Universidade Federal de Pernambuco
Graduação em Ciência da Computação

Sintetizador de Imagens Metafóricas de Execução Musical

Jarbas Jácome de Oliveira Júnior

TRABALHO DE GRADUAÇÃO

Recife
11 de março de 2005

Universidade Federal de Pernambuco
Centro de Informática

Jarbas Jácome de Oliveira Júnior

Sintetizador de Imagens Metafóricas de Execução Musical

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Sílvio Melo
Co-orientador: Geber Ramalho

Recife
11 de março de 2005

Agradecimentos

Papai, Mamãe, Huguinho e Daniel, vovó Geni e vovó Lupécia, véi Toinho e véi Jandila, Nessinha (e Pedrão, Patrícia, Kássia, Kassandra e Gabênsius), Gandhi, Mavi, Fred e Dedeco, Diogo e nossa querida família, Asas, Márcio, Regiane, Nix e Tex, Geber Ramalho, Clylton Galamba, Sílvio Meira, Sílvio Melo, Alejandro Frery, Ives, Fábio Heday e Gato Mestre (Bruno Bourbon), NeHe, Re:combo, MediaSana (Queops e Igor), Sebba, Negroove, Turma 2000-2/2001-1, Barbelinhos, Magão, Capiau, Parcira, Melo, Lúcia, Ináicia, Edilene, Roberto, Antônio (do CTG), Cin e UFPE, Apodi, Patu, Paris (Pedrão e Lawrence), Londres(Isabelle), Berlim(Gabi e Ivanzinho), Recife, Olinda, Porto Trombetas e Natal, Perez Prado, *The Meters*, *The Beatles*, Chico Science, Bezerra da Silva, Jr. Black, Jorge Ben, Jackson do Pandeiro e Zé Areia

Resumo

Este projeto tem por finalidade a pesquisa e o estudo de tecnologias de infra-estrutura de *softwares* multimídia que possam ser usados no desenvolvimento de sistemas capazes de gerar imagens e efeitos visuais. Os usuários desses sistemas podem ser artistas que se apresentam em público exibindo vídeos sintetizados em tempo real. Entre os principais interesses do projeto, mais especificamente, está a síntese de imagens e efeitos em vídeo capturado em tempo real, utilizando a análise da entrada de áudio como parâmetro desses efeitos e da síntese.

Sumário

Índice de figuras

Introdução

> "Jovem suíça (27 anos, musicista) é capaz de saborear literalmente a música", título de uma reportagem de [Presse, 2005] para o jornal [FolhaOnLine, 2005], de 2 de Março de 2005: "consegue ver cores ao ouvir uma música (...) Um intervalo musical de segunda menor a induz a sentir acidez, enquanto o de segunda maior deixa um gosto amargo em sua boca. A terça menor é salgada e a terça maior, doce. (...) ela apresenta o caso mais extremo jamais visto de **sinestesia**, no qual **a música estimula uma resposta em outros órgãos sensoriais.**"

A criação, produção e manipulação de imagens projetadas para exibição ao público, durante concertos musicais é considerada uma nova linguagem artística e vem sendo cada vez mais difundida nos últimos anos. Desde muito tempo atrás, os espetáculos de execução musical já são acompanhados por elementos visuais como a iluminação. Atualmente, graças ao avanço tecnológico e queda do preço do *hardware* necessário, além do artista iluminador, está se tornando comum em apresentações públicas de grupos musicais, a presença do *VJ* (*Video Jockey*), um artista que opera *softwares* em um computador ligado a um projetor durante o concerto, para que este exiba imagens criadas ou transformadas por ele naquele momento. Na Europa, é muito comum que grupos musicais possuam um *VJ* como componente fixo da banda.

Existe, assim, uma demanda por sistemas, que podemos chamar de **instrumentos visuais** capazes de auxiliar esses novos artistas a expressarem sua arte, e concretizarem, em imagem, suas idéias. As possibilidades de operação, funcionamento e interface destes sistemas são incontáveis e a maioria ainda não foi imaginada, devido ao fato de ser uma modalidade artística muito recente.

O objetivo do projeto aqui apresentado, chamado **ViMus** (**Vi**sual+**Mus**ica) é a pesquisa e estudo de tecnologias de infra-estrutura de *software* multimídia que possam potencialmente ser usadas no desenvolvimento desse tipo de sistema. Entre os principais interesses do projeto, mais especificamente, está a síntese de imagens e efeitos em vídeo capturado em tempo real, utilizando a análise da entrada de áudio como parâmetro desses efeitos e da síntese. A Figura abaixo mostra um esquema de funcionamento do **ViMus**.

Figura 1. Exemplo de aplicação do *ViMus*.

A pesquisa foi além do estudo das tecnologias e obteve como resultado um modelo e a implementação de um *software* (também chamado **ViMus**) já bastante funcional, com alguns exemplos de **instrumentos visuais** construídos. Os primeiros protótipos (mesmo sem interface para edição de efeitos que não entrou no escopo do projeto por enquanto) já despertaram o interesse de grupos importantes de *VJs* e músicos da cidade como [Re:combo, 2005] e [MediaSana, 2005].

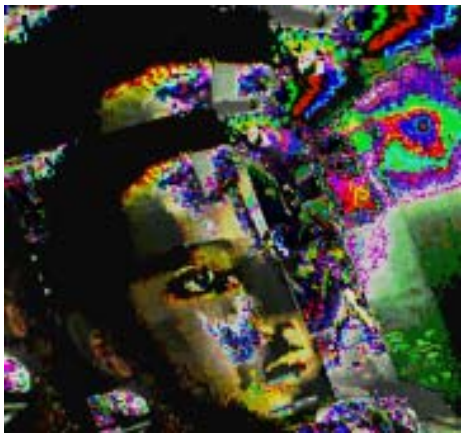

Figura 2. Exemplo de saída do ViMus. Imagem gerada em tempo real a partir da captura de uma *webcam* e com efeitos baseados na entrada de áudio.

1. Projetos Relacionados

Neste capítulo serão apresentados alguns trabalhos já existentes que foram analisados antes e durante o processo de desenvolvimento do **ViMus**. Utilizaremos a seguinte classificação de *softwares* de *VJ* adaptada de [PIXnMIX, 2005]:

- **orientado a amostras**: permite que o usuário carregue várias amostras de vídeo e dispare a qualquer momento através do teclado, ou através de interface MIDI.
- **orientado a efeitos**: o principal foco do sistema é em efeitos, tanto de síntese de imagens como de transformações em vídeo capturados em tempo real
- **infra-estrutura:** sistemas que oferecem uma *API* de alguma finalidade multimídia específica, que pode ser utilizada por outras aplicações

1.1 VVVV

Classificação: Orientado a efeitos (principalmente de síntese) e infra-estrutura

Esta ferramenta [VVVV, 2005] de nome curioso é usada para síntese de vídeo em tempo real, com a facilidade de manipulação de vários dispositivos físicos multimídia. É muito poderosa, permitindo *clustering* de PCs para criar multi-projeções, por exemplo. Não foi criada especificamente para *VJs*, e sim para qualquer aplicação multimídia que

necessite utilizar vários dispositivos, sendo o estado da arte da síntese de imagens 3D em tempo real parametrizável por qualquer tipo de entrada (áudio, vídeo, MIDI, etc.).

