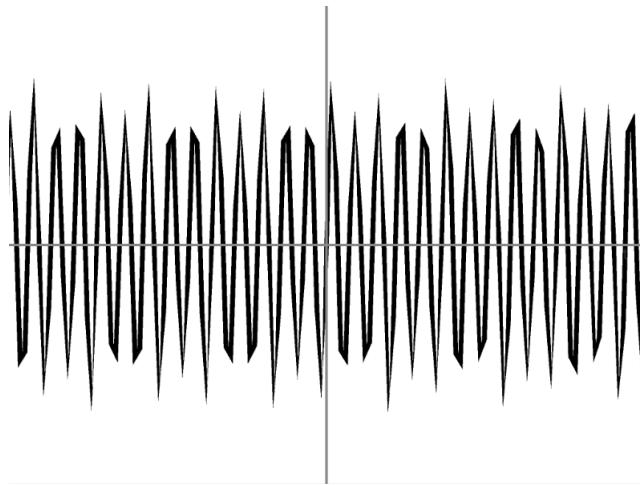
**Figura 4.16:** Plot de sinal tonal na Unity, com frequência igual a  $440\text{ Hz}$ , e 512 pontos.



**Fonte:** Autoria Própria

Note que o gráfico apresentado possui boa resolução, representando bem o áudio no tempo. O mesmo não pode ser dito para o arquivo de frequência igual a  $10\ kHz$ . Observe pela Figura 4.17, que a resolução do gráfico é bastante prejudicada, devido a alta oscilação do sinal.

**Figura 4.17:** Plot de sinal tonal na Unity, com frequência igual a  $10\ kHz$ , e 512 pontos.



**Fonte:** Autoria Própria

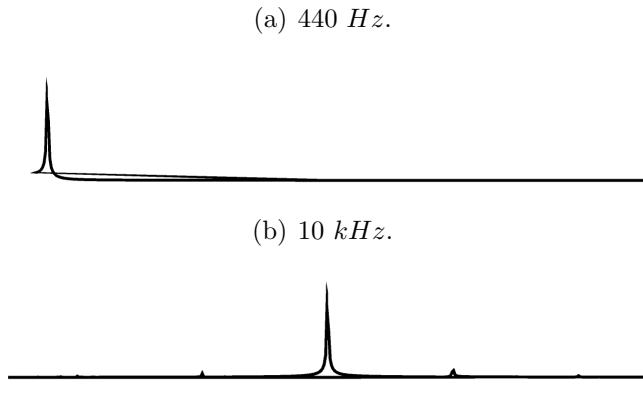
Caso houver um aumento do número de pontos, e diminuição do passo entre cada um deles, a resolução irá melhorar. No entanto, o aumento desta característica deixa o *software* lento, e pode prejudicar outras funcionalidades. Mesmo com a baixa qualidade, é perceptível a diferença de frequência entre os dois sons (Figuras 4.16 e 4.17).

### 4.5.2 Plot do Espectro do Sinal

Para plotar os espectros na frequência, também foram utilizados 512 pontos, com escala no eixo das frequências igual 0, 05. Para a programação de um sistema que permite escolher o plot no tempo ou frequência, o uso do mesmo número de pontos para ambos facilita a programação.

A Figura 4.18 mostra o espectro de dois sinais tonais, um com frequência igual a  $440\ Hz$ , e outro  $10\ kHz$ . Apesar dos desenhos não apresentarem as respectivos valores dos eixos, pois o espectro possui apenas um papel de auxílio visual no projeto, é possível perceber a validade do gráfico pela distância entre ambos os sinais. Note que o sinal de  $440\ Hz$  está bem próximo da origem, pelo motivo da escala utilizada ter sido pequena.

**Figura 4.18:** Espectro de sinais tonais na Unity, com frequência igual a  $10\text{ kHz}$  e  $440\text{ Hz}$ , e 512 pontos.



**Fonte:** Autoria Própria

Já a Figura 4.19 mostra o espectro de um áudio de onda quadrada, e frequência igual a  $440\text{ Hz}$ . Observe que em relação aos sinais tonais apresentados pela Figura 4.18, este possui as harmônicas, características do sinal de onda quadrada. Como o aplicativo tem cunho educacional, este detalhe pode servir como aprendizado, já que está diretamente ligado a teoria das Séries de Fourier.

**Figura 4.19:** Espectro do áudio de onda quadrada na Unity, com frequência igual a  $440\text{ Hz}$ , e 512 pontos.



**Fonte:** Autoria Própria

Assim como os gráficos dos sinais no tempo, vale ressaltar que a escolha de 512 pontos, não é perfeita para resoluções gráficas. Porém esta quantidade foi mantida, devido à apresentação de um resultado condizente com os objetivos do projeto, por deixar o programa mais rápido, e também facilitar a programação do sistema.

### 4.5.3 Distortion Filter

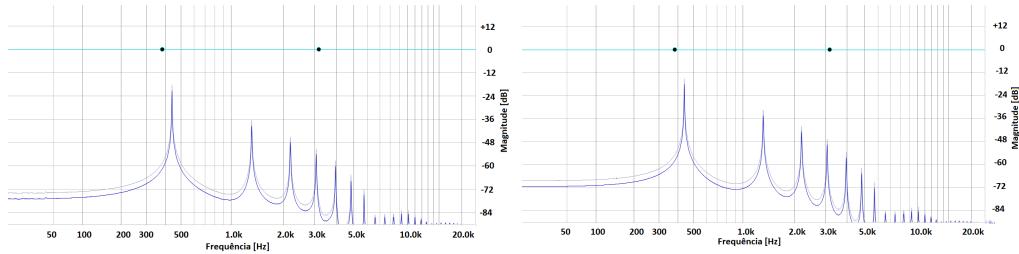
Assim como a simulação, no estudo dos resultados apresentados pelo filtro de distorção da Unity, foi utilizado o mesmo arquivo de áudio tonal com frequência igual a  $440\text{ Hz}$ . E assim, o efeito foi aplicado, em que seu resultado foi extraído utilizando o *software* Audacity. Em seguida, a extração foi alocada no Reaper para análise.

Como já mencionado, a componente tem seu nível de distorção variável entre 0 a 1. Dessa forma, este parâmetro foi regulado até encontrar sons semelhantes com os apresentados na Seção 4.4.2.

A Figura 4.20 mostra um comparativo dos espectros com nível 0,8 de distorção na Unity, e ganho igual a 5 da simulação. Note que os resultados são bem parecidos, existindo apenas uma diferença bem sutil de ganho entre elas. Isso implicou em um áudio bastante semelhante.

**Figura 4.20:** Comparativo do efeito da Distorção da Unity com simulação no Matlab.

(a) Nível de Distorção do filtro na Unity igual a 0,8.  
(b) Ganho de Distorção igual a 5 na simulação do Matlab.

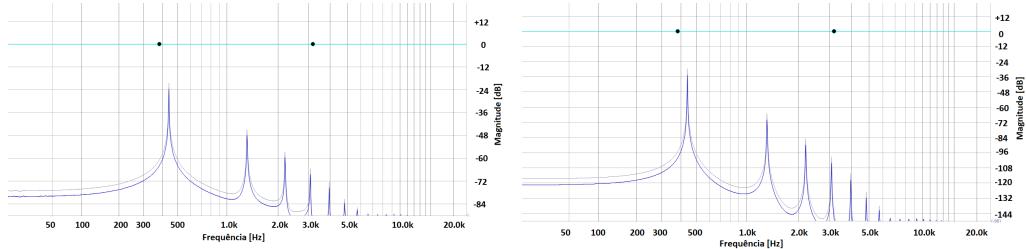


**Fonte:** Autoria Própria

O mesmo ocorre para o nível 0,5 de Distorção na Unity, em comparação com ganho igual a 2. O resultado pode ser visto na Figura 4.21.

**Figura 4.21:** Comparativo do efeito da Distorção da Unity com simulação no Matlab.

(a) Nível de Distorção do filtro na Unity igual a 0,5. (b) Ganho de Distorção igual a 2 na simulação do Matlab.

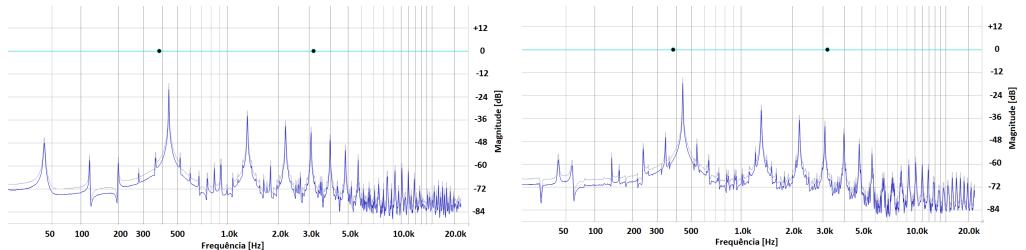


**Fonte:** Autoria Própria

Por fim, para nível de distorção máximo e igual a 1 na Unity, o resultado sonoro se equivale ao ganho de 1.000 da simulação. Isso pode ser evidenciado pela Figura 4.22.

**Figura 4.22:** Comparativo do efeito da Distorção da Unity com simulação no Matlab.

(a) Nível de Distorção do filtro na Unity igual a 1. (b) Ganho de Distorção igual a 1.000 na simulação do Matlab.



**Fonte:** Autoria Própria

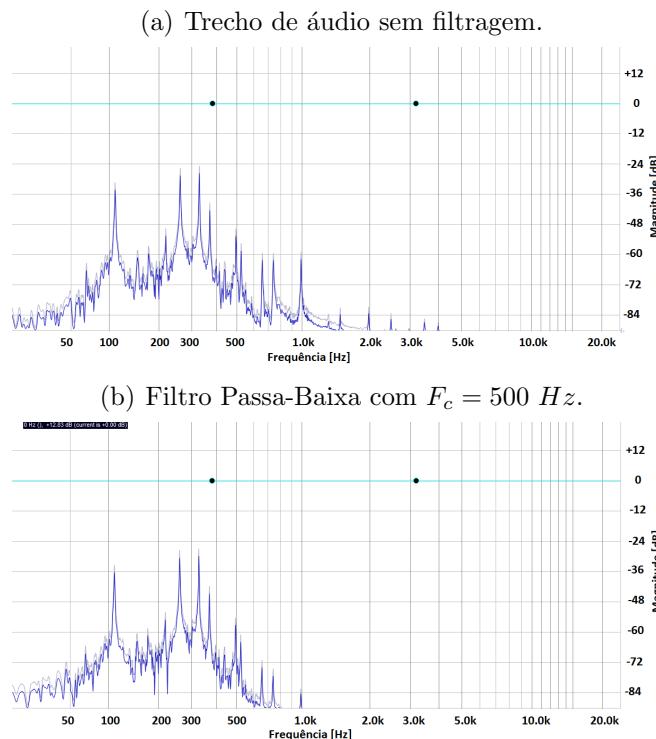
Portanto, o filtro de distorção disponível na Unity, de fato aplica o efeito esperado, pois em comparação com as simulações, os resultados foram semelhantes. Além disso, também é possível identificar que o modelamento matemático da componente, é equivalente a proposta por Zölzer (2011).

#### 4.5.4 Filtragem

O resultado foi analisado de maneira auditiva, e visualmente pelo espectro. Para isso, foi utilizado um trecho sonoro de guitarra, extraído do vídeo de De-Marco (2009). A partir desse áudio, foram aplicados tanto filtro Passa-Baixas (FPB), quanto Passa-Altas (FPA).

O FPB foi setado com uma frequência de corte ( $F_c$ ) equivalente a  $500\text{ Hz}$ . A Figura 4.23 mostra um comparativo do sinal sem e com filtro. Note que de fato uma atenuação ocorre para altas frequências, e está de acordo com o esperado.

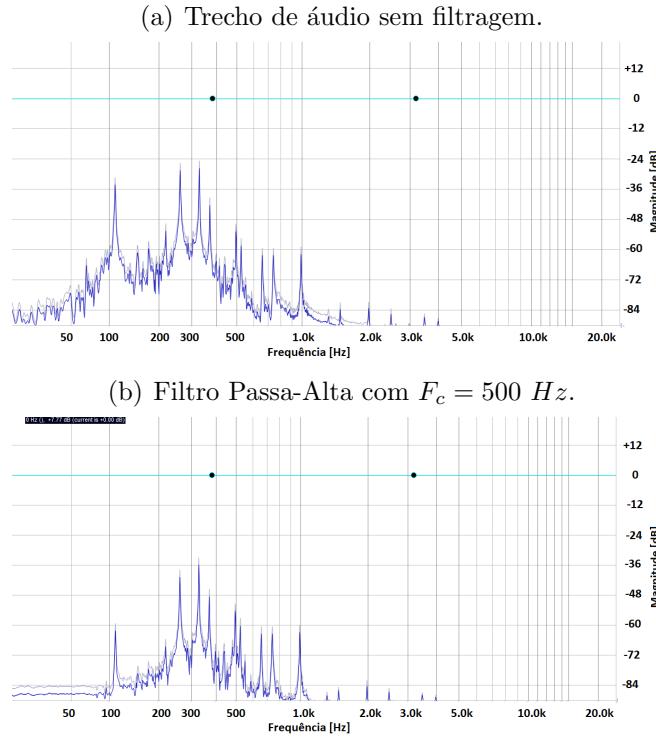
**Figura 4.23:** Comparativo de som proveniente de um trecho de guitarra com e sem filtragem passa-baixa.



**Fonte:** Autoria Própria

O mesmo foi realizado com o Filtro Passa-Altas, que também foi setado para uma  $F_c = 500\text{ Hz}$ . A Figura 4.24 mostra o comparativo, onde é possível observar uma certa atenuação para baixas frequências.

**Figura 4.24:** Comparativo de som proveniente de um trecho de guitarra com e sem filtragem passa-alta.



**Fonte:** Autoria Própria

Os resultados são suficientes para os objetivos do projeto, pois projetar a ordem do filtro, topologia, e outras características não fazem parte do escopo. O esperado foi a filtragem do sinal, independente dos atributos que vão além da frequência de corte.

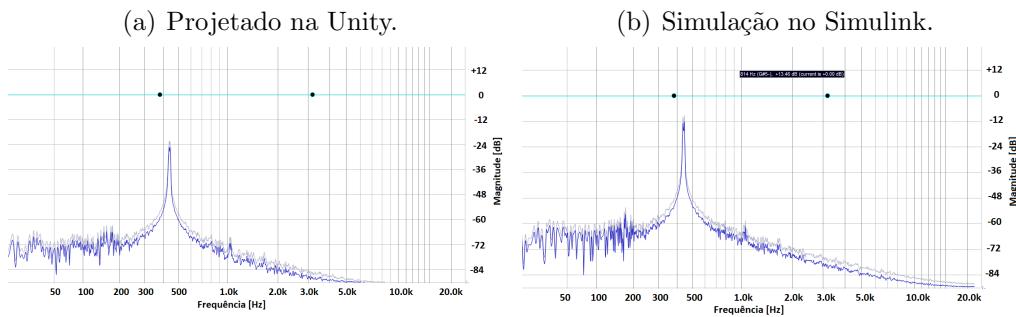
#### 4.5.5 Chorus

Para o Chorus desenvolvido na Unity, foram utilizados os mesmos parâmetros da simulação (Seção 4.4.3). Ou seja  $Delay = 441$ ,  $Depth = 100$ , e  $Rate$  variável. Aplicado a um sinal tonal de frequência igual a  $440 \text{ Hz}$ . O ganho de cada cópia ( $G$ ), foi setado com  $0,3$ . Como já explicado na seção referida, esta configuração representa um Chorus bem forte, porém com estes valores torna a análise do espectro mais nítida.

Primeiramente, pela análise auditiva do efeito, com aplicações em um trecho de guitarra, discurso e sinais tonais, percebeu-se que o efeito estava conforme o esperado. Toda esta percepção, ocorreu devido a referências de outras músicas que possuem o efeito.

Em comparação com os resultados obtidos com a simulação, o Chorus desenvolvido na Unity apresentou resultados satisfatórios, porém não idênticos com os simulados, o que é natural, já que ambos foram aplicados em *softwares* distintos. Para um *Rate* igual a  $1\text{ Hz}$ , da mesma forma como resultado no Simulink, as cópias do sinal são perceptíveis, porém com uma variação de atraso leve, e com pouca distorção no sinal. O comparativo pode ser observado na Figura 4.25. Note que o espetro resultante foi semelhante com o simulado, da mesma forma como o som.

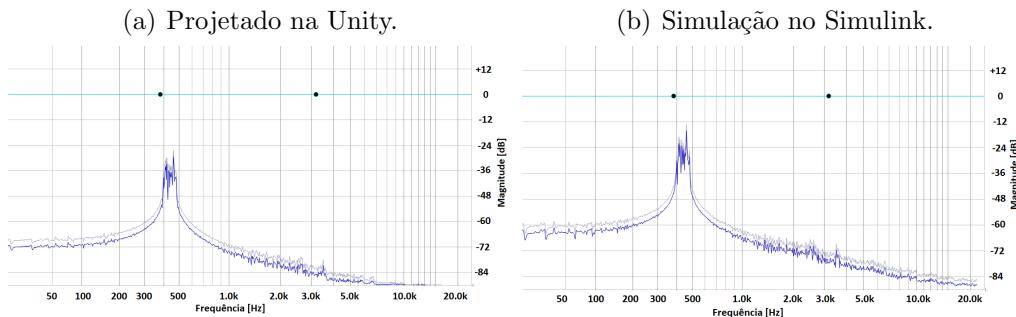
**Figura 4.25:** Comparativo do espetro com Chorus extraído do Simulink e Unity, aplicado a um sinal tonal de  $440\text{ Hz}$ ,  $\text{Delay} = 441$ ,  $\text{Depth} = 100$ ,  $\text{Rate} = 1\text{ Hz}$ .



**Fonte:** Autoria Própria

Conforme o aumento do *Rate*, a variação de atraso e as cópias se tornaram mais evidentes. Em consequência disso, mais distorcido o sinal ficou. Para um *Rate* igual a  $5\text{ Hz}$ , o som resultado pelo Chorus da Unity também se assemelhou com da simulação. A Figura 4.26 mostra os espectros de ambos os áudios, onde é possível perceber a semelhança dos sinais na frequência.

**Figura 4.26:** Comparativo do espetro com Chorus extraído do Simulink e Unity, aplicado a um sinal tonal de  $440\text{ Hz}$ ,  $\text{Delay} = 441$ ,  $\text{Depth} = 100$ ,  $\text{Rate} = 5\text{ Hz}$ .

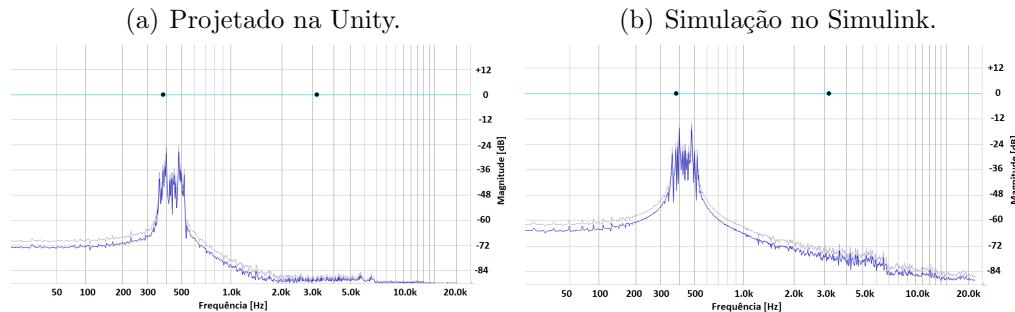


**Fonte:** Autoria Própria

O mesmo ocorre para um *Rate* igual a  $10\text{ Hz}$ , ou seja, o áudio resultante

pelo efeito projetado na Unity foi semelhante com o simulado para os mesmos parâmetros. O espectro comparativo é apresentado pela Figura 4.27.

**Figura 4.27:** Comparativo do espectro com Chorus extraído do Simulink e Unity, aplicado a um sinal tonal de  $440\text{ Hz}$ ,  $\text{Delay} = 441$ ,  $\text{Depth} = 100$ ,  $\text{Rate} = 10\text{ Hz}$ .



**Fonte:** Autoria Própria

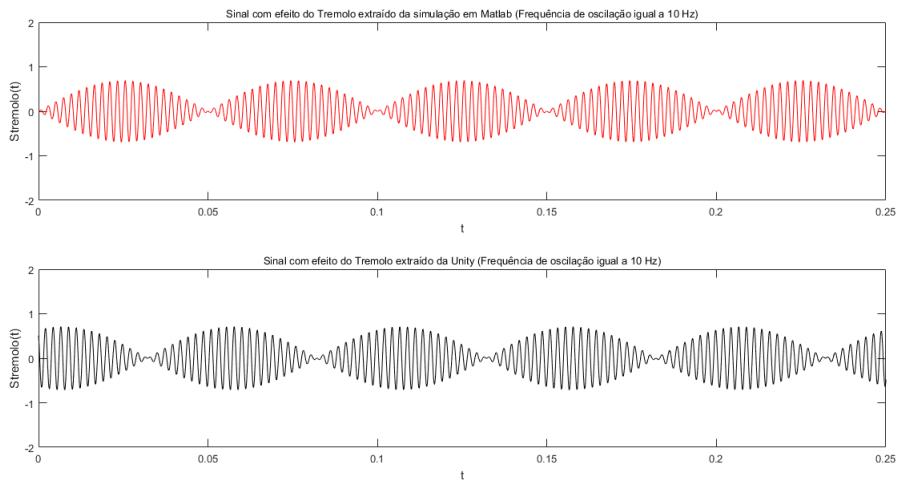
Portanto, o Chorus programado na Unity apresentou resultados satisfatórios, e condizentes com o objetivo do projeto. O efeito cria a percepção sonora de coro, com várias cópias do sinal de atraso variável. No entanto, o parâmetro referente ao ganho ( $G$ ) não pode ser muito alto, podendo ocasionar uma distorção indesejada ao som. Em arquivos “.wav” o desempenho é melhor em relação a “.mp3”, já que devido à qualidade do primeiro, permite valores de ganhos maiores, sem ocasionar ruídos.

#### 4.5.6 Tremolo

Da mesma maneira como realizado nos outros efeitos, o áudio foi extraído e comparado com a simulação. Neste caso, o sinal foi analisado no tempo, com o uso do Matlab. O arquivo de som tonal, de frequência  $440\text{ Hz}$  foi aplicado.

A Figura 4.28 mostra os resultados de Tremolo, com oscilação de frequência igual a  $10\text{ Hz}$ . Note que o efeito desenvolvido na Unity é muito semelhante à simulação, o que evidencia a qualidade do resultado. Com isso, o Tremolo programado no *software* de fato aplica o efeito esperado.

**Figura 4.28:** Comparativo do Tremolo desenvolvido na Unity com a simulação em Matlab.



**Fonte:** Autoria Própria

## 4.6 Aplicativo de Efeitos de Áudio Unity 3D

Da mesma forma como foi apresentado a etapa do jogo, as cores do aplicativo também foram alteradas, com o objetivo de melhorar a impressão do trabalho. Com isso, a Figura 4.29 mostra a interface principal, onde é possível escolher o arquivo de som, apertar *play*, pausar, ajustar o volume, e adicionar os efeitos.

**Figura 4.29:** Interface principal do aplicativo de áudio.



**Fonte:** Autoria Própria

Em seguida, a Figura 4.30 mostra o arquivo de som tonal, de frequência igual a 440 Hz, sendo tocado. O *software* exibe seu espectro, a um certo volume, e sem nenhum efeito aplicado.

**Figura 4.30:** Áudio tonal de frequência igual a 440 Hz sendo tocado, com espectro exibido.



**Fonte:** Autoria Própria

Para trocar a forma de exibição, basta clicar na opção tempo. A Figura 4.31 mostra o resultado em questão. A troca entre os tipos de plot é rápida.

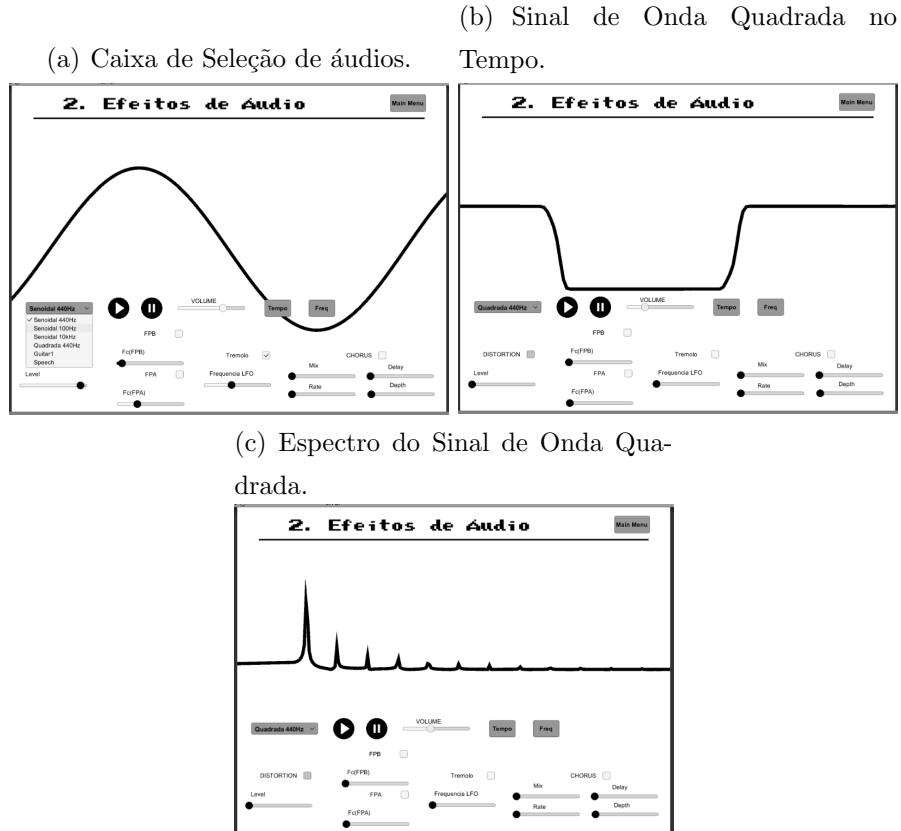
**Figura 4.31:** Áudio tonal de frequência igual a 440 Hz sendo tocado, com sinal no tempo exibido.



**Fonte:** Autoria Própria

Para trocar o áudio, basta clicar na caixa de listagem no lado esquerdo. Existem ao todo 6 opções, sendo 3 sons tonais, 1 onda quadrada, 1 trecho de guitarra, e um discurso fornecido pela própria Unity. A Figura mostra as opções em aberto, e outro áudio, uma onda quadrada de 440 Hz sendo tocada.

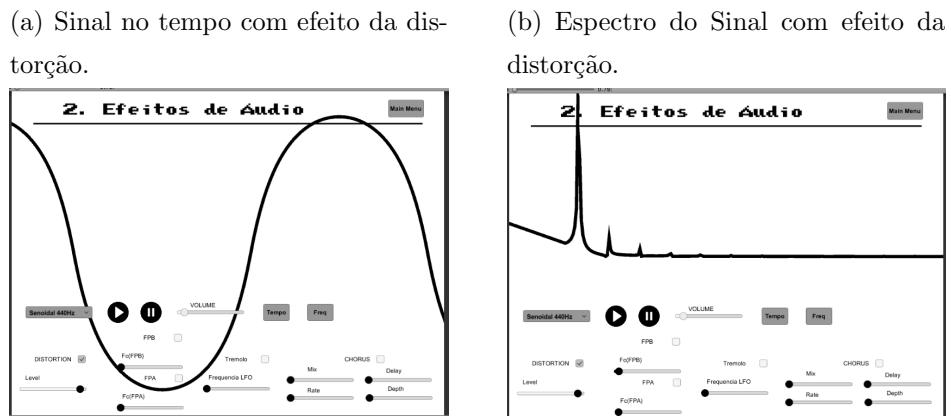
**Figura 4.32:** Seleção de áudios, com o um arquivo de onda quadrada e frequência de 440 Hz.



**Fonte:** Autoria Própria

Os efeitos estão disponibilizados abaixo, que são habilitados por meio de *checkbox*, em seguida basta ajustar o parâmetro desejado. A Figura 4.33 mostra um sinal tonal de 440 Hz, com efeito da distorção aplicado. Note que o formato do sinal no tempo, e seu espectro, são bastante característicos do efeito, conforme já apresentado anteriormente.

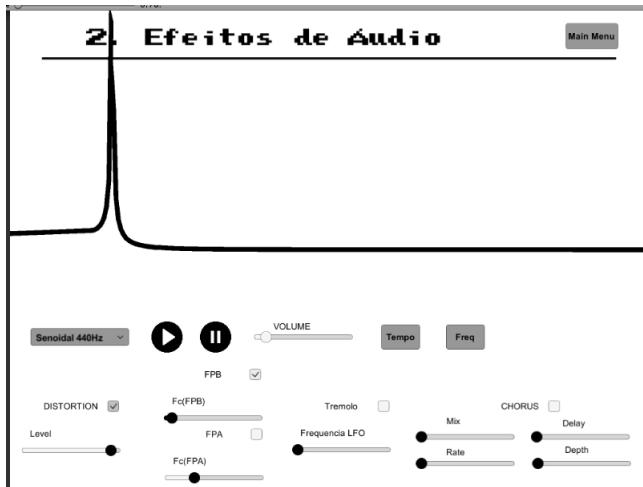
**Figura 4.33:** Efeito da distorção aplicado ha um sinal tonal de frequência 440 Hz.



**Fonte:** Autoria Própria

Por fim, é possível também aplicar os efeitos de maneira simultânea, configurados em cascata. Observe a Figura 4.34, onde em um sinal distorcido, foi aplicado um filtro Passa-Baixas, para atenuar as harmônicas adicionadas pela distorção.

**Figura 4.34:** Audio tonal de frequência igual a 440 Hz sendo tocado, com distorção e FPB ativados.



**Fonte:** Autoria Própria

O aplicativo dispõe ao usuário a oportunidade do aprendizado pela experimentação, pois, a partir de um arquivo sonoro, o aluno pode aplicar os efeitos desejados, verificar seu comportamento auditivo e gráfico, podendo relacionar com a teoria de Séries de Fourier. Por exemplo, pode-se tocar o arquivo de uma onda quadrada, e em seguida filtrar as frequências acima do triplo da fundamental, isso implicará na transformação do sinal original em uma senoide.

## 4.7 Validação

Ao todo compareceram 9 alunos, onde 5 formaram a Turma Sem *Software*, e 4 tiveram acesso ao aplicativo para fazer a prova (Turma Com *Software*). Um número baixo, porém ainda assim foi importante para a coleta de dados da validação. Todas as questões valiam 1 ponto, assim a prova possui um total de 4. Os dados são apresentados por aluno, que tiveram seus nomes ocultados, e seu desempenho de acerto por questão e geral.

A Tabela 4.1 mostra o desempenho dos estudantes que não usaram o aplicativo. Note que o aproveitamento foi bem fraco, cerca de 15% na média, tendo 2 alunos que zeraram e outros 3 que acertaram apenas 1 pergunta. A Questão 1 (Q1) foi a mais acertada, com 2 no total, e a Questão 4 (Q4) em seguida com 1 acerto. As perguntas que envolviam Séries de Fourier, Questões 2 e 3 (Q2 e Q3), não houve acertos. O teste pode ser visto no Apêndice A.

**Tabela 4.1:** Desempenho da Turma Sem *Software*

	Q1	Q2	Q3	Q4	Teve aula sobre Séries de Fourier	Desempenho
Aluno 1	1	0	0	0	Não	25%
Aluno 2	1	0	0	0	Não	25%
Aluno 3	0	0	0	0	Sim	0%
Aluno 4	0	0	0	0	Sim	0%
Aluno 5	0	0	0	1	Não	25%

**Fonte:** Autoria Própria

Já a turma que usou o *software* para fazer a prova, teve um desempenho melhorado. A Tabela 4.2 mostra os resultados. Note que 2 alunos obtiveram aproveitamento de 50% na prova, e 1 com mais da metade de acerto (75%). Isso resultou em uma performance média de 50% da Turma Com *Software* na prova, maior que a Turma Sem *Software*.

**Tabela 4.2:** Desempenho da Turma Com *Software*

	Q1	Q2	Q3	Q4	Teve aula sobre Séries de Fourier	Desempenho
Aluno 6	1	0	0	1	Sim	50%
Aluno 7	0	1	0	0	Sim	25%
Aluno 8	1	1	0	0	Sim	50%
Aluno 9	0	1	1	1	Não	75%

**Fonte:** Autoria Própria

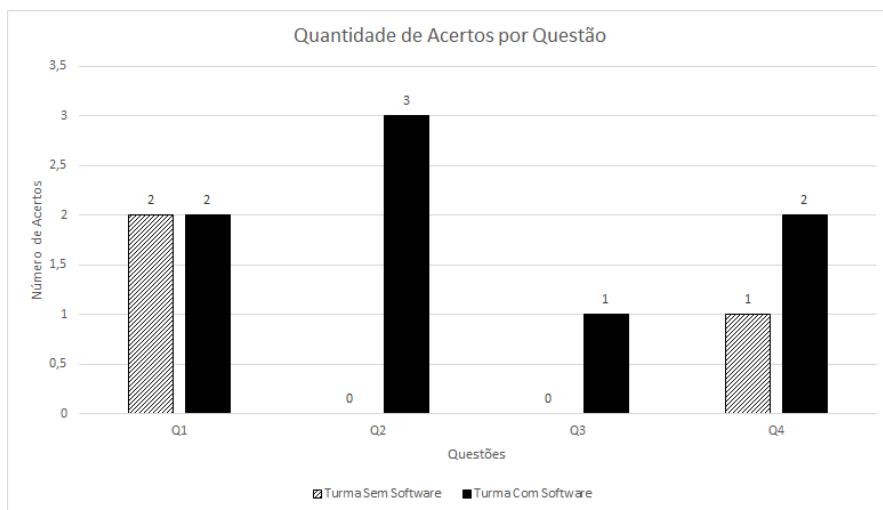
O resultado melhor da Turma Com *Software* fica mais claro no gráfico da Figura 4.35. A primeira questão não apresentou diferenças, com 2 acertos cada. Já nas questões 2 e 3 houve disparidade, principalmente em Q2, onde tiveram 3 acertos no total da turma que usou o *software*, e nenhum da outra. E na questão 4 teve 1 acerto a mais em relação a turma que não usou o aplicativo.

Outro elemento a ser destacado, é o fato dos alunos terem visto o assunto de Séries de Fourier não influenciou no desempenho das provas. Basta observar pelas Tabelas 4.1 e 4.2, que o aluno com maior nota (Aluno 9) não tinha visto a matéria, já os Alunos 3 e 4, que viram o tema, obtiveram desempenho mínimo (0%). Além disso, o professor aproveitou a disciplina de Circuitos Elétricos 1 para ensinar o assunto de forma resumida, em uma aula anterior, que teve o uso de recursos computacionais. No dia da validação uma breve aula sobre o tema também foi passada para os alunos da Turma Sem *Software*.

A interação com as Séries de Fourier que o *software* oferece, proporciona ao estudante o aprendizado experimental e visual da teoria. Ao acertar um nível, ele aprende a lógica por trás da representação de uma onda quadrada e triangular. O fato do desempenho nas questões relativas a este tema (Q2 e Q3), terem sido melhor na turma que usou o aplicativo, mostra o bom funcionamento do mecanismo didático do projeto.

Já o aplicativo de áudio proporciona o aprendizado por descoberta, onde o aluno pode testar sons e sinais aplicados a filtros e efeitos, o que liga com a teoria. Assim, a turma que usou o aplicativo também teve um aproveitamento melhor na última questão.

**Figura 4.35:** Comparativo de quantidade de acertos por questão das Turmas com e sem o uso do *software*.



**Fonte:** Autoria Própria

# 5 Conclusões e Sugestões de Trabalhos Futuros

No capítulo anterior foram apresentados os Resultados colhidos do projeto. Neste, portanto, serão descritas as conclusões obtidas a partir das informações coletadas. Além disso, serão retratados as sugestões de trabalhos futuros, com propostas de melhoria, e evolução do trabalho.

## 5.1 Conclusões

O jogo digital apresentou um bom funcionamento técnico, representando fielmente os gráficos matemáticos controlados pelo jogador. Além disso, as fases apresentam boa fundamentação teórica com as séries de Fourier, o que conecta o jogador/estudante com a teoria a ser estudada. Esta etapa do projeto resultou também no artigo “DESENVOLVIMENTO DE SOFTWARE PARA APRIMORAMENTO DO ENSINO EM ENGENHARIA ELÉTRICA”, aprovado e apresentado no XLV Congresso de Educação em Engenharia (COBENGE 2017), que pode ser visto no Apêndice E.

O aplicativo de áudio resultou em uma boa fidelidade com os conceitos teóricos dos efeitos. Em comparação com as simulações, os blocos desenvolvidos na Unity, ou utilizações de componentes prontas, tiveram semelhanças sonoras e visuais (Análise no tempo ou frequência). Além disso, o sistema programado funcionou adequadamente, permitindo a aplicação de todos efeitos de áudio, realizar a transição de *plots*, e trocar os arquivos de música de forma eficaz.

Por fim, quando todo o *software* foi apresentado aos alunos da disciplina de Circuitos Elétricos 1, o trabalho muito bem recebido pelos estudantes. A grande maioria interagiu bastante, interessando-se pelo tema proposto, e o desafio que o jogo proporcionava. Os resultados da prova aplicada contiveram um bom desempenho da turma que usou o aplicativo para resolver, tendo um aumento 35% em relação à performance média da turma que não utilizou o *software*.

Portanto, o trabalho resolvido cumpriu com seu propósito, de desenvolver um *software* educativo, que transformasse a teoria vista em sala de aula, em um aprendizado lúdico e visual. Vale ressaltar que o *software* não substitui o método tradicional, e sim funciona como um meio complementar à aula, que pode ser explorado pelo docente e discente, já que toda a teoria apresentada em sala, pode ser revista, experimentada, e trabalhada com os desafios do jogo, e também com o aplicativo de áudio, que permite a conexão da teoria com a prática.

## 5.2 Sugestões de Trabalhos Futuros

O *software* funciona muito bem, porém seu desempenho no quesito de programação pode ser mais eficiente. Um estudo mais aprofundado da linguagem C#, e comunicação de Scripts da Unity, pode contribuir para uma melhoria de velocidade do *software*, para que o controle dos gráficos fique mais rápido no jogo. No aplicativo de áudio, quando alguns efeitos como Chorus e Tremolo são aplicados, o *software* fica um pouco lento, e assim, com essa sugestão este processamento pode ser melhorado, e consequentemente será possível aumentar a resolução do *plot* dos sinais.

Realizar o desenvolvimento deste projeto para dispositivos mobile, pode contribuir com sua acessibilidade, e também ampliar os recursos que podem ser implementados. A Unity possibilita a realização de projetos em várias plataformas diferentes, e seu método de programação e montagem não é muito diferente do que foi apresentado neste trabalho. Existem algumas diferenças a ser levadas em consideração, como resolução de tela, e capacidade de processamento do dispositivo.

Outra sugestão é o acréscimo de novos níveis ao jogo, onde abordem outros sinais periódicos, que podem ser dos mais simples aos complexos. Com isso, é possível adicionar níveis onde sejam incrementados outras harmônicas, além das já utilizadas na representação. Por exemplo, no caso das onda quadrada e triangular, apenas as harmônicas ímpares foram fornecidas, pois as harmônicas pares destes sinais são nulas. Assim, outro nível que forneça também as pares para controle, pode estimular um aprendizado mais completo.

No quesito dos efeitos de áudio, um desenvolvimento mais aprofundado para processamento digital de áudio, possibilita uma melhoria na qualidade sonora. O que pode torná-los cada vez mais fieis, e sem ruídos, como é o caso do Chorus, que dependendo dos valores de ganho, o sinal apresenta distorções. Do mesmo

modo que é possível explorar outros efeitos, fazer uma interface com microfone do computador, processamento de áudio com mais canais, projeto de filtros digitais, entre outras aplicações.

# Referências

- AGUIAR, A. et al. A importância das atividades práticas no ensino de instalações elétricas prediais - um estudo de caso. In: CONGRESSO BRASILEIRO DE EDUCAÇÃO EM ENGENHARIA, 44., 2016, Natal. *Anais...* Natal: Associação Brasileira de Educação em Engenharia, 2016.
- BOISCLAIR, C. *Press Start*. 2001. [Online; acesso 12 de Set. 2017]. Disponível em: <<http://www.dafont.com/pt/press-start.font>>.
- CAELUM. *C# e Orientação a Objetos*. 2013. [Online; acesso 03 de Dez. 2017]. Disponível em: <<https://www.caelum.com.br/apostila-csharp-orientacao-objetos/>>.
- CHAMBERLIN, H. *Musical Applications of Microprocessors*. 2<sup>a</sup>. ed. [S.l.]: Hayden Books, 1987.
- CHAVES, R. et al. Desigmps: Um jogo de apoio ao ensino de modelos de qualidade de processos de software, baseado em mapas conceituais. In: CONGRESSO BRASILEIRO DE EDUCAÇÃO EM ENGENHARIA, 39., 2011, Blumenau. *Anais...* Blumenau: Associação Brasileira de Educação em Engenharia, 2011.
- CZUBAK, A. J.; RAHEJA, G. Guitar effects processor using DSP. 09 2017. [Online; acesso 20 de Jan. 2018]. Disponível em: <<http://ee.bradley.edu/projects/proj2008/gegudps/>>.
- DEMARCO, J. *EFFECTS 101 Chorus*. 2009. [Online; acesso 19 de Nov. 2017]. Disponível em: <<https://www.youtube.com/watch?v=zmN7fK3fKUE>>.
- FALCÃO, M. *Efeitos para Guitarra e Outros instrumentos*. [S.l.]: Funalfa, v. 2, 2015.
- HEADUP GAMES. *Bridge Constructor*. 2017. [Online; acesso 02 de Jan. 2018]. Disponível em: <<http://www.bridgeconstructor.com>>.
- LIMA, C. G. M. *Criação, construção, uso e análise de um jogo digital voltado ao ensino de circuitos elétricos*. Dissertação (Mestrado Profissional em Ensino de Física) — Curso de Mestrado Nacional Profissional de Ensino de Física, IFRN, Natal, 2015.
- LOTUFO, A. D. P. Séries de fourier. UNESP, 2014. [Online; acesso em: 13 de Mai.2017. Disponível em: <<http://www.feis.unesp.br/Home/departamentos/engenhariaeletrica/mcap03.pdf>>.
- MCDONALD, E. The global games market will reach \$108.9 billion in 2017 with mobile taking 42%. NEWZOO, 2017. [Online; acesso 18 de Jan. 2018]. Disponível em: <<https://newzoo.com/insights/articles/the-global-games-market-will-reach-108-9-billion-in-2017-with-mobile-taking-42/>>.

- MICROSOFT DOCS. *Hello World – seu primeiro programa (Guia de Programação em C#)*. 2015. [Online; acesso 03 de Dez. 2017]. Disponível em: <<https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/inside-a-program/hello-world-your-first-program>>.
- MORSCH I.B.; ROCHA, M. Jogos didáticos aplicados ao ensino de engenharia projeto e construção de catapultas do tipo trabuco. In: CONGRESSO BRASILEIRO DE EDUCAÇÃO EM ENGENHARIA, 39., 2011, Blumenau. *Anais...* Blumenau: Associação Brasileira de Educação em Engenharia, 2011.
- NATIONAL INSTRUMENTS. O que é condicionamento de sinal? 2012. [Online; acesso 29 de Nov. 2017]. Disponível em: <<http://www.ni.com/white-paper/10630/pt/>>.
- NAVARRO M.P.; MARQUES, A. E. A. I. Jogo para aprendizagem vivencial de conceitos de restrições de projetos no ensino de engenharia. In: CONGRESSO BRASILEIRO DE EDUCAÇÃO EM ENGENHARIA, 42., 2014, Juiz de Fora. *Anais...* Juiz de Fora: Associação Brasileira de Educação em Engenharia, 2014.
- NILSSON J. W.; RIEDEL, S. A. *Circuitos Elétricos*. 8<sup>a</sup>. ed. [S.I.]: PEARSON Prentice Hall, 2009.
- NINTENDO. *Brain Age: Concentration Training*. 2017. [Online; acesso 02 de Jan. 2018]. Disponível em: <<http://brainage.nintendo.com/why-concentration-training/>>.
- ONIRIA. *Simulador Didático SimMag 3D Automatus*. 2016. [Online; acesso 04 de Dez. 2017]. Disponível em: <<https://oniria.com.br/simulador-didatico-simmaq-3d-automatus/>>.
- OPPENHEIM A. V.; SCHAFER, R. W. *Sinais e Sistemas*. 2<sup>a</sup>. ed. [S.I.]: PEARSON Prentice Hall, 2010.
- PACCOLA, F. et al. Jogo do barco: Uma versão inovadora incluindo mapeamento de fluxo de valor. In: CONGRESSO BRASILEIRO DE EDUCAÇÃO EM ENGENHARIA, 42., 2014, Juiz de Fora. *Anais...* Juiz de Fora: Associação Brasileira de Educação em Engenharia, 2014.
- PATSKO, L. F. *Processamento Digital de Sinais de Áudio com STM32F4*. Monografia (Especialização em Sistemas Eletrônicos Embarcados.) — UEL, Londrina, 2015.
- REAPER. *About*. 2017. [Online; acesso 03 de Dez. 2017]. Disponível em: <<https://www.cockos.com/reaper/about.php>>.
- RODRIGUES, J. et al. Desenvolvimento de jogos educativos para dispositivos portáteis: aliando ensino de engenharia, computação e ciências. In: CONGRESSO BRASILEIRO DE EDUCAÇÃO EM ENGENHARIA, 42., 2014, Juiz de Fora. *Anais...* Juiz de Fora: Associação Brasileira de Educação em Engenharia, 2014.
- SAVI R.; ULBRICHT, V. R. Jogos digitais educacionais: benefícios e desafios. *Revista Novas Tecnologias na Educação*, v. 6, n. 1, p. 1–10, 2008.

- SCHEGOCHESKI, A. et al. Jogo das Águas: Conhecendo o processo de tratamento da Água. In: CONGRESSO BRASILEIRO DE EDUCAÇÃO EM ENGENHARIA, 42., 2014, Juiz de Fora. *Anais...* Juiz de Fora: Associação Brasileira de Educação em Engenharia, 2014.
- SCHELL, J. *The Art of Game Design*. [S.l.]: Morgan Kaufmann Publishers, 2008.
- SMITH, S. *The Scientist and Engineer's Guide to Digital Signal Processing*. 2<sup>a</sup>. ed. [S.l.]: California Technical Publishing, 1999.
- SQUAD. *About Kerbal Space Program*. 2017. [Online; acesso 02 de Jan. 2018]. Disponível em: <[https://kerbalspaceprogram.com/en/?page\\_id=7](https://kerbalspaceprogram.com/en/?page_id=7)>.
- UNITY. *Audio Clip*. 2017a. [Online; acesso 18 de Set. 2017]. Disponível em: <<https://docs.unity3d.com/Manual/class-AudioClip.html>>.
- UNITY. *Audio Distortion Filter*. 2017b. [Online; acesso 02 de Out. 2017]. Disponível em: <<https://docs.unity3d.com/Manual/class-AudioDistortionFilter.html>>.
- UNITY. *Audio High Pass Filter*. 2017c. [Online; acesso 02 de Out. 2017]. Disponível em: <<https://docs.unity3d.com/Manual/class-AudioHighPassFilter.html>>.
- UNITY. *Audio Listener*. 2017d. [Online; acesso 20 de Set. 2017]. Disponível em: <<https://docs.unity3d.com/Manual/class-AudioListener.html>>.
- UNITY. *Audio Low Pass Filter*. 2017e. [Online; acesso 02 de Out. 2017]. Disponível em: <<https://docs.unity3d.com/Manual/class-AudioLowPassFilter.html>>.
- UNITY. *Audio Source*. 2017f. [Online; acesso 18 de Set. 2017]. Disponível em: <<https://docs.unity3d.com/Manual/class- AudioSource.html>>.
- UNITY. *AudioClip.GetData*. 2017g. [Online; acesso 07 de Set. 2017]. Disponível em: <<https://docs.unity3d.com/ScriptReference/ AudioClip.GetData.html>>.
- UNITY. *AudioClip.SetData*. 2017h. [Online; acesso 07 de Set. 2017]. Disponível em: <<https://docs.unity3d.com/ScriptReference/ AudioClip.SetData.html>>.
- UNITY. *AudioListener.GetSpectrumData*. 2017i. [Online; acesso 18 de Set. 2017]. Disponível em: <<https://docs.unity3d.com/ScriptReference/ AudioListener.GetSpectrumData.html>>.
- UNITY.  *AudioSource.GetOutputData*. 2017j. [Online; acesso 18 de Set. 2017]. Disponível em: <<https://docs.unity3d.com/ScriptReference/ AudioSource.GetOutputData.html>>.
- UNITY. *Button.onClick*. 2017k. [Online; acesso 20 de Nov. 2017]. Disponível em: <<https://docs.unity3d.com/ScriptReference/UI.Button-onClick.html>>.
- UNITY. *Creating and Using Scripts*. 2017l. [Online; acesso 17 de Mai. 2017]. Disponível em: <<https://docs.unity3d.com/Manual/CreatingAndUsingScripts.html>>.

- UNITY. *GameObjects*. 2017m. [Online; acesso 20 de Nov. 2017]. Disponível em: <<https://docs.unity3d.com/Manual/GameObjects.html>>.
- UNITY. *Introduction to components*. 2017n. [Online; acesso 20 de Nov. 2017]. Disponível em: <<https://docs.unity3d.com/Manual/Components.html>>.
- UNITY. *Line Renderer*. 2017o. [Online; acesso 07 de Set. 2017]. Disponível em: <<https://docs.unity3d.com/Manual/class-LineRenderer.html>>.
- UNITY. *LineRenderer.SetPositions*. 2017p. [Online; acesso 07 de Set. 2017]. Disponível em: <<https://docs.unity3d.com/ScriptReference/LineRenderer.SetPositions.html>>.
- UNITY. *MonoBehaviour.Start()*. 2017q. [Online; acesso 20 de Nov. 2017]. Disponível em: <<https://docs.unity3d.com/ScriptReference/MonoBehaviour.Start.html>>.
- UNITY. *MonoBehaviour.Update()*. 2017r. [Online; acesso 20 de Nov. 2017]. Disponível em: <<https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html>>.
- UNITY. *O que é?* 2017s. [Online; acesso 17 de Mai. 2017]. Disponível em: <<https://unity3d.com/pt/unity>>.
- UNITY. *Scenes*. 2017t. [Online; acesso 20 de Nov. 2017]. Disponível em: <<https://docs.unity3d.com/Manual/CreatingScenes.html>>.
- UNITY. *Vector3*. 2017u. [Online; acesso 20 de Nov. 2017]. Disponível em: <<https://docs.unity3d.com/ScriptReference/Vector3.html>>.
- ZACHTRONICS. *About Ruckingenur II*. 2008. [Online; acesso 04 de Dez. 2017]. Disponível em: <<http://www.zachtronics.com/ruckingenur-ii/>>.
- ZACHTRONICS. *About KOHCTPYKTOP: Engineer of the People*. 2009. [Online; acesso 04 de Dez. 2017]. Disponível em: <<http://www.zachtronics.com/kohctpyktop-engineer-of-the-people/>>.
- ZACHTRONICS. *TIS-100*. 2015. [Online; acesso 04 de Dez. 2017]. Disponível em: <<http://www.zachtronics.com/tis-100/>>.
- ZÖLZER, U. *Digital Audio Signal Processing*. [S.l.]: John Wiley & Sons Ltd, 1999.
- ZÖLZER, U. *DAFX - Digital Audio Effects*. 2<sup>a</sup>. ed. [S.l.]: John Wiley & Sons Ltd, 2011.

# Apêndice A - Teste Aplicado para Validação

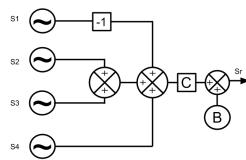
## A.1 Teste

### Teste - Projeto TCC - Séries de Fourier

#### Circuitos Elétricos 1

Orientador: Ernesto Ferreyra Ramirez  
Orientando: Matheus Raphael Elero  
Aluno:  
Data: 24/08/17

**1** - Dado o Diagrama de Blocos a baixo. Qual deve ser  $S_1$  para que o sinal resultante ( $S_r$ ) seja igual a zero.

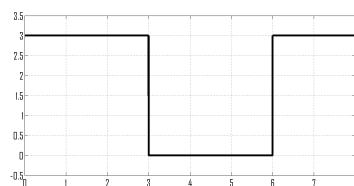


#### Dados:

$C = 1$ ,  $B = 0$ ,  $S_2 = 2 \cdot \text{Cos}(2 \cdot \pi \cdot f \cdot t)$ ,  $S_3 = \text{Cos}(2 \cdot \pi \cdot f \cdot t - 90^\circ)$  e  $S_4 = -2 \cdot \text{Sin}(2 \cdot \pi \cdot f \cdot t + 90^\circ)$

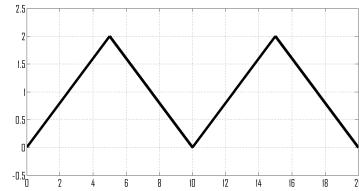
- a)  $S_1 = -\text{Sin}(2 \cdot \pi \cdot f \cdot t + 90^\circ)$
- b)  $S_1 = -\text{Sin}(2 \cdot \pi \cdot f \cdot t)$
- c)  $S_1 = \text{Sin}(2 \cdot \pi \cdot f \cdot t)$
- d)  $S_1 = -\text{Cos}(2 \cdot \pi \cdot f \cdot t)$
- e)  $S_1 = \text{Cos}(2 \cdot \pi \cdot f \cdot t)$

**2** - Dado a onda quadrada abaixoo, calcule  $a_m$  e  $b_m$



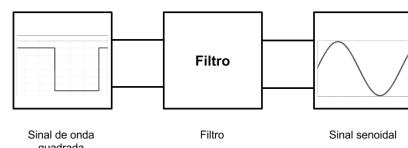
- a)  $a_m = \frac{3}{\pi m} [1 - \text{Cos}(m\pi)]$  e  $b_m = 0$
- b)  $a_m = 0$  e  $b_m = \frac{3}{\pi m} [1 - \text{Cos}(\frac{1}{\pi m})]$
- c)  $a_m = 0$  e  $b_m = \frac{3}{\pi m} [1 - \text{Cos}(\frac{3}{\pi m})]$
- d)  $a_m = 0$  e  $b_m = \frac{3}{\pi m} [1 - \text{Cos}(m\pi)]$
- e)  $a_m = 0$  e  $b_m = \frac{\pi m}{3} [1 - \text{Cos}(\frac{1}{\pi m})]$

**3** - Dado a onda triangular abaixoo, calcule  $a_m$  e  $b_m$



- a)  $a_m = \frac{4}{\pi^2 m^2} [\text{Cos}(m\pi) - 1]$  e  $b_m = 0$
- b)  $a_m = \frac{2}{\pi m} [\text{Cos}(m\pi) - 1]$  e  $b_m = 0$
- c)  $a_m = \frac{4}{\pi^2 m^2} [1 - \text{Cos}(m\pi)]$  e  $b_m = 0$
- d)  $a_m = 0$  e  $b_m = \frac{4}{\pi^2 m^2} [\text{Cos}(m\pi) - 1]$
- e)  $a_m = 0$  e  $b_m = \frac{2}{\pi m} [\text{Cos}(m\pi) - 1]$

**4** - Seja um sinal de onda quadrada com frequência  $f$ , e amplitude  $A$ . É possível converte-lo em um sinal senoidal através do uso de filtros.



Assinale a alternativa correta:

- a) FPA(Filtro Passa Alta) com  $f_c = 3 \cdot f$
- b) FPA(Filtro Passa Alta) com  $f_c = 3 \cdot f$
- c) FPB(Filtro Passa Baixa) com  $f_c = 5 \cdot f$
- d) FPB(Filtro Passa Baixa) com  $f_c = 7 \cdot f$
- e) FPB(Filtro Passa Baixa) com  $f_c = 3 \cdot f$

Muito obrigado por contribuir com o trabalho, fique a vontade para deixar suas críticas e sugestões a respeito do trabalho:

## A.2 Resolução

### A.2.1 Questão 1

$$S_r = [S_1 \cdot (-1) + S_2 + S_3 + S_4] \cdot C + B \quad (\text{A.1})$$

$$S_r = [S_1 \cdot (-1) + 2 \cdot \cos(2\pi ft) + \cos(2\pi ft - 90^\circ) - 2 \cdot \sin(2\pi ft + 90^\circ)] \cdot C + B \quad (\text{A.2})$$

Como  $C = 1$  e  $B = 0$ :

$$S_r = S_1 \cdot (-1) + 2 \cdot \cos(2\pi ft) + \cos(2\pi ft - 90^\circ) - 2 \cdot \sin(2\pi ft + 90^\circ). \quad (\text{A.3})$$

Como  $\sin(2\pi ft + 90^\circ) = \cos(2\pi ft)$ :

$$S_r = S_1 \cdot (-1) + \cos(2\pi ft - 90^\circ). \quad (\text{A.4})$$

Para  $S_r = 0$ , seja a seguinte relação:

$$S_1 = \cos(2\pi ft - 90^\circ). \quad (\text{A.5})$$

Já que  $\sin(2\pi ft) = \cos(2\pi ft - 90^\circ)$ , a resposta correta é a letra c.

$$S_1 = \sin(2\pi ft) \quad (\text{A.6})$$

### A.2.2 Questão 2

Primeiramente, deve-se equacionar o sinal periódico em uma  $f(x) = f(x+T)$ , ou seja:

$$f(x) = \begin{cases} 3 & \text{se } 0 \leq x \leq 3 \\ 0 & \text{se } 3 \leq x \leq 6 \end{cases} \quad (\text{A.7})$$

Neste caso, o parâmetro L (largura do pulso) corresponde à metade do período, ou seja  $L = 3$ . Assim, os parâmetros são calculados.

$$a_0 = \frac{1}{3} \cdot \int_{-3}^3 f(x) \cdot \cos(0) dx = 3 \quad (\text{A.8})$$

$$a_m = \frac{1}{3} \cdot \int_{-3}^3 f(x) \cdot \cos\left(\frac{m \cdot \pi \cdot x}{3}\right) dx = 0 \quad (\text{A.9})$$

$$b_m = \frac{1}{3} \cdot \int_{-3}^3 f(x) \cdot \sin\left(\frac{m \cdot \pi \cdot x}{3}\right) dx = \frac{3}{m \cdot \pi} \cdot [1 - \cos(m \cdot \pi)] \quad (\text{A.10})$$

Portanto a alternativa correta é a letra d.

### A.2.3 Questão 3

Primeiramente, deve-se equacionar o sinal periódico em uma  $f(x) = f(x+T)$ , ou seja:

$$f(x) = \begin{cases} \frac{2x}{5} & \text{se } 0 \leq x \leq 5 \\ \frac{2x}{5} + 4 & \text{se } 5 \leq x \leq 10 \end{cases} \quad (\text{A.11})$$

O parâmetro L corresponde a metade do período, ou seja  $L = 5$ . Assim, os parâmetros são calculados,

$$a_m = \frac{1}{5} \cdot \int_{-5}^5 f(x) \cdot \cos\left(\frac{m \cdot \pi \cdot x}{5}\right) dx = \frac{2}{5} \cdot \int_0^5 \frac{2x}{5} \cdot \cos\left(\frac{m \cdot \pi \cdot x}{5}\right) dx \quad (\text{A.12})$$

Colocando as constantes fora da integral,

$$a_m = \frac{4}{25} \cdot \int_0^5 x \cdot \cos\left(\frac{m \cdot \pi \cdot x}{5}\right) dx \quad (\text{A.13})$$

Com o uso da relação de integração  $\int u dv = u \cdot v - \int v du$ . Assumindo que  $u = x$ ,  $dv = \cos\left(\frac{m \cdot \pi \cdot x}{5}\right)$ , em consequência que  $du = dx$  e  $v = \frac{5}{m\pi} \sin\left(\frac{m \cdot \pi \cdot x}{5}\right)$ .

Tem-se a seguinte resolução da integral,

$$a_m = \frac{4}{25} \cdot \left[ \frac{5}{m\pi} \sin\left(\frac{m \cdot \pi \cdot x}{5}\right) \Big|_0^5 - \int_0^5 \frac{5}{m\pi} \sin\left(\frac{m \cdot \pi \cdot x}{5}\right) dx \right] \quad (\text{A.14})$$

$$a_m = \frac{4}{m^2 \cdot \pi^2} \cdot [\cos(m \cdot \pi - 1)] \quad (\text{A.15})$$

$$b_m = \frac{1}{5} \cdot \int_{-5}^5 f(x) \cdot \sin\left(\frac{m \cdot \pi \cdot x}{5}\right) dx \quad (\text{A.16})$$

$$b_m = \frac{2}{5} \left[ \cdot \int_{-5}^0 \left( \frac{-2x}{5} + 4 \right) \cdot \sin\left(\frac{m \cdot \pi \cdot x}{5}\right) dx + \int_0^5 \left( \frac{2x}{5} \right) \cdot \sin\left(\frac{m \cdot \pi \cdot x}{5}\right) dx \right] \quad (\text{A.17})$$

Utilizando a mesma relação de integração utilizada para cálculo do  $a_m$ , onde  $u = x$ ,  $dv = \sin\left(\frac{m\pi x}{5}\right)$ ,  $du = dx$ , e  $v = \frac{-5}{m\pi} \cdot \cos\left(\frac{m\pi x}{5}\right)$ , com o objetivo de resolver a integral  $\int x \sin\left(\frac{m\pi x}{5}\right)$ . Calcula-se que

$$b_m = 0 \quad (\text{A.18})$$

Portanto a alternativa correta é a letra a.

#### A.2.4 Questão 4

Seja um sinal de onda quadrada de frequência  $f$ . Sua representação de Fourier mostra que o sinal é composto por uma senoide de frequência  $f$ , somada com outras de frequência  $m \cdot f$ , onde  $m$  é ímpar. A amplitude das harmônicas decai em razão ímpar de  $m$ .

Ou seja, caso as senoides que compõem este sinal sejam filtradas, de forma que fique apenas a principal de frequência  $f$ , o sinal será convertido em uma senoide. Para isso, filtra-se as frequências acima de  $3 \cdot f$ , ou seja um filtro passa-baixa (FPB) de  $f_c = 3 \cdot f$  é utilizado. Portanto letra e é a correta.

# Apêndice B – Script completos das simulações de Séries de Fourier no Matlab

## B.1 Onda Quadrada

```

%%Definição do Tempo
t = 0:.0001:10;
w0=2*10*pi;
%%Parâmetros de Fourier
L = 2;
Onda1 = square(2*pi*0.25*t) + 1;
%%Fourier
Onda2 = L/2 + (2*L/pi)*sin(pi*t);
Onda3 = L/2 + (2*L/pi)*(sin(pi*t/L)+(1/3)*sin(3*
pi*t/L));
Onda4 = L/2 + (2*L/pi)*(sin(pi*t/L)+(1/3)*sin(3*
pi*t/L)+(1/5)*sin(5*pi*t/L));
Onda5 = L/2 + (2*L/pi)*(sin(pi*t/L)+(1/3)*sin(3*
pi*t/L)+(1/5)*sin(5*pi*t/L)+(1/7)*sin(7*pi*t
/L));
Onda6 = L/2 + (2*L/pi)*(sin(pi*t/L)+(1/3)*sin(3*
pi*t/L)+(1/5)*sin(5*pi*t/L)+(1/7)*sin(7*pi*t
/L)+(1/9)*sin(9*pi*t/L));
OndaInf = L/2;
for m=1:+2:1000
    OndaInf = OndaInf +(2*L/pi)*(1/m)*sin(m*pi*t/L);
end
subplot(2,1,2),plot(t,Onda1)
hold on
subplot(2,1,2),plot(t,Onda2,'g')

```

```

hold on
subplot(2,1,2),plot(t,Onda3,'r')
hold on
subplot(2,1,2),plot(t,Onda4,'m')
hold on
subplot(2,1,2),plot(t,Onda5,'y')
hold on
plot(t,Onda6,'k')
hold on
subplot(2,1,1), plot(t,OndaInf,'b')
axis([0,10,-1,3])

```

## B.2 Onda Triangular

```

clear all
clc
close all
%%Definição do Tempo
t = 0:.0001:10;
w0=2*10*pi;
%%Parâmetros de Fourier
L = 2;
%%Geração de Onda Quadrada
Onda1 = sawtooth(2*pi*0.25*t,0.5) + 1;
%%Fourier
Onda2 = L/2 - (4*L/(pi^2))*cos(pi*t/L);
Onda3 = L/2 - (4*L/(pi^2))*(cos(pi*t/L)+(1/(3^2))
    *cos(3*pi*t/L));
Onda4 = L/2 - (4*L/(pi^2))*(cos(pi*t/L)+(1/(3^2))
    *cos(3*pi*t/L)+(1/(5^2))*cos(5*pi*t/L));
Onda5 = L/2 - (4*L/(pi^2))*(cos(pi*t/L)+(1/(3^2))
    *cos(3*pi*t/L)+(1/(5^2))*cos(5*pi*t/L)
    +(1/(7^2))*cos(7*pi*t/L));
Onda6 = L/2 - (4*L/(pi^2))*(cos(pi*t/L)+(1/(3^2))
    *cos(3*pi*t/L)+(1/(5^2))*cos(5*pi*t/L)
    +(1/(7^2))*cos(7*pi*t/L)+(1/(9^2))*cos(9*pi*t/L));

```

```
OndaInf = L/2;
for m=1:+2:1000
    OndaInf = OndaInf +(4*L/(pi^2))*(1/(m^2))*cos(m*pi
        *t/L);
end
subplot(2,1,2),plot(t,Onda1)
hold on
subplot(2,1,2),plot(t,Onda2,'g')
hold on
subplot(2,1,2),plot(t,Onda3,'r')
hold on
subplot(2,1,2),plot(t,Onda4,'m')
hold on
subplot(2,1,2),plot(t,Onda5,'y')
hold on
plot(t,Onda6,'k')
hold on
subplot(2,1,1), plot(t,OndaInf,'b')

axis ([0,10,-1,3])
```

# Apêndice C – Scripts de Desenvolvimento do jogo

A seguir são apresentados os 3 Scripts que compõem o nível 3 do jogo. Nesta cena, o sinal controlado é um somatório de senoides simples, com frequências diferentes. O jogador deve obter o sinal  $S_{esperado}(x) = \text{Sin}(\pi \cdot x)$ .

## C.1 Controle de Sinal $S_r$

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;

public class ControleSinais13 : MonoBehaviour {

    float i = -7;//i é o equivalente ao valor de x inicial
    int pos = 0; //Pos é o indice do vetor de pontos a serem
tracadas as retas
    float yvalue; // Valor em y da função y = f(x)

    Vector3[] positions = new Vector3[300]; //Vetor de
posições
    private LineRenderer lr; //LineRenderer lr
    public Slider A1; //Slider A1, A2...
    public Slider A2;
    public Slider A3;
    public Slider A4;

    void Start () {
```

```

lr = GetComponent<LineRenderer>(); //lr recebe a
// componente LineRenderer do GameObject que contém o
// Script
}

// Update is called once per frame
void Update () {
//A função calcula o valor de y para cada número de x(i),
//que começa em -12 e incrementa 0.1 até completar 300
//pontos no total.
if (pos <= 300) {
//A1. valeu é o valor retornado de cada Slider
//yvalue = L/2f + (A1. value*Mathf.Sin(Mathf.PI*i/L)+A2.
//value*Mathf.Sin(3f*Mathf.PI*i/L)+A3. value*Mathf.Sin(5f*
//Mathf.PI*i/L)+A4. value*Mathf.Sin(7f*Mathf.PI*i/L));
yvalue = (A1. value*Mathf.Sin(Mathf.PI*i)+A2. value*Mathf.
Sin(2*Mathf.PI*i)+A3. value*Mathf.Sin(Mathf.PI*i)+A4.
value*Mathf.Sin(2*Mathf.PI*i));
//Positions é um vetor de Vector3, onde cada um de seu
//valor consta um Vector3 com posições de i e yvalue (y=f
//(x))
positions [pos] = new Vector3 (i , yvalue + 2 , 0.0f);
//Incremento de i e posições
i = i + 0.05f;
pos++;
//Caso pos for igual a 300 ele é reiniciado para que o
//Gráfico seja sempre plotado na Tela
if (pos == 300) {
pos=0;
i = -7;
}
}
//lr.SetPositions seta as posições do vetor de pontos do
//LineRenderer
lr . SetPositions ( positions );
}
}

```

## C.2 Game Controller

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;
using UnityEngine.SceneManagement;

public class GameController13 : MonoBehaviour {

    //Declaração dos Sliders e botões
    public Slider A1; //A1,A2... Sliders do controle das
                      senoides
    public Slider A2;
    public Slider A3;
    public Slider A4;

    //Declaração dos Textos
    public Text AMP1; //Textos das Senoides
    public Text AMP2;
    public Text AMP3;
    public Text AMP4;
    public Text Timer; //Texto do TIMER
    public Text TimerRegister; //Registro de tempo de
                           conclusão da fase

    //Declaração de variáveis importantes para o Algorítimo
    float Result; //Sinal Resultante
    enum StateGame { Inicio = 1, Jogando, Acertou, Errou,
                    DiagramadeBlocos, Verificando};
    private float StartTime;
    private float StartVerifingTime;
    private float TimerResul;

    //Declaraçāode GameObjects
    public GameObject WIN; //Tela de WIN
    public GameObject LOOSE; //Tela de Loose
    public GameObject WINCANVAS; //Tela de WIN(Canvas)
    public GameObject LOOSECANVAS; //Tela de Loose(Canvas)
```

```
public GameObject Controle; // Sliders de Controle para
    serem desabilitados
public GameObject ResultadoEsperado;
public GameObject SinalControlado;
public GameObject Inicio;
public GameObject InicioCANVAS;
public GameObject DiagramadeBlocosSprite;
public GameObject DiagramadeBlocosCanvas;

//BOTÕES
public Button Iniciar;
public Button RetryWIN;
public Button RetryLOSE;
public Button ProximoLevel;
public Button QUIT;
public Button Diagrama;
public Button CloseDiagrama;
public Button Verificar;
public Button Quit; //QUIT DO HUB
public Button Menu; //VOLTA AO MENU PRINCIPAL

// Use this for initialization
StateGame GameState;
void Start () {
//AchandoGameObjects
A1 = GameObject.Find("Sinal1").GetComponent<Slider>();
A2 = GameObject.Find("Sinal2").GetComponent<Slider>();
A3 = GameObject.Find("Sinal3").GetComponent<Slider>();
A4 = GameObject.Find("Sinal4").GetComponent<Slider>();

AMP1 = GameObject.Find("Sin1").GetComponent<Text>();
AMP2 = GameObject.Find("Sin2").GetComponent<Text>();
AMP3 = GameObject.Find("Sin3").GetComponent<Text>();
AMP4 = GameObject.Find("Sin4").GetComponent<Text>();
Timer = GameObject.Find("Timer").GetComponent<Text>();
TimerRegister = GameObject.Find("TempoResultado").
    GetComponent<Text>();
WIN = GameObject.Find("GameMenuWin");
```

```
LOOSE = GameObject.Find("GameMenuLoose");
WINCANVAS = GameObject.Find("WINCANVAS");
LOOSECANVAS = GameObject.Find("LOOSECANVAS");
Controle = GameObject.Find("HUB");
ResultadoEsperado = GameObject.Find("ResultadoEsperado");
SinalControlado = GameObject.Find("ControleDeSinais");
Inicio = GameObject.Find("StageInicio");
InicioCANVAS = GameObject.Find("INICIOCANVAS");
DiagramadeBlocosSprite = GameObject.Find("DiagramaSprite")
;
DiagramadeBlocosCanvas = GameObject.Find("DiagramaCanvas")
;

Iniciar = GameObject.Find("Iniciar").GetComponent<Button>();
RetryWIN = GameObject.Find("/WINCANVAS/Retry").GetComponent<Button>();
RetryLOOSE = GameObject.Find("/LOOSECANVAS/Retry").GetComponent<Button>();
ProximoLevel = GameObject.Find("Próximo Level").GetComponent<Button>();
QUIT = GameObject.Find("qUIT").GetComponent<Button>();
Diagrama = GameObject.Find("Diagrama").GetComponent<Button>();
CloseDiagrama = GameObject.Find("Fechar").GetComponent<Button>();
Verificar = GameObject.Find("Verificar").GetComponent<Button>();
Menu = GameObject.Find("MenuPrincipal").GetComponent<Button>();
Quit = GameObject.Find("Quit").GetComponent<Button>();

//VARIAVEIS INICIAIS
StartTime = Time.time;
//ESTADOS INICIAIS
GameState = StateGame.Inicio;
WIN.gameObject.SetActive (false);
WINCANVAS.gameObject.SetActive (false);
```

```
Controle.gameObject.SetActive ( false );
LOSE.gameObject.SetActive ( false );
LOOSECANVAS.gameObject.SetActive ( false );
ResultadoEsperado.gameObject.SetActive ( false );
Inicio.gameObject.SetActive ( false );
InicioCANVAS.gameObject.SetActive ( false );
SinalControlado.gameObject.SetActive ( false );
DiagramadeBlocosSprite.gameObject.SetActive ( false );
DiagramadeBlocosCanvas.gameObject.SetActive ( false );

//SCRIPT DE BOTÕES
Button btn = Iniciar.GetComponent<Button>();
btn.onClick.AddListener ( AoClicarIniciar );

Button btn1 = RetryWIN.GetComponent<Button>();
btn1.onClick.AddListener ( AoClicarRetryWin );

Button btn2 = RetryLOSE.GetComponent<Button>();
btn2.onClick.AddListener ( AoClicarRetryLoose );

Button btn3 = ProximoLevel.GetComponent<Button>();
btn3.onClick.AddListener ( AoClicarProximoLevel );

Button btn4 = QUIT.GetComponent<Button>();
btn4.onClick.AddListener ( AoClicarSair );

Button btn5 = Diagrama.GetComponent<Button>();
btn5.onClick.AddListener ( AoClicarDiagramaDeBlocos );

Button btn6 = CloseDiagrama.GetComponent<Button>();
btn6.onClick.AddListener ( AoClicarFecharDiagramaDeBlocos );

Button btn7 = Verificar.GetComponent<Button>();
btn7.onClick.AddListener ( AoClicarVerificar );

Button btn8 = Menu.GetComponent<Button>();
btn8.onClick.AddListener ( AoClicarMainMenu );
```

```
Button btn9 = Quit.GetComponent<Button>();
btn9.onClick.AddListener(AoClicarQuitEnquantoJoga);

}

void AoClicarMainMenu(){
SceneManager.LoadScene ("Menu", LoadSceneMode.Single);
}

void AoClicarQuitEnquantoJoga(){
Application.Quit();
}

void AoClicarIniciar(){
GameState = StateGame.Jogando;
StartTime = Time.time; // Inicio do Cronometro
}

void AoClicarRetryWin(){
GameState = StateGame.Inicio;
}

void AoClicarRetryLoose(){
GameState = StateGame.Inicio;
}

void AoClicarProximoLevel(){
SceneManager.LoadScene ("14", LoadSceneMode.Single);
}

void AoClicarSair(){
Application.Quit();
}
```

```
void AoClicarDiagramaDeBlocos (){  
  
    DiagramadeBlocosSprite .gameObject .SetActive ( true );  
    DiagramadeBlocosCanvas .gameObject .SetActive ( true );  
  
}  
  
void AoClicarFecharDiagramaDeBlocos (){  
    DiagramadeBlocosSprite .gameObject .SetActive ( false );  
    DiagramadeBlocosCanvas .gameObject .SetActive ( false );  
  
}  
  
void AoClicarVerificar (){  
    GameState = StateGame .Verificando ;  
}  
  
// Update is called once per frame  
void Update () {  
  
    //TIMER  
    float TIMER = 60 - ( Time .time - StartTime ); // Inicia o  
    // Timer , X - Time .time ( X é o valor de tempo a ser  
    // decrescido ).  
    Timer .text = "TEMPO:" + TIMER .ToString (" f0 " ); // Atualizao  
    // Texto do Timer  
  
    switch ( GameState )  
    {  
        case StateGame .Inicio :  
        {  
            WIN .gameObject .SetActive ( false );  
            WINCANVAS .gameObject .SetActive ( false );  
            Controle .gameObject .SetActive ( false );  
            LOOSE .gameObject .SetActive ( false );  
            LOOSECANVAS .gameObject .SetActive ( false );  
            ResultadoEsperado .gameObject .SetActive ( false );  
        }  
    }  
}
```

```
Inicio.gameObject.SetActive (true);
InicioCANVAS.gameObject.SetActive (true);
SinalControlado.gameObject.SetActive (false);
//Estado Inicial dos Sliders
A1.value = 0;
A2.value = 0;
A3.value = 0;
A4.value = 0;

break;
}

case StateGame.Jogando:
{

WIN.gameObject.SetActive (false);
WINCANVAS.gameObject.SetActive (false);
Controle.gameObject.SetActive (true);
LOSE.gameObject.SetActive (false);
LOSECANVAS.gameObject.SetActive (false);
ResultadoEsperado.gameObject.SetActive (true);
Inicio.gameObject.SetActive (false);
InicioCANVAS.gameObject.SetActive (false);
SinalControlado.gameObject.SetActive (true);

if (TIMER <= 0.001f) {
GameState = StateGame.Errou;
}

break;
}

case StateGame.Acertou:
{
WIN.gameObject.SetActive (true);
WINCANVAS.gameObject.SetActive (true);
Controle.gameObject.SetActive (false);
LOSE.gameObject.SetActive (false);
LOSECANVAS.gameObject.SetActive (false);
```

```
ResultadoEsperado.gameObject.SetActive (true);
Inicio.gameObject.SetActive (false);
InicioCANVAS.gameObject.SetActive (false);
SinalControlado.gameObject.SetActive (true);

TimerRegister.text = "Tempo: " + TimerResul.ToString ("f2
") + " s";
break;
}
case StateGame.Errou:
{
WIN.gameObject.SetActive (false);
WINCANVAS.gameObject.SetActive (false);
Controle.gameObject.SetActive (false);
LOSE.gameObject.SetActive (true);
LOOSECANVAS.gameObject.SetActive (true);
ResultadoEsperado.gameObject.SetActive (true);
Inicio.gameObject.SetActive (false);
InicioCANVAS.gameObject.SetActive (false);
SinalControlado.gameObject.SetActive (true);
break;
}
case StateGame.Verificando:
{
// Resultado
Result = A2.value + A4.value; //O Sinal resultante é a
soma de todos os outros, como estão com a mesma
frequência, é a soma das amplitudes.
if ((Result <= 2.1f && Result >= 1.9f) ) {
GameState = StateGame.Acertou;
TimerResul = 60f - TIMER;
} else {
GameState = StateGame.Jogando;
}
break;
}
}
```

```
//TEXT ( Atualiza o texto dos sinais com a variação dos A1,
A2 . . . )
AMP1.text = "Sinal1 =" + A1.value.ToString("f2") + "*Sin(
PI)";
AMP2.text = "Sinal2 =" + A2.value.ToString("f2") + "*Sin(
2*PI)";
AMP3.text = "Sinal3 =" + A3.value.ToString("f2") + "*Sin(
PI)";
AMP4.text = "Sinal4 =" + A4.value.ToString("f2") + "*Sin(
2*PI)";
}
}
```

### C.3 Plot de Sinal Esperado

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;

public class Graph213 : MonoBehaviour {
    private LineRenderer lr;

    float i = -7;
    float angle;
    int pos = 0;
    Vector3[] positions = new Vector3[300];

    void Start () {
        lr = GetComponent<LineRenderer>();

        for (pos = 0 ; pos < 300; pos++)
        {
            positions [pos] = new Vector3(i , 2* Mathf.Sin (i *2* Mathf.PI)
                + 2, 0.0f);
            i = i + 0.05f;
        }
    }
}
```

```
lr . SetPositions ( positions ) ;  
  
}  
  
// Update is called once per frame  
void Update () {  
  
}  
}
```

## Apêndice D – Script do aplicativo de áudio completo

```

using UnityEngine;
using UnityEngine.UI;
using System.Collections;
using UnityEngine.SceneManagement;

public class Graph3Effects : MonoBehaviour {
    int pos = -12; //Pos é o indice do vetor de pontos
                  a serem traçadas as retas
    float yvalue; // Valor em y da função y = f(x)
    Vector3[] positions = new Vector3[512]; //Vetor de
                  posições
    private LineRenderer lr; //LineRenderer lr
    enum StateAudio {Freq = 1, Tempo};
    int i = 0;
    //Botões
    public Button Temp;
    public Button Frquencia;
    public Button Play;
    public Button Stop;
    public Slider Volume;
    public Dropdown ListadeMusicas;
    public Button MainMenu;

    //CHORUS
    public Slider DepthSlider;
    public Slider RateSlider;
    public Slider DelaySlider;
    public Slider DryMix;
}

```

```
public Toggle EnableChorus;

//Tremolo
public Slider FTremolo;
public Toggle EnableTremolo;

//Distortion
public Slider LevelDistSlider;
public Toggle EnableDistortion;

//FPB
public Slider FPB;
public Toggle EnableFPB;

//Distortion
public Slider FPA;
public Toggle EnableFPA;

//AudiosClips
public AudioClip [] Clips;
StateAudio Estado;

//TESTES

 AudioSource audio;
int indiceAudio = 0;
float [] samples ;
float [] samples2;
float [] samples3;

void Start () {
Estado = StateAudio.Freq;
lr = GetComponent<LineRenderer>(); //lr recebe a
componente LineRenderer do GameObject que
contém o Script
audio = GetComponent<
```

```
Button btn = Temp.GetComponent<Button>();  
btn.onClick.AddListener(AoClicarTempo);  
  
Button btn1 = Frquencia.GetComponent<Button>();  
btn1.onClick.AddListener(AoClicarFrequencia);  
  
Button btn2 = Play.GetComponent<Button>();  
btn2.onClick.AddListener(AoClicarPlay);  
  
Button btn3 = Stop.GetComponent<Button>();  
btn3.onClick.AddListener(AoClicarStop);  
  
Button btn4 = MainMenu.GetComponent<Button>();  
btn4.onClick.AddListener(AoClicarMainMenu);  
//OnValueChange Dropdown  
ListadeMusicas.onValueChanged.AddListener(delegate  
{  
    selectvalue(ListadeMusicas);  
});  
  
///INICIAL AUDIOS  
samples = new float [audio.clip.samples * audio.  
    clip.channels];  
samples2 = new float [audio.clip.samples * audio.  
    clip.channels];  
samples3 = new float [audio.clip.samples * audio.  
    clip.channels];  
  
audio.clip.GetData(samples, 0); //Armazena os  
    valores das amostras em Samples, com OffSet = 0  
}  
private void selectvalue(Dropdown gdropdown)  
{  
  
    audio.clip = Clips[ListadeMusicas.value];  
    samples = new float [audio.clip.samples * audio.  
        clip.channels];
```

```
samples2 = new float[ audio . clip . samples * audio .
    clip . channels ];
samples3 = new float[ audio . clip . samples * audio .
    clip . channels ];
audio . clip . GetData( samples , 0 ); //Armazena os
valores das amostras em Samples , com OffSet = 0

print ( "Oi" );
}

void AoClicarTempo(){
Estado = StateAudio . Tempo;
print(" Clicou Tempo" );
}

void AoClicarFrequencia(){
Estado = StateAudio . Freq;
print(" Clicou Freq" );
}

void AoClicarPlay(){
audio . Play();
print(" Clicou Play" );
}

void AoClicarStop(){
audio . Pause();
print(" Clicou Pause" );
}

void AoClicarMainMenu(){
SceneManager . LoadScene ( "Menu" , LoadSceneMode .
    Single);
print(" Clicou MainMenu" );
}

void Update () {
```

```
float [] spectrum = new float [512]; //Declaração do
                                         Vetor para coletar os valores de Espectro
float [] spectrum2 = new float [512];

//print (ListadeMusicas . value );
switch(Estado){
case StateAudio . Freq:
{
    PlotarAudioNaFrequencia (spectrum2 , pos , yvalue ,
    positions ,10 ,0 ,0.05 f );
    break;
}
case StateAudio . Tempo:
{
    PlotarAudioNoTempo (spectrum , pos , yvalue , positions
    ,0 ,0 );
    break;
}
}
/*
if (EnableChorus . isOn == true ){
GetComponent<AudioChorusFilter>(). enabled = true ;
print ("EnableChorus" );
} else{
GetComponent<AudioChorusFilter>(). enabled = false ;
print ("NotEnable" );
}
*/
if ( EnableDistortion . isOn == true ){
GetComponent<AudioDistortionFilter>(). enabled =
    true ;
//      print ("EnableChorus" );
} else{
GetComponent<AudioDistortionFilter>(). enabled =
    false ;
//      print ("NotEnable" );
}
```

```
if(EnableFPB.isOn == true){
    GetComponent<AudioLowPassFilter>().enabled = true;
    //      print("EnableFPB");
} else{
    GetComponent<AudioLowPassFilter>().enabled = false
    ;
    //      print("NotEnable");
}

if(EnableFPA.isOn == true){
    GetComponent<AudioHighPassFilter>().enabled = true
    ;
    print("EnableFPA");
} else{
    GetComponent<AudioHighPassFilter>().enabled =
        false;
    //      print("NotEnable");
}

if(EnableTremolo.isOn == true){
    TremoloEffect(samples2,samples,FTremolo.value,
        indiceAudio);
} else{
    CleanAudio(samples2,samples,indiceAudio);
}
if(EnableChorus.isOn == true){
    ChorusEffect(samples3,samples2,DryMix.value,(int)
        DelaySlider.value,DepthSlider.value,RateSlider.
        value,indiceAudio);
} else{
    CleanAudio(samples3,samples2,indiceAudio);
}
audio.clip.SetData(samples3, 0);

audio.volume = Volume.value;
//GetComponent<AudioChorusFilter>().delay =
DelaySlider.value;
```

```
//GetComponent<AudioChorusFilter>().depth =
    DepthSlider.value;
//GetComponent<AudioChorusFilter>().rate =
    RateSlider.value;
GetComponent<AudioDistortionFilter>().
    distortionLevel = LevelDistSlider.value;
GetComponent<AudioLowPassFilter>().cutoffFrequency
    = (float)FPB.value; //Filtro Passa baixa
    recebe como frequência de corte o Slider F1
GetComponent<AudioHighPassFilter>().
    cutoffFrequency = (float)FPA.value; //Idem para
    o FPA

}

void PlotarAudioNoTempo( float [] spectrum , int pos ,
    float yvalue , Vector3 [] positions , int AjusteX
    , int AjusteY)
{
//AudioListener.GetSpectrumData( spectrum , 0,
    FFTWindow.Rectangular ); //A função Coleta os
    valores de magnetude do Espectro e aloca no
    Vetor Spectrum
AudioListener.GetOutputData( spectrum , 0);

//////Printa o gráfico com os valores do Vetor
    Espectro
for( pos = 0; pos < spectrum.Length; pos++ )
{
    yvalue = spectrum[ pos ];
    positions[ pos ] = new Vector3( pos*0.1f + AjusteX ,
        7*yvalue + AjusteY , 0 );
    lr.SetPositions( positions );
}
}

void PlotarAudioNaFrequencia( float [] spectrum , int
    pos , float yvalue , Vector3 [] positions , int
    AjusteX , int AjusteY , float RangeX)
```

```
{  
    AudioListener.GetSpectrumData( spectrum , 0 ,  
        FFTWindow.Rectangular ); //A função Coleta os  
        valores de magnitude do Espectro e aloca no  
        Vetor Spectrum  
  
    //////Printa o gráfico com os valores do Vetor  
    //Espectro  
    for( pos = 1; pos < spectrum.Length ; pos++ )  
    {  
        yvalue = spectrum[ pos ];  
        positions[ pos ] = new Vector3( pos*RangeX + AjusteX ,  
            7*yvalue + AjusteY ,0 );  
        //      positions [pos] = new Vector3(Mathf.Log10(  
        //      pos)*RangeX+ AjusteX , 7*yvalue + AjusteY,0 );  
        lr.SetPositions ( positions );  
    }  
}  
  
///////////////Tremolo  
  
void TremoloEffect( float [] SamplesEffect , float []  
    SamplesAudio , float LFOFrequency , int Indice)  
{  
  
    while ( Indice < SamplesAudio.Length ) {  
        SamplesEffect [ Indice ] = SamplesAudio [ Indice ] *  
            Mathf.Cos(2 f*Mathf.PI*Indice*LFOFrequency/audio  
            . clip . frequency ); //Tremolo  
        ++Indice ;  
    }  
    //Indice = 0;  
    //    audio . clip . SetData( SamplesEffect , 0 );  
}  
  
///////////////CHORUS
```

```

void ChorusEffect( float [ ] SamplesEffect , float [
    SamplesAudio , float DryMixGain , int DELAY, float
    DEPTH, float RATE, int Indice )
{
    float DryMix1;
    float DryMix2;
    float DryMix3;

    while ( Indice < SamplesAudio . Length - (DELAY+DEPTH)
        ) {
        //      DryMix1 = ( float ) DELAY + DEPTH*Mathf.Sin
        //          (2f*Mathf.PI*Indice*RATE/44100 );
        //      DryMix2 = ( float ) DELAY + DEPTH*Mathf.Sin
        //          (2f*Mathf.PI*Indice*RATE/44100 - (Mathf.PI/2f)
        //          );
        //      DryMix3 = ( float ) DELAY + DEPTH*Mathf.Sin
        //          (2f*Mathf.PI*Indice*RATE/44100 + (Mathf.PI/2f)
        //          );
        DryMix1 = ( float ) DELAY + DEPTH*Mathf.Sin (2 f *
            Mathf.PI*Indice*RATE/audio . clip . frequency );
        DryMix2 = ( float ) DELAY + DEPTH*Mathf.Sin (2 f *
            Mathf.PI*Indice*RATE/audio . clip . frequency - (
            Mathf.PI/2 f ));
        DryMix3 = ( float ) DELAY + DEPTH*Mathf.Sin (2 f *
            Mathf.PI*Indice*RATE/audio . clip . frequency + (
            Mathf.PI/2 f ));

        SamplesEffect [ Indice ] = DryMixGain*SamplesAudio [
            Indice ] + DryMixGain*SamplesAudio [ Indice + ( int
            ) DryMix1 ] + DryMixGain*SamplesAudio [ Indice + (
            int ) DryMix2 ] + DryMixGain*SamplesAudio [ Indice
            + ( int ) DryMix3 ]; //DELAY
        //      SamplesEffect [ Indice ] = SamplesAudio [
        //          Indice ] + SamplesAudio [ Indice + ( int ) DryMix1 ]
        //          + SamplesAudio [ Indice + ( int ) DryMix2 ] +
        //          SamplesAudio [ Indice + ( int ) DryMix3 ]; //DELAY

        ++Indice ;
}

```

```
    }

    print( Indice );

    //Indice = 0;
    audio . clip . SetData( SamplesEffect , 0);
}

void CleanAudio( float [ ] SamplesEffect , float [ ]
    SamplesAudio , int Indice)
{

    while ( Indice < SamplesAudio . Length ) {
        SamplesEffect [ Indice ] = SamplesAudio [ Indice ];
        ++Indice;
    }
    //Indice = 0;
    //    audio . clip . SetData( SamplesEffect , 0);
}

}
```

## **Apêndice E – Artigo apresentado no COBENGE 2017**

Nas próximas páginas está disponível o artigo “DESENVOLVIMENTO DE SOFTWARE PARA APRIMORAMENTO DO ENSINO EM ENGENHARIA ELÉTRICA”, que foi elaborado pelo Orientador Ernesto F. Ferreyra Ramírez, e orientando Matheus Elero. O trabalho foi aprovado no XLV Congresso Brasileiro de Educação em Engenharia (COBENGE 2017), e apresentado no evento na sessão poster. O trabalho foi bem recebido pelos avaliadores, e congressistas que assistiram à apresentação.



## DESENVOLVIMENTO DE SOFTWARE PARA APRIMORAMENTO DO ENSINO EM ENGENHARIA ELÉTRICA

**Matheus R. Elero** – matheuselero1@gmail.com

**Ernesto F. Ferreyra Ramírez** – ferreyra@uel.br

Universidade Estadual de Londrina, Departamento de Engenharia Elétrica

Rod. PR-445 km 380 – Caixa Postal 10.011

86.057-970 – Londrina – PR

**Resumo:** Alguns conceitos teóricos de áreas da Engenharia Elétrica podem ser bastante abstratos e de difícil compreensão para alunos dos primeiros anos do curso, que é o caso por exemplo das séries de Fourier. Com o objetivo de contribuir para a solução desse problema, este trabalho mostra o desenvolvimento de um jogo de caráter educativo, desenvolvido no curso de Engenharia Elétrica da Universidade Estadual de Londrina, que apresenta uma interação lúdica e visual acerca do tema. O jogo foi construído utilizando o software Unity 3D e programação em C#, que é uma ferramenta gratuita para projetos sem fins lucrativos, e destinada para desenvolvimento de jogos digitais.

**Palavras-chave:** Séries de Fourier, Resposta em Frequência, Unity, MatLab, Jogos Educativos.

### 1. INTRODUÇÃO

O mercado de jogos digitais vem crescendo cada vez mais, com um rendimento mundial aproximadamente igual a 89,4 bilhões (NEWZOO, 2016). Dessa forma, os jogos digitais se mostram cada vez mais presentes no dia a dia, atraindo um público de todas as idades. Por toda essa popularidade, e suas características que permitem aos usuários resolverem problemas, se divertir e interagir, a utilização de jogos na educação pode ser muito benéfica.

Existem alguns componentes básicos que são comumente encontrados em um jogo digital, como papel ou personagem, regras, metas e objetivos, quebra-cabeças, problemas ou desafios, história, interações do jogador, estratégias e feedbacks (SAVI et al., 2008). Cada um deles possui sua importância de acordo com os objetivos do projeto. Por exemplo, há jogos que possuem um desafio elevado, porém sem nenhuma história. Em compensação, existem outros que possuem um nível de dificuldade praticamente igual a zero, mas a história contada pelo jogo é mais importante.

Sabendo disso, os jogos proporcionam alguns benefícios educacionais, como o desenvolvimento de habilidades cognitivas, aprendizado por descoberta, experiência de novas identidades, socialização, coordenação motora, facilitador de aprendizado e motivacional (SAVI et al., 2008).

Organização



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA



Educação e Tecnologia

Promoção



Associação Brasileira de Educação em Engenharia



O tópico Séries de Fourier é lecionado no curso de Engenharia Elétrica da Universidade Estadual de Londrina (UEL) na disciplina de “Circuitos Elétricos 1”, oferecida no segundo ano do curso. Depois, este conceito é aprofundado em outras disciplinas ao longo da graduação. É indiscutível a relevância deste assunto para a formação do engenheiro eletricista, já que é aplicado nas áreas de Telecomunicações, Processamento de Sinais, Circuitos Eletrônicos, Controle e Automação, entre outros.

Assim, este trabalho relata a confecção de um jogo digital para usufruir das suas vantagens na educação. O objetivo é propiciar uma interface simples e visual, para que o aluno resolva, de forma lúdica, problemas relacionados a Séries de Fourier, assunto relevante para a formação do engenheiro eletricista.

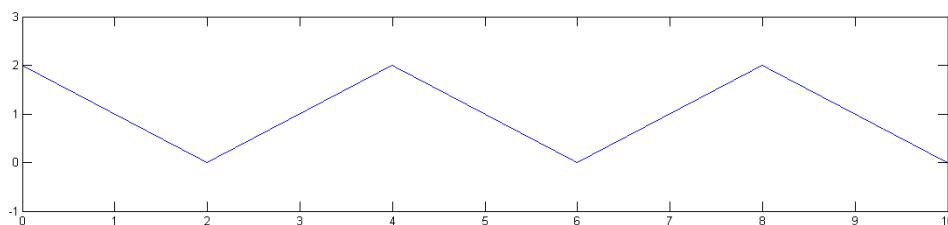
## 2. SÉRIES DE FOURIER

Séries de Fourier são casos particulares das séries de Taylor. Caso uma série formada pelo somatório de senos e cossenos seja convergente, é denominada de Série Trigonométrica de Fourier (LOTUFO, 2014). A Equação (1) mostra a forma de uma Série Trigonométrica de Fourier aplicada a uma função  $f(x)$ .

$$f(x) = \frac{a_0}{2} + \sum_{m=1}^{\infty} \left( a_m * \cos\left(\frac{m * \pi * x}{L}\right) + b_m * \sin\left(\frac{m * \pi * x}{L}\right) \right) \quad (1)$$

Na Engenharia Elétrica, geralmente há a necessidade de processar sinais elétricos periódicos, sendo a aplicação de Séries de Fourier primordial para estes trabalhos. Basicamente, este importante conceito é utilizado para sintetizar e facilitar a resposta em frequência de funções periódicas, tais como ondas quadradas, triangulares e dentes de serra. A Figura 1 mostra um exemplo de função periódica triangular.

Figura 1 - Sinal periódico triangular gerado no MatLab.



Para que um função  $f(x)$  possa ser representada por uma série de potências, ela deve ser infinitamente derivável, e a fórmula de Taylor deve possuir resto tendendo para zero. Assim para uma série trigonométrica, também deve-se analisar sua convergência (LOTUFO, 2014).

Dessa forma, para representar um sinal periódico ( $f(x) = f(x+T)$ ) através de uma Série de Fourier, deve-se calcular os coeficientes  $a_m$  e  $b_m$  vistos na Equação (1), os quais são calculados utilizando as Equações (2) e (3).

Organização



**UDESC**  
 UNIVERSIDADE  
 DO ESTADO DE  
 SANTA CATARINA



Educação e Tecnologia

Promoção



Associação Brasileira de Educação em Engenharia

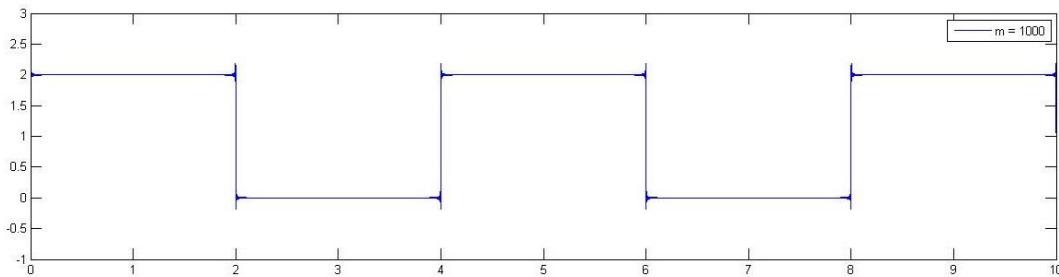


$$a_m = \frac{1}{L} * \int_{-L}^L f(x) * \cos\left(\frac{m * \pi * x}{L}\right) dx, \quad m = 0, 1, 2, 3, \dots \quad (2)$$

$$b_m = \frac{1}{L} * \int_{-L}^L f(x) * \sin\left(\frac{m * \pi * x}{L}\right) dx, \quad m = 1, 2, 3, \dots \quad (3)$$

Como exemplo, será descrito o processo para encontrar a Série de Fourier correspondente a uma quadrada de período 4 e amplitude 2, mostrada na Figura 2.

Figura 2 - Sinal periódico quadrado gerado em MatLab



Primeiramente, deve-se equacionar o sinal periódico em uma  $f(x)$ , ou seja:

$$f(x) = \begin{cases} f(x) = 2, & 0 \leq x \leq 2 \\ f(x) = 0, & 2 \leq x \leq 4 \end{cases} \quad f(x + 4) = f(x) \quad (4)$$

Neste caso, o parâmetro  $L$  (*largura do pulso*) corresponde à metade do período, ou seja  $L = 2$ . Assim, os parâmetros são calculados utilizando as Equações (2) e (3),

$$a_0 = \frac{1}{2} * \int_{-2}^2 f(x) * \cos(0) dx = 2 \quad (5)$$

$$a_m = \frac{1}{2} * \int_{-2}^2 f(x) * \cos\left(\frac{n * \pi * x}{2}\right) dx = \frac{1}{2} * \int_0^2 2 * \cos\left(\frac{n * \pi * x}{2}\right) dx = 0 \quad (6)$$

$$b_m = \frac{1}{2} * \int_{-2}^2 f(x) * \sin\left(\frac{m * \pi * x}{2}\right) dx = \frac{2}{n * \pi} * [1 - \cos(m * \pi)] \quad (7)$$

Organização



**UDESC**  
 UNIVERSIDADE  
 DO ESTADO DE  
 SANTA CATARINA



Educação e Tecnologia

Promoção



Associação Brasileira de Educação em Engenharia



Disto, os parâmetros calculados são,

$$a_0 = 2 \quad (8)$$

$$a_m = 0, \quad m = 1, 2, 3, \dots \quad (9)$$

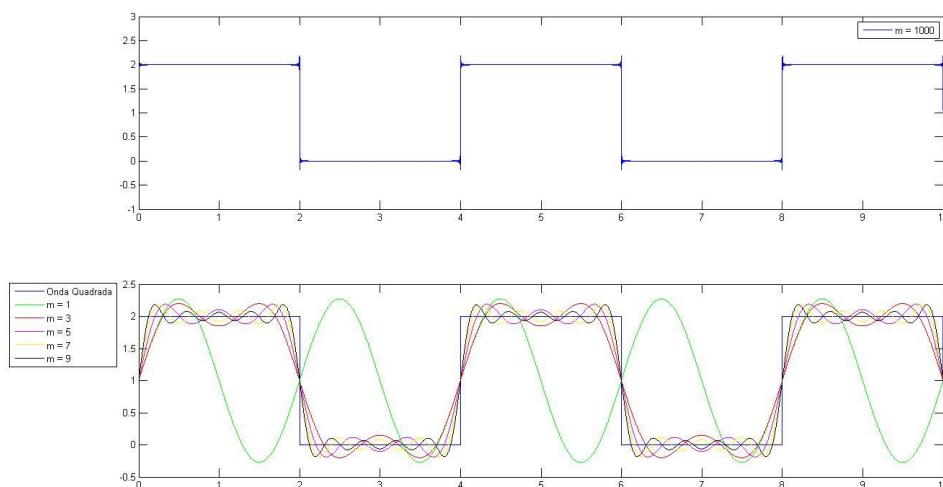
$$b_m = \begin{cases} 0, & m \text{ par} \\ \frac{4}{m * \pi}, & m \text{ ímpar} \end{cases} \quad (10)$$

Portanto, a função  $f(x)$  da Figura 2 e Equação (4) pode ser representada pela série da Equação (11).

$$f(x) = 1 + \frac{4}{\pi} * \left[ \operatorname{Sen}\left(\frac{\pi * x}{2}\right) + \frac{1}{3} * \operatorname{Sen}\left(\frac{3 * \pi * x}{2}\right) + \frac{1}{5} * \operatorname{Sen}\left(\frac{5 * \pi * x}{2}\right) \dots \right] \quad (11)$$

Observe que existe uma lei de formação do sinal periódico de onda quadrada em Série de Fourier, onde este sinal é composto pelo somatório apenas de senoides, com decréscimo de amplitude e acréscimo de frequência apenas em termos ímpares. Ou seja, a soma possui seu índice  $m$  assumindo apenas valores ímpares. Com o software MatLab, foi realizada uma simulação com o somatório de senoides variando o número máximo de  $m$  para comprovar a representação calculada. O resultado é mostrado na Figura 3.

Figura 3 - Simulação utilizando MatLab do somatório de senoides variando o número máximo de  $m$ .



Organização



**UDESC**  
 UNIVERSIDADE  
 DO ESTADO DE  
 SANTA CATARINA



Promoção





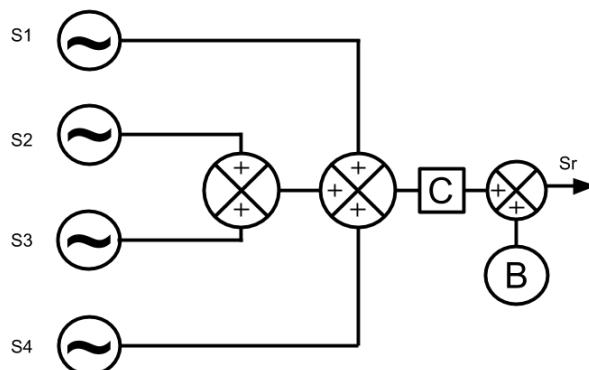
### 3. O JOGO

A matemática muitas vezes pode ser abstrata para todos os alunos por diversos motivos. Um deles é a carência de algum recurso visual, que possibilita transformar toda fórmula e equação em algo mais palpável. Foi partindo deste ponto que os autores do trabalho elaboraram as regras, desafios e conceitos do jogo.

De acordo com (SCHELL, 2008, p.41, tradução nossa) “um jogo pode ser definido como uma atividade de resolução de problemas, ligado com uma atitude lúdica”. Partindo desta definição, pode-se construir o conceito do jogo, já que a formação de qualquer engenheiro está diretamente ligada com a capacidade de resolver problemas. Com isso, o jogo apresentado neste trabalho se propõe a deixar o aprendizado de forma lúdica.

A premissa do jogo apresentado neste trabalho é bastante simples, o jogador tem à sua disposição 4 sinais (Senos ou Cossenos), os quais serão somados, como mostra o diagrama de blocos da Figura 4. A partir destes recursos disponibilizados, o jogador pode ajustar a amplitude de cada sinal, e tem como objetivo sintetizar um sinal periódico proposto em um determinado período de tempo, seguindo o conceito de Séries de Fourier.

Figura 4 - Diagrama Genérico de Blocos do jogo



Os sinais a serem controlados podem ser representados conforme a Equação (12), onde a amplitude pode ser ajustada.

$$S_i = A_i * \text{Sin}(2 * \pi * f), \quad i = 1,2,3 \text{ e } 4 \quad (12)$$

Na Equação (13) é mostrada a representação matemática do diagrama de blocos da Figura 4.

$$S_r = [A_1 * \text{Sin}(2 * \pi * f_1) + A_2 * \text{Sin}(2 * \pi * f_2) + A_3 * \text{Sin}(2 * \pi * f_3) + A_4 * \text{Sin}(2 * \pi * f_4)] * C + B \quad (13)$$

Fica evidente que o sinal resultante não é exato em comparação com o requerido. Entretanto, com 4 sinais, ou  $m = 4$ , o resultado é aproximado, e é possível entender o processo como um todo. O jogo plota o gráfico da Equação (13) em tempo real conforme as alterações das amplitudes. Basta o jogador apertar um botão para verificar o resultado.

Organização



**UDESC**  
 UNIVERSIDADE  
 DO ESTADO DE  
 SANTA CATARINA



Educação e Tecnologia

Promoção

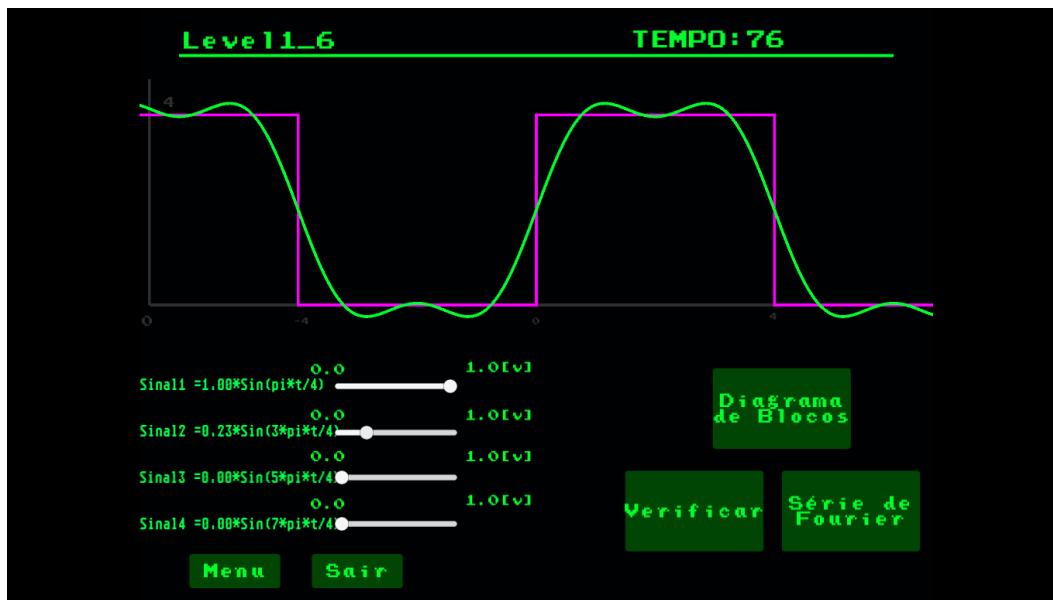


Associação Brasileira de Educação em Engenharia



O jogo propõe ao aluno o desafio de transformar os sinais iniciais em um outro requerido, e também permite que ele analise padrões na Série de Fourier, como o obtido pela Equação (11). Além disso, possibilita a percepção visual para entender o processo de representação de um sinal periódico. A Figura 5 mostra uma imagem do jogo já pronto.

Figura 5 - Captura de imagem de uma fase do jogo



#### 4. UNITY

Unity é um software voltado para desenvolvimento de jogos, criado pela empresa de mesmo nome. Também conhecido como Unity 3D, a ferramenta permite criar tantos jogos em 3D quanto em 2D, além de poder desenvolverlos para inúmeras plataformas, como celulares, tablets, computadores, web e videogames (UNITY,2017c).

Essa ferramenta suporta 2 linguagens de programação, C# e JavaScript. O desenvolvimento de algum projeto utilizando a ferramenta se baseia na manipulação de objetos, ou seja, cada cena do jogo é composta por objetos que são dos mais variados tipos, como áudios, imagens, textos, botões, entre outros. Cada objeto é composto por componentes, que dão a cara ao objeto, por exemplo, um objeto de jogo que toca uma música, só consegue executar tal funcionalidade pois nele existe um componente específico para executar esta função. Além disso, também é possível anexar Scripts programados pelo desenvolvedor, para executar funções específicas, manipular objetos e componentes (UNITY,2017a).

Todas as funções disponíveis para programação pertencem ao conjunto de funcionalidades denominado de MonoBehaviour (UNITY,2017a). No site da empresa contém documentações, manuais e tutoriais a respeito de todas as funcionalidades do software.

Organização



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA



Educação e Tecnologia

Promoção



Associação Brasileira de Educação em Engenharia



## 5. DESENVOLVIMENTO

Todo o trabalho foi desenvolvido utilizando a ferramenta Unity, como ambiente principal de desenvolvimento, MatLab para realização de estudos e alguns testes, como já apresentados pelas Figuras 1, 2 e 3, e por fim o Google Desenhos, ferramenta do Google Drive para a parte gráfica do jogo.

Como primeiro passo para desenvolvimento do jogo, foram realizados pesquisas e estudos a respeito do manuseio da Unity. Em seguida, foi elaborado um script com a função de plotar um gráfico em tempo real, e o alterasse conforme o usuário execute alguma ação, como apertar um botão, ajustar uma barra, entre outros.

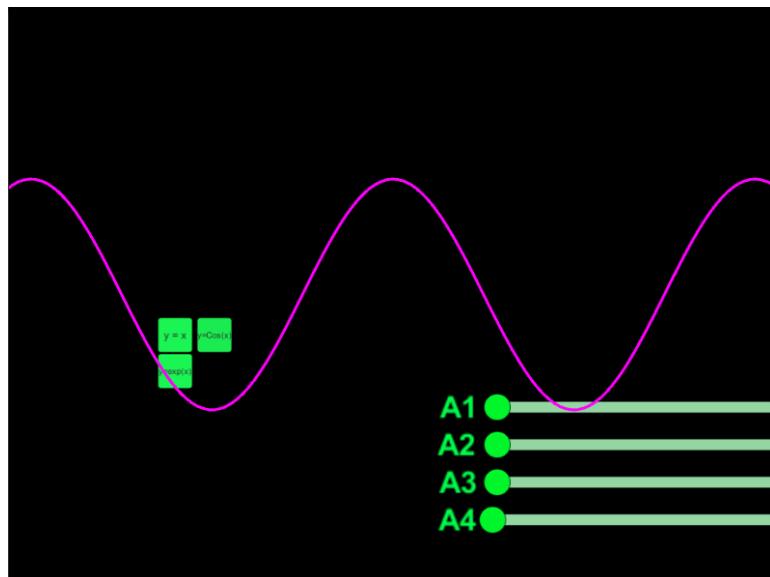
A Unity é um ambiente bastante versátil, possui um espaço em 3D com coordenadas e geometria analítica. Para este projeto foi utilizado apenas um plano 2D, eixos  $x$  e  $y$ , desconsiderando o eixo  $z$ . Assim, foi criado um objeto denominado “Graph”, que contém a componente principal denominada “LineRenderer”, que desenha uma linha entre 2 pontos no plano cartesiano, e que pode ser expandida para um vetor de pontos (UNITY,2017b). Ou seja, é o princípio básico de um gráfico na matemática, são diversos pontos dispostos em  $x$  e  $y$ , que entre cada um deles são traçados uma linha, e quanto maior a quantidade de pontos, mais suave é o gráfico.

O Script trabalha em torno de 300 pontos, com passo de 0.05 em  $x$ , e  $y$  é o resultado em função de  $x$ , basicamente o que mostra a Equação (14).

$$y = f(x) \quad (14)$$

Com a manipulação do componente LineRenderer utilizando o Script, foi possível plotar gráficos com as mais diversas funções. A Figura 6 mostra os resultados obtidos nos primeiros testes.

Figura 6-Gráfico plotado nos primeiros testes.



Organização



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

**UNISOCIESC**  
Educação e Tecnologia

Promoção

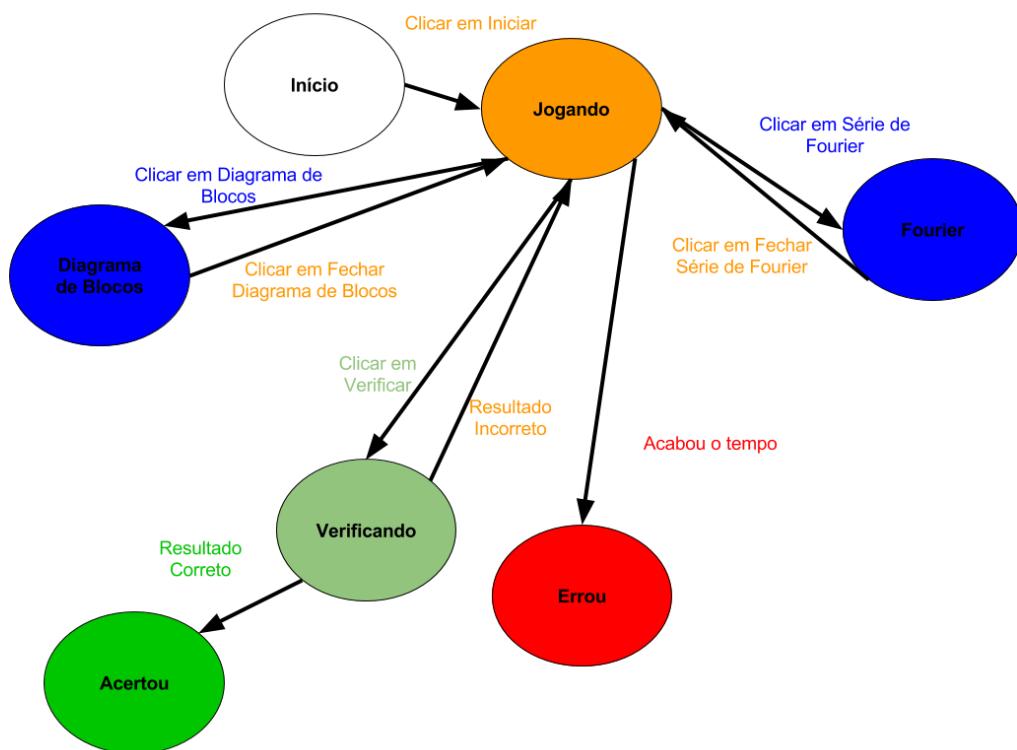
**ABENGE**  
Associação Brasileira de Educação em Engenharia



Com o script para plotar gráficos pronto, foi possível desenvolver o restante do jogo, que foi dividido em 8 cenas, sendo os diferentes níveis de jogo e um menu geral. As fases iniciais são simples, e não possuem muita ligação com as Séries de Fourier, apenas para que o jogador se acostume e aprenda as ações do jogo. A interação consiste no ajuste das amplitudes através de “Sliders”, que é uma barra ajustável, como se fosse uma barra de volume de som. Além disso, o jogador tem a sua disposição um botão que mostra o diagrama de blocos, como o da Figura 4, e também um com o teorema de Fourier, para o auxiliar na resolução do problema. Quando o jogador quiser verificar sua resposta, basta clicar em um botão específico e intuitivo. Todas estas funcionalidades podem ser vistas nas Figuras 5.

Todo este processo é controlado por uma máquina de estados em um Script específico denominado de “GameController”. São 7 estados que contém um conjunto de funções específicas para a cena, são eles, “Início”, “Jogando”, “Acertou”, “Errou”, “Diagrama de Blocos”, “Fourier” e “Verificando”. Cada botão ou determinado evento realiza a transição destes estados. A Figura 7 mostra um diagrama da máquina de estado base para cada nível.

Figura 7 - Máquina de estados de cada nível do jogo.



O estado “Início” apenas apresenta uma interface explicando os objetivos do nível, com algumas equações e instruções. Ao clicar no botão para iniciar, o estado passa para “Jogando”, que inicia um cronômetro regressivo, e libera o menu de jogo, permitindo ao jogador ajustar as amplitudes dos sinais, verificar o resultado, e clicar em ações de exibição. Ao acionar o botão “Diagrama de Blocos”, o estado é mudado para outro de

Organização



**UDESC**  
 UNIVERSIDADE  
 DO ESTADO DE  
 SANTA CATARINA



**UNISOCIESC**  
 Educação e Tecnologia

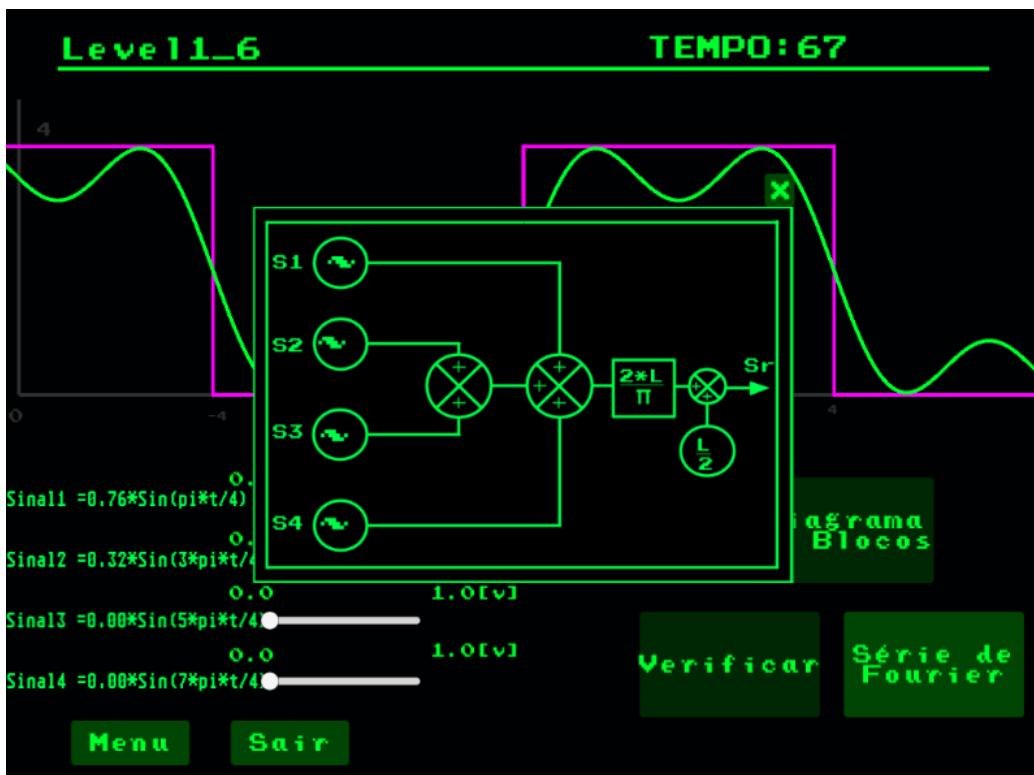
Promoção





mesmo nome, e neste é exibido um desenho do diagrama, que também auxilia no aprendizado do aluno, já que a leitura deste esquemático também é importante. Para fechar a exibição do diagrama, basta clicar em um botão fechar, como mostra a Figura 8. O mesmo ocorre ao clicar no botão denominado “Série de Fourier”, porém com exibição das equações fundamentais. Vale ressaltar que o cronômetro não é interrompido com estas ações de exibição.

Figura 8- Exibição do estado "Diagrama de Blocos"



Em “Verificando” o script compara o resultado ajustado pelo jogador ao esperado, caso for correto ocorre uma transição para o estado “Acertou”, que mostra o tempo obtido pelo jogador ao completar o nível, e opções de tentar novamente ou seguir em frente. Por fim, caso acabe o tempo o estado é transitado para “Errou”, que fornece a opção sair do jogo, ou tentar novamente o nível. Além disso existem as opções de fechar o jogo, e retornar para o menu principal durante o estado “Jogando”.

## 6. CONSIDERAÇÕES FINAIS

Disseminar conhecimentos de engenharia e matemática de forma palpável já foi um desafio muito maior, hoje com a tecnologia disponível está dificuldade diminuiu. E os jogos digitais são um destes recursos, que em vantagem a outros como vídeo, imagem e áudio, este cria uma interatividade, que estimula o aluno/jogador a participar de forma ativa da resolução de um problema.

Organização



**UDESC**  
 UNIVERSIDADE  
 DO ESTADO DE  
 SANTA CATARINA



Educação e Tecnologia

Promoção





Além de promover aos estudantes de Engenharia uma fonte de ensino que complemente seus estudos, com os benefícios já citados, este trabalho também serve de estímulo, para que outros projetos como este sejam desenvolvidos. Já que o curso de Engenharia Elétrica promove a qualquer estudante, a capacidade adequada para que projetos ainda maiores como este sejam desenvolvidos.

O jogo desenvolvido fornece aos estudantes benefícios como entender o processo de geração de sinais periódicos, como onda quadradas e triangulares, ludicidade ao executar um exercício a respeito do tema, visibilidade acerca da aplicação da Equação (1) das Séries de Fourier, resolver um problema em um determinado tempo utilizando como ferramenta a teoria, e por fim pode despertar um interesse maior no aluno com o assunto ensinado.

## REFERÊNCIAS BIBLIOGRÁFICAS

LOTUFO, A. D. P., Séries de Fourier, UNESP, 2014, Disponível em <<http://www.feis.unesp.br/Home/departamentos/engenhariaeletrica/mcap03.pdf>> Acesso em: 13 de Mai.2017

NEWZOO,2016 GLOBAL GAMES MARKET REPORT, Disponível em <[https://cdn2.hubspot.net/hubfs/700740/Reports/Newzoo\\_Free\\_2016\\_Global\\_Games\\_Market\\_Report.pdf](https://cdn2.hubspot.net/hubfs/700740/Reports/Newzoo_Free_2016_Global_Games_Market_Report.pdf)> Acesso em: 11 de Out.2016

SAVI, R.; ULBRICHT, V. R. Jogos Digitais Educacionais: Benefícios e Desafios, CINTED-UFRGS, 2008.

SCHELL, J. The Art of Game Design. 1 ed. Morgan Kaufmann Publishers, 2008. P.26 – 46.

UNITY, Creating and Using Scripts, 2017 Disponível em <<https://docs.unity3d.com/Manual/CreatingAndUsingScripts.html>> Acesso em: 17 de Mai.2017

UNITY, Line Renderer, 2017, Disponível em <<https://docs.unity3d.com/Manual/class-LineRenderer.html>> Acesso em: 17 de Mai.2017

UNITY, O que é?, 2017, Disponível em <<https://unity3d.com/pt/unity>> Acesso em: 17 de Mai.2017

Organização



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA



Educação e Tecnologia

Promoção



Joinville/SC – 26 a 29 de Setembro de 2017  
UDESC/UNISOCIESC  
“Inovação no Ensino/Aprendizagem em  
Engenharia”



**COBENGE 2017**

XLV CONGRESSO BRASILEIRO DE EDUCAÇÃO EM ENGENHARIA

## DEVELOPMENT OF SOFTWARE FOR ENHANCEMENT OF EDUCATION IN ELECTRICAL ENGINEERING

**Abstract:** Some theoretical concepts of Electrical Engineering areas could be a lot abstract, and for hard comprehension to students of the first years of graduation, for example this is the case of Fourier series. In order to contribute to solving this problem, this paper shows the development of an educational game, that was made at the Eletrical Engineering course of State University of Londrina, which presents a playful and visual interactivity about the theme. The game was built using the software Unity 3D and programming in C#, which is a free tool to non-profits projects, and made for digital games developments.

**Key-words:** Fourier Series, Frequency Response, Unity, MatLab, Educational Games.

Organização



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA



Educação e Tecnologia

Promoção



Associação Brasileira de Educação em Engenharia