UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
ESCOLA POLITÉCNICA
DEPARTAMENTO DE ELETRÔNICA E DE COMPUTAÇÃO

Editor Gráfico de Sinais de Áudio Multiplataforma

Autor:

Daniel Levitan

Orientadores:

Prof. Luiz Wagner Pereira Biscainho

Prof. Marcelo Luiz Drumond Lanza

Examinadores:

Prof. Sérgio Lima Netto

Filipe Castello da Costa Beltrão Diniz

DEL
Junho de 2005

Ao mundo.

Agradecimentos

Este é o ápice de um processo, o ponto onde culminaram os anos de estudo do colégio e posteriormente a faculdade. O aprendizado nunca acaba, e é contínuo. Para desenvolvermos qualquer atividade sempre precisamos, dependemos e somos influenciados por pessoas que nos circundam. Este espaço é uma oportunidade de agradecer-lhes.

Antes de mais nada devo agradecer ao Criador, sem o qual nada seria possível. Seguindo o raciocínio devo agradecer aos meus pais, Alberto Levitan e Vera Levitan, por terem sido tão pacientes, terem sempre me estimulado e acima de tudo me dando o suporte psicológico de que qualquer pessoa necessita para um bom desenvolvimento. Igual mérito em minha criação têm duas outras pessoas, minha irmã Patricia e a minha segunda mãe, a pessoa que realmente me criou, Maria do Carmo Pereira, e a qual devo grande parte de minha pessoa.

A família de um modo geral sempre me deu suporte:

Meus avós, José Levitan, Josefina Levitan, Maurício Stolar e Bela Stolar. Meus tios, Jaime Luis Levitan, Marlene Balassiano, Marly Zilberman e Telma Stolar. Meus primos, Déborah Balassiano, Flávio Balassiano, Michelle Nigri Levitan, Marcela Levitan, Eduardo Zilberman, Rafael Zilberman, Natália Klein e Marcos Klein.

Passo agora aos meus amigos, que nos trazem alegria pela caminhada. Agradeço ao mestre em planejamento energético, Ítalo Lopes Borges, pelas inúmeras horas de conversa que tivemos, onde compartilhamos visões do mundo e discutimos problemas atuais.

Agradeço a Roberto Barreto de Moraes por inúmeras horas compartilhando ensaios com violão, e troca de informação entre bandas e músicas; agradeço ainda a Igor Monteiro Moraes por ter me mostrado a cidade do outro lado da ponte. Não posso esquecer Rodrigo Franceski Prestes que sempre foi um pingo da visão comercial de todos os aspectos da vida; agradeço a Norberto Guimarães Lopez que mostrou que é possível sim conciliar o tempo para fazer tudo que quisermos.

Não posso me esquecer de Gabriel Epsztejn e Eduardo Rapoport, por terem sido fiéis companheiros desde os tempos de segundo grau, sempre partilhando visões sobre assuntos de interesse comum; ainda agradeço a Marco Dias Dutra Bicudo pela companhia nas tardes ao sol nas areias de Ipanema. Rafael Pinaud Laufer foi responsável por memoráveis festas em seu condomínio, e ainda tenho que agradecer a muito mais gente que foi igualmente importante para mim nessa caminhada.

Do ponto de vista desse trabalho agradeço a Luis Fernando Lugon por ter cedido sua biblioteca gráfica para tratamento de sinais, de onde comecei a escrever o código de exibição de formas de onda em tela. Agradeço a Rodrigo Meirelles por ter cedido as rotinas de execução de áudio, das quais eu adaptei meu código. Agradeço aos meus orientadores, Luiz Wagner Biscainho e Marcelo Lanza, que sempre estiveram dispostos a me ajudar, aturando minha ansiedade.

A caminhada apenas chega a um ponto em que se define, pois ela é longa e contínua.

Resumo

O Editor Gráfico de Sinais de Áudio Multiplataforma é um editor de arquivos de áudio. As funções básicas de um editor, tais como recortar, colar, dar ganho e realizar a conversão entre mono e estéreo estão incluídas nesse projeto.

Este editor apresenta uma interface gráfica que permite ao usuário maior precisão na manipulação de sinais. O *software* foi desenvolvido em código aberto e permite que futuras contribuições lhe sejam agregadas.

Abstract

The Cross-Platform Graphic Audio Wave Editor is a wave file editor. The basic functions of an editor such as copy, cut, paste, gain and mono-stereo conversion are included in this project.

This software holds a graphic interface that gave the user a better precision in manipulating audio waves. The work was developed under open source to make future contributions possible.

Palavras Chave

Processamento de Sinais

Wave

MP3

Codificação

Decodificação

Restauração

Filtro Digital

Processamento de Áudio

Equalização

Sumário

SUMÁRIO

Lista de Figuras

Capítulo I

Introdução

I.1 Motivação

A evolução digital trouxe ao mundo os computadores. Em todos os campos do conhecimento e dos negócios os profissionais puderam tirar grande proveito da nova ferramenta de que dispunham. E não foi diferente com os profissionais da área de áudio. O trabalho e as dificuldades que anteriormente existiam em efetuar pequenas modificações em sinais tornaram-se mais simples com a utilização dos computadores.

Os computadores continuaram a sua evolução, e percebendo o aumento da capacidade de processamento, os profissionais que utilizavam tais máquinas para este fim começaram a exigir a execução de tarefas cada vez mais complexas.

O desenvolvimento tanto das máquinas quanto dos programas que processam áudio chegou ao ponto, em que, atualmente, existe a possibilidade de possuirmos uma estação habilitada a editar arquivos de áudio em casa, permitindo assim que um usuário leigo realize suas modificações.

Antes da utilização dos computadores para o tratamento de áudio, as gravações em vinil e em fita magnética eram os equivalentes a arquivos de áudio, pois continham a informação do sinal. Com a introdução dos computadores, estes sinais foram convertidos

para o padrão utilizado pelos computadores, e transformaram-se em sinais digitais. A evolução das técnicas de tratamento de arquivos digitais evoluiu a tal ponto que permitiu o desenvolvimento de técnicas de compressão, reduzindo assim, o tamanho dos arquivos.

Criou-se então um novo cenário no qual uma ampla troca de músicas se concretizou. Esse enorme volume de músicas e de ferramentas de manipulação impele os usuários a efetuarem modificações nos arquivos de áudio, já que nem sempre os arquivos vêm arrematados como em uma gravação profissional.

O público que tem acesso às ferramentas e aos arquivos de música espera conseguir realizar as modificações de que tem necessidade. O grande desafio passa a ser disponibilizar uma ferramenta que tenha a maior flexibilidade possível levando em conta o tipo de usuário.

Além de possuir esta flexibilidade, a ferramenta deve, a partir de uma interface que requeira o mínimo de conhecimento, permitir que um usuário realize modificações a partir de funções tais como marcar trechos para executar modificações, realizar compensação de volume, *cross-over* de faixas e desvanescimento, dentre outras.

O Editor Gráfico de Sinais de Áudio Multiplataforma é uma ferramenta que permite que sejam efetuadas modificações e reparos simples em sinais de áudio, como concatenar arquivos diferentes e corrigir distorções de volume, entre outros.

O editor é útil principalmente para usuários não profissionais com um pouco mais de conhecimento do que um completo leigo, oferecendo, contudo, inúmeras facilidades.

A intenção de construir uma ferramenta com tal finalidade advém do interesse de manipulação de arquivos de áudio em dimensão não profissional. Sua implementação baseou-se nas necessidades mais corriqueiras de um usuário comum, aliada a uma interface não complexa e o mais intuitiva possível, permitindo que a ferramenta seja usada por pessoas com limitado nível de conhecimento técnico.

Algumas ferramentas com o mesmo propósito existem no mercado. Algumas são soluções pagas; dentre elas, podemos destacar *SoundForge* e *ProTools*. Outras são livres

e de código aberto, como o *Audacity* [1]. Porém, essas ferramentas já estão em um nível de desenvolvimento bem elevado, e não possibilitam o entendimento das menores partes de um *software* do gênero, sendo assim, este é mais um fator motivador para o desenvolvimento do projeto.

I.2 Histórico

Como nosso tema principal nesse projeto é a edição de arquivos de áudio, vale percorrer desde o passado os inúmeros passos que levaram ao que hoje entendemos por edição de áudio.

Até aproximadamente 1877 não existia nenhuma forma de reproduzir a música, a não ser quando executada ao vivo por músicos [2]. Em 1877, Thomas Alva Edison fez a primeira gravação da voz humana no então primeiro cilindro fonográfico. Ele construiu uma máquina capaz de reproduzir o som.

Até esse ponto, inúmeras tentativas haviam sido executadas, cilindros de metais crivados com saliências e depressões produziam sons, mas não eram, até então, gravações de fato, e apenas reproduziam sons que eram pré-selecionados para a gravação.

Em 1881, surgem os primeiros discos de corte lateral, muito parecidos em formato com os discos de vinil, invento de Charles Tainter. O grande problema deste invento é que não existia nenhuma máquina capaz de reproduzir o que havia sido gravado, pela falta de amplificação.

Um segundo tipo de fonógrafo foi inventado por Chichester Bell e Charles Tainter em 1885, em que o som era reproduzido a partir de um cilindro vertical coberto por cera no qual eram feitas as gravações.

Mais um tipo de fonógrafo, o terceiro, foi inventado por Emile Berliner em 1887. O invento foi chamado de gramofone e não utilizava cilindro de cera, mas sim um disco de borracha rígida. O processo de gravação consistia em fazer um molde de zinco que era

corroído por ácido, e depois disso, as cópias de borracha eram prensadas. Foi o primeiro a produzir cópias, podendo assim ser considerado o primeiro a atingir a produção em massa.

Apesar da constante evolução nos aparelhos de reprodução e nas mídias utilizadas, até então, tudo se tratava sempre da mesma idéia, até que em 1898 o dinamarquês Valdemar Poulsen inventou o primeiro gravador baseado em princípios magnéticos, chamado de telegrafone.

Muitos anos se passaram entre essa descoberta e a implementação de fato da tecnologia para a gravação em fitas magnéticas. A mesma só se deu em 1928 quando Fritz Pfleumer patenteou um aparato que permitia manipular papel e filme. Ele mesmo começou a desenvolver aparelhos que fossem capazes de gravar e reproduzir sons.

Devemos observar que houve um intervalo muito grande entre os primeiros discos e os aparatos magnéticos. Nesse ínterim, as tecnologias de gravação foram melhoradas, nasceram grandes corporações de música, conhecidas atualmente, e ainda houve muita disputa em torno de um mercado que se mostrava cada vez mais promissor.

Nesse meio tempo, em 1894, o rádio também foi inventado, e logo surgiram aparelhos que foram adaptados para reproduzir a música em programas de rádio.

Com o desenvolvimento da fita magnética, surgem as primeiras idéias de edição. Como hoje, as gravações eram feitas em vários *takes*; se uma parte da gravação ficava melhor em um e outra parte melhor em outro trecho, os pedaços eram cortados, colados e gravados para outra fita. Isso constitui o primórdio da edição.

A evolução do computador e a possibilidade de gravar informações em componentes magnéticos desempenharam papel fundamental no desenvolvimento do áudio digital.

As primeiras tentativas de criar áudio digital datam de 1962 e foram feitas por Tom Stocklham do MIT (Massachusetts Institute of Technology). Suas tentativas consistiram em, através de conversores A/D (Analógico Digital) e D/A (Digital Analógico), passar o som para o computador, e reproduzi-lo através do mesmo.

I.2 Histórico

A empresa Mellotron criou o primeiro teclado com amostras de sons de instrumentos diferentes. Não nos referimos a sons de teclados, mas por exemplo, sons de violino ou mesmo de flauta. Isso aconteceu em 1963 e foi usado por ícones da música como *The Beatles*, *Pink Floyd* e *Led Zeppelin*.

Até esse ponto, os teclados apenas reproduziam sons pré-gravados, até que em 1979, Peter Vogel e Kim Ryrie desenvolveram o *Fairlight Computer Musical Instrument*, que permitia que sons fossem gravados em discos flexíveis.

As idéias para utilizar computadores e processamento de áudio não pararam por aí. Em 1969, o professor Francis Lee e o engenheiro Chuck Bagnaschu já haviam criado a American Data Sciences (que se tornou Lexicon em 1969) para desenvolver aparatos de áudio baseados no atraso digital para monitoramento cardíaco e que fora projetado pelo próprio Lee. Barry Blesser desenvolveu uma linha de atraso de 100ms e, com a ajuda de Steven Temmer da Gotham Áudio, vendeu a estúdios de gravação processadores de efeitos e equipamentos que realizavam o efeito de reverberação. Além destes aparelhos, eles desenvolveram a Estação de Áudio Digital OPUS.

A empresa Intel produziu circuitos integrados que foram usados nos primeiros aparelhos de áudio digital em 1971. Tais circuitos foram utilizados em processadores de áudio digital por empresas como Sony, Mitsubishi, Hitachi e JVC, e também foram usados nos *compact discs* da Philips.

A Philips demonstrou seu videodisco ótico em setembro de 1972 usando um reprodutor a laser e um disco de vidro. No mesmo ano, a Nippon Columbia desenvolveu um aparelho para masterizar faixas utilizando a tecnologia PCM, e como unidade de armazenamento uma fita de vídeo com um *range* dinâmico de 87 dB.

Em 1973, a IBM desenvolveu o disco rígido, com dois pratos de 30 MB cada. Já em 1975 o sintetizador digital Synclavier foi desenvolvido no Dartmouth College. Seus criadores fundaram a New England Digital Corporation para vender equipamentos profissionais para a indústria fonográfica.

Em 1980, Philips/Sony terminaram de desenvolver o projeto de *compact disc* e cada

uma delas começou separadamente a fabricar e vender seus próprios produtos.

É digna de nota a grande facilidade que o processamento digital de áudio (e sinais de um modo geral) trouxe. Se antes necessitávamos de processos mecânicos para a gravação, reprodução e processamento, hoje estamos livres disso. Os computadores e os equipamentos digitais permitiram um modo mais eficaz de registrar os arquivos, uma qualidade insuperável do ponto de vista de reprodução e facilitaram enormemente o processamento dos mesmos arquivos.

A captação para a gravação e a reprodução continuam bastante dependentes dos conceitos mecânicos antigos. Isso porque o processo para captar o som produzido por um instrumento ou a voz de um cantor baseia-se na conversão de ondas mecânicas em sinal elétrico. E depois, para serem reproduzidos, os sinais elétricos precisam ser convertidos em ondas mecânicas.

Porém, o processamento de sinais não precisa ser feito em cima das ondas mecânicas (registros análógicos), o que nos fornece como vantagem um ganho em precisão. Modernos algoritmos tratam sinais de áudio aplicando-lhes efeitos, tanto corretivos quanto para mixagem, e atualmente podemos contar com uma nova dinâmica no que se refere a arquivos de áudio.

I.3 Objetivos

O cenário atual é extremamente favorável para a implementação de um *software* que permita a manipulação de arquivos de áudio. Existem inúmeros arquivos de áudio transitando na rede mundial de computadores (*internet*), muito deles com qualidade não tão boa ou com alguns arremates não tão precisos. A facilidade de possuirmos máquinas capazes de efetuar processamento é uma realidade.

Um objetivo que temos é disponibilizar uma ferramenta livre, que possa ser utilizada com o intuito de realizar manipulação de áudio, como dito anteriormente. Para atingirmos nossos objetivos alguns pontos básicos devem ser alcançados, e abaixo enumeramos os

mesmos:

- Normalização de amostras e compensação de volume

  É comum encontrarmos inúmeros arquivos em que as músicas estão gravadas com volumes incompatíveis ou trechos muito mais altos ou baixos do que se espera. Por isso é desejável poder normalizar a amplitude das amostras.

- *Cross-Over* de faixas

  Quando existe a reprodução em seqüência de duas faixas e não queremos aguardar o silêncio entre o fim de uma música e o início da outra podemos diminuir o volume da faixa anterior e aumentar o volume da faixa posterior simultaneamente.

- Desvanecimento

  A redução gradativa de amplitude do sinal é uma forma de concluir uma seleção musical.

- Mono e Estéreo

  Por questão de compatibilidade, pode tornar-se interessante a função de transformar a fonte de mono (um canal) para estéreo (dois canais) ou vice-versa.

- Identificar tempo (índice da amostra) x amplitude (visualisação gráfica)

  Um dos principais objetivos do aplicativo é a visualização de arquivos *Wave*. Esta funcionalidade proverá uma interface gráfica na qual as amostras serão exibidas. Cada amostra tem uma posição definida no tempo e possui determinada amplitude. A forma de exibição respeitará essas informações.

- Marcar trechos (início - final) a processar

  Para que as amostras possam ser seletivamente editadas torna-se necessário que, através da interface gráfica, exista a possibilidade da marcação de trechos determinados a fim de processá-los.

I.3 Objetivos

- *Zoom*

  Quando estamos visualizando um arquivo de música ao longo de sua extensão completa não temos uma visão minuciosa do sinal. A necessidade de editar um trecho em especial, ou somente estudar mais de perto as nuanças do mesmo, traz a necessidade de "aproximar" as amostras de nossa visão para que elas se tornem o centro de nossa atenção.

- Execução

  Uma funcionalidade indispensável é a execução de um arquivo, que consiste na interpretação dos dados, e por conseguinte, a tradução do mesmo para os dispositivos de reprodução.

- Apagar

  Recortar algum trecho do sinal e ligar os dois pontos que o delimitam.

- Dar ganho

  Aumentar ou diminuir o valor das amplitudes das amostras, enfatizando ou atenuando o trecho processado.

- Localizar máxima amplitude

  A localização do ponto de máxima amplitude permite que o arquivo musical seja normalizado. Um exemplo claro de utilização é quando o arquivo musical foi gravado com baixo volume. A partir da normalização, podemos aumentar o valor de todas as amostras, conservando a relação entre elas.

- Conversão *Wave* - MP3

  Permitir a compressão de arquivos editados no formato MP3 e a descompressão dos mesmos.

I.4 Estrutura do Texto

O texto expõe a teoria relacionada ao processamento de sinais e a base para o desenvolvimento das funções no Capítulo II. A modelagem do *software* é apresentada no Capítulo III. No Capítulo IV, descreveremos a implementação. As conclusões são apresentadas no Capítulo V.

O apêndice A lista os acrônimos, o apêndice B indica os requisitos mínimos para a instalação do *software*. O apêndice C aborda os requisitos mínimos para compilar o código do *software*, citando as bibliotecas auxiliares utilizadas e ainda como instalá-las no sistema. O apêndice D mostra o processo de instalação do *software* e por fim, o apêndice E é um manual de utilização do mesmo.

O código-fonte do aplicativo está contido na mídia eletrônica que acompanha o texto.

Capítulo II

Teoria

II.1 Visão do Processamento de Sinais

Apesar de ser um projeto intimamente ligado ao desenvolvimento de *software*, devemos nos deter em alguns conceitos importantes no campo do processamento de sinais. As funções que podem ser aplicadas ao sinal de áudio derivam de tais conceitos, e o propósito desse capítulo é relacionar as funções implementadas à teoria.

II.2 Amostragem

Um sinal é a descrição no tempo de algum fenômeno físico. Normalmente esse sinal é correlacionado a alguma informação inerente à sua forma. Um arquivo de áudio, portanto, contém um sinal elétrico que varia sua amplitude no tempo e carrega informação pertinente aos sons, como freqüências e volume a ele associados. Normalmente os sinais são representados como sendo funções matemáticas.

Podemos dizer que uma função é contínua, se para cada valor de sua abcissa (eixo x), existe um valor correspondente em sua ordenada (eixo y) e ainda não existem saltos entre valores possíveis de sua função. Isso equivale dizer que para cada ponto de seu

domínio existe um valor correspondente para a função. Quando estamos nos referindo a funções que possuem o tempo como domínio, podemos dizer que para cada valor no tempo corresponde um valor da função. Esses sinais são chamados de analógicos.

Por outro lado, uma função é dita discreta quando sua ordenada não varia continuamente com os valores de seu domínio. Mais que isso, seu valor é apenas considerado em instantes de tempo múltiplos inteiros de um determinado intervalo de tempo que é o inverso da freqüência de amostragem. Ao amostrarmos um sinal, o fazemos discreto. Depois de quantizá-lo, isto é, adaptar os níveis do sinal em valores pré-estabelecidos, temos o sinal discreto. Seu domínio passa a ser um conjunto limitado de valores, e conseqüentemente, os valores que a função assume só fazem sentido nesses pontos específicos. Esses são os chamados sinais digitais.

A versão discreta para a representação de sinais é, como o seu nome alternativo indica, mais fácil para o processamento em ambientes digitais, como é o caso dos microcomputadores. Dessa forma um sinal que é analógico passa a ser armazenado em um formato digital equivalente para o processamento.

O processo que leva um sinal da forma analógica para a forma discreta chama-se amostragem. A amostragem é o processo de tomar algumas amostras da onda em determinados pontos no tempo, sendo que essa distância é constante. Algumas peculiaridades envolvem a amostragem; dentre elas podemos citar a sua freqüência de captura de amostras.

Como dito anteriormente, é através do processo de amostragem que um sinal analógico é transformado em uma série de amostras, que são normalmente espaçadas uniformemente no tempo. Para que esse processo faça sentido, devemos escolher uma freqüência de amostragem apropriadamente, de forma que a seqüência de amostras defina unicamente o sinal analógico. Essa é basicamente a premissa do teorema da amostragem, que é apresentado a seguir [3].

Consideremos um sinal arbitrário $g(t)$, que possui valor para qualquer t. Um segmento desse sinal pode ser visto na Figura II.1 . Suponhamos que o sinal seja amostrado instan-

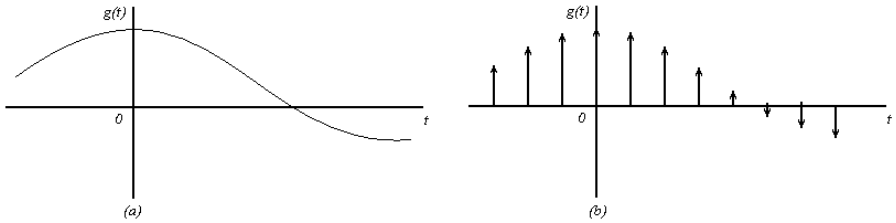

Figura II.1: Amostragem: (a) Sinal analógico. (b) Versão amostrada do sinal analógico.

taneamente e a uma taxa uniforme, a cada T_s segundos. Como resultado, obtemos uma seqüência infinita de amostras espaçadas de T_s uma da outra. Chamemos cada amostra dessas de $g(nT_s)$, onde n é qualquer inteiro. Chamemos a T_s de período de amostragem, e assim, $f_s = 1/T_s$ é sua freqüência de amostragem. Essa forma de amostragem é chamada de amostragem instantânea.

Agora chamemos de $g_\delta(t)$ o sinal obtido após multiplicarmos os elementos de uma função δ periódica espaçados de T_s por uma seqüência de números $g(nT_s)$ como mostrado na Figura II.1.

$$g_\delta(t) = \sum_{n=-\infty}^{\infty} g(nT_s)\delta(t - nT_s) \tag{II.1}$$

onde $g_\delta(t)$ é o sinal amostrado ideal. O termo $g(nT_s)\delta(t - nT_s)$ representa a função delta posicionada no tempo $t = nT_s$. Lembramos que esta função possui área unitária.

Consideremos, agora, a transformada de Fourier:

$$g_\delta(t) \rightleftharpoons f_s \sum_{m=-\infty}^{\infty} G(f - md_s) \tag{II.2}$$

onde $G(f)$ é a transformada de Fourier do sinal original $g(t)$, e f_s é a taxa de amostragem.

Outra expressão usual para a transformada de Fourier idealmente amostrado do sinal $g_\delta(t)$ pode ser obtida tomando-se a transformada de Fourier de ambos os lados da equação (II.1) e notando que a transformada de Fourier da função delta $\delta(t - nT_s)$ é igual a

$exp(-j2\pi nfT_s)$. Denotemos $G_\delta(f)$ a transformada de Fourier de $g_\delta(t)$. Dessa forma podemos escrever:

$$G_\delta(f) = \sum_{n=-\infty}^{\infty} g(nT_s)exp(-j2\pi nfT_s) \tag{II.3}$$

Essa relação é chamada de *Transformada de Fourier Discreta no Tempo*. Ela pode ser vista como uma representação de uma Transformada de Fourier Complexa do sinal periódico $G_\delta(f)$, com a seqüência de amostras $g(nT_s)$ definindo os coeficientes da expansão.

As relações, como mostradas aqui, são aplicáveis a quaisquer sinais contínuos no tempo $g(t)$ de energia finita e duração infinita. Suponha, no entanto, que o sinal $g(t)$ seja limitado em banda, com nenhuma componente de freqüência maior do que W Hertz. Em outras palavras, a transformada de Fourier $G(f)$ do sinal $g(t)$ tem a propriedade de possuir valor zero para $|f| \geq W$, como ilustrado na figura II.2. Supondo que tenhamos escolhido o período de $T_s = 1/2W$. Então o espectro correspondente $G_\delta(f)$ do sinal amostrado $g_\delta(t)$ é mostrado na Figura II.2. Ao substituirmos o valor $T_s = 1/2W$ na equação (II.3) ficamos com

$$G_\delta(f) = \sum_{n=-\infty}^{\infty} g\left(\frac{n}{2W}\right) exp\left(-\frac{j\pi nf}{W}\right) \tag{II.4}$$

Da equação (II.2), vemos que a transformada de Fourier de $g_\delta(t)$ também pode ser expressa por

$$G_\delta(f) = f_s G(f) + f_s \sum_{\substack{m=-\infty \\ m\neq 0}}^{\infty} G(f - mf_s) \tag{II.5}$$

Quando ocorrem as condições abaixo:

1. $G(f) = 0$ para $|f| \geq W$

2. $f_s = 2W$,

então, a partir da equação (II.5) temos

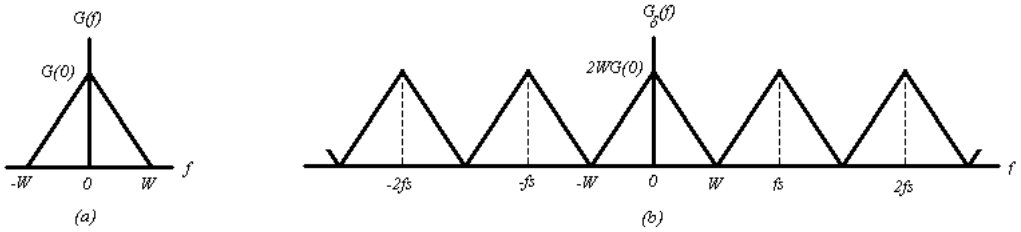

Figura II.2: (a) Espectro de um sinal g(t) limitado em banda. (b) Espectro de uma versão amostrada do sinal g(t), com um período de amostragem de 2T.

$$G(f) = \frac{1}{2W} G_\delta(f), \qquad -W < f < W \tag{II.6}$$

Substituindo a equação (II.5) na equação (II.6), obtemos

$$G(f) = \frac{1}{2W} \sum_{n=-\infty}^{\infty} g\left(\frac{g}{2W}\right) exp\left(-\frac{j\pi nf}{W}\right), \qquad -W < f < W. \tag{II.7}$$

Assim, se os valores das amostras $g(n/2W)$ de um sinal $g(t)$ são especificados para todos os valores de n. A transformada de Fourier $G(f)$ é unicamente determinada usando a Transformada de Fourier Discreta no Tempo da equação (II.7). Devido ao fato de $g(t)$ ser relacionada a $G(f)$ pela Inversa da Transformada de Fourier, segue que o sinal $g(t)$ é unicamente determinado pelas amostras $g(n/2W)$ para $-\infty < n < \infty$. Em outras palavras, a seqüência $g(n/2W)$ tem toda a informação contida em $g(t)$.

Considere agora o problema de reconstruir $g(t)$ a partir da seqüência de amostras $g(n/2W)$. Substituindo a equação (II.7) na fórmula da Inversa da Transformada de Fourier e definindo $g(t)$ em termos de $G(f)$, temos

$$g(t) \;=\; \int_{-\infty}^{\infty} G(f) exp(j2\pi nf) df$$

$$= \int_{-W}^{W} \frac{1}{2W} \sum_{n=-\infty}^{\infty} g\left(\frac{n}{2W}\right) exp\left(-\frac{j\pi nf}{W}\right) exp(j2\pi nf)df \quad \text{(II.8)}$$

Modificando a ordem do somatório e da integração,

$$g(t) = \sum_{n=-\infty}^{\infty} g\left(\frac{n}{2W}\right) \frac{1}{2W} \int_{-W}^{W} exp\left[j2\pi f\left(t - \frac{n}{2W}\right)\right] df \quad \text{(II.9)}$$

Resolvendo a integral na equação acima, obtemos

$$\begin{aligned} g(t) &= \sum_{n=-\infty}^{\infty} g\left(\frac{n}{2W}\right) \frac{sin(2\pi Wt - n\pi)}{2\pi Wt - n\pi} \\ &= \sum_{n=-\infty}^{\infty} g\left(\frac{n}{2W}\right) sinc(2Wt - n), -\infty < t < \infty \end{aligned} \quad \text{(II.10)}$$

Essa última equação nos fornece uma fórmula para reconstruir o sinal original $g(t)$ a partir da seqüência de amostras $g(n/2W)$, com a função $sinc(2Wt)$ exercendo o papel de função de interpolação. Cada amostra é multiplicada por uma versão atrasada da função de interpolação, e todos os resultados são somados para que se obtenha o sinal $g(t)$.

II.3 Teoria Referente às Funções

Depois dessa visão sobre amostragem de sinais abordaremos alguns aspectos teóricos relevantes das funções utilizadas na nossa ferramenta.

- A função de normalização de amostras utiliza um conceito bastante simples. Ela multiplica todo o sinal por um fator de correção para que nenhum trecho da gravação seja privilegiado em relação a outros.

  O fator de correção citado anteriormente é na verdade a relação entre o máximo valor que pode ser representado em um arquivo *WAVE* dividido pelo maior valor encontrado nas amostras do arquivo que se pretende normalizar.

- A compensação de volume utiliza um pensamento semelhante. Temos como diferença o fato de podermos marcar um trecho para editarmos. Outro aspecto relevante é a possibilidade que se abre por não termos o fator de correção fixo, ou seja, podemos aumentar o valor das amostras de um a partir da multiplicação por um fator de correção que não aquele determinado pela razão entre o valor máximo que pode ser representado em um arquivo *WAVE* e a maior amplitude do sinal.

- A função de *cross-over* de faixas utiliza a sobreposição de sinais como idéia básica. Uma sobreposição sem nenhum arremate no ponto de encontro pode não ser perfeita, ou seja, o final do primeiro sinal pode não encaixar no início do segundo sinal de forma que suas ondas se complementem. Se as ondas não se encontrarem num ponto em que possuam o mesmo valor de amplitude da amostra, ocorre um *click*, que é perceptível ao ouvido.

  Desta forma precisamos fazer uma pequena correção no ponto de encontro dos sinais. O que utilizamos como artifício é diminuir gradativamente o valor das amostras do primeiro sinal, e ao mesmo tempo, aumentar gradativamente o valor das amostras do segundo sinal.

  Para obtermos o efeito de diminuir ou aumentar gradativamente o valor das amostras, multiplicamos o sinal original por uma função monotônica crescente ou decrescente, de acordo com o efeito que desejamos obter. Assim, mesmo que o sinal não possua o mesmo valor no ponto de encontro não percebemos a emenda ocorrendo.

- A função de desvanecimento é bastante similar à função anterior. Só que como lidamos apenas com um sinal, aplicamos a função monotônica decrescente, de forma que gradativamente, cada amostra tenha o seu valor decrescido. Um exemplo de utilização seria em gravações ao vivo, as quais normalmente se encerram com uma salva de palmas. A figura II.3 mostra a o processo de multiplicarmos um sinal de áudio ordinário (no topo da figura) por uma função monotônica decrescente (no centro da figura) e obtemos como resultado o produto entre as duas (na parte inferior da figura).

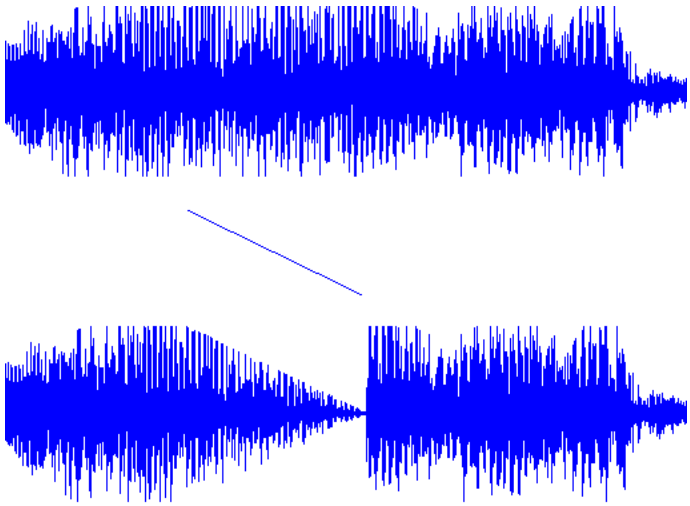

Figura II.3: Multiplicação de um sinal de áudio ordinário por uma função monotônica decrescente e seu produto.

- A transformação de um arquivo *WAVE* de mono para estéreo ou vice-versa é trivial. Se possuímos um arquivo estéreo (dois canais) e queremos chegar a um arquivo mono (um canal) precisamos somar os sinais de ambos os canais e dividirmos por dois. Dessa forma mantemos o volume do sinal, porém obtemos as duas informações em apenas um canal.

  O processo de transformar um arquivo mono em estéreo é o reverso do anterior, ou seja, o valor do sinal é entregue idêntico a cada canal.

- A função de dar ganho em algum trecho marcado ou ao arquivo inteiro segue uma idéia já utilizada previamente. Se o ganho for linear, multiplicamos as amostras do trecho em questão por um fator multiplicativo previamente escolhido. Se o ganho for logarítmico, devemos multiplicar o trecho por uma função logarítmica.

II.3.1 Compressão MP3

Na computação atual, o espaço necessário para guardar arquivos e para realizar trocas é um fator importante. Para isso várias técnicas de compressão são utilizadas. Para cada

tipo de arquivo existe uma forma de compressão específica que faz uso de características próprias ao tipo de arquivo. A utilização desses fatores gera arquivos menores, com maior grau de compactação. Algoritmos distintos fazem realmente diferença para cada tipo de arquivo usado. Para examinar esse comportamento, podemos tentar compactar algum arquivo com extensão *wav* com as ferramentas do algoritmo *zip*, o tamanho quase não se modifica.

Existem duas formas de compactação: sem e com perdas. A primeira utiliza as redundâncias dos arquivos para diminuir a informação transmitida. Um exemplo de algoritmo que realiza compressão sem perdas é o *Ape*. A segunda utiliza aspectos piscoacústicos para realizar a compressão. Ou seja, só comprime o que é necessário para que se reproduza e cause a sensação que o arquivo original gerava. Dessa forma muitos dados são descartados.

Um dos grandes fatores que impulsionaram a troca de arquivos de áudio foi a criação de técnicas que permitiram a diminuição do tamanho dos arquivos. Alguns grupos de estudiosos formaram-se com o intuito de criar algoritmos que permitissem a redução do tamanho dos arquivos originais. Uma técnica oriunda de um desses grupos de trabalho foi a que realiza a compressão de arquivos do tipo *raw* para o formato citado.

Arquivos do tipo *raw* são aqueles encontrados em arquivos de áudio comuns, como por exemplo, a forma em que encontramos as faixas em um *compact disc* comum. MP3 é a abreviatura de MPEG 1, layer 3, MPEG significa *Moving Picture Experts Group*. *Layer* 3 indica a faixa alvo de velocidades.

Abordaremos as técnicas de compressão e descompressão, ou seja, o caminho de ida e o de volta da transformação de arquivos do tipo *raw* para o tipo MP3. Para isso contaremos com o auxílio do desenho em alto nível dos blocos que compõem o sistema [4].

Vemos na figura II.4 o esquema de um codificador perceptual de áudio. Tipicamente, a entrada de dados recebe um áudio no formato PCM. O sinal é então mapeado para o domínio da freqüência utilizando algum tipo de banco de filtros. Os dados no domínio da freqüência são quantizados depois e empacotados em um *stream* de bits. A quantiza-

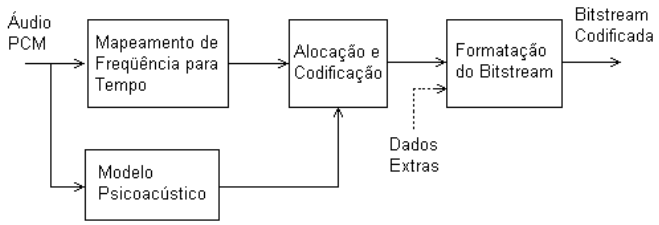

Figura II.4: Blocos básicos para a construção de um codificador.

ção é realizada utilizando uma alocação de bits que visa a maximizar a taxa sinal-ruído (SNR) menos a taxa sinal-máscara (SMR) de cada bloco de dados. O estágio do modelo psicoacústico analisa o sinal de entrada, determina o nível de mascaramento em cada componente freqüencial e computa os valores da taxa SMR. A rotina de alocação de bits aloca um número determinado de bits de mantissa para o banco de dados do domínio da freqüência sobre as componentes mais fortes do sinal e suas taxas SMR relativas.

O *stream* de bits contém o áudio codificado propriamente dito e outros dados que podem ser utilizados futuramente para a descompressão, realizada em cada execução do arquivo. Esses dados compõem um cabeçalho para serem acessados mais facilmente na descompressão.

Análise Detalhada dos Blocos

O processo de codificação começa passando pelo bloco que mapeia o sinal do domínio do tempo para o domínio da freqüência. A idéia básica do mapeamento do domínio do tempo para o domínio da freqüência é reduzir a redundância em um sinal de áudio através da subdivisão de seu conteúdo em suas componentes freqüenciais e então alocar apropriadamente um conjunto de bits. Sinais altamente tonais possuem componentes de freqüência esparsas que variam lentamente no tempo. A quantidade de dados necessária para descrever espectralmente esses sinais em blocos pode ser significativamente menor do que os que seriam necessários para descrever os mesmos sinais amostra-a-amostra à medida que o tempo passa [4].

II.3 Teoria Referente às Funções

A técnica básica do mapeamento do domínio do tempo para a freqüência é passar o sinal através de um banco de filtros em multitaxa que divide o sinal em K bandas distintas de freqüências. O sinal relativo a cada banda de freqüência será quantizado com um número variável de bits, alocando a maior parte do ruído de quantização em faixas de freqüência que são menos audíveis.

Paralelamente à passagem do sinal pelo bloco mapeador do tempo para a freqüência, o sinal é processado pelo bloco psicoacústico. A psicoacústica é a ciência que estuda as relações estatísticas entre os estímulos acústicos e as sensações de percepção auditiva. Através do entendimento de como nosso corpo, especificamente o aparelho auditivo, reage aos estímulos, podemos ser menos fiéis aos sinais de áudio nas regiões onde nosso corpo não tem bom discernimento, ou seja, onde não compreende com clareza os estímulos. O que resulta desse estudo é o desenvolvimento de modelos matemáticos que descrevem a maior importância relativa às regiões freqüênciais citadas anteriormente, privilegiando as que mais percebemos.

Um fenômeno fundamental considerado no modelo matemático previamente citado é o mascaramento freqüêncial. Um exemplo bastante conhecido ocorre quando dois instrumentos que geram sons em freqüências próximas são executados ao mesmo tempo: o de maior intensidade sonora (som mais alto), mascara o outro.

Um outro mascaramento, conhecido como temporal também pode acontecer quando dois sons ocorrem muito próximos no tempo: o mais forte encobre o mais fraco. Divide-se em dois casos: o pré-mascaramento e o pós-mascaramento. Mesmo que seja inesperado, o pré-mascaramento ocorre, e é importante no desenvolvimento de filtros para suprimir informação da região onde ele ocorre. Já o pós-mascaramento é mais compreensível, e representa o nível de decaimento após o sinal responsável pelo mascaramento ter cessado.

O efeito do mascaramento pode ser computado das seguintes formas:

- Identificando o mascaramento de sinais no domínio da freqüência.
- Desenvolvendo curvas de mascaramento freqüenciais e temporais baseadas nas ca-

racterísticas de cada mascarador identificado.

- Combinando as curvas de mascaramento individuais com cada uma das demais e com o limite do silêncio para criar um limite total de audibilidade do sinal.

Esse limite de audibilidade geral ou máscara de limite é então utilizado para identificar as componentes inaudíveis do sinal e para determinar o número de bits necessários para quantizar as compontentes do sinal.

Após passar por esses dois blocos concorrentemente, o sinal entra, então, no bloco de alocação e codificação. A alocação de bits utiliza o conhecimento anterior acerca das partes do sinal em que a fidelidade deve ser privilegiada, regiões que possuem componentes com grande energia devem ser codificadas com um número grande de bits, enquanto faixas de freqüência que não possuem componentes ou que possuem componentes com baixo valor de energia podem, inclusive, não ser codificadas. Dessa forma, o ruído da quantização pode ser controlado separadamente em cada banda. Ainda, levando em consideração os padrões de mascaramento gerados por cada componente, o ruído de quantização pode ser construído para ser inaudível.

Manter a qualidade do sinal igual em toda sua extensão com um número fixo de bits gera taxas de amostragens diferentes nos blocos que o constituem. Encontramos também aplicações em que a taxa de amostragem precisa ser fixa, então o que se busca é a melhor qualidade possível dentro desse novo limite. Essas situações mostram algumas formas de codificar o sinal com respeito à quantidade utilizada de bits.

Depois de ter sofrido a alocação e a codificação, o sinal já está pronto para ser empacotado. No bloco de formatação, cada bloco do sinal recebe um cabeçalho contendo informações a respeito dos dados codificados que ele carrega. Dessa forma o decodificador será capaz de recuperar o áudio codificado contido naquele bloco.

O *stream* de bits codificados inclui o áudio codificado, isto é, as mantissas, e os fatores de escala, como a alocação de bits; e ainda quaisquer outros parâmetros de controle, tais como extensão do bloco e tipo de janelamento, que são necessários para informar ao

decodificador como realizar seu trabalho com os dados constituintes do *stream* de bits.

A palavra de sincronização, a taxa de amostragem e a taxa de dados, entre outros, são normalmente componentes do cabeçalho e são transmitidos ao decodificador em intervalos de tempo pré-determinados. Finalmente, os códigos de correção de erros, as marcas de sincronização de tempo e outros dados auxiliares também podem ser incluídos no *stream* de dados. O *stream* de dados pode ser armazenado ou pode ser transmitido diretamente para o decodificador.

Descompressão

Os blocos básicos de um decodificador de áudio são exibidos na figura II.5. Primeiro, o *stream* de bits é desempacotado em suas partes constituíntes, ou seja, dados de áudio, parâmetros de controle e eventuais dados agregados. A informação de alocação de bits é utilizada para desquantizar os dados de áudio presentes em cada sub-banda e recuperar da melhor forma possível os dados originais na representação do sinal no domínio da freqüência. Estes são combinados utilizando-se o banco de filtros apropriado. Os dados reconstruídos no domínio do tempo contêm ruído de quantização, mas se o modelo psicoacústico desempenhou corretamente o seu papel, gera-se o áudio com ruído inaudível ou muito próximo disso. Depois dessa reconstrução, os dados de áudio no formato PCM vão formar uma *stream* de bits de saída[4].

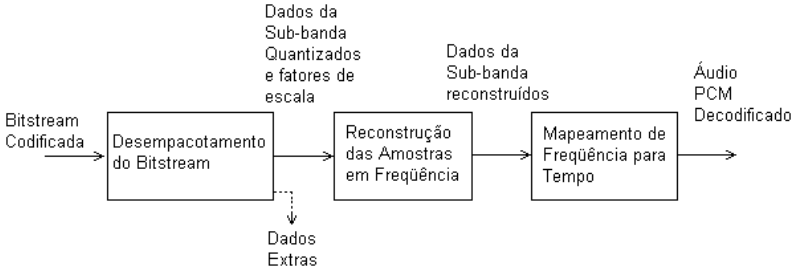

Figura II.5: Blocos básicos para a construção de um decodificador.

Capítulo III

Modelagem

Neste capítulo iremos abordar a modelagem dos objetos utilizados no projeto em questão com a finalidade de desenvolver o *software*. Utilizamos alguns conceitos muito conhecidos sob o escopo da UML (*Unified Modeling Language*), tais como o processo de desenvolvimento, casos de uso, diagrama de classes e pacotes e colaborações.

Ao longo do capítulo iremos explorar com maiores detalhes cada ponto relevante e de interesse. Iremos explicitar também o motivo de nossas escolhas. Deixamos para o capítulo posterior as dificuldades de implementação e a abordagem dos principais pontos do código propriamente dito.

III.1 Processo de Desenvolvimento

A UML é uma metodologia de análise e projeto orientado a objetos (OOA & D, Object Oriented Analysis and Design) que surgiu no final dos anos oitenta e no início dos anos noventa. Ela unifica três métodos que vinham sendo desenvolvidos em paralelo e o resultado final é uma padronização baseada nos três modelos [5].

Na realidade, UML é uma linguagem de modelagem e não um método. Esta linguagem é uma notação (principalmente gráfica) utilizada para expressar partes de um projeto. É

importante notar que ela não apresenta uma seqüência de passos a serem seguidos para a elaboração de um projeto.

O processo que utilizamos para o desenvolvimento do *software* foi iterativo e incremental (evolucionário), nada foi implementado antes do fim da etapa de concepção de projeto para cada incremento. Foram estipuladas fases e a construção foi realizada em partes. Podemos dividir nosso processo em quatro partes básicas: concepção, elaboração, construção e transição. Podemos ver na figura III.1 a forma temporal como o processo é dividido.

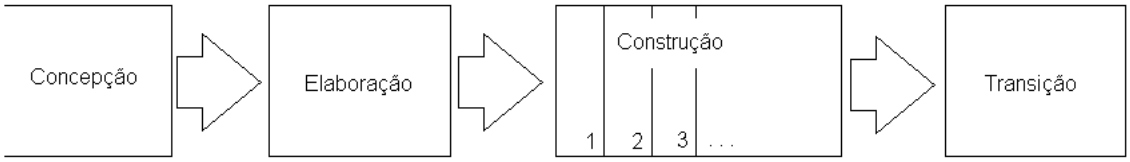

Figura III.1: Esboço do processo de desenvolvimento.

A fase de concepção foi o período no qual surgiu a idéia de implementação do *software*. Inicialmente a idéia era apenas construir um conversor de padrão *WAVE* para MP3 e vice-versa. Mas a percepção das necessidades de diversos usuários indicava a elaboração de uma ferramenta mais ampla. O desenvolvimento de um programa capaz de realizar as operações mais básicas de edição de áudio foi a resposta para a percepção das necessidades comuns aos entusiastas da edição de áudio. O objetivo passou a ser construir um editor de sinais contidos em arquivos de áudio que possuísse interface gráfica.

O interesse surgiu mediante alguns problemas enfrentados por usuários amadores de programas de áudio para a execução de tarefas comumente realizadas. Nesse contexto a fase de elaboração tomou lugar. Precisávamos fixar alguns objetivos e estratégias a serem seguidos no desenvolvimento.

Nessa fase tomamos a decisão de utilizar as linguagens C e C++ para a programação do código. Também gostaríamos que o programa seguisse a orientação de código aberto (*open source*) para que melhorias, futuras implementações e participação externa fosse possível.

Um outro fator motivante foi o conceito de programas multiplataforma (a princípio *Unix* e *Windows*). Dessa forma nenhum usuário sería excluído pela adoção de um sistema operacional em particular.

A idéia se concretizou no Editor Gráfico de Sinais de Áudio Multiplataforma. Para a sua implementação o projeto foi dividido em fases de construção.

Em cada uma dessas fases, construímos algumas funcionalidades que deveriam ser implementadas, realizamos testes sobre o que já havia sido implementado e portanto garantimos o funcionamento. As tarefas foram divididas em funções responsáveis pela parte gráfica e funções responsáveis por alterações no sinal.

Depois da fase de construção passamos à fase de transição, na qual otimizamos diversas funções e verificamos a compatibilidade das diversas partes do programa.

III.2 Casos de Uso

A utilização da técnica de Casos de Uso formalizou uma prática que era muito despojada e sem nenhuma documentação ou formalização de idéias. Antes de definir um caso de uso, torna-se necessário a definição do conceito de cenário. Um cenário é definido como uma seqüência de passos que descreve uma interação entre o usuário e o sistema.

Agora podemos definir um caso de uso. Um caso de uso é um conjunto de cenários amarrados por um objetivo comum de um usuário. De uma forma geral, um caso de uso possui um cenário em que tudo funciona como o esperado e vários outros em que diversos tipos de erros ocorrem.

Um caso de uso é descrito por uma seqüência de passos que podem ser representados por um diagrama. A seguir explicitaremos os casos de uso de nosso sistema, um a um.

III.2 Casos de Uso

Os seguintes casos de uso são relativos ao *Menu File* da implementação:

Criação de Arquivo

1. O usuário seleciona no menu *File* a entrada *New*.
2. A caixa de seleção de tipo de arquivo abre e o usuário seleciona um tipo para o arquivo.
3. O sistema abre um arquivo novo e o exibe graficamente em tela.

Abertura de Arquivo

1. O usuário seleciona no menu *File* a entrada *Open*.
2. A caixa de seleção de arquivo abre e o usuário seleciona um arquivo no formato WAVE e aperta o botão *Open*.
3. O sistema abre o arquivo e o exibe graficamente em tela.

Alternativa: Escolha de um arquivo com a extensão diferente de "wav".

Alternativa: Não é feita uma escolha de arquivo.

Salvar Arquivo

1. O usuário seleciona no menu *File* a entrada *Save*.
2. O sistema salva o arquivo com o nome "result.wav".

Salvar Arquivo com nome específico

1. O usuário seleciona no menu *File* a entrada *Save As...*
2. A caixa de seleção de arquivo abre e o usuário seleciona um arquivo para salvar ou escreve um novo nome.

3. O sistema salva o arquivo.

Exportar arquivo no formato *MP3* com nome específico

1. O usuário seleciona no menu *File* a entrada *Export MP3 As...*
2. A caixa de seleção de arquivo abre e o usuário seleciona um arquivo para salvar ou escreve um novo nome.
3. O sistema salva o arquivo no formato indicado.

Fechamento de Arquivo

1. A pré-condição é a existência de um arquivo aberto.
2. O usuário seleciona a entrada *Close* no menu *File*.
3. O sistema fecha o arquivo, deixa de exibir graficamente em tela e possibilita a abertura de outros arquivos.

Fechamento do Programa

1. O usuário seleciona a entrada *Exit* no menu *File*.
2. O programa é encerrado.

Os seguintes Casos de Uso são relativos ao *Menu Edit* da implementação:

Função *Copy*

1. A pré-condição é a existência de um arquivo aberto.
2. O usuário seleciona um trecho de onda com o *mouse*.
3. O usuário seleciona a entrada *Copy* no menu *Edit*.
4. O sistema copia para a memória o valor das amostras do trecho selecionado.

Função *Paste*

1. A pré-condição é a existência de um arquivo aberto e um trecho deste arquivo previamente selecionado e copiado para a memória.
2. O usuário seleciona com o *mouse* um ponto para colar o trecho que está em memória.
3. O usuário seleciona a entrada *Paste* no menu *Edit*.
4. O sistema copia para o vetor que representa a onda os valores das amostras que estavam em memória e exibem a mudança em tela.

Função *Cut*

1. A pré-condição é a existência de um arquivo aberto.
2. O usuário seleciona um trecho de onda com o *mouse*.
3. O usuário seleciona a entrada *Cut* no menu *Edit*.
4. O sistema copia para a memória o valor das amostras no trecho selecionado, retira as amostras do trecho selecionado do vetor que guarda os valores das amostras que estão sendo exibidas em tela e por fim exibe o vetor sem as amostras selecionadas.

Função *Select All*

1. A pré-condição é a existência de um arquivo aberto.
2. O usuário seleciona a entrada *Select All* no menu *Edit*.
3. O sistema sombreia toda a onda exibida graficamente em tela.

Os seguintes Casos de Uso são relativos ao *Menu View* da implementação:

Função *Exhibition Sample*

1. A pré-condição é a existência de um arquivo aberto.

2. O usuário seleciona a entrada *Exhibition Sample* no menu *View*.

3. O sistema passa a exibir os pontos de marcação de onda em amostras.

Função *Exhibition Time*

1. A pré-condição é a existência de um arquivo aberto.

2. O usuário seleciona a entrada *Exhibition Time* no menu *View*.

3. O sistema passa a exibir os pontos de marcação de onda em seu valor correspondente em tempo.

Função *Zoom Horizontal In*

1. A pré-condição é a existência de um arquivo aberto.

2. O usuário seleciona a entrada *Zoom Horizontal In* no menu *View*.

3. O sistema aproxima a exibição de amostras no eixo x.

Função *Zoom Horizontal Out*

1. A pré-condição é a existência de um arquivo aberto.

2. O usuário seleciona a entrada *Zoom Horizontal Out* no menu *View*.

3. O sistema distancia a exibição de amostras no eixo x.

Função *Fit*

1. A pré-condição é a existência de um arquivo aberto.

2. O usuário seleciona a entrada *Fit* no menu *View*.

3. O sistema exibe todo o arquivo no espaço disponível em tela (tamanho da janela).

Função *Expand*

1. A pré-condição é a existência de um arquivo aberto e que um trecho do arquivo esteja marcado.
2. O usuário seleciona a entrada *Expand* no menu *View*.
3. O sistema exibe o trecho selecionado no arquivo no espaço disponível em tela (tamanho da janela).

Função *Max. Amplitude*

1. A pré-condição é a existência de um arquivo aberto.
2. O usuário seleciona a entrada *Max. Amplitude* no menu *View*.
3. Se nenhum trecho do arquivo estiver selecionado, uma caixa de diálogo exibe o maior valor do módulo das amostras do arquivo, se algum trecho estiver marcado, o maior valor encontrado naquele trecho é exibido.

Função *Separate*

1. A pré-condição é a existência de um arquivo estéreo aberto.
2. O usuário seleciona a entrada *Separate* no menu *View*.
3. O sistema passa a tratar a marcação dos canais independentemente.

Os seguintes Casos de Uso são relativos ao *Menu Effect* da implementação:

Efeito *Null*

1. A pré-condição é a existência de um arquivo aberto.
2. O usuário seleciona um trecho de onda com o *mouse*.
3. O usuário seleciona a entrada *Null* no menu *Effect*.

4. O sistema assume valor zero para as amostras do trecho selecionado e exibe em tela o novo vetor de amostras.

Efeito *Null* Reverso

1. A pré-condição é a existência de um arquivo aberto.
2. O usuário seleciona um trecho de onda com o *mouse*.
3. O usuário seleciona a entrada *Null* no menu *Effect*.
4. O sistema assume valor zero para as amostras do trecho que não foi selecionado e exibe em tela o novo vetor de amostras.

Efeito *Gain*

1. A pré-condição é a existência de um arquivo aberto.
2. O usuário seleciona a entrada *Gain* no menu *Effect*.
3. É aberta uma caixa de diálogo e o usuário entra com um valor no formato *float*.
4. O sistema multiplica o valor recebido pela caixa de diálogo pelo arquivo todo, ou por um trecho se esse tiver sido selecionado.

Efeito *Fade In*

1. A pré-condição é a existência de um arquivo aberto.
2. O usuário seleciona um trecho de onda com o *mouse*.
3. O usuário seleciona a entrada *Fade In* no menu *Effect*.
4. O sistema multiplica o valor de uma função de *Fade In* pelo valor das amostras e exibe em tela o resultado.

Efeito *Fade Out*

1. A pré-condição é a existência de um arquivo aberto.
2. O usuário seleciona um trecho de onda com o *mouse*.
3. O usuário seleciona a entrada *Fade Out* no menu *Effect*.
4. O sistema multiplica o valor de uma função de *Fade Out* pelo valor das amostras e exibe em tela o resultado.

Efeito *Cross Over*

1. A pré-condição é a existência de um arquivo aberto.
2. O usuário seleciona um trecho de onda com o *mouse*.
3. O usuário seleciona a entrada *Cross Over* no menu *Effect*.
4. O sistema multiplica o valor de uma função de *Cross Over* pelo valor das amostras e exibe em tela o resultado.

Efeito *to Stereo*

1. A pré-condição é a existência de um arquivo mono aberto.
2. O usuário seleciona a entrada *to Stereo* no menu *Effect*.
3. O sistema transforma o arquivo em Estéreo.

Efeito *to Mono*

1. A pré-condição é a existência de um arquivo estéreo aberto.
2. O usuário seleciona a entrada *to Mono* no menu *Effect*.
3. O sistema transforma o arquivo em Mono.

III.3 Diagrama de Classes

Outra forma de exposição de partes de projetos bastante conhecida é a visão dada pela utilização do diagrama de classes. Em um diagrama de classes descrevemos os tipos de objetos utilizados no sistema e os vários tipos de relacionamentos estáticos que existem entre eles. Os tipos principais de relacionamentos estáticos são associações e subtipos.

Na chamadas associações dizemos quais tipos de objetos se ligam a objetos que os ajudam a descrever; por exemplo, um arquivo de música pode ter associados a ele inúmeros instrumentos. Já nos subtipos dizemos, por exemplo, quais instrumentos são instrumentos de corda. Os exemplos abordados nesse ponto são meramente ilustrativos.

Na figura III.2 tratamos de maneira genérica as diversas classes e seus relacionamentos. Vamos explicitar as diversas partes das classes `RotateVector` , `walkGraphic` e da `track`, já que nelas se deu grande parte do desenvolvimento.

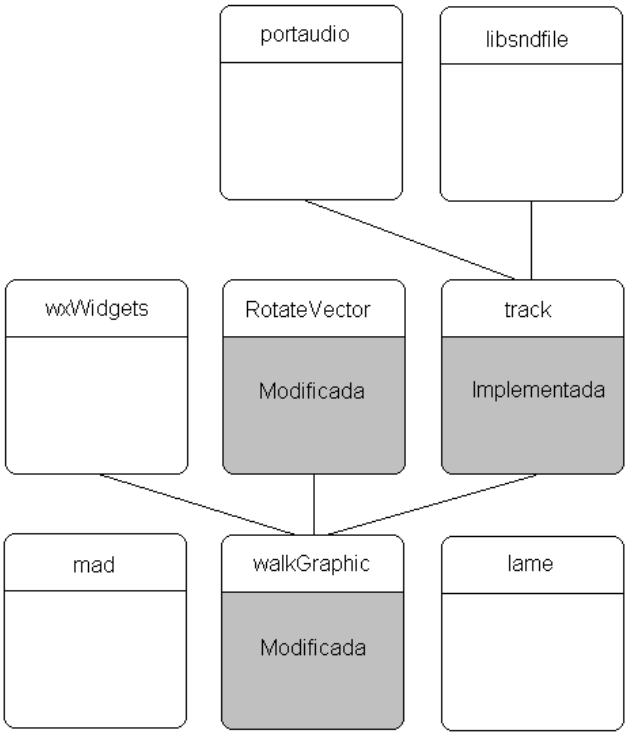

Figura III.2: Diagrama de Classes.

III.3 Diagrama de Classes

A base da manipulação de arquivos é a classe `Class RotateVector` (ver seção IV.9). Ela possibilita que armazenemos vetores do tipo `float` que representam o nosso sinal de áudio. Ela possui métodos que nos dão a liberdade de manipulação dos valores de diversas formas, e ainda possui atributos que facilitam a tarefa de exibir em tela. Além disso ela armazena o número total de valores e assim por diante.

A classe `Class WalkGraphic` foi construída baseada na `Class RotateVector`. Ela é a responsável por exibir em tela os gráficos que são correspondentes aos valores armazenados na classe `Class RotateVector`. Ela possui métodos para exibir seleção de trechos, para o desenho de escala, para a mudança de escala, para selecionar as cores em que o gráfico será exibido, dentre outras.

As duas classes citadas anteriormente, `Class RotateVector` e `Class WalkGraphic`, foram modificadas para que pudessem satisfazer as especificações do projeto atual. Rotinas para realizar seleção de trechos, para realçar as cores das ondas e outros métodos tiveram que ser implementados para adaptar as classes originais às novas necessidades encontradas.

A classe `Class WalkGraphic` é o ponto de ligação entre a `Class RotateVector`, responsável pelo armazenamento dos valores dos arquivos de áudio, e a biblioteca *wxWidgets*. Esta última é de fato quem possui os métodos para a exibição gráfica. Esta classe ainda herda atributos e métodos de outra classe, a classe `track`, que será abordada abaixo.

A classe `track`, que foi implementada por inteiro, encapsula os atributos de outras duas bibliotecas, *libsndfile* e *portaudio*. Dessa forma a classe armazena uma estrutura na qual os dados do arquivo de áudio ficam guardados. Tais atributos são indicadores de marcação de pontos no ambiente gráfico (que são utilizados para a seleção de trechos), ponteiro para o arquivo de áudio, indicação de tamanho de área utilizada para exibição, dentre outros.

A classe *lame* é utilizada para implementar o algoritmo de compressão (ou codificação) de arquivos de áudio do tipo *raw*, no formato *wave*, para arquivos comprimidos no formato *MP3*. A classe *mad* por sua vez é utilizada para realizar a descompressão de arquivos no

formato *MP3* para arquivos *raw*.

A atuação destas duas bibliotecas se dá por meio de funções que não foram inseridas em classes, mas sim tratadas como módulos a parte. O mesmo ocorre com a reprodução de áudio que utiliza a biblioteca *portaudio*. A utilização desta biblioteca não se restringiu à reprodução e por isso teve de ser inserida no contexto de herança da classe *track*.

Capítulo IV

Implementação

Neste capítulo serão abordados os principais pontos da implementação do projeto. A forma como um programa *wxWidgets* é escrito obedece a algumas normas e deve seguir um procedimento padrão de utilização. Seguindo o mesmo raciocínio as chamadas às funções da biblioteca *libSndFile* também têm uma ordem a ser seguida. Paralelamente alguns passos devem ser seguidos para a execução de um arquivo de áudio através da biblioteca *portaudio* e também para utilizar a biblioteca *lame*.

As figuras IV.1 e IV.2 nos ajudam a visualizar quais funções são chamadas dentro de cada rotina. As funções serão abordadas neste capítulo nas próximas seções.

IV.1 Introduzindo a *wxWidgets*

A biblioteca *wxWidgets* [6] foi desenvolvida em C++ e tem como objetivo implementar a interface gráfica de programas, ou seja, é a biblioteca responsável pelo desenvolvimento da GUI (*General User Interface*) dentro do corrente projeto. Uma vantagem desta biblioteca é que ela é multiplataforma, isto é, com mínima ou nenhuma alteração no código somos capazes de levar o aplicativo de um sistema operacional a outro. A implementação final foi testada tanto no sistema operacional *Windows* quanto no *Linux*. Testes foram realizados no *Conectiva Linux* e no *Fedora Core*.

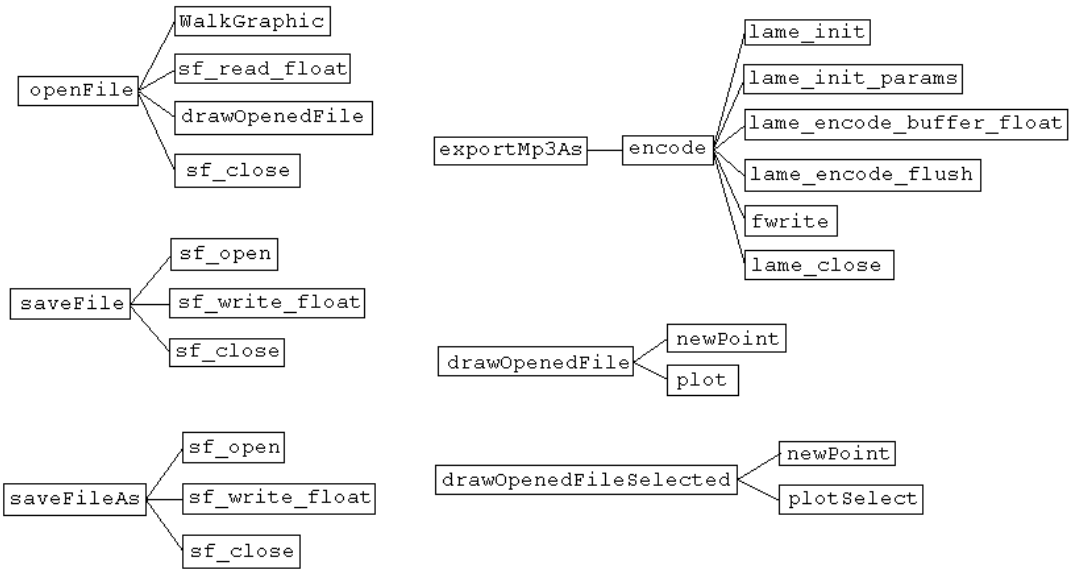

Figura IV.1: Funções e chamadas as funções.

Todo programa utilizando a *wxWidgets* precisa da declaração de um objeto que seja herdeiro da classe *wxApp* e que sobrescreva o método *OnInit*. A classe *wxApp* representa a aplicação em si, ativando algumas propriedades que devem estar relacionadas ao programa e implementando o sistema de janelas. É neste objeto que declaramos um *wxFrame*, que é criado na *wxApp::OnInit*.

Como exemplo, imaginemos que um programa em *wxWidgets* é um quadro. Utilizamos um objeto da classe *wxFrame* como moldura e um objeto da classe *wxScrolledWindow* como painel para a pintura. Esse é um tipo de painel especial, que permite que você pinte em uma área maior que a moldura, de forma que, movimentando o painel, a parte visível se enquadre à moldura.

Existem diversas formas de realizarmos as pinturas. Uma delas pode ser simplesmente colar um recorte no painel. Isso se dá através de um *Device Context*, representado pela classe *wxDC*, que é inserido num objeto da classe *wxScrolledWindow*. Por sua vez, um *Device Context* recebe uma espécie de quadro de *bitmap* representado pela classe *wxBitmap*.

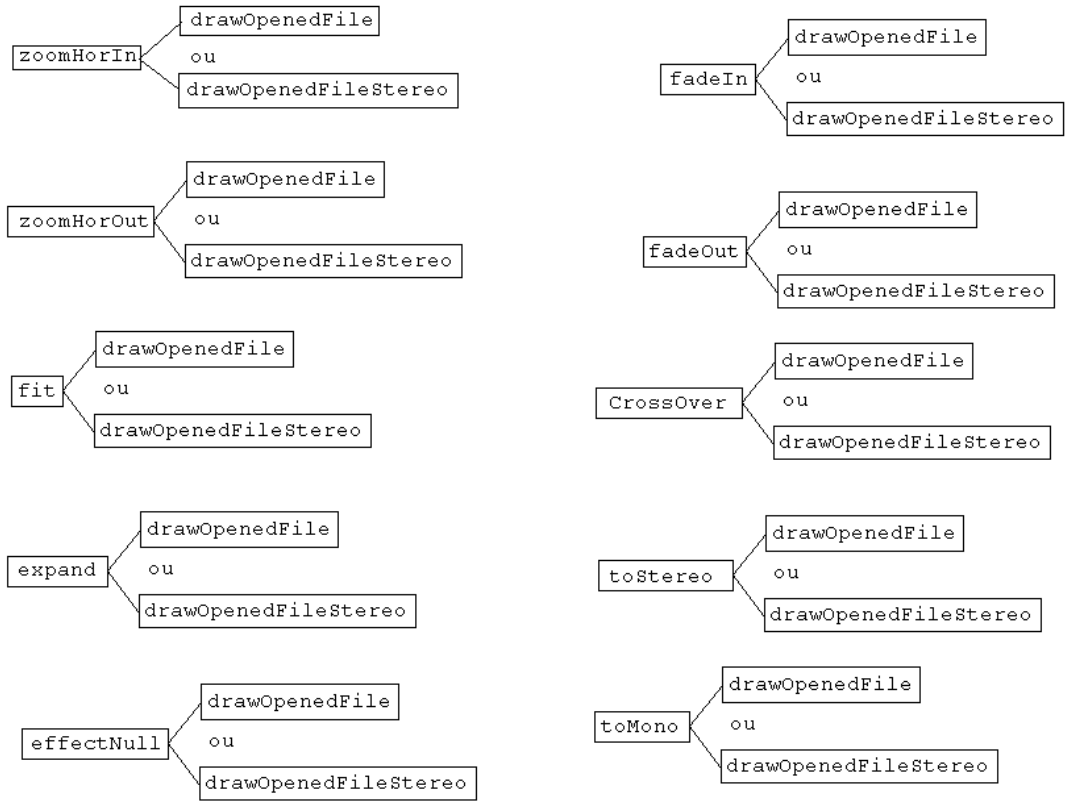

Figura IV.2: Funções e chamadas as funções.

Dessa forma, a *wxFrame* é quem de fato representa o papel da janela utilizada na aplicação. É ela, como dito anteriormente, a responsável pela moldura na qual inseriremos a representação da pintura. Esta classe ainda é responsável pelo tratamento de eventos. Os eventos são acontecimentos tais como a ativação de um botão, o deslocamento da moldura, a captação do movimento do *mouse*, etc.

A *wxWidgets* trabalha de forma a conectar esses acontecimentos ao processamento de funções ou trechos de código. Um exemplo bastante razoável para citarmos nesse momento é a abertura de um arquivo. Quando o usuário abre o *menu* de arquivos, e escolhe a opção de abrir arquivo, é exibida na tela uma caixa onde ele pode escolher o arquivo. Esse é o exemplo de uma interação que corresponde a um evento gerado pelo usuário, um processamento, ou seja, a abertura da caixa para a escolha do arquivo.

O esquema geral do tratamento dos eventos segue um padrão. A cada função exibida no *menu* corresponde uma função correspondente no *MyFrame*. Essa função por sua vez realiza o controle do *frame*, como habilitar ou desabilitar botões, abrir caixas de diálogo, entre outros.

Depois desse processamento inicial, a função do *menu* chama uma função que foi especificada na *MyCanvas* que é nosso painel. Deve-se ressaltar que o objeto *MyCanvas* é herdeiro do objeto *wxScrolledWindow*. Em *MyCanvas* estão implementadas as rotinas que desenham a tela.

IV.2 Introduzindo a *libsndfile*

A biblioteca *libsndfile*[7] foi escrita em C com o objetivo de ler e escrever arquivos de áudio com diversos formatos tais como por exemplo *WAVE*. A biblioteca está disponível para os ambientes *Linux* e *Windows*, que são os sistemas operacionais que devem ser atendidos por este projeto.

No projeto a biblioteca *libsndfile* foi utilizada de duas formas: para a leitura de arquivos para o processamento e para a gravação do resultado dos arquivos processados. As funções da biblioteca utilizadas para isso são respectivamente a *sf_ read_ float* e a *sf_ write_ float*. Antes da utilização de cada uma dessas funções devemos abrir o arquivo, e para isso utilizamos a função *sf_ open*. Após a leitura ou escrita devemos fechar o arquivo com a função *sf_ close*.

A função *sf_ open* tem como parâmetros o caminho onde se encontra o arquivo, o modo de abertura e um ponteiro para uma estrutura padrão da biblioteca que é preenchida quando ocorre o sucesso da abertura do arquivo. A função ainda devolve um ponteiro para o tipo SNDFILE que é utilizada por outras funções. O protótipo da função é:

```
SNDFILE* sf_open (const char *path, int mode, SF_INFO *sfinfo)
```

O modo de abertura do arquivo pode ser:

- SFM_READ - somente leitura
- SFM_WRITE - somente escrita
- SFM_RDWR - leitura e escrita

A biblioteca utiliza a estrutura abaixo para devolver os parâmetros do cabeçalho do arquivo de áudio:

```
typedef struct
      {    sf_count_t  frames ;     /* Used to be called samples. */
           int         samplerate ;
           int         channels ;
           int         format ;
           int         sections ;
           int         seekable ;
      } SF_INFO ;
```

Depois de abrir um arquivo, o programa pode ler a partir ou escrever no mesmo. As funções responsáveis por isso serão analisadas a seguir.

A função *sf_read_float* tem como parâmetros um ponteiro para um descritor do tipo *SNDFILE*. Este descritor possui as informações a respeito do arquivo que foi anteriormente aberto. Possui um ponteiro para o tipo *float* onde serão armazenadas as amostras lidas e um tipo *sf_count_t*, que é um inteiro de 64 bits, que representa o número de amostras que devem ser lidas. A função devolve um outro valor do mesmo tipo (*sf_count_t*). Quando a função obtém sucesso ela retorna como valor, o mesmo número que havia sido passado como parâmetro. Parâmetro esse que indicou número de ítens que deveriam ser lidos.

O protótipo da função é:

```
sf_count_t sf_read_float (SNDFILE *sndfile, float *ptr, sf_count_t items).
```

A outra possibilidade que temos é escrever em um arquivo. Para isso utilizamos a função *sf_ write_ float*. Essa função recebe como parâmetros um ponteiro para *SNDFILE* que é o arquivo que foi aberto e no qual serão escritas as amostras, um ponteiro para *float* que possui as amostras que serão escritas e um tipo *sf_ count_ t* que define o número de amostras que serão escritas no arquivo. Quando a função obtém sucesso ela devolve o mesmo valor que foi passado como parâmetro e que corresponde ao número de ítens escritos.

O protótipo da função é:

```
sf_count_t sf_write_float(SNDFILE *sndfile, float *ptr, sf_count_t items).
```

Por fim, o arquivo que foi aberto ou para leitura ou para escrita precisa ser fechado, para isso utilizamos a função *sf_ close* que tem o protótipo:

```
int sf_close (SNDFILE *sndfile)
```

Essa função recebe somente o ponteiro para o descritor de arquivo do tipo *SNDFILE*.

IV.3 Introduzindo a *portaudio*

A biblioteca *portaudio* [8] tem como objetivo facilitar a entrada e a saída de sinais de áudio através da placa de som. Neste projeto utilizamos a parte responsável pela escrita das amostras lidas de um arquivo na placa de som para a reprodução de áudio.

Assim como as demais bibliotecas utilizadas neste projeto, essa biblioteca também dispõe de uma rotina de chamada de suas funções para que possa ser atingindo um determinado objetivo. Dois passos básicos do processo são a inicialização da biblioteca e a abertura de um *stream* de áudio para entrada ou saída.

Depois devemos ter uma função de *callback* que será chamada repetidamente pela biblioteca para reproduzir o áudio. A função de *callback* lê as amostras do *buffer* de entrada e escreve no *buffer* de saída. Para encerrarmos a reprodução a função de *callback*

deve retornar o valor 1 para indicar que não será chamada de novo. Quando quisermos encerrar a reprodução devemos fechar o *stream* e executar a função responsável por liberar as variáveis utilizadas internamente pela biblioteca.

De acordo com os passos anteriores, devemos chamar *Pa_ Initialize*. Essa função busca os dispositivos de reprodução e captação de áudio presentes no sistema. Ela retorna um código de erro em casos de fracasso e *PaNoError* em caso de sucesso.

A abertura de um *stream* se dá através da chamada da função *Pa_ OpenStream*. Essa função possui inúmeros parâmetros, que são:

- *stream*

  Esse é o endereço do ponteiro que irá receber o ponteiro do novo *stream*.

- *inputDevice*

  Esse campo representa o identificador do dispositivo de entrada. Se não utilizar nenhum dispositivo de entrada utiliza-se *paNoDevice*.

- *numInputChannels*

  É o número de canais de entrada. Se o *PaDeviceID* for *paNoDevice*, como é nosso caso, esse valor é ignorado.

- *inputSampleFormat*

  Identifica o formato do *buffer* de entrada que é disponibilizado pela função de *call-back*.

- *inputDriverInfo*

  Esse é o ponteiro para uma estrutura que suporta algum *driver* específico de entrada (opcional). Em nosso caso é nula.

- *outputDevice*

  Esse campo representa o identificador do dispositivo de saída, no qual é reproduzido o áudio. Utilizamos a saída padrão.

- *numOutputChannels*

  Número de canais que a função de *callback* precisa fornecer para reprodução.

- *outputSampleFormat*

  Identifica o formato do *buffer* de saída que é disponibilizado pela função de *callback*.

- *outputDriverInfo*

  Esse é o ponteiro para uma estrutura que suporta algum *driver* específico de saída (opcional). Em nosso caso é nula.

- *sampleRate*

  Taxa de amostragem para entrada e saída.

- *framesPerBuffer*

  Esse campo representa o comprimento dos *buffers* que passam as amostras para a saída de áudio.

- *numberOfBuffers*

  É o número de *buffers* utilizados para comunicação na saída.

- *streamFlags*

  Controlam o comportamente do processo de *streaming*.

- *callback*

  É o ponteiro para a função implementada pelo usuário para o processo de reprodução do som.

- *userData*

  É o ponteiro de dados que é passado para a função de *callback*.

O protótipo da função *Pa_ OpenStream* é o seguinte:

```
PaError Pa_OpenStream( PortAudioStream** stream,
PaDeviceID inputDevice,
int numInputChannels,
PaSampleFormat inputSampleFormat,
void *inputDriverInfo,
PaDeviceID outputDevice,
int numOutputChannels,
PaSampleFormat outputSampleFormat,
void *outputDriverInfo,
double sampleRate,
unsigned long framesPerBuffer,
unsigned long numberOfBuffers,
PaStreamFlags streamFlags,
PortAudioCallback *callback,
void *userData );
```

Após a abertura do stream, precisamos iniciar sua reprodução, isso se dá pelo método *Pa_StartStream*, cujo protótipo é:

```
PaError Pa_StartStream( PortAudioStream *stream )
```

Essa função apenas indica o início da comunicação e reprodução de áudio. Para garantir que os dados não sejam corrompidos, é necessário que não os deixemos ser acessados. Isso é feito utilizando a função *Pa_Sleep*, cujo protótipo é:

```
void Pa_Sleep( long msec )
```

O tempo que passamos como parâmetro para a função *Pa_Sleep* é justamente o tempo necessário para que a reprodução do som aconteça, e portanto deve ser calculada com exatidão.

Após a reprodução devemos fechar o stream através da função *Pa_ CloseStream* e por fim liberar as variáveis utilizadas pela biblioteca através da função *Pa_ Terminate*. Os protótipos da função são respectivamente,
`PaError Pa_CloseStream( PortAudioStream* )` e
`PaError Pa_Terminate( void )`.

IV.4 Introduzindo a *lame*

A biblioteca *lame* [9] implementa as rotinas de codificação e decodificação entre um arquivo do tipo *WAVE* e um do tipo *MP3*. Como as demais bibliotecas já abordadas no projeto, esta também precisa seguir uma ordem de chamada de funções tanto para a codificação quanto para a decodificação.

Para realizar a codificação devemos em primeiro lugar iniciar a biblioteca através da função `lame_global_flags * CDECL lame_init(void)`. Esta função retorna uma variável do tipo *lame_ global_ flags* que na realidade é uma estrutura *lame_ global_ struct* que guarda parâmetros do arquivo aberto ou que ainda será aberto e que serão utilizados para codificação ou decodificação.

O padrão adotado pela biblioteca e que fica registrado na variável *lame_ global_ flags* é um arquivo estéreo, de 44.1KHz, com codificação CBR (*constant bitrate*) realizada a 128kbps.

Depois a função `int CDECL lame_init_params(lame_global_flags * const)` é chamada. Essa função realiza configurações baseadas nos dados anteriormente especificados, e ainda procura possíveis problemas ao iniciar a biblioteca.

Depois de iniciada a biblioteca, chega a hora de codificar os dados através da função *lame_ encode_ buffer_ float*, cujo protótipo segue:

```
int CDECL lame_encode_buffer_float(
    lame_global_flags*  gfp,
```

```
const float     buffer_l [],
const float     buffer_r [],
const int           nsamples,
unsigned char*      mp3buf,
const int           mp3buf_size );
```

Os parâmetros que a função recebe são:

- *lame_global_flags* gfp*

  Uma estrutura que armazena as características do arquivo que será criado.

- *const float buffer_l []*

  Dados a serem codificados relativos ao canal esquerdo.

- *const float buffer_r []*

  Dados a serem codificados relativos ao canal direito.

- *const int nsamples*

  Número de amostras por canal.

- *unsigned char* mp3buf*

  Ponteiro associado à variável que irá armazenar o resultado, ou seja, os dados codificados.

- *const int mp3buf_size*

  Número de octetos válidos no *stream*.

Depois da codificação é necessário que os dados finais que porventura não tenham sido alocados na variável de retorno sejam descarregados através da função *lame_encode_flush*:

```
int CDECL lame_encode_flush(
        lame_global_flags *  gfp,
        unsigned char*       mp3buf,
        int                  size);
```

Os parâmetros seguem a mesma descrição da função anterior. Para finalizar o processo de codificação, precisamos liberar as variáveis utilizadas pela biblioteca através da função `int CDECL lame_close (lame_global_flags *)`.

IV.5 Introduzindo a *mad*

A biblioteca *mad* [10] foi escrita em C com o intuito de realizar a operação de descompressão de arquivos *MP3* para arquivos *raw*. Apesar da biblioteca *Lame* nos fornecer uma funcionalidade para este fim, a biblioteca *mad* tem mais recursos e é direcionada para este propósito, tornando a tarefa mais simples. Assim como a biblioteca *portaudio*, a biblioteca *mad* funciona através de *callbacks*.

A função principal, que realiza o trabalho de decodificação espera receber o ponteiro para três funções. São elas uma função de entrada de dados, uma de retorno (saída) de dados e uma função de erro, responsável por tratar exceções e erros.

A implementação dessas funções deve ser feita pelo usuário da biblioteca. Além de implementar as três funções, este também, deve passar uma estrutura própria para manter controle do processo de decodificação. A chamada de funções segue uma ordem específica.

O comportamento da biblioteca segue a seguinte lógica: Através da função de entrada, dados para descompressão são armazenados, quando uma quantidade suficiente de dados é armazenada a biblioteca chama a função de cabeçalho, decodifica os dados, chama a função de filtro, chama a função de saída e finalmente retorna para a função de entrada. Esse laço é realizado até que acabe o arquivo de entrada.

A biblioteca é iniciada através da função *mad_ decoder_ init* cujo protótipo é visto a

seguir:

```
void mad_decoder_init(
struct mad_decoder *,
void *,
enum mad_flow (*)(void *, struct mad_stream *),
enum mad_flow (*)(void *, struct mad_header const *),
enum mad_flow (*)(void *, struct mad_stream const *, struct mad_frame *),
enum mad_flow (*)(void *, struct mad_header const *, struct mad_pcm *),
enum mad_flow (*)(void *, struct mad_stream *, struct mad_frame *),
enum mad_flow (*)(void *,
void *,
unsigned int *));
```

Os parâmetros que a função recebe são:

- *struct mad_ decoder **

  Um ponteiro para uma estrutura *mad_ decoder* que servirá para armazenar informações sobre a *stream* decodificada, como as funções de entrada e saída, função de erro, dentre outras.

- *void **

  Esse parâmetro recebe uma estrutura de dados particular, onde as informações de controle do processo podem ser armazenadas.

- *enum mad_flow (*)(void *, struct mad_ stream *)*

  Esse parâmetro recebe a função de entrada responsável por fornecer as amostras em formato *MP3* a serem decodificadas.

- *enum mad_flow (*)(void *, struct mad_ header const *)*

  Esse parâmetro recebe a função responsável por obter dados de cabeçalho a partir do arquivo.

- *enum mad_flow (*)(void *, struct mad_stream const *, struct mad_frame *)*

  Esse parâmetro recebe a função responsável por filtrar os dados antes de devolvê-los para a função de saída.

- *enum mad_flow (*)(void *, struct mad_header const *, struct mad_pcm *)*

  Esse parâmetro recebe a função de saída, que por sua vez recebe os dados descomprimidos. Cabe a essa função realizar o tratamento para os dados fornecidos pela biblioteca.

- *enum mad_flow (*)(void *, struct mad_stream *, struct mad_frame *)*

  Esse parâmetro recebe a função de erro para tratamento de erros e exceções.

- *enum mad_flow (*)(void *, void *, unsigned int *)*

  Esse parâmetro recebe uma função para informar o usuário do processo através de mensagens.

Em seguida a função *mad_decoder_run* é chamada, ela é responsavel pelo ordenamento das chamadas das funções de *callback* e pela descompressão do arquivo. O protótipo da função é:

```
int mad_decoder_run(struct mad_decoder *, enum mad_decoder_mode);
```

Os parâmetros que a função recebe são:

- *struct mad_decoder **

  O primeiro parâmetro é um ponteiro para uma estrutura *mad_decoder* que servirá para armazenar informações sobre a *stream* decodificada, como as funções de entrada e saída, função de erro, dentre outras.

- *enum mad_decoder_mode*

  O modo de decodificação escolhido para realizar o processo.

E finalmente a função *mad_ decoder_ finish* é chamada. Tem como protótipo `mad_decoder_finish(&decoder)` e seu único parâmetro é a estrutura do tipo *mad_ decoder* que é liberada da memória quando ocorre a chamada desta função.

IV.6 Esquema de funcionamento do Editor

O Editor Gráfico de Sinais de Áudio Multiplataformas funciona seguindo principalmente as regras de programação da biblioteca *wxWidgets*, mas além disso ele possui algumas características que precisam ser esclarecidas neste momento.

Após ser aberta uma nova instância do Editor, uma nova aplicação *wxWidgets* é aberta, possuindo portanto todas as variáveis pertinentes à nova instância, além disso alguns outros objetos são associados a ele.

Um objeto da classe *Track* possui uma variável do tipo *SF_ INFO* na qual ficam armazenadas as informações do cabeçalho do arquivo e possui variáveis do tipo *float** que representam as amostras do arquivo de áudio dentro outras variáveis.

Cada abertura de arquivo associa ao objeto instanciado *m_ track* um cabeçalho de arquivo e, um ponteiro para as amostras, que assim podem ser exibidas em tela.

A partir deste ponto o arquivo está preparado para sofrer as possíveis alterações desejadas pelo usuário. As funções de efeito são as principais responsáveis por essas modificações. Após o usuário realizar o comando para submeter o arquivo ou trecho de arquivo à modificação, é gerado um evento que é processado pela biblioteca *wxWidgets* e é executado o trecho de código relativo ao evento em questão.

Além dos efeitos, o usuário pode estar interessado em modificar a forma como a onda está sendo exibida para obter maior precisão em suas ações. Qualquer comando que altere a forma de exibição em tela além de executar uma rotina associada a um evento também ativa a função *OnPaint*.

Após suas modificações o usuário pode querer salvar seu novo arquivo, acionando assim

algum comando relativo ao arquivo, gerando um outro evento associado a algum trecho de código com esse objetivo. Por fim o usuário encerra o aplicativo e todas as variáveis utilizadas pelo mesmo são liberadas da memória.

Toda a ligação entre os eventos e o código executado pela ativação do evento teve de ser implementado. Assim como essa ligação, o próprio código para realizar as funções foi implementado. As funções serão descritas na próxima seção.

IV.7 Principais funções

As funções de nosso programa podem basicamente ser divididas em dois grandes blocos: as funções que executam rotinas disponíveis ao usuário diretamente pelo *menu* que também estão disponíveis por atalhos em determinados casos, e as funções que são utilizadas para a execução de parte de uma ou mais funções. As primeiras são chamadas de funções principais, enquanto que as últimas são chamadas de funções auxiliares.

IV.7.1 *Menu File*

As funções abaixo pertencem ao *menu File*:

- Função `void MyCanvas::openFile(const char *filename)`

  Esta função tem como objetivo abrir o arquivo musical e passar seus parâmetros para um objeto que armazenará os valores de seu cabeçalho e os valores das amostras que representam de fato o arquivo. Recebe como parâmetro o ponteiro para o arquivo que irá abrir.

  A abertura do arquivo através da função *sf_open* , associa ao objeto instanciado *m_track* da classe *Track* o cabeçalho do arquivo de áudio e ainda um ponteiro do tipo SNDFILE que é a forma de comunicação que mantemos com o arquivo de áudio. Se obtivermos sucesso com a abertura, tratamos de ler as amostras do arquivo.

Antes de efetuarmos a leitura do arquivo, alocamos memória para a variável associada ao ponteiro do tipo *float** pertencente ao objeto *m_track* da classe *Track* citada anteriormente. É na variável associada a esse ponteiro que as amostras lidas serão armazenadas. Ainda instanciamos um objeto da classe *WalkGraphic* para armazenar as amostras que serão desenhadas na tela.

A leitura das amostras do arquivo é realizada pela função *sf_read_float*. O procedimento difere no caso de o arquivo ser estéreo ou mono. No caso mono apenas um objeto da classe *WalkGraphic* e apenas um *float** são utilizados. No caso estéreo temos dois de cada. Ainda no caso estéreo as amostras devem ser divididas entre duas variáveis de armazenamento.

No final do procedimento o arquivo é fechado. A opção por trabalhar com o arquivo em memória se deve à necessidade de velocidade de processamento.

O trecho de código abaixo mostra o fluxo de chamada às funções:

```
...
// Abertura de arquivo e recuperação de cabeçalho e amostras
if ((m_track->infile = sf_open(filename, SFM_READ, &m_track->trackInfo)) == NULL ){
...
// Criação de uma nova instância do objeto que estará associado às amostras lidas
m_walkGraphic = new WalkGraphic( m_track->trackInfo.frames );
...
// Exibição em tela da forma de onda de acordo com as amostras
drawOpenedFileStereo(m_track->trackFptrLeft,m_track->trackFptrRight, m_track->trackInfo.frames);
...
sf_close(m_track->infile);   // Fechamento do arquivo
```

- Função `void MyCanvas::saveFile()`

  O objetivo desta função é salvar as alterações realizadas no arquivo. Como as alterações são salvas no mesmo arquivo, não recebe parâmetro algum.

  A função identifica o sistema em que está trabalhando e depois apaga o arquivo atual. O próximo passo é abrir um arquivo de nome idêntico ao apagado para escrita através da função *sf_open*. A diferença de chamada desta função para a anterior, é que nessa passamos como parâmetro para *sf_open SFM_WRITE* em oposição a *SFM_READ*.

A função utilizada para a escrita no arquivo é *sf_ write_ float*. Existe uma diferença quando estamos salvando o arquivo estéreo. Um arquivo estéreo tem as amostras intercaladas. Precisamos realizar o processo inverso ao realizado na função de abertura de arquivo.

Ao final do processo fechamos o arquivo e liberamos a memória do ponteiro que fora usado como *buffer* temporário.

O trecho de código abaixo mostra a ordem em que as funções são chamadas:

```
...
if ((outfile  = sf_open(filename, SFM_WRITE, &info)) == NULL ){     // Abertura de arquivo
...
writeCnt = sf_write_float(outfile, m_track->trackFptr, readcntf);   // Escrita em arquivo (gravação)
sf_close(outfile);   // Fechamento do arquivo
```

- Função `void MyCanvas::saveFileAs(const char *filename)`

  Essa é a função responsável por salvar o conteúdo do arquivo em um arquivo novo, ou no mesmo. Para isso recebe como parâmetro o nome do arquivo que será salvo.

  Essa função tem o procedimento semelhante ao da função anterior. O diferencial dela é que a abertura do arquivo para escrita é feita com o novo nome que desejamos que o arquivo possua.

- Função `void MyCanvas::exportMp3As(const char *filename)`

  Essa é a função responsável por codificar o conteúdo de um arquivo PCM comum para um do tipo *Mp3*, salvar o resultado em um arquivo novo, ou no mesmo. Para isso recebe como parâmetro o nome do arquivo que será salvo.

  Essa função tem o procedimento semelhante ao da função anterior. O diferencial dela é que antes de salvar o arquivo, ele é codificado. Um trecho do código da função de codificação pode ser visto abaixo:

```
...
gfp = lame_init(); // Iniciando a biblioteca
imp3 = lame_init_params(gfp);     // Configurações baseadas nos parâmetros da função anterior
```

```
...
// Codificando alguns dados
imp3 = lame_encode_buffer_float(gfp, left, right, info.frames, mp3buffer, mp3buffer_size);
ret_code = lame_encode_flush(gfp, mp3buffer, mp3buffer_size); // Retornando os restante dos dados
...
writecnt = fwrite(mp3buffer, 1, imp3, outfile);   // Escrevendo os dados convertidos em arquivo
...
lame_close(gfp);                                  // Fechando a biblioteca
```

- Função `void MyFrame::OnClose()`

  Apesar de pertencer ao *menu File* que até então só possui funções da classe *MyFrame* que chamavam apenas funções da classe *MyCanvas*, a função *OnClose* é de fato implementada pela classe *MyFrame*. Ela é responsável por liberar as variáveis de controle da aplicação.

  O arquivo sempre é fechado pela própria função que o abre, para que não seja perdido esse controle, cabe a essa função habilitar ou desabilitar comandos do menu e ainda chamar a função que limpa a tela (*MyCanvas::clearArea()*).

IV.7.2 *Menu Edit*

As funções abaixo pertencem ao *menu Edit*.

- `void MyCanvas::copy()`

  O objetivo desta função é copiar para um *buffer* o valor das amostras que foram previamente marcadas em tela. Não possui parâmetros de entrada.

  A função faz uso das marcações realizadas previamente pelas funções que detectam o movimento do *mouse*, são elas as `OnMouseButtonDown` e `OnMouseButtonUp`. Essas marcações ficam armazenadas nas variáveis `int m_startPoint`, `int m_endPoint`, `float m_startSample` e `float m_endSample`.

  Os pontos que indicam início e final de trecho são os dois últimos, dessa forma esses valores são utilizados como limites para o *loop* no qual são copiados para memória

os valores dentro da seleção. No caso do arquivo estéreo, é necessário copiar as amostras de ambos os canais.

O *buffer* temporário é implementado por meio de um arquivo auxiliar onde as amostras selecionadas ficam armazenadas e prontas para serem transferidas ou copiadas.

- `void MyCanvas::paste()`

  Esta função é responsável por copiar para o *buffer* que é exibido em tela os valores das amostras que foram previamente colocadas na memória pelo processo de *copy* ou *cut*.

  Essa rotina tem função complementar a anterior. Ela é responsável por recuperar as amostras do arquivo que fora criado com as amostras selecionadas. A função insere as amostras copiadas em memória entre as amostras limite do ponto de marcação previamente feito pelo *mouse*.

  Para inserir as amostras nesse ponto, um novo *buffer* recebe as amostras até o ponto de corte. Em seguida são inseridas nesse *buffer* temporário as amostras que estão em memória e por fim as amostras restantes do sinal original, no final a variável associada ao ponteiro que contém as amostas da classe *Track* é redimensionada e o *buffer* temporário é copiado para lá.

- `void MyCanvas::cut()`

  Essa é a função responsável por copiar para a memória as amostras dentro do trecho marcado e retirá-las do buffer que está em exibição. Ela portanto é bastante parecida com a função copy.

  Ela primeiro copia para o *buffer* de memória as amostras que estão delimitadas pela marcação. A seguir é feita uma cópia de todas as amostras do *buffer* que estão sendo exibidos na tela, com exceção das amostras retiradas. Por fim, a variável associada ao ponteiro para guardar as amostras da classe *Track* é redimensionada e preenchida com os novos valores.

IV.7.3 *Menu View*

As funções abaixo pertencem ao *menu View*.

- `void MyCanvas::ams()`

  Essa função é acessível pela entrada Exibition → Sample do *menu View*. Ela apenas força o valor da variável `bool showAms` como o oposto do valor de inicialização que é *false*. Dessa forma, as funções que são responsáveis pela detecção da movimentação do *mouse* exibem o valor da amostra correspondente àquela abcissa e ainda exibem o ponto equivalente do dispositivo em questão.

  Para que os valores, que são exibidos na barra de *status*, deixarem de ser mostrados, basta novamente escolher a entrada Exibição → Amostra e assim o valor da variável é forçado para o oposto do que fora escolhido previamente.

- `void MyCanvas::temp()`

  Essa função ativa pela entrada Exibition → Time do *menu* em questão. Ela exibe o tempo total do arquivo na barra de *status*. Além disso, essa função exibe o tempo correspondente ao local por onde passa o mouse, isso é feito forçando o valor da variável `bool showTemp` para o oposto do valor inicial.

  Da mesma forma que a função anterior, as funções que detectam a movimentação do *mouse* fazem uso dessa variável para a exibição ou não desse valor.

- `void MyCanvas::zoomHorIn()`

  O objetivo desta função é permitir uma visão detalhada do arquivo ao longo do tempo, permitindo, portanto, o detalhamento horizontal. O valor de aumento desta visualização é pré-fixado. Para um ajuste específico deve-se utilizar a função *fit*.

  A rotina utiliza os membros *ini_width* e *zoomFactor* da classe *Track* instanciada para realizar a modificação na exibição. Como citado anteriormente, um arquivo ao ser aberto armazena o valor da janela *wxWidgets* para servir como parâmetro de tamanho de exibição do arquivo em tela. Quando uma função recebe um *zoom*,

esse valor é modificado e assim o que ocorre é que a área disponível para desenho aumenta ou diminui.

No caso da função de *zoom Horizontal In* a área é aumentada, e portanto o distanciamento entre os pontos, um novo valor é atribuído *ini_width*, no caso é acrescentado 10% ao campo de visão, e este acréscimo é contabilizado no membro *zoomFactor*. Após esse procedimento é chamada a função *drawOpenedFile* ou *drawOpenedFileStereo* que redesenham a onda levando em consideração esse novo valor de *ini_width*.

A função ainda desabilita as funções de colar (*paste*) para que o usuário seja obrigado a fazer uma nova seleção na escala natural, caso seja sua vontade editar o arquivo nesta nova visualização. O trecho de código abaixo exibe as principais partes da função descrita:

```
...
m_track->ini_width = m_track->ini_width*1.1;       // Incrementando em 10% o tamanho
m_track->zoomFactor = m_track->zoomFactor + 1; // Medida de alteração do tamanho original
...
// Redesenhando a forma de onda
drawOpenedFileStereo(m_track->trackFptrLeft,m_track->trackFptrRight,m_track->trackInfo.frames);
...
SetVirtualSize(m_track->ini_width, 0 );    // Redimensionando a figura por trás da moldura
```

- `void MyCanvas::zoomHorOut()`

  O objetivo desta função é permitir uma visão menos detalhada do arquivo, ou de outro ponto de vista, uma visão mais ampla do arquivo. Ela é executada quando da seleção da entrada *Zoom Horizontal Out*.

  De forma análoga à função anterior, essa função diminui em 10% a área de exibição, utilizando-se para isso da diminuição do campo *ini_width* da classe *Track*. Ainda contabiliza esta diminuição no membro *zoomFactor* da mesma classe.

  Após a mudança no valor das variáveis, a tela é redesenhada e os comandos de colar (*paste*) são desabilitados.

- `void MyCanvas::fit()`

  Essa função redimensiona o tamanho da região de exibição em tela. Para isso ela recupera o valor da área dentro da janela pertencente ao programa e atribui esse valor ao campo *ini_ width* da classe *Track*. Além disso ela reajusta o *zoom* através da atribuição do valor unitário ao campo *zoomFactor* da mesma classe.

- `void MyCanvas::expand()`

  De modo similar à anterior, essa função redimensiona o tamanho da região de exibição em tela, só que no intuito de dar prioridade a exibição de um trecho em particular. Para isso ela recupera o valor da área dentro da janela pertencente ao programa, calcula a parcela de tela que representa o número de amostras no trecho, realiza um cálculo de proporção e atribui esse novo valor ao campo *ini_ width* da classe *Track*. Além disso ela reajusta o *zoom* através da atribuição do valor unitário ao campo *zoomFactor* da mesma classe.

- `void MyCanvas::maxAmp()`

  Essa função mostra ao usuário o maior valor de amplitude encontrado nas amostras do arquivo. Se existir um trecho marcado no arquivo, somente aquele trecho é examinado no intuito de buscar o maior valor. Esse valor é exibido por uma janela de diálogo.

IV.7.4 *Menu Effect*

- `void MyCanvas::effectNull()`

  O efeito de zerar as amostras selecionadas é obtida por esta função. Ela faz uso dos limites de seleção previamente armazenados pelas funções que captam a movimentação do *mouse* e para cada amostra dentro desse limite, seu valor é estabelecido como zero, após modificar as amostras, a tela é redesenhada.

  O trecho de código a seguir mostra como as amostras são zeradas:

```
...
for (int i = start; i < end; i++) m_track->trackFptr[i] = 0;
// Redesenhando em tela
drawOpenedFileSelect(m_track->trackFptr,m_track->trackInfo.frames,
 m_track->m_startPoint, m_track->m_endPoint);
...
```

- `void MyCanvas::effectReverseNull()`

  Funciona de forma semelhante à função anterior, porém as amostras que têm seu valor transformados em zero são as que ficam fora dos limites de seleção.

- `void MyCanvas::gain()`

  Essa função exibe uma caixa de diálogos na qual o usuário entra com um valor do tipo *float*. O trecho marcado então é multiplicado por esse valor. Se nenhum trecho estiver marcado, o arquivo inteiro é multiplicado pelo fator de ganho.

- `void MyCanvas::fadeIn()`

  Essa função, usualmente utilizada no início de gravações, tem como objetivo o aumento gradativo do valor das amostras, até que seu nível atinja o nível das demais amostras.

  Para realizar esse aumento, a função primeiramente busca o maior valor e o menor valor encontrados entre as amostras com o objetivo de torná-los parâmetro de uma função reta crescente que será multiplicada pelo trecho de arquivo marcado.

  Depois disso, a função original é multiplicada ponto a ponto pela nova função reta, o resultado da multiplicação é uma forma de onda que guarda os aspectos freqüênciais das amostras (o que faz com que a amplitude da envoltória do sinal aumente).

- `void MyCanvas::fadeOut()`

  Essa função tem funcionamente bastante semelhante à função anterior. Porém é normalmente utilizada no final dos arquivos por sua característica de diminuir gradativamente o valor das amostras.

Tal como a função anterior, recupera o maior e o menor valor das amostras do arquivo, utilizando-o para estabelecer uma função reta decrescente que será multiplicada pelo trecho marcado do arquivo em questão. Dessa forma as amostras têm seus valores decrescidos gradativamente sem alterar o conteúdo frequencial.

- `void MyCanvas::crossOver()`

  A função é normalmente utilizada para emendar trechos de gravações distintas. Dessa forma ela inicialmente diminui o valor das amostras selecionadas para depois fazer o aumento gradual de seus valores. Dessa forma garantimos uma transição suave e sem *clicks* entre dois trechos que porventura tenham sido colados.

  Ela funciona de forma similar às duas funções anteriores, busca primeiro o maior e o menor valor das amostras no trecho em questão, e então inicia a multiplicação pela função reta crescente na primeira parte do trecho para em seguida aplicar a função reta decrescente. O resultado é uma marcação em forma de X que caracteriza a função (ver figura II.3).

- `void MyCanvas::toStereo()`

  A função cria um arquivo estéreo a partir de um arquivo mono. Para isso as amostras do canal mono são dividas entre os dois canais estéreos, para manter a compatibilidade, o valor de suas amostras é dividido por dois. O cabeçalho do arquivo também é alterado para conter a informação do número pertinente de canais.

- `void MyCanvas::toMono()`

  Tal como a função anterior, esta função cria um arquivo mono a partir de um arquivo estéreo. Para manter a compatibilidade com o arquivo original, o valor das amostras de ambos os canais é somado e dividido por dois. O cabeçalho do arquivo também é alterado para conter a informação sobre o número pertinente de canais.

IV.8 Funções Auxiliares

Essa parte elucida o papel de cada função que é usada na maior parte das funções acima descritas. As funções pertinentes à representação gráfica do sinal de áudio propriamente dito pertencem a Classe WalkGraphic. Essa classe por sua vez, utiliza a classe RotateVector. Iremos abordar ambas as classes para mostrar de que forma as demais funções são implementadas.

- `void MyCanvas::drawOpenedFile(float *fptr, SF_INFO sfinfo)`

  Esta é a função responsável por exibir as amostras em forma de onda. O objetivo dela é desenhar na tela arquivos de só um canal. Ela tem como parâmetros um ponteiro para as amostras (*float**) e ainda recebe uma variável do tipo *SF_INFO*, que é a estrutura que armazena as informações relativas ao cabeçalho do arquivo de áudio.

  A função começa recuperando o quanto a tela foi deslocada pelas *scroll bars*, se ela tiver sido deslocada. Em seguida ela recupera o tamanho da área disponível para desenho do aplicativo e ainda registra esse valor numa variável da instância da classe *Track*.

  A combinação desses dois valores são úteis para redesenhar a tela em caso de um evento *OnPaint*, para isso devemos levar em conta o comprimento do retângulo que representa a faixa de áudio e ainda o eventual deslocamento que porventura essa faixa possa ter sofrido na visualização. É importante notar que num momento inicial, o valor do retângulo onde é exibido a onda representando o arquivo de áudio é escolhido como sendo o próprio tamanho da tela do aplicativo.

  Um novo objeto da classe *WalkGraphic* é instanciado para que novos valores possam ser associados a essa classe. Juntamente com um novo objeto *WalkGraphic*, instanciamos um novo objeto do tipo *wxBitmap* com o tamanho já atualizado pelo processo anteriormente descrito.

  Como dito na seção introdutória da biblioteca *wxWidgets*, o objeto *wxScrolledWin-*

dow realiza o papel da tela onde um recorte, *wxBitmap*, é colado. Na realidade qualquer *Device Context* pode realizar essa função. O trecho de código a seguir mostra essa associação:

```
...
m_buffer = new wxBitmap(m_width,m_height);    // Uma nova instância do objeto
...
m_bufferDC.SelectObject(*m_buffer);    // Selecionando o objeto que desempanhará o papel
                                       // de Device Context
m_bufferDC.Clear();                   // Limpando o Device Context
m_walkGraphic->setDC(&m_bufferDC);    // Passando para o objeto da classe walkGraphic qual
...                                   // o Device Context que ela utilizará
```

No objeto *wxBitmap* serão desenhados os retângulos que representam os canais. Se o arquivo for mono, apenas um retângulo é desenhado, quando o arquivo é estéreo, falamos de dois retângulos. Além dos retângulos, ainda desenhamos nesse *bitmap* a escala e o fundo de tela.

Depois de especificarmos o *bitmap*, fazemos a associação do retângulo responsável por conter a representação da onda de áudio, à classe *WalkGraphic*, que por sua vez é associada à classe *Track*. Até esse ponto temos os objetos que representam o retângulo (*wxRect*) , no qual desenhamos a onda representando o som e temos também o objeto *wxBitmap*, no qual iremos inserir o retângulo e a escala.

Os locais onde são desenhadas as formas de onda são os retangulos citados anteriormente. Depois de desenhar as ondas representando os sinais de áudio, é necessário que o desenho seja associado à algum dispositivo para exibição. O aplicativo em questão utiliza a placa de vídeo para exibir o desenho na tela. Essa função é desempenhada por um *Device Context*, o objeto *wxBitmap*. Ainda precisamos associar esse *Device Context* à classe *WalkGraphic* porque a classe desenha diretamente nele.

Depois de escolhermos em que *Device Context* iremos desenhar, são escolhidas as cores de fundo, de linha de desenho as quais são associadas aos objetos que de fato executam o desenho. Um laço insere cada ponto da amostra na variável responsável

por armazená-los na classe *WalkGraphic*. Depois de terminado o laço, a função *plot* da própria classe é chamada para executar o desenho dos pontos.

Depois disso exibimos em tela o resultado do que fora previamente desenhado em *background*. Para que o resultado seja exibido em tela, o seguinte trecho de código se faz necessário:

```
m_bufferDC.SelectObject( wxNullBitmap );
wxClientDC screen_dc(this);
screen_dc.DrawBitmap(*m_buffer,-xx*10,0);
```

- `void MyCanvas::drawOpenedFileStereo(float *fptrLeft,float *fptrRight, SF_INFO sfinfo)`

  Esta função é responsável por exibir em tela o sinal de áudio quando este for estéreo. Tem como parâmetros de entrada as amostras de ambos os canais, e ainda uma estrutura que armazena os dados a respeito do sinal.

  A função segue uma rotina muito semelhante à função anteriormente descrita, porém como são desenhadas duas formas de onda (uma para representar cada canal), são necessárias duas instâncias da classe *WalkGraphic*, a área de *bitmap* precisa ser instanciada levando-se em consideração que ela abrigará dois retângulos representando cada canal.

  Por sua vez, a cada canal é associado um *wxRect* pertencente à instância da classe *Track*. E por fim, a cada instância da *WalkGraphic* são adicionados pontos que serão exibidos no *Device Context*, que é comum a ambos.

- `void MyCanvas::clearArea()`

  Tal como as funções anteriores, essa função recupera o tamanho da área disponível para desenho, associa um novo bitmap e uma ou duas instâncias da classe *WalkGraphic*, dependendo do número de canais do arquivo. Após associar o novo *bitmap* e a nova instância de *WalkGraphic* ao *Device Context* em questão, a tela apenas desenha o fundo de tela da cor padrão, de forma que a tela esteja limpa para os próximos desenhos.

IV.8 Funções Auxiliares

A próximas funções detectam eventos que são associados a elas. Dessa forma, se o tamanho da tela é redimensionado, se a tela fica em segundo plano e volta a primeiro plano, se o mouse passa por determinada região e alguns outros eventos, essas funções respondem a esses acontecimentos, ora redesenhando o conteúdo pertencente à janela, ora apenas redimensionando ou qualquer outro tratamento necessário para a exibição do aplicativo.

- `void MyCanvas::OnPaint(wxPaintEvent &event)`

  Quando o conteúdo da janela de exibição de um programa em *wxWidgets* precisa ser redesenhado, um evento *wxPaintEvent* é gerado para nos alertar sobre isso. A função *OnPaint* é responsável por realizar essa tarefa.

  O início da função recupera o *Device Context* em questão através da função. A biblioteca *wxWidgets* exige que seja associado um objeto da classe *wxPaintDC* para que exista a possibilidade de se desenhar a partir da função *OnPaint*. Precisamos ainda passar uma janela para que realize o *scroll* através do *PrepareDC*. Dessa forma não temos a necessidade de realizar o cálculo necessário para ajustar o *scroll* . O trecho de código a seguir mostra como o *Device Context* é recuperado:

```
wxPaintDC dc(this);
PrepareDC (dc);
```

  Se o arquivo não estiver aberto, a função mantém as condições que foram encontradas antes da geração do evento, se o arquivo estiver aberto ela leva em consideração o tamanho *ini_ width* para desenhar o arquivo na mesma proporção. Se o arquivo tiver sido deslocado pela barra a função *OnPaint* contabiliza esse deslocamento e o realiza na hora de redesenhar.

- `void MyCanvas::OnMouseButtonDown(wxMouseEvent &event)`

  A função recebe um evento do tipo *wxMouseEvent*, essa classe de eventos contém toda a informação pertinente aos eventos gerados pelo mouse, incluindo o pressionamento de botões, seu deslocamento entre outros.

As funções pertinetes ao tratamento de eventos do mouse só funcionam quando o arquivo estiver aberto. Depois disso a função recupera o tamanho da área disponível para desenho e então recupera a posição do mouse. Essa posição é convertida para a unidade lógica de deslocamento do *Device Context*. A posição é recuperada pela linha de código `wxPoint pos = event.GetPosition();`

Quando o mouse está dentro da área que representa o arquivo um novo cursor é associado a ele, a seleção tem início e as coordenadas do ponto onde o mouse se encontra são guardadas. Tanto o ponto físico (*m_ startPoint*) quanto a amosta correspondente (*m_ startSample*) são salvas nos membros da classe *Track*.

- `void MyCanvas::OnMouseButtonUp(wxMouseEvent &event)`

  Essa função trabalha em conjunto com a função anterior. Assim como a função anterior, ela recebe um *wxMouseEvent*, verifica se existe um arquivo aberto, verifica se o mouse está dentro da área onde está desenhado o arquivo de áudio, e verifica se o início da seleção ocorreu dentro dessa área.

  Se o evento satisfizer essas condições, o ponto onde a marcação se encerra é gravado na classe *Track* nos membros *m_ endPoint* e *m_ endSample*.

- `void MyCanvas::OnMouseMove(wxMouseEvent &event)`

  Essa função é responsável por identificar a localização do *mouse*, indicando quando o cursor deve ter sua forma alterada, quando o cursor está ativo para realizar marcações e quando ele está indisponível.

IV.9 Classe *WalkGraphic* e Classe *RotateVector*

A base de controle de todo o programa está na parte do código que implementa um aplicativo do tipo *wxWidgets*, porém a exibição das informações deve-se basicamente a duas classes que são abordadas a seguir.

A classe *RotateVector* encapsula um vetor do tipo *float*, correspondente ao eixo y e

um *label* correspondente no eixo *x*. Ela nos permite ainda associar uma variável booleana para que nos utilizemos dela na classe *WalkGraphic* para permitir ou não a exibição do label do eixo *x*.

A classe WalkGraphic encapsula a Classe RotateVector e associa a ela elementos gráficos. Ela possui um construtor que devolve a área de um retângulo, especificado em sua chamada, e ainda disponibiliza um vetor de float encapsulado (RotateVector). Ela associa um *device context* que é o meio pelo qual a *wxWidgets* desenha em tela.

Outras possibilidades interessantes é a possibilidade de desenhar linhas pontilhadas paralelas aos eixos, para aplicações onde a precisão de visualização seja requerida.

A principal função desta classe é a plotVector que de fato é responsável pelo desenho do gráfico. Ela define o valor do passo em cada eixo, e associa retas aos pontos. Ou seja, liga dois pontos consecutivos por meio de uma reta.

Além disso possui uma função para atualizar os valores da escala quando ocorre mudança nos valores das amostras, outra característica importante é a possibilidade de exibir em tela os trechos marcados.

Uma das principais funções em todo o programa é a responsável pelo desenho da tela. DrawOpenedFile e DrawOpenedFileStereo, além das variantes que permitem desenhar a tela quando a onda exibida esta marcada.

O primeiro passo desta função é calcular o tamanho do comprimento do local onde o sinal de áudio será exibido. Para isso, se é a primeira vez que aquele sinal é exibido, ele é calculado para ser exibido no tamanho atual da janela, caso contrario, ele utiliza o tamanho que vinha sendo utilizado.

A classe WalkGraphic é instanciada, e um novo buffer de bitmap (local onde de fato será desenhado) é associado a ele. A área onde a onde será desenhada também é associada a essa nova instância da WalkGraphic. Um device context é associado a WalkGraphic, onde mais tarde será feita a associação entre o buffer de bitmap e esse device context, para assim ser exibido em tela.

Para cada ponto do arquivo, que recebemos por meio de um ponteiro para float, é adicionado um novo ponto na WalkGraphic. Depois deste processo estamos aptos a chamar a função que atualiza o vetor de acordo com os pontos que temos na instância, exibimos as linhas verticais e horizontais se for o caso e ainda a função que desenha no device context dado as retas ligando os pontos que formará a onda.

IV.10 Dificuldades na implementação

A construção de uma ferramenta gráfica não é tarefa simples. A integração de diversas bibliotecas, sob uma interface gráfica, transformam a tarefa, que já não é trivial em algo maior, e por conseqüência mais difícil. Este projeto possui essas características. Além de ser uma ferramenta gráfica integrando diversas bibliotecas, esse projeto ainda tem como objetivo funcionar em dois sistemas operacionais distintos.

As dificuldades não foram poucas. O primeiro passo foi relativo à transformação de dados binários em sinais de áudio representados visualmente. Essa parte pode ser considerada a base do projeto e foi onde grande parte da estrutura de classes teve de ser desenhada. As possibilidades deveriam estar previstas desde esse passo fundamental. Neste ponto ocorreu a integração da biblioteca *wxWidgets*.

Apesar de a instalação desta biblioteca não ter sido um fator impeditivo, alguns exemplos deixavam dúvidas, que tiveram de ser sanadas com inúmeros testes. Incontáveis problemas foram encontrados na implementação ligada a esta biblioteca. Alguns problemas podem ser citados: o redimensionamento da área onde os sinais de áudio são representados, a inserção do deslocamento de janelas (*scroll*), a inserção de caixas de diálogos sofisticadas, a ordenação necessária para a instanciação de objetos, entre outros.

O passo seguinte foi incorporar a biblioteca *libsndfile*, para que as amostras pudessem ser recuperadas dos arquivos de áudio. Juntamente com essa biblioteca, foi incorporada a biblioteca *portaudio* que permite a reprodução de sinais de áudio. Essas duas bibliotecas foram instaladas e utilizadas com relativa facilidade visto que a documentação é vasta e

os exemplos são de fácil compreensão.

A compressão e descompressão entre sinais *Wave* e *MP3* consumiram bastante tempo. A biblioteca utilizada para a compressão, *Lame*, tem pouca documentação, e o arquivo proposto a ajudar não é bastante claro, apesar de ter sido bastante útil. Alguns passos precisaram ser deduzidos. A documentação da biblioteca *Mad* é inexistente, a melhor documentação encontrada foram os *e-mails* de sua lista de discussão. O processo tomou bastante tempo.

Capítulo V

Conclusão

V.1 Conclusões e possíveis extensões

Este trabalho teve como objetivo a implementação de um sistema de manipulação de arquivos de áudio. Foram adotados como premissas os conceitos multiplataforma e código aberto. A proposta foi alcançada satisfatoriamente, concluindo grande parte dos nossos objetivos.

Sugerimos como trabalho futuro a implementação de um módulo capaz de capturar dados a partir de uma placa de som como entrada. Ainda sugerimos algumas modificações do ponto de vista estético, onde, por exemplo uma régua de tempo pudesse ser utilizada, e além de uma espécie de etiqueta em cada canal para exibir os dados do arquivo em questão.

A possibilidade de desenhar trechos de onda seria útil no caso de se recuperar manualmente os pontos onde houvessem falhas mais graves.

Outras funções bastante interessantes seriam as responsáveis por modificar o *pitch*, modificar o *timing* e alterar a taxa de amostragem.

Uma outra alteração interessante seria modificar o código de modo a receber novas funções na forma de *plug-ins*; assim, não seria necessário recompilar o código sempre que

uma nova função fosse incorporada.

Do ponto de vista da implementação como um todo, é sugerido buscar-se uma forma de tornar o processamento mais leve.

Referências Bibliográficas

[1] *http://audacity.sourceforge.net/.*

[2] *http://history.acusd.edu/gen/recording/notes.html.*

[3] S. Haykin, *Communication Systems.* John Willey & Sons Inc., quarta ed., 2001.

[4] K. Brandenburg, *Applications of Digital Audio processing to Audio and Acoustics.* Kluwer Academic Publishers, 1998.

[5] M. Fowler, *UML Essencial - Um Breve Guia para a Linguagem-Padrão de Modelagem de Objetos.* Bookman, segunda ed., 2000.

[6] *http://www.wxwidgets.org.*

[7] *http://www.mega-nerd.com/libsndfile/.*

[8] *http://www.portaudio.com.*

[9] *http://lame.sourceforge.net/.*

[10] *http://www.underbit.com/products/mad/.*

[11] *http://www.lightlink.com/tjweber/StripWav/WAVE.html.*

[12] *http://ccrma.stanford.edu/CCRMA/Courses/422/projects/WaveFormat/.*

[13] *http://www.noisebetweenstations.com/personal/essays/audio_on_the_internet/FileFormats.html.*

Apêndice A

Lista de acrônimos

LP :	*Long Play*;
MIT :	*Massachusetts Institute of Technology*;
A/D :	*Conversor Analógico Digital*;
D/A :	*Conversor Digital Analógico*;
GUI :	*General User Interface*;
PCM :	*Pulse Code Modulation*;
MPEG :	*Moving Picture Experts Group*;
SNR :	*Signal to Noise Ratio*;
SMR :	*Signal Mask Ratio*;
MP3 :	*Moving Picture Experts Group Audio Layer 3*;
UML :	*Unified Modeling Language*;
OOA & D :	*Object Oriented Analysis and Design*;

Apêndice B

Requisitos mínimos para a instalação

O projeto foi desenvolvido em um computador com um processador AMD Athlon XP 2200+, 1,8GHz, com 256MB de memória e placa de vídeo NVIDIA GeForce4 MX 440 com AGP8X. O programa foi testado também uma máquina com processador Pentium 2, com apenas 64MB de memória e placa de vídeo inferior. Obviamente o desempenho ficou aquém do esperado.

Apêndice C

Requisitos mínimos para compilar

Para poder compilar o programa bem como para fazer alterações no código fonte é fundamental um conjunto de bibliotecas: *WxWidgets*, *libsndfile*, *Portaudio*, *Lame* e *mad*. A biblioteca *wxWidgets* é responsável por toda a parte gráfica da ferramenta. Por sua vez *libsndfile* é a biblioteca que permite ler e escrever arquivos que contêm formatos de áudio, enquanto que *Portaudio* é a biblioteca que nos permite executar de fato os arquivos de áudio. *Lame* é a biblioteca responsável pela conversão do arquivo para o formato *MP3* e finalmente *mad* é a biblioteca responsável pela desconversão do arquivo, ou conversão de *MP3* para *Wave*.

Os usuários de *Linux* que desejarem recompilar o programa para *Windows* têm a possibilidade de fazê-lo utilizado o *Cygwin*, *software* que emula um terminal *Linux* no sistema operacional *Windows*. Ainda no ambiente *Windows* o código também pode ser compilado no *Microsoft Visual C++*, onde o código fonte foi testado e compilado sem problemas.

No ambiente *Linux* o usuário pode utilizar o compilador *g++*.

A biblioteca *wxWidgets* deve ser compilada, não possuindo distribuição de binários para nenhuma plataforma. O mesmo ocorre para a biblioteca *portaudio*.

As bibliotecas *libsndfile*, *lame* e *mad* posuem versões *apt* para o *Linux*.

C.1 Instalando a *wxWidgets*

O primeiro passo para o desenvolvimento de qualquer aplicativo gráfico multiplataforma começa pela seleção e instalação de uma biblioteca de suporte gráfico. A biblioteca selecionada neste projeto como base para a programação gráfica foi a *wxWidgets*.

C.1.1 No ambiente *Windows*

No ambiente *Windows* existem duas possibilidades: instalar o *Microsoft Visual C++* ou o *Cygwin*.

Em caso de escolha da primeira opção é necessário possuir a mídia para a instalação do programa. A forma de instalação segue o padrão de caixas de diálogo com opções. Opte pela instalação padrão. Essa opção irá instalar todos os componentes necessários para a compilação/*link* edição do projeto em questão.

O primeiro passo depois de instalar o Visual C++ é fazer o *download* do arquivo *wxMSW-2.4.2-setup.zip* (esse arquivo era a versão estável do executável mais recente em dezembro de 2003, versões superiores mantêm o suporte a versões mais recentes e podem ser utilizadas). Este arquivo está disponível no endereço `http://www.wxwidgets.org/`.

É aconselhável fazer o *download* do *help* tanto no formato *pdf* quanto no formato para *Windows*. Os arquivos são respectivamente, *wxWindows-2.4.2-PDF.zip* e *wxWindows-2.4.2-WinHelp.zip*, encontrados na mesma página. Novamente atentamos para estes serem os arquivos mais recentes em dezembro de 2003, versões mais recentes estão disponíveis.

Após descompactar o arquivo *wxMSW-2.4.2-setup.zip* para um diretório de trabalho, executamos o arquivo *setup.exe*. Depois de concluída a instalação, crie o arquivo *setup.h* no diretório $C:\backslash winap\backslash wxWindows\_2.4.0\backslash include\backslash wx\backslash setup.h$.

Para criar este arquivo podemos abrir um novo arquivo com o *bloco de notas* e escrevemos a linha, `#include "msw/setup.h"`.

O próximo passo então é compilar o projeto no Microsoft Visual C++. Localize o arquivo *wxWindows.dsw*, esse arquivo já estará vinculado ao Visual C++. Abra a aba *Build*, e depois selecione a opção *Batch Build*. Selecione todas as caixas que aparecerão e depois clique em *Build*. O *Microsoft Visual C++* criará as bibliotecas do *wxWindows*.

Se o passo anterior for concluído com sucesso, alguns testes que estarão no subdiretório *sample* do diretório de trabalho do *wxWindows* poderão ser executados. Para isso é necessário que associar o caminho da biblioteca ao projeto. Selecione a aba *projects* e clique em *Projects Setting*. Agora selecione a aba *Link* e na entrada *Category* escolha *Input*. Na entrada *Additional Library Path* adicione o endereço de onde ficam as bibliotecas da *wxWidgets*, em nosso caso, $C:\backslash wxWindows-2.4.2\backslash lib\backslash$.

Utilizando o *Cygwin* devemos instalar o software encontrado no endereço `cygwin`. O procedimento é similar ao *Linux* que será abordado na seção seguinte.

C.1.2 No ambiente Linux

A forma tradicional de instalar a biblioteca *wxWidgets* no Linux consiste em executar *./configure –with-gtk* a partir do diretório onde foi extraído o código fonte. Se este comando foi executado com sucesso, execute os comandos *make* e *make install*. Este último comando deverá ser executado como super-usuário (*root*). Depois executamos o comando *ldconfig*.

C.2 Instalando a libSndFile

O próximo passo é a instalação da biblioteca *libsndfile*. O arquivo de instalação da biblioteca é o *libsndfile-1.0.10.tar.gz*(versão de dezembro de 2003) e encontra-se no site `http://www.mega-nerd.com/libsndfile/`. Este arquivo deve ser descompactado para um diretório de trabalho. Existe diferenças na instalação da biblioteca em *Linux* e no ambiente *Windows*.

C.2.1 No ambiente *Windows*

No diretório de trabalho procure pelo arquivo *Win32_ Makefile.msvc*. Esse arquivo deve ser alterado. Localize a linha que começa com MSVCDir e altere o valor desta linha para que corresponda à localização onde se encontra o seu Visual C++. Isso é necessário para que o compilador encontre os *headers* padrão. Deve-se ressaltar que os nomes não podem ter mais de oito caracteres. A seguir copie os arquivos *sndfile.h* e *config.h* do diretório *Win32* para o diretório *Src*.

Abra uma janela *DOS* e execute o comando *nmake -f Win32_ Makefile.msvc* a partir do diretório onde a biblioteca foi instalada. O comando *nmake* deve estar em algum diretório que faça parte da variável de ambiente *PATH*. Para se certificar de que o arquivo libsndfile.dll foi construído corretamente execute o comando *nmake -f Win32_ Makefile.msvc check*.

No final do procedimento descrito acima, os arquivos *libsndfile.dll* e *libsndfile.lib* estarão no diretório corrente. Estes dois arquivos mais o arquivo sndfile.h que se encontra no diretório *Win32* são os componentes necessários para serem usados em um projeto que utilize a biblioteca.

C.2.2 No ambiente Linux

A forma mais simples de compilar a biblioteca *libsndfile* no ambiente Linux é seguir alguns passos básicos. Em primeiro lugar abra um *shell* e execute o comando *cd* para o diretório que contém o código fonte da biblioteca. Neste diretório executamos o comando *./configure*. Quando a execução do comando *configure* terminar, compile o programa através do comando *make*. Para conferir se a compilação ocorreu com sucesso podemos executar o comando *make check*.

Para instalar o programa, execute o comando *make install*. Se o usuário quiser ele pode remover os códigos binários e os arquivos objeto executando o comando *make clean*. Se quiser remover os arquivos criados pelo comando *configure* o usuário deve executar

make distclean.

C.3 Portaudio

A biblioteca Portaudio, disponível no endereço `http://www.portaudio.com/`, é a biblioteca capaz de escrever amostras no dispositivo de som e assim reproduzir sons. Ela é escrita em C e é disponibilizada em versões para compilação de diversas plataformas.

C.3.1 No ambiente Windows

Para compilar a biblioteca *Portaudio* no ambiente Windows, temos três opções. *DirectSound API*, *Windows MultiMedia Extensions API* (conhecida como *WMME* ou *WAVE*) e ainda *Steinberg's ASIO API*. A forma mais abrangente e recomendada é a WMME.

Para qualquer Windows, a inclusão dos arquivos a seguir é obrigatório:

- `pa_common\pa_lib.c`
- `pa_common\portaudio.h`
- `pa_common\host.h`

Precisamos ainda fazer a ligação (*link*) com a biblioteca `winmm.lib`. No projeto que usaremos devemos escolher a opção `Project`, aba `Settings...`, escolher no menu `All Configurations` e então ir à aba `Link`. No campo `object/library modules` devemos entrar com o nome do arquivo *winmm.lib*.

Projetos utilizando o *WMME* devem ainda adicionar o arquivo fonte `pa_win_wmme\pa_win_wmme.c`. Projetos utilizando a *DirectSound* precisam ainda adicionar os arquivos `pa_win_ds\dsound_wrapper.c` e `pa_win_ds\pa_dsound.c`.

Para todos os projetos ainda é necessário ligar as bibliotecas `dsound.lib` e `winmm.lib` utilizando o procedimento já mencionado.

C.3.2 No ambiente Linux

Para compilar a biblioteca, você precisa estar no diretório `pa_unix_oss`, depois editar o arquivo `Makefile` e descomentar um dos testes, use por exemplo o `patest_sine.c`, que é muito simples. Agora basta executar o comando `gmake run`.

C.4 Instalando a *Lame*

Lame não é uma biblioteca, ela é um conjunto de códigos fonte abertos que devem ser necessariamente ser compilados em qualquer plataforma. O código fonte está disponível no endereço `http://lame.sourceforge.net/download/download.html`. Após descompactar o arquivo para um diretório de trabalho, os processos de compilação se diferenciam.

C.4.1 No ambiente Windows

Um programa que utilize *Lame* para compactar arquivos para o formato MP3 deve usar o arquivo de *header lame.h*, deve possuir o arquivo *lame_ enc.lib* e ainda a biblioteca compilada (*lame_ enc.dll*). No diretório principal onde foi descompactado o arquivo, execute o arquivo *lame.bat*, que ajusta as variáveis de ambiente para trabalhar com o *Visual C++* no modo texto. Depois o comando *copy configMS.h config.h* deve ser executado para substituir o arquivo e por fim deve ser executado o comando *nmake -f Makefile.MSVC comp=msvc asm=no*.

Ambos os arquivos necessários para compilar o programa no ambiente *Windows* estarão disponíveis no diretório de trabalho utilizado.

C.4.2 No ambiente Linux

No *Linux* devemos seguir a execução dos comandos *./configure*, *make* e por fim *make install*. Basta incluirmos a opção *-lmp3lame* na linha de chamada do compilador *gcc*.

C.5 Instalando a *Mad*

A biblioteca *Mad* foi desenvolvida em C e tem como objetivo disponibilizar uma rotina eficiente e de alta qualidade para a descompressão de arquivos no formato *MP3* para o formato *Wave*. Cada usuário é responsável pela compilação do código fonte, disponível no endereço `http://www.underbit.com/products/mad/`.

C.5.1 No ambiente Windows

Dois arquivos devem ser adquiridos, são eles *libmad-0.15.1b.tar.gz*, que contém a bibloteca, e o arquivo *madplay-0.15.2b.tar.gz* que contém uma versão compilável de um programa que utiliza a biblioteca. A necessidade desse segundo arquivo se deve ao fato de possuir o projeto *madplay.dsw* que permite a criação do arquivo *libmad.lib*.

Após descompactar ambos os arquivos dentro do mesmo diretório, um dirétorio relativo ao *madplay* será criado. Dentro do diretório $madplay - 0.15.2b\backslash msvc++$ encontra-se o projeto *madplay.dsw*. Abra o arquivo com o *MSVC++*, dentro do arquivo ative o projeto *libmad* através do menu *Project*, opção *Set As Active Project*, ítem *Libmad*. Através da opção *Build*, selecione o ítem *Build libmad.lib*. A biblioteca será criada na pasta $libmad - 0.15.1b\backslash msvc++\backslash Debug$ com origem no diretório de trabalho.

C.5.2 No ambiente Linux

No *Linux* devemos seguir a execução dos comandos *./configure*, *make* e por fim *make install*. Basta incluirmos a opção *-lmad* na linha de chamada do compilador *gcc*.

Apêndice D

Instalando o Editor Gráfico de Sinais de Áudio Multiplataforma

Para a utilização do programa basta extrair o arquivo `plotGraphic.zip` para um diretório padrão. Já incluímos alguns arquivos de teste, sendo dois monos e um estéreo.

No ambiente *Windows* o binário existente já pode ser utilizado. No ambiente Linux, o usuário pode utilizar o arquivo *makefile* através do comando *make* e digitar a linha *./open* a partir do diretório de trabalho para iniciar o uso do programa.

Apêndice E

Manual de Utilização

E.1 Funções de manipulação de arquivo

Para utilizar o programa basta executar o arquivo `plotGraphic.exe` no *Windows* ou executar *./open* no diretório de trabalho no *Linux*. Tendo sido iniciado com êxito, o programa permite a abertura de arquivos ou a criação de um arquivo novo. Um arquivo novo é criado com um tamanho padrão e exibido em tela. Esse tipo de abertura de arquivo faz mais sentido quando queremos captar dados através da placa de som, porém, essa versão preliminar do programa não possui tal funcionalidade.

Uma outra opção para a abertura de um arquivo novo é colar trechos selecionados de outros arquivos. O programa permite que múltiplas instâncias do programa sejam abertas e que exista a troca de informação entre elas.

Nesta versão do programa temos como principal objetivo a edição de um arquivo existente.

A criação de um novo arquivo se dá pela escolha do ítem *New* no *menu File*. Essa funcionalidade também pode ser atingida pelo uso do ícone que lembra uma folha de papel em branco. Na figura E.1 podemos ver o ícone que representa a funcionalidade e também a caixa de diálogo que é aberta.

E.1 Funções de manipulação de arquivo

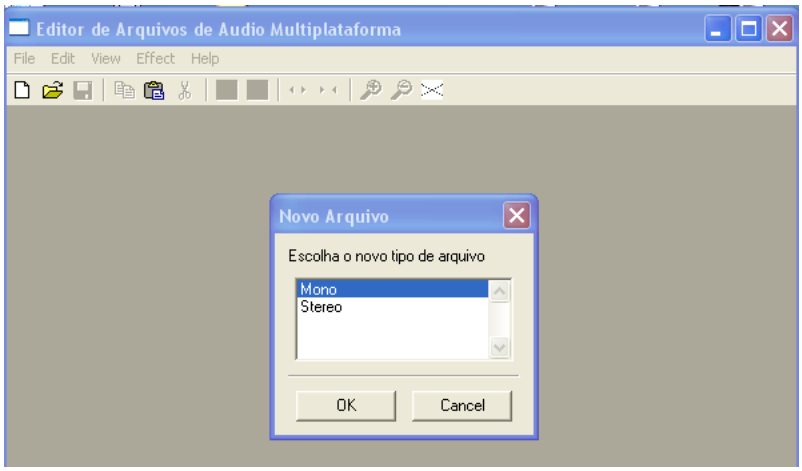

Figura E.1: Caixa de diálogo de novo arquivo.

Depois de selecionarmos essa funcionalidade, uma caixa de diálogo nos permite escolher entre a criação de um arquivo mono ou estéreo que depois será exibido em tela. Uma alternativa a criação de um arquivo é a abertura de um já existente.

A abertura de um arquivo pode ser realizada pela seleção do *menu File* seguida pela escolha da entrada *Open*. Uma forma alternativa é utilizar o ícone da barra de ferramentas, representado pela figura de uma pasta. Esses casos são mostrados na figura E.2.

Após a escolha da opção *Open*, uma caixa de diálogo será exibida. A figura E.3 mostra a caixa de diálogo. A configuração do programa utiliza um filtro e exibe os arquivos que podem ser abertos com o editor. As extensões correspondentes são .wav e .mp3, que representam os arquivos nos padrões utilizados no programa.

Após a escolha do arquivo, escolhemos o botão *Open*. Este botão dará início à exibição do arquivo em tela. Podemos ver na figura E.4 um arquivo de um só canal sendo exibido em tela. Quando um arquivo está aberto, não podemos abrir outro arquivo. Por esse motivo as entradas correspondentes à abertura de arquivos permanecem desabilitadas. Por outro lado, quando um arquivo está aberto, as entradas relativas ao processo de salvar as alterações dos arquivos permanecem habilitadas (*Save* e *Save As*).

A entrada *Save* salva as alterações feitas no arquivo, sobrescrevendo o arquivo original,

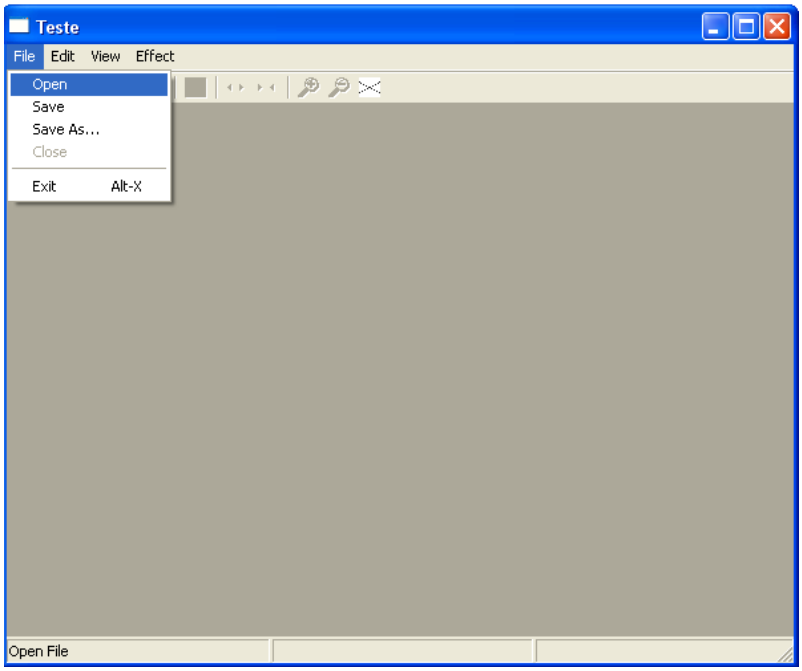

Figura E.2: Menu de arquivo, entrada *Open* e ícone correspondente.

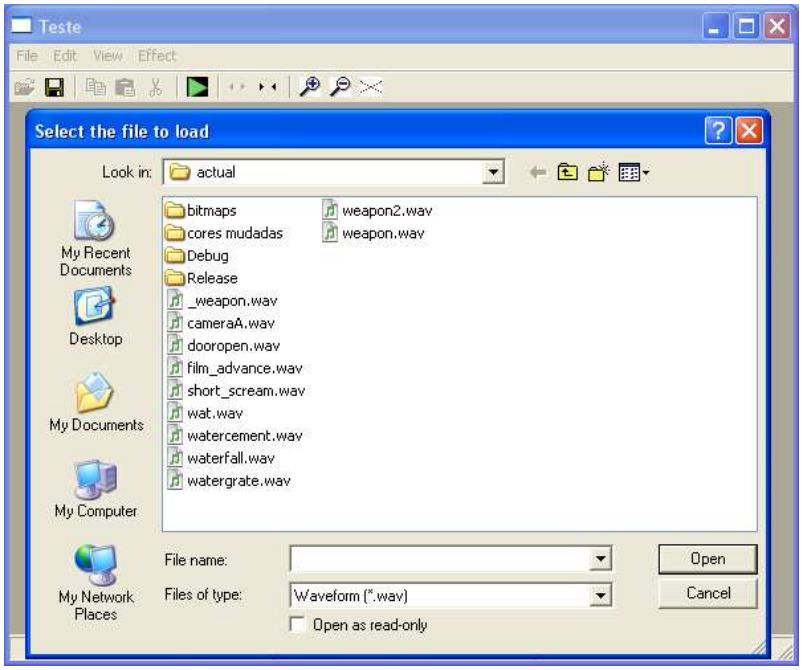

Figura E.3: Caixa de diálogo para a escolha do arquivo.

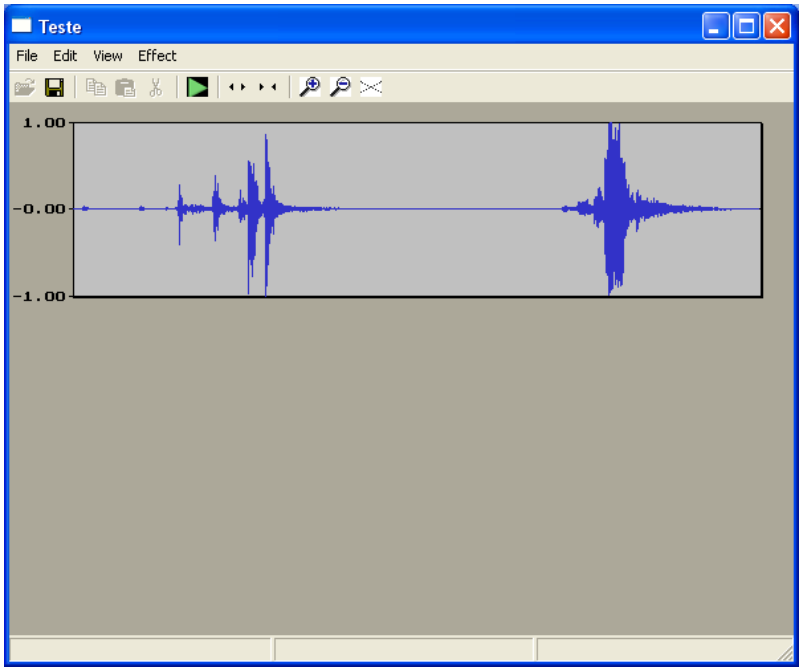

Figura E.4: Arquivo mono exibido.

enquanto que a entrada *Save As* realiza a operação em um arquivo que o próprio usuário pode selecionar.

A escolha da entrada *Save As* abre uma caixa de diálogo semelhante àquela aberta na operação *Open*. Esta caixa pode ser vista na figura E.6.

A última entrada do *menu File* é a função *Close* que realiza o fechamento de um arquivo para o processamento de outro arquivo.

E.2 Funções de edição

Uma das principais tarefas que auxiliam o usuário no processamento de arquivos é a seleção de um trecho do arquivo. Para efetuar uma seleção de trecho basta posicionar o cursor do *mouse* sobre o trecho desejado do arquivo (o cursor muda para o modo de marcação) e clicar com o botão direito no ponto inicial onde se deseja realizar a marcação, arrastar o *mouse* até o ponto final e soltar o botão. A seleção de trecho pode ser vista

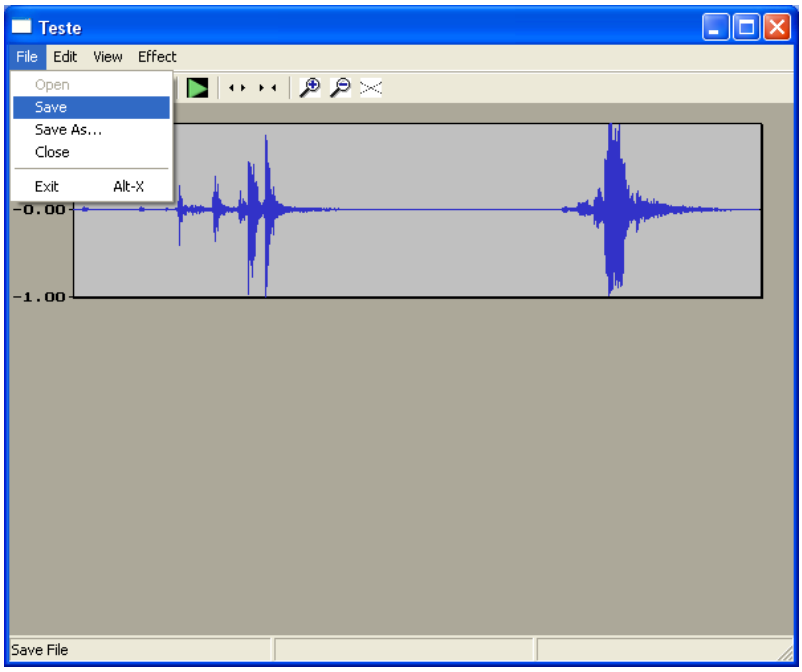

Figura E.5: Opções para salvar um arquivo a partir do *Menu File*.

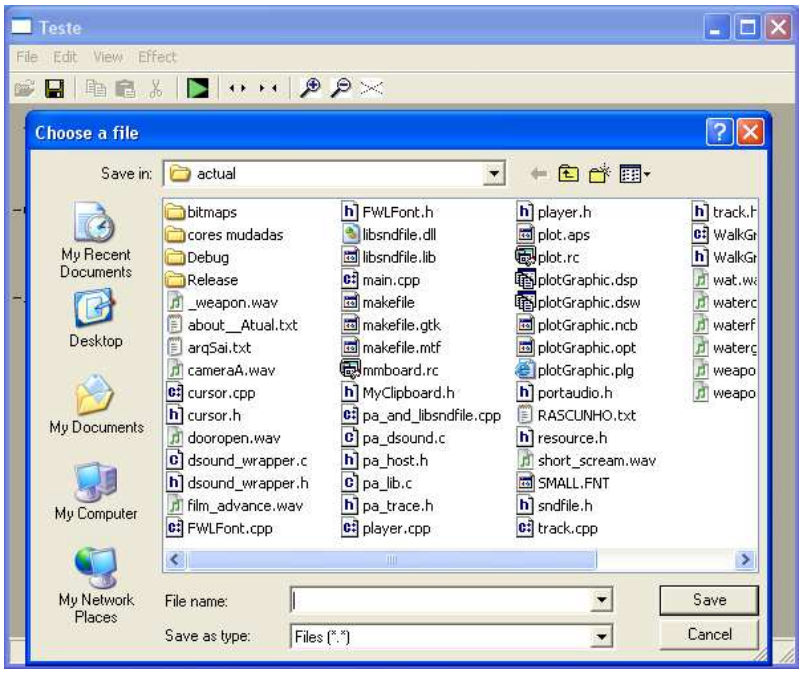

Figura E.6: Caixa de diálogo da função *Save As*.

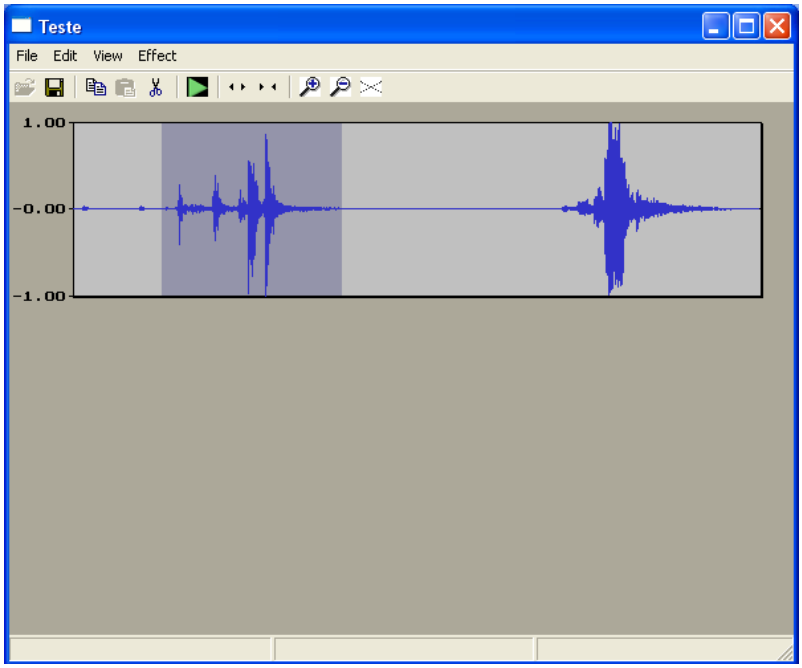

Figura E.7: Seleção de um trecho do arquivo de exemplo, o cursor está no modo de seleção.

na figura E.7. Muitas funções trabalham em trechos marcados, por isso a função citada anteriormente é de extrema importância no programa.

A partir de uma seleção temos as funções responsáveis por trocar trechos de arquivos ou trocar de lugar dentro de um mesmo arquivo. As funções referidas aqui são *Copy*, *Cut* e *Paste*.

A função *Copy* pode ser utilizada pelo *menu Edit*, na entrada *Copy* ou ainda pelo atalho `CTRL+C`. Essa função transfere para a memória o trecho de arquivo copiado, permitindo a transferência de trechos entre arquivos. Outra função que é utilizada para mudar, dentro de um mesmo arquivo posições de trechos específicos é a função *Cut*. Essa opção tem uma entrada no mesmo *menu* que a função *Copy* e funciona também através do atalho `CTRL+X`.

A função que complementa as duas anteriores é a função *Paste*. Essa função recupera da memória o trecho de arquivo que fora anteriormente marcado e copiado ou recortado, e colando-o no lugar marcado pelo mouse. Essas são as três formas de movimentação e

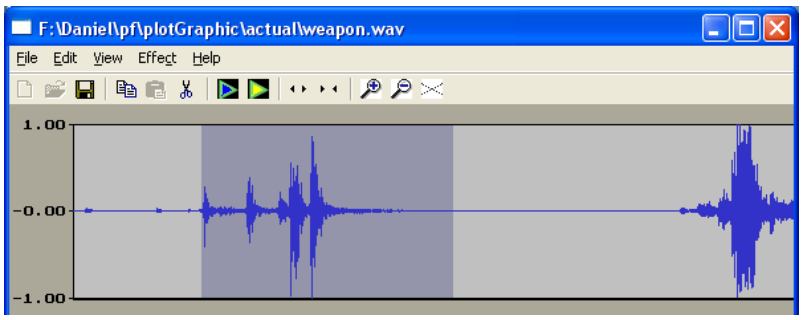

Figura E.8: *Zoom In* em um trecho marcado.

transferência de trechos em um arquivo.

Uma outra funcionalidade disponibilizada neste *menu* é a entrada *Select All*. Essa função marca todo o trecho do arquivo aberto evitando que tenhamos de fazer o processo manualmente com o *mouse*.

E.3 Funções de visualização

Duas funções que auxiliam o usuário na marcação e melhor compreensão do arquivo de áudio são as funções que ficam no *menu Edit*, entrada *Exibition*, e são elas *Sample* e *Time*. A função *Sample* permite ao usuário visualizar no menu de *Status* o ponto físico e a amostra correspondente ao local por onde o mouse passa. Essa função auxilia diretamente na marcação de trechos específicos.

A função *Time* exibe na última coluna da barra de *Status* o tempo correspondente à amostra por onde o *mouse* esta passando.

Nesse mesmo menu estão as duas funções de *Zoom*, especificamente as funções que se preocupam em melhorar a visão no eixo x. As entradas correspondentes são *Zoom Horizontal In* e *Zoom Horizontal Out*. A função *Zoom Horizontal In* dá um maior destaque ao arquivo espaçando as amostras uma das outras, o que pode ser visto na figura E.8.

A outra entrada, *Zoom Horizontal Out* permite que o processo anterior seja revertido.

E.3 Funções de visualização

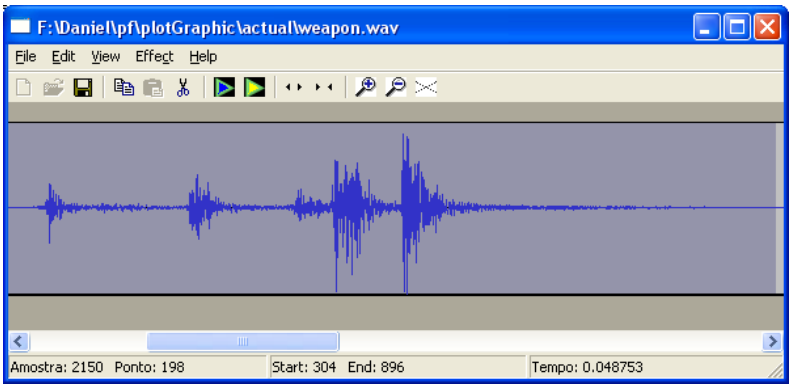

Figura E.9: Expansão de um trecho marcado.

O espaço de separação das amostras é reduzido e assim damos prioridade a uma visão mais global do arquivo. Outras duas funções ativas pela barra de ferramentas e que usam o princípio das duas funções citadas anteriormente são as funções *Expand* e *Fit*. A função *Expand* pode ser executada a partir de uma seleção, ela expande o tamanho da seleção para a tela inteira, executando portanto um *zoom* que realize tal procedimento. A função *Expand* pode ser vista na figura E.9.

Por outro lado, a função *Fit* encaixa todo o arquivo aberto no espaço em tela que dispomos, o arquivo fica reduzido visualmente. A outra função do mesmo menu é a função *Máxima Amplitude* que nos fornece por meio de uma caixa de diálogo o maior valor absoluto de uma amostra no arquivo. Se existir algum trecho do arquivo selecionado, somente aquela parte do arquivo é inspecionado para obter esse valor.

A última entrada desse *menu* é a função *Separate*. Essa função permanece ativa nos arquivos estéreos e permite que os canais sejam tratados separadamente. Na figura figura E.10 é mostrado um arquivo estéreo com a marcação sendo exercida somente no canal esquerdo (superior).

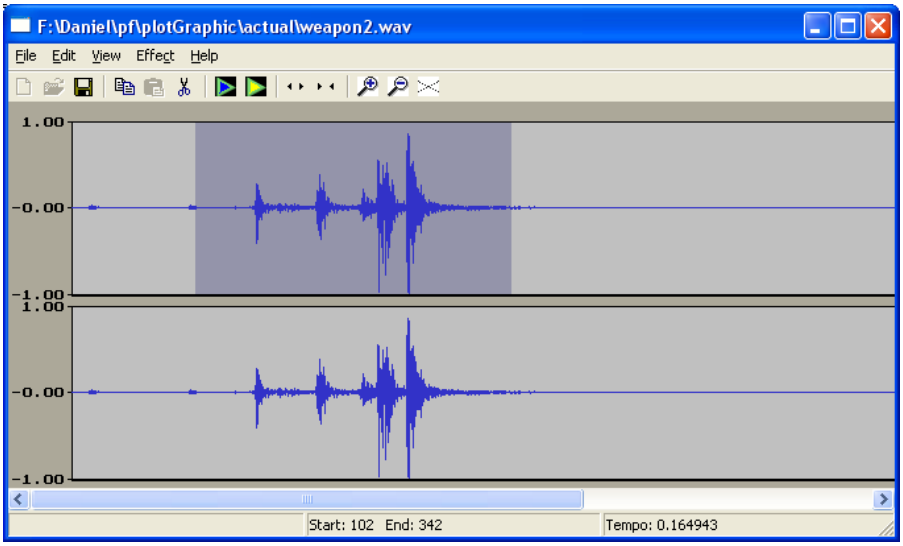

Figura E.10: Marcação de somente um canal de um arquivo estéreo.

E.4 Funções de efeitos

Alguns efeito são restritos a aplicação em trechos limitados do arquivo, e para isso a execução de qualquer uma das funções requer a prévia marcação. A primeira função que abordaremos é o Efeito Nulo (*Null*). Essa função zera o valor das amostras selecionadas e pode ser utilizada para suprimir ruídos, estalos e *clicks*.

A função Nulo Reverso (*Reverse Null*) zera todas as amostras fora da seleção. O uso dessa função se dá quando queremos dar destaque a algum trecho, para que por exemplo possamos posteriormente colar em um outro arquivo, ou mesmo fazer um efeito de *loop* dentro de um arquivo. A figura E.11 mostra o efeito sendo aplicado.

Os efeitos *Fade in* e *Fade out* tem a função de gradativamente aumentar o volume e diminuir o volume de um trecho. São utilizadas principalmente no início e final de arquivo respectivamente.

O efeito de *Cross Over* é utilizado quando queremos emendar dois trechos de arquivos diferentes e queremos impedir que ocorram *clicks* nesta ligação. Podemos ainda querer reduzir o efeito de palmas em concertos ao vivo. Os três efeitos são mostrados na

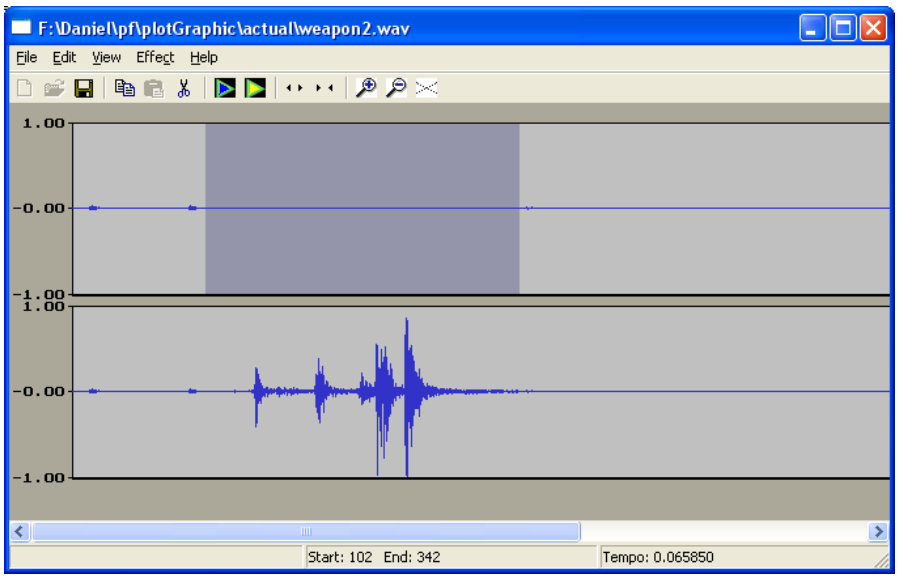

Figura E.11: Aplicação do efeito *Null* em um trecho de arquivo.

figura E.12.

As duas outras funções do Menu são a conversão de um arquivo mono em estéreo e também a conversão de um arquivo estéreo em um arquivo mono.

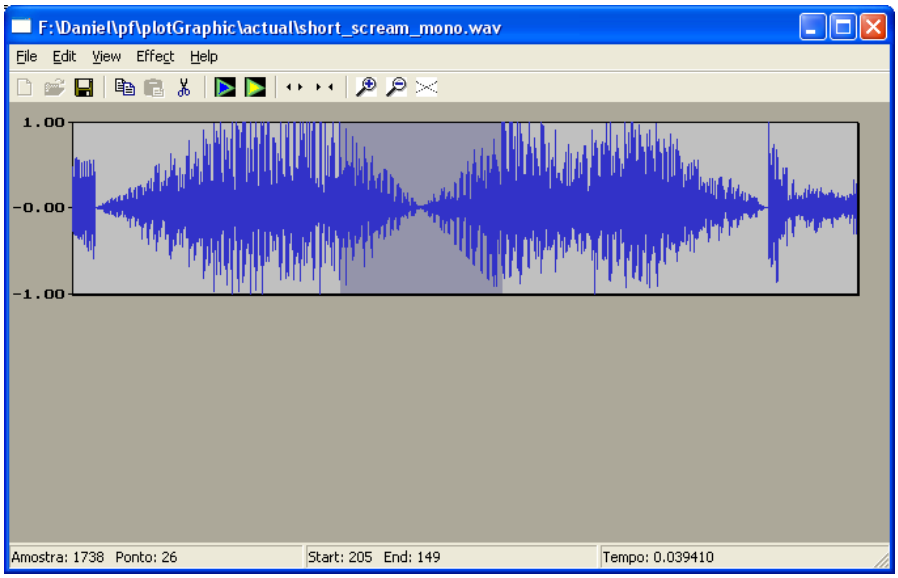

Figura E.12: Aplicação dos efeito *Fade In*, *Cross Over* e *Fade Out* respectivamente no arquivo.