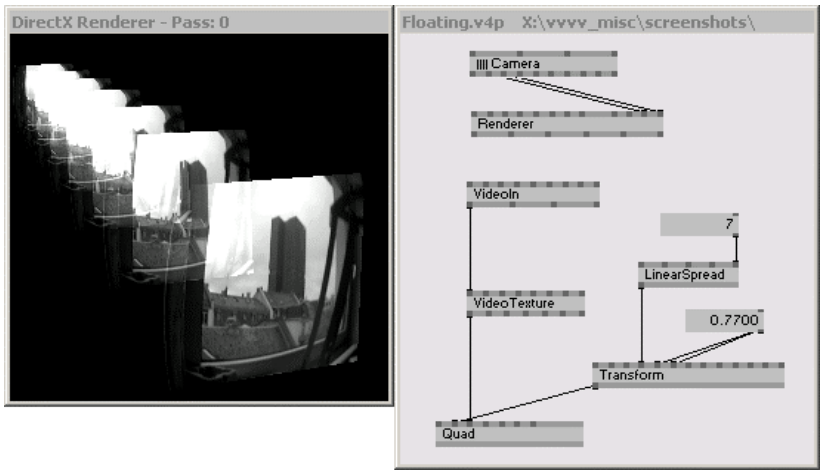

Figura 3. VVVV: Efeitos de entrada de vídeo como textura em objetos 3D.

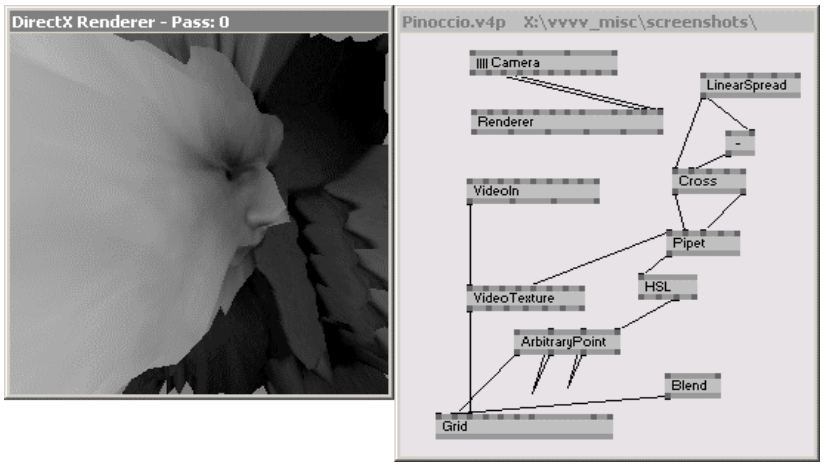

Figura 4. VVVV: Criação de objetos 3D em tempo real, baseado no vídeo.

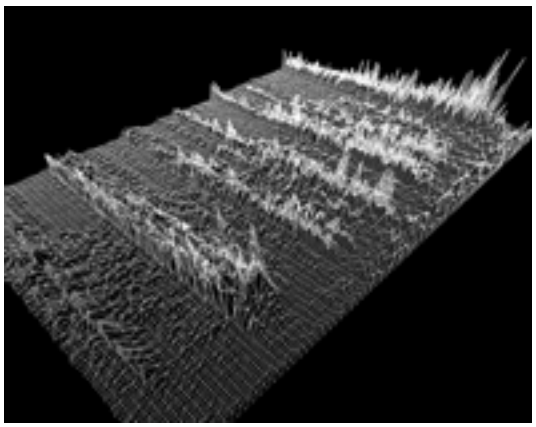

Figura 5. VVVV: Análise espectral da entrada de áudio

Um ponto fraco do VVVV é a falta de interface amigável para o usuário que precisa ser um especialista para poder utilizá-lo. Isso, porém é justificável pelo fato de não ser uma ferramenta direcionada ainda para nenhuma aplicação específica como um *VJ software*. Além disso, ainda está nas versões beta e em constante desenvolvimento. O código é fechado, porém a arquitetura é totalmente extensível, implementando-se interfaces de módulos definidas pelo sistema. É desenvolvido por um laboratório de multimídia em Frunkfurt chamado Meso [Meso, 2005] que trabalha na fronteira entre a arte e a tecnologia.

1.2 Resolume

Classificação: orientado a amostras e orientado a efeitos

Eleito pela [VJForums, 2005] o melhor *software* de VJs do momento pela sua "excelente interface gráfica e curva de aprendizado". Realmente, é um *software* totalmente direcionado a apresentações ao vivo, e tem uma interface boa. Porém não consegui testar de forma alguma a captura de vídeo, ou efeitos em vídeos. Além disso, não possui um bom sintetizador - não gera gráficos 3D com geometria baseada na entrada de Áudio como o próprio **ViMus,** por exemplo. E seu código é fechado.

Figura 6. *Resolume*: boa interface. Cada quadrado desses no inferior da tela pode armazenar um sample de vídeo que é disparado pela tecla correspondente no teclado.

1.3 Visual Jockey

Classificação: orientado a efeitos

Visual Jockey [Visual Jockey, 2005] possui um excelente sintetizador, totalmente parametrizável pela entrada de áudio, ou seja, possibilitando a síntese baseada em análise do som. Porém, após várias tentativas, não consegui executar vídeo capturado em tempo real para ser processado baseado na entrada de áudio. Código fechado e *shareware*.

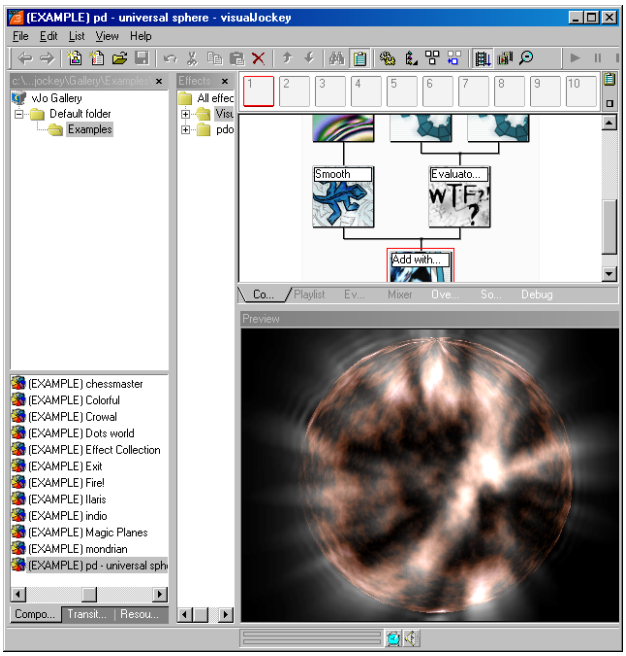

Figura 7. Interface gráfica do *Visual Jockey*: ótimos efeitos de síntese.

1.4 *Plugin AVS* do *Winamp*

Classificação: orientado a efeitos

Este já foi provavelmente o mais famoso sistema de visualização para uso doméstico. Ocorre um fenômeno interessante: o AVS [AVS, 2005] parece ser simplesmente ignorado pela comunidade de *VJs*, não havendo nem citações sobre ele em páginas especializadas. Entretanto existe uma comunidade enorme de *designers hackers* seguidores do AVS que além de artistas, estudam matemática e muitas vezes são programadores de C++ e Assembly. Passam horas estudando funções geométricas, para

conseguir a fórmula do melhor efeito de todos a ser executado no *AVS*. Talvez, uma das explicações para que este sistema não seja utilizado para apresentações ao vivo por VJs é o fato de que poucos saibam que existe a possibilidade de utilizar uma entrada de áudio do microfone e obter a geração de imagem a partir dela em tempo real, através de um *input plugin* do *Winamp*. Outro motivo para que a maioria dos VJs não se interessem por essa ferramenta é que o AVS não é, de forma alguma "orientado a amostra" que é o tipo de *software* preferido pelos *VJs*.

Porém, apesar de todas suas limitações, o AVS possui um sintetizador de imagens 2D (e pseudo-3D) superior a todos os analisados. Seu segredo está em sua flexibilidade e a possibilidade de entrada de fórmulas matemáticas para a síntese dos efeitos e é claro no talento dos artistas que criam as fórmulas matemáticas para se chegar ao efeito estético desejado.

Como veremos em **5.4 Effect**, a arquitetura de efeitos do **ViMus** foi inspirada no esquema de lista de efeitos do *AVS*.

2. Conhecimentos de domínio da aplicação

Para uma melhor compreensão do conteúdo deste documento, são necessários alguns conhecimentos básicos relacionados a processamento de som e imagem digital. Este capítulo apresenta alguns conceitos de forma a introduzir o assunto a pessoas com pouca experiência na área.

2.1 Áudio Digital

Citando definição da [Wikipedia, 2005], o som é uma compressão mecânica ou onda longitudinal que se propaga através de um meio (sólido, líquido ou gasoso). Um som possui uma velocidade de oscilação ou freqüência que se mede em hertz (Hz) e uma amplitude ou energia que se mede em decibéis. Os sons audíveis pelo ouvido humano têm uma freqüência entre 20 Hz e 20 Khz. Como representá-lo em uma máquina discreta?

2.1.1 Formato *PCM*

Para representar esta informação digitalmente, foi estabelecido um padrão utilizado nos CDs, por exemplo, chamado PCM. Um som no formato PCM consiste em um *array* de valores correspondentes a amostras de amplitude no decorrer do tempo. Quanto maior a freqüência de captura para obter esse *array*, mais amostras existirão para um intervalo de tempo e a onda sonora será mais fielmente representada e por isso esse valor determina a qualidade do som digitalizado. Valores comuns de freqüência de amostragem, também chamada resolução, são 8100 Hz, 11025 Hz, 22050 Hz, 44100 Hz, por exemplo. Quanto ao número de bits, o mais comum é que cada amostra seja representada por 8 ou 16. Com 16 bits, é alcançada uma melhor resolução de amplitude.

Formato de dados	Valor máximo	Valor do meio
8-bit PCM	255 (0xFF)	0 128 (0x80)
16-bit PCM	32,767 (0x7FFF)	- 32,768 (0x8000)

Tabela 1. Resolução de amplitude para 8 e 16 bits.

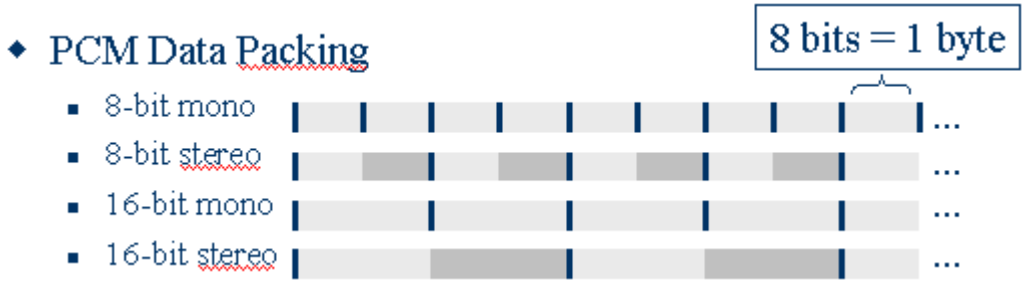

Figura 8. Empacotamento de dados em streams no formato PCM.

A figura acima mostra 4 possibilidades de formato de *stream* de áudio e são justamente as 4 suportadas pelo sistema ViMus.

2.1.2 MIDI

Nos anos 80, diversas empresas fabricantes de instrumentos musicais eletrônicos se reuniram para a definição de um protocolo de comunicação entre controladores (teclados) e sintetizadores. Este protocolo chamado MIDI – Musical Instrument Digital Interface, define mensagens informando, por exemplo, notas a serem tocadas, em uma determinada intensidade e duração. Este protocolo é muitas vezes interpretado por *softwares* de *VJs* por quê possibilita, por exemplo, que o mesmo utilize um teclado, controlador MIDI para disparar vídeos, ou efeitos de vídeo e ao mesmo tempo tocar aquela nota correspondente a tecla pressionada.

2.2 Processamento Digital de Sinais

Como vimos na sessão anterior, o som em formato digital é uma seqüência de amostras (*samples*) de amplitude no decorrer do tempo. Porém, faz-se muitas vezes

necessário saber a amplitude do som em uma determinada freqüência para, por exemplo, analisar o espectro de onda produzido por uma origem sonora, avaliando se está predominantemente grave, ou predominantemente agudo. Para sinais digitais, utiliza-se para isso a transformada discreta de Fourier cuja entrada é uma seqüência de amostras de amplitude no tempo e a saída é a amplitude em uma determinada freqüência [Embree, 1998]. Calculando-se essa amplitude para várias freqüências, obteremos um gráfico representando o espectro de freqüência da onda.

2.3 Processamento Gráfico e Vídeo Digital

2.3.1 Bitmap

Um *bitmap*, como o nome sugere, consiste em uma matriz bidimensional de elementos, em que cada elemento representa a cor de um ponto de uma imagem a ser exibida na tela do computador.

2.3.2 Sistema de cores RGB 24 bits

Para desenhar um *bitmap* na tela, o dispositivo gráfico precisa saber qual o sistema de cores o *bitmap* está usando para poder imprimir a cor corretamente. Um sistema muito comum é o RGB 24 bits no qual cada ponto (*pixel*) do *bitmap* é representado por 3 bytes (24 bits) onde o primeiro representa a componente vermelha, o segundo a verde e a terceira a azul da cor final. A cor final é resultante do processo de "mistura" dessas três componentes, que na verdade se trata da simulação de interseção luminosa da luz vermelha, verde e azul. A intensidade de cada componente varia de 0 a 255.

Exemplos de cores:

R	G	B		Cor
0,	0,	0	=	preto
255,	255,	255	=	branco
255,	255,	0	=	amarelo

2.3.3 Vídeo Digital

Um vídeo digital consiste em uma seqüência de *bitmaps* chamados *frames* ou quadros, que são exibidos a uma determinada freqüência denominada “quadros por segundo” ou *frames per second* (*fps*).

Observações importantes sobre *fps*

A quantidade de *fps* determina a **fluência** de exibição do vídeo e o ideal é que ela seja a maior possível. Existe, porém um efeito chamado *motion blur* provocado quando uma imagem é capturada pela câmera que consiste num “borramento” de elementos da cena que estejam em alta velocidade, quanto maior a velocidade, mais “borrado” o elemento fica. Este efeito é utilizado no cinema e na televisão e por isso, essas mídias utilizam os padrões de 24 *fps* para Cinema e 30 *fps* para TV. Pois, graças ao *motion blur* não percebemos mudanças bruscas de um quadro para outro.

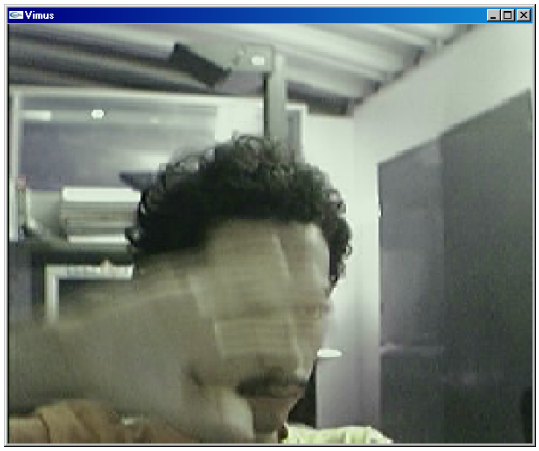

Figura 9. *Motion blur* da mão, provocado pela câmera de captura. Possibilita que 24 *fps* sejam suficientes para garantir a fluência dos movimentos.

Devido a essa padronização nas mídias principais, convencionou-se que a quantidade suficiente de *fps* para garantir fluência em uma animação é de 24. Chegou-se até a criar o mito absolutamente incorreto de que a visão humana não é capaz de perceber mais que 30 *fps*. Esta afirmação é totalmente equivocada, como defende [Brand, 2005],

pois até o momento não foi encontrado um limite prático para o número de quadros máximo que a visão humana é capaz de captar.

Para o caso de animações geradas e exibidas no computador como, desenhos 3D renderizados em tempo real, o *motion blur* ainda não é comumente usado devido ao alto custo de processamento necessário para produzir este efeito. Dessa forma, é perceptível a descontinuidade dos movimentos mesmo a 24 *fps*. Para reverter esse problema, placas de vídeo de última geração tentam elevar essa a taxa a números muitas vezes até maiores que 60 *fps* dependendo do número de objetos a serem renderizados. Entretanto só possível atingir essas taxas quando não há muitos objetos complexos na cena.

3. Tecnologias estudadas

Por ser um de seus principais objetivos, uma grande parte do tempo dedicado a esse projeto foi utilizada no estudo e aprendizado de algumas tecnologias específicas para este tipo de aplicação. A finalidade deste capítulo então é apresentar e registrar o que foi aprendido.

3.1 Captura de Áudio

O primeiro desafio tecnológico a ser enfrentado foi como fazer a captura de áudio que serviria de base para a síntese de imagens. Este problema foi inicialmente abordado pelo grupo que participei no projeto de um primeiro protótipo do que veio a se tornar o Vimus, chamado Guitarra Muda, para a disciplina de Computação Musical e Processamento de Som. Primeiramente, foram levantadas quais as *API*s de captura de áudio disponíveis para C++ na plataforma Windows. As duas possibilidades encontradas foram: *DirectShow*[DirectShow, 2005] e *Windows Multimedia API*[Windows Media API, 2005]. A primeira consiste num sistema multimídia (utilizado tanto para som, como imagem) orientado a filtros e conexões entre filtros e será discutida na sessão de captura de vídeo. A segunda fornece comandos de mais baixo nível e específicos para captura de som (inclusive seus comandos são utilizados pelo *DirectShow*). Buscando acesso direto a um nível mais baixo, para obter mais flexibilidade e até mesmo uma melhor compreensão do processo, já que o fim primordial é a aprendizagem, optamos pela própria *Windows Multimedia API*.

3.1.1 O processo de captura

Apesar de apresentarmos a seguir uma *API* específica e totalmente direcionada a um sistema operacional, o entendimento de seu funcionamento e de como devemos utilizá-la, implica num conhecimento que é muito útil e aplicável a várias outras *APIs* de

captura de som, pois as funções são parecidas e sua operação não varia muito de uma plataforma para outra.

A estrutura de dados objeto central dessa discussão é o ***buffer*** de áudio, que consiste numa seqüência de amostras de amplitude do som no decorrer do tempo, em uma determinada resolução – freqüência de amostra. Podemos identificar dois atores principais que participam do processo de captura: o nosso sistema e o dispositivo de captura. O primeiro fica todo momento lendo o ***buffer*** e o segundo escrevendo. Porém, deve existir um controle de acesso dessa área da memória que está sendo lida e escrita. Vejamos a seguir como este controle é feito tomando como exemplo a *API* do *Windows*.

3.1.2 *Windows Multimedia API*

A *Windows Multimedia API* é um conjunto de bibliotecas para C e C++, de arquitetura orientada a mensagens (como a maioria das *APIs* do *Windows*) que fornece uma interface entre o programador e os dispositivos multimídia do computador. As principais funções desta *API* que utilizamos são as cinco seguintes:

waveInOpen (HWAVEIN &device, WAVEFORMATEX format, ...)
Inicia a comunicação com o *hardware*, fazendo com que o ponteiro ***device*** aponte para o dispositivo de captura, que será configurado para enviar o som no formato digital especificado em ***format*** (ex.: 44100Hz, estéreo e 16 bits).
waveInPrepareHeader (HWAVEIN device, WAVEHDR buffer, ...)
Aloca o ***buffer*** para ser preenchido pelo dispositivo determinado por ***device***.
waveInAddBuffer (HWAVEIN device, WAVEHDR buffer, ...)
Envia o ***buffer*** para o dispositivo determinado por ***device***. Quando o ***buffer*** é totalmente preenchido a o bit WHDR_DONE é alterado no membro dwFlags da *struct* WAVEHDR.
waveInStart (HWAVEIN device)
Inicia captura.
waveInUnprepareHeader (HWAVEIN device, WAVEHDR buffer, ...)
Desaloca ***buffer***, liberando para que seja acessado por qualquer outro processo.

Podemos então, dividir o processo em duas partes: iniciando captura e lendo o ***buffer*** (o que é feito sempre que um buffer é preenchido). Estes dois processos estão ilustrados nos esquemas abaixo.

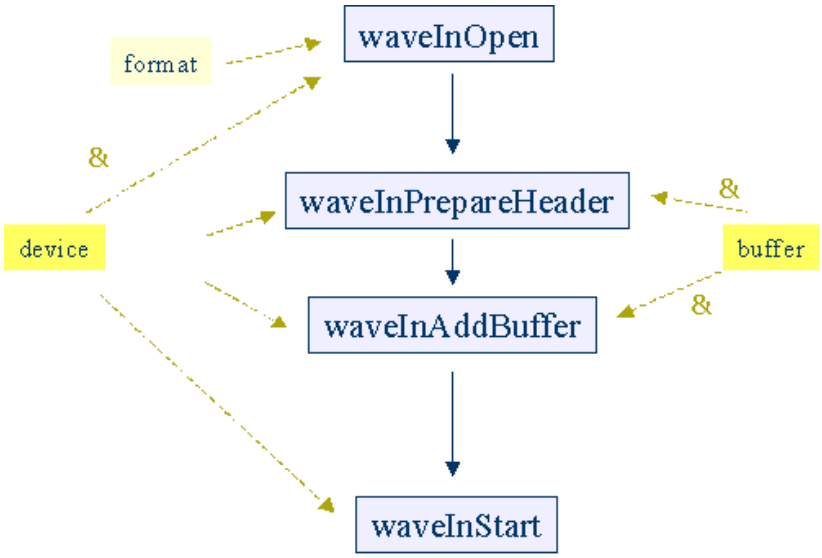

Figura 10. Iniciando captura.

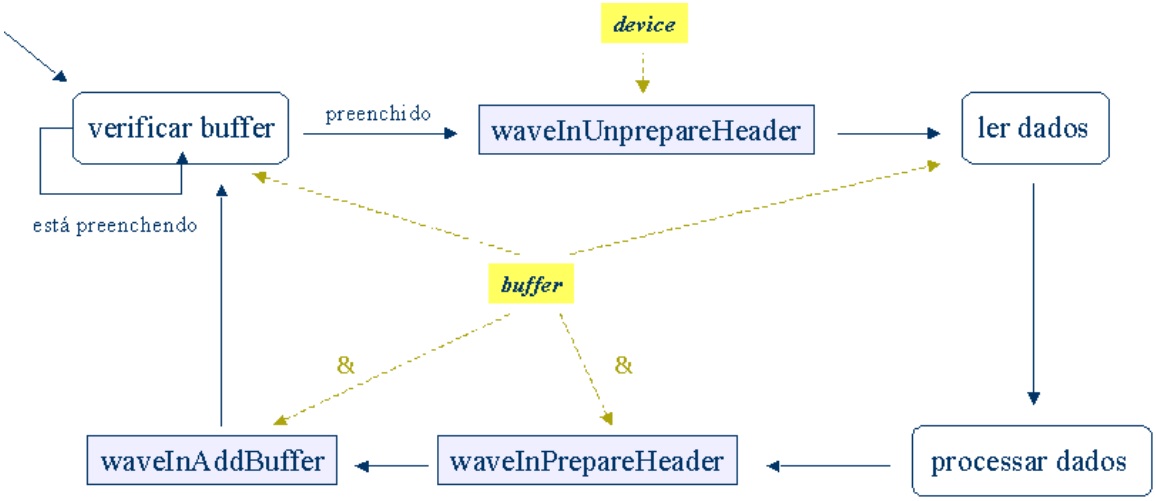

Figura 11. Verifica se o buffer foi preenchido. Em caso positivo libera o *buffer*, para seus dados serem lidos e o prepara novamente.

Como explicitado nos esquemas acima, o processo se dá da seguinte forma: após a inicialização da captura, o sistema operacional começa a encher o ***buffer*** adicionado.

Poderá existir um processo na aplicação que de quando em quando verifica se este **buffer** foi preenchido, em caso positivo, lê os novos dados e prepara novamente o ***buffer*** para ser novamente adicionado. Pode-se perceber que este processo fica mais eficiente se usarmos mais de um ***buffer*** para que, enquanto um está sendo preenchido, outro já possa ser lido. Isto é possível, adicionando-se mais de um buffer utilizando o comando ***waveInAddBuffer()***. Mais eficiente ainda poderia ser a utilização de uma fila circular de ***buffers*** que são todos (exceto o último) adicionados durante a inicialização e após começada a captura, quando um ***buffer*** estiver completo, os outros seguintes, já estão completos ou prestes a serem completados.

Porém, é necessário analisar a necessidade real de uma quantidade maior de ***buffers*** que depende do tipo da aplicação. Em um cliente de streaming de rádio pela Internet, por exemplo, o que mais importa é que a música não seja interrompida. É melhor que o processo de *buffering* demore e a música seja tocada sem interrupções até o final, do que o contrário. Por isso a quantidade de *buffers* deve ser proporcional à lentidão da conexão com o servidor, ou seja, quanto mais lenta a conexão, maior o tempo de *buffering* para evitar interrupções no meio da música.

Entretanto, isso não se aplica ao nosso sistema, pois não tocamos a música e por tanto não há problemas se deixarmos de processar algum pedaço da música não precisando de muitos *buffers*. Na verdade, o mais importante é o tempo de resposta do sistema à entrada de som, ou seja, assim que o áudio alterar, a imagem deve alterar o mais rápido possível. Esta discussão será retomada em **7.2 Tempo de resposta ao estímulo sonoro** no capítulo de análise dos resultados do sistema.

3.2 Captura de Vídeo

Independente de qual *API* seja usada para captura de vídeo, o requisito básico para o Vimus é que seja possível obter acesso ao último *frame* capturado o mais rápido possível. Nas primeiras pesquisas em busca de *APIs* para C++ que fornecessem acesso à sistemas de *hardware* de captura, foram novamente encontradas referências à *Windows Multimedia API*, utilizada por alguns *softwares* de vídeo (inclusive alguns maciçamente utilizados como o [*VirtualDub*, 2005]. As funções, estruturas e constantes relacionadas à captura estão mais especificamente agrupadas na biblioteca "*vfw.h*" (*Video for Windows*) [*VfW*, 2005]. O problema desta *API*, é que se trata de uma implementação muito antiga, dos tempos do Windows 3.11, e por isso opera em 16 bits, existindo uma espécie de conversor "vfw32.dll" que intermedia sua operação com o Windows. Isso limita a capacidade e velocidade de um sistema utilizando essa biblioteca.

Mas o problema mais sério, especificamente para sua utilização no Vimus é que esta *API* simplesmente não dá acesso direto ao *frame* capturado. A única forma de obter o *frame* enquanto o mesmo ainda está na memória é através de uma janela na qual o vídeo já é automaticamente desenhado e exibido na tela do computador, mas não é fornecido um ponteiro ao *frame* desenhado. Por isso a *VfW* é dita uma API de captura de vídeo "baseada em janelas" [Davies, 2003]. Podemos apenas criar ou destruir essa janela, como um elemento gráfico fechado, do qual não temos acesso ao *buffer* do *bitmap* que está sendo exibido.

O que explica o fato de que mesmo com essas diversas limitações esta *API* ainda é tão utilizada é que, *softwares* como *VirtualDub*, tem como finalidade principal, a captura de vídeo para arquivos (função que a *VfW* executa muito bem) e não para processamento e exibição na tela em tempo real. Ainda assim, existe uma tentativa de adaptação da *VfW*, que pode ser encontrada em "http://www.codeguru.com/Cpp/G-M/multimedia/audio/article.php/c1579", para programas que queiram utilizá-la para captura e exibição em tempo real. Alguns testes, porém, mostraram que o desempenho (quase meio segundo de intervalo de um *frame* para outro) ainda é muito abaixo do necessário para uma aplicação como o ViMus.

3.2.1 *DirectShow*

A insistência dos desenvolvedores em utilizar o *VfW* ocorreu também, provavelmente, apenas para evitar mudanças de código, pois já em 1996, a Microsoft lançou a primeira versão do *ActiveMovie*, que fornecia um controle *ActiveX* para executar vídeos MPEG1, AVI e QuickTime, assim como arquivos de áudio no *Windows*. Junto, foi obviamente desenvolvida uma *SDK* que provia ferramentas e informações para desenvolvimento de filtros e aplicações. Porém, para a segunda versão lançada um ano depois, o nome foi mudado para *DirectShow* para ter um esquema de nomes coerente com as outras tecnologias de multimídias recém lançadas (DirectX, DirectSound). Geraint Davies, ex-funcionário da Microsoft, e nessa época, líder do projeto *Quartz* - como era internamente conhecido o ActiveMovie - disponibiliza na página eletrônica [Davies, 2003] de sua própria empresa uma série de artigos sobre o *DirectShow* e captura de vídeo em geral.

A escolha do *DirectShow* foi determinada pelo desempenho incomparavelmente superior à VfW. Apesar disso, é uma *API* muito mais complexa, sobretudo para o programador sem experiência em *COM* - *Component Object Model*. *COM* consiste em "um mecanismo do *Windows* para identificação e comunicação entre componentes de aplicações de uma maneira genérica" [Davies, 2005]. Esse mecanismo é usado pelo desenvolvedor para criar componentes de *software* reusáveis, interligar componentes para construir novas aplicações e utilizar serviços do *Windows*. Os filtros do *DirectShow* são objetos *COM* dos quais podemos utilizar suas interfaces para criar nossos próprios filtros. Para auxiliar testes e experimentos de possíveis conexões entre os filtros, existe a ferramenta *GraphEdit* do conjunto de utiliários do *DirectX*. A Figura abaixo apresenta um esquema de filtros usado para capturar vídeo de uma *WebCam*, montado no *GraphEdit* e usado para efetuar os testes de desempenho da captura do *DirectShow*.

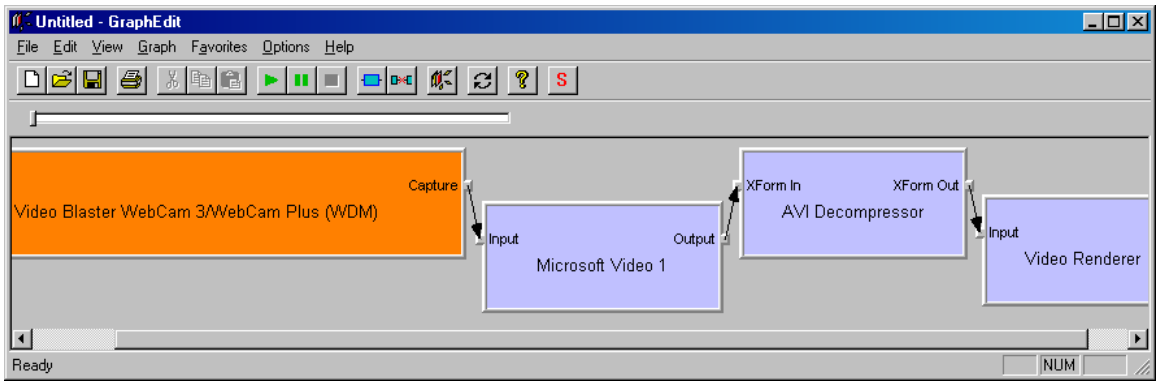

Figura 12. Grafo de filtros para captura de vídeo visto no GraphEdit (DirectShow).

Observe que o ponto inicial do grafo na figura acima é a origem do vídeo: o *hardware* de captura e o ponto final é um **filtro renderizador**. Ao executar esse grafo, uma janela com suporte a *DirectX* é aberta e o vídeo vindo da câmera é exibido. Assim como esse grafo foi construído no GraphEdit, pode ser instanciado em nosso software, carregando-se esses filtros no código através de chamadas utilizando *COM*. Porém, como ficou claro quando apresentamos a *VfW*, não nos interessa uma janela exibindo o vídeo, o que nos interessa é o ponteiro para o último *frame* capturado. Porém, não existe um filtro que faça isso entre os que já vêm implementados com a biblioteca *DirectShow*: um filtro que receba uma entrada AVI e armazene em um *buffer*, cada *frame* assim que recebido.

Foi necessário ser implementado um novo *VideoRenderer* com um atributo que é um ponteiro para o útlimo *frame* capturado. Este novo filtro renderizador, ao qual foi dado o nome externo de ***FrameGrabber,*** implementa a classe abstrata *COM* chamada *CbaseVideoRenderer* do conjunto de objetos *COM* disponibilizados pelo *DirectShow*. Como esta é um uma classe COM, pode ser instanciada em qualquer aplicação que seja executada no momento em que está carregada. E para ser identificada individualmente pelo sistema operacional, é dado um número como explicitado na primeira linha do código abaixo, da interface (*.h*) de nosso novo filtro.

```
//------------------------------------------------------------------------
// Define GUID for FrameGrabber used in Vimus
// {D7A2CE2F-8221-4b80-B086-B795D9C845F5}
//------------------------------------------------------------------------
struct __declspec(uuid("{D7A2CE2F-8221-4b80-B086-B795D9C845F5}"))
CLSID_FrameGrabber;

class DirectShowVideoRenderer : public CBaseVideoRenderer
{
public:
    DirectShowVideoRenderer(LPUNKNOWN pUnk, HRESULT *phr);
    ~DirectShowVideoRenderer();

      unsigned char * getFrameBuffer();
      VIDEOINFO * getVideoInfo();

private:
    HRESULT CheckMediaType(const CMediaType *pmt );
    HRESULT SetMediaType(const CMediaType *pmt );
    HRESULT DoRenderSample(IMediaSample *pMediaSample);
      LONG m_lVidWidth; // Video width
      LONG m_lVidHeight;// Video Height
      LONG m_lVidPitch; // Video Pitch

      unsigned char * frameBuffer;                  <--frameBuffer

      VIDEOINFO * videoInfo;
};
```

O método ***DoRenderSample()*** que é chamado pelo *DirectShow* todo momento em que um *frame* é capturado, simplesmente armazena o ponteiro para esse *bitmap* em ***frameBuffer***. Finalmente, a qualquer momento, ***getFrameBuffer()*** pode ser chamado para obtermos acesso ao *buffer*.

3.3 Computação Gráfica

Por se tratar de um sistema sintetizador de imagens inclusive em 3 dimensões, a escolha da *API* gráfica é uma das mais importantes. Ela é responsável por fornecer todos os comandos gráficos básicos necessários para o desenho das imagens criadas. As duas opções mais plausíveis para o *Windows* são *OpenGL* [OpenGL, 2005] ou *DirectX*[DirectX, 2005]**.**

A questão sobre qual das duas *APIs* é a melhor rende dezenas de páginas em fóruns de desenvolvedores e sua resposta não é sempre a mesma, pois depende dos requisitos de cada sistema. Existe um senso comum de que DirectX tem melhor desempenho. Porém, isso depende do tipo de operação que a aplicação faz e do *hardware* utilizado. Para placas *ATI* [ATI, 2005], por exemplo, provavelmente DirectX terá um melhor desempenho realmente, o que não é verdade para placas *Nvidia* [Nvidia, 2005]. É provável, porém, que este senso comum tenha se estabelecido, por que inicialmente a própria *Microsoft* desenvolveu os *drivers* de *OpenGL* e supõe-se que não houve um esforço por parte da empresa de desenvolvê-los tão eficientes quanto os *drivers* do *DirectX*. Posteriormente, a *Silicon Graphics* lançou suas versões dos *drivers* de OpenGL para Windows e a diferença entre uma API agora varia pouco dependendo do *hardware* de vídeo.

OpenGL foi escolhida para o projeto Vimus pelas seguintes razões:

- **Portabilidade:** planeja-se seriamente desenvolver-se no futuro, uma versão para Linux e Macintosh. Ambas as plataformas têm suporte a *OpenGL*.
- **Familiaridade:** eu já havia trabalhado com OpenGL em projetos anteriores.
- **Complexidade:** a curva de aprendizado de OpenGL é melhor que a de DirectX.
- **Curvas paramétricas:** *OpenGL* oferece ótimo suporte a curvas paramétricas: curvas construídas a partir de alguns poucos parâmetros chamados pontos de controle.

3.3.1 OpenGL

OpenGL significa *Open Graphics Library* e surgiu como uma iniciativa da empresa Silicon Graphics [SGI, 2005] de criar uma API gráfica 2D/3D independente de plataforma. O projeto ganhou força com a montagem de um consórcio de várias empresas da área que definem uma especificação de como implementar essa API gráfica para grupos ou indústrias de dispositivos gráficos e sistemas operacionais. Da mesma forma os programadores e desenvolvedores consultam essa especificação para aprender como utilizar a *API* em sua aplicação.

A operação de OpenGL se dá através de mensagens contendo instruções gráficas enviadas para o sistema operacional que pode interpretá-las e executá-las ou repassar diretamente para o hardware gráfico, caso este tenha suporte a *OpenGL*. O que torna a API independente de plataforma é o fato da sintaxe e efeito dessas mensagens ser exatamente igual em qualquer plataforma, pois todas implementações devem seguir a especificação padrão.

Veja a seguir um exemplo de mensagens *OpenGL* e seu efeito:

```
glTranslatef(0.0,0.0,5.0);          //translada em 5.0 ao eixo Z
glRotatef (this->rot++*0.1f, 1.0f, 0.0f, 0.0f); //rotação X
glRotatef (this->rot*0.1f, 0.0f, 1.0f, 0.0f);   //rotação Y
glRotatef (this->rot*0.1f, 0.0f, 0.0f, 1.0f);   //rotação Z
glBegin(GL_QUADS);
      // Begin Drawing A Cube
      // Front Face
      glNormal3f( 0.0f, 0.0f, 0.5f);
      glVertex3f(-1.0f, -1.0f,  1.0f);
      glVertex3f( 1.0f, -1.0f,  1.0f);
      glVertex3f( 1.0f,  1.0f,  1.0f);
      glVertex3f(-1.0f,  1.0f,  1.0f);
      // Back Face
      glNormal3f( 0.0f, 0.0f,-0.5f);
      glVertex3f(-1.0f, -1.0f, -1.0f);
      glVertex3f(-1.0f,  1.0f, -1.0f);
      glVertex3f( 1.0f,  1.0f, -1.0f);
      glVertex3f( 1.0f, -1.0f, -1.0f);
      // Top Face
      glNormal3f( 0.0f, 0.5f, 0.0f);
      glVertex3f(-1.0f,  1.0f, -1.0f);
      glVertex3f(-1.0f,  1.0f,  1.0f);
      glVertex3f( 1.0f,  1.0f,  1.0f);
      glVertex3f( 1.0f,  1.0f, -1.0f);
      // Bottom Face
      glNormal3f( 0.0f,-0.5f, 0.0f);
      glVertex3f(-1.0f, -1.0f, -1.0f);
```

```
        glVertex3f( 1.0f, -1.0f, -1.0f);
        glVertex3f( 1.0f, -1.0f,  1.0f);
        glVertex3f(-1.0f, -1.0f,  1.0f);
        // Right Face
        glNormal3f( 0.5f, 0.0f, 0.0f);
        glVertex3f( 1.0f, -1.0f, -1.0f);
        glVertex3f( 1.0f,  1.0f, -1.0f);
        glVertex3f( 1.0f,  1.0f,  1.0f);
        glVertex3f( 1.0f, -1.0f,  1.0f);
        // Left Face
        glNormal3f(-0.5f, 0.0f, 0.0f);
        glVertex3f(-1.0f, -1.0f, -1.0f);
        glVertex3f(-1.0f, -1.0f,  1.0f);
        glVertex3f(-1.0f,  1.0f,  1.0f);
        glVertex3f(-1.0f,  1.0f, -1.0f);
glEnd();
```

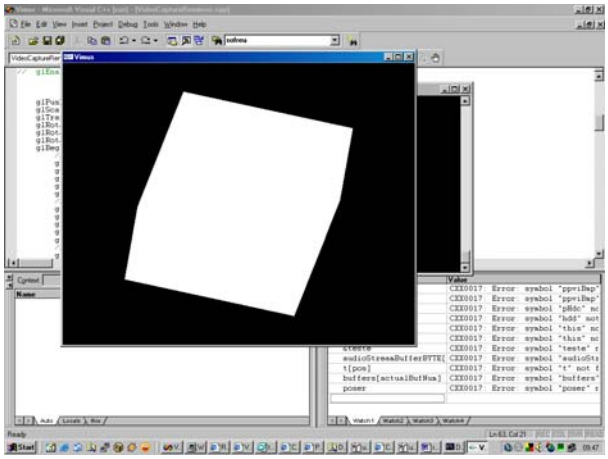

Figura 13. Desenhando cubo de 6 faces.

3.4 Sistema de janelas e tratamento de eventos

Por se tratar de uma aplicação gráfica, seu ambiente de execução é um sistema operacional com modo gráfico (como Microsoft Windows, ou StartX no caso de uma implementação em Linux) e por isso poderá fazer uso do sistema de janelas e eventos de mouse e teclado. Existem vários sistemas do tipo disponíveis, porém o próprio grupo de especificação de *OpenGL* criou uma extensão do mesmo chamada *GLUT* [Kilgard, 1996], que fornece uma série de funções de acesso ao sistema de janelas e de tratamento de eventos do sistema operacional, seja ele qual for. Ou seja, utilizando GLUT, o código da interface gráfica fica facilmente portável para qualquer outra plataforma.

3.4.1 GLUT

Além dos comandos triviais de criação de janela, com opções para tela cheia, GLUT possui um sistema de tratamento de eventos de entrada do usuário (teclado e mouse). Para utilizar este sistema deve-se criar funções estáticas especificando o que fazer quando algum determinado evento for disparado. Essas funções são “instaladas” no GLUT e são chamadas assim que o evento ocorre. O único problema desse sistema é que funções estáticas não podem fazer chamadas a métodos de objetos não estáticos. A solução para isso é descrita em **5.1.1 Problema das funções *callback*** na sessão sobre o módulo de interface gráfica do ViMus.

4. Sistema Vimus

Neste capítulo será finalmente apresentado o produto de *software* resultante da pesquisa realizada.

4.1 Processo de desenvolvimento

O processo de desenvolvimento do ViMus se deu através das seguintes etapas: definição de requisitos, estudo e escolha de tecnologias, implementação de protótipos funcionais e independentes (como exibição de vídeo capturado utilizando OpenGL, por exemplo), testes dos protótipos, modelagem, implementação do sistema integrando os módulos antes separados e agora reimplementados utilizando classes de forma bem estruturada e organizada, respeitando a arquitetura definida na modelagem. Este processo, que poderíamos ver grosseiramente como uma mistura de *extreme programming* e processo em espiral, já se tornou comum em meus projetos de *software* e é resultado de experiências práticas e teóricas da graduação.

4.2 Arquitetura

Objetivando facilitar a implementação, bem como minimizar os custos de manutenção, a arquitetura do sistema teve importância fundamental para o sucesso na realização deste projeto. Uma boa arquitetura é necessária para o desenvolvedor lidar com a complexidade e o tamanho de sistemas, fazendo uso de princípios como ocultação de informações, modularidade, flexibilidade, etc [Mendes, 2002]. Neste capítulo apresentaremos primeiramente como a arquitetura do sistema ViMus foi concebida e em seguida “navegaremos” pelos vários módulos, suas ramificações e inter-relações.

4.2.1 Processo de concepção

A concepção da arquitetura foi realizada sem muitas dificuldades, graças à decisão de projeto de primeiro fazer independentemente os protótipos de estudo das tecnologias utilizadas, antes de iniciar o sistema como um todo. Pois, no processo de implementação desses protótipos, foi adquirido experiência e conhecimentos das tecnologias (como OpenGL, captura de imagem, captura de vídeo) que influenciaram nas escolhas de arquitetura.

Assim, com os protótipos finalizados, e tendo adquirido certo domínio das tecnologias, partiu-se para a modelagem do sistema que se deu da seguinte forma:

4.2.1.1 *Brainstorm* de entidades

A primeira atividade foi escrever a lápis de forma caótica e livre, em páginas em branco, nomes de possíveis módulos, classes básicas ou entidades que fossem sendo pensados no momento e que, possivelmente, iriam existir no sistema, independente de suas inter-relações, e até de sua própria natureza e real necessidade dos mesmos.

4.2.1.2 Classificação em grupos de afinidade

Ao fim do levantamento de entidades, tinha-se dezenas de nomes correspondentes a entidades não bem definidas ainda, tendo-se apenas uma vaga idéia do que cada nome representaria, apenas pela semântica das palavras que os compunham. Uma primeira organização dada a esse conjunto, foi o agrupamento desses elementos de acordo com sua natureza, sua propriedade “relacionado a ...”. Por exemplo, os nomes *Audio*, *Audio Analyzer*, *AudioSpectrumBuffer*, *AudioStreamBuffer*, *AudioCapture*, são todos obviamente relacionados a áudio e mais especificamente à entrada de áudio e por tanto pertenceram a um mesmo grupo.

4.2.1.3 Estabelecimento de hierarquias

Após a definição dos grupos, já foi possível identificar relações de hierarquias entre elementos e adicionar ou eliminar alguns nomes redundantes ou desnecessários. Por exemplo: para satisfazer o escopo atual do projeto, era necessário apenas trabalhar com entrada de áudio, assim, a entidade *Audio* que encapsula todas as rotinas de áudio teve o nome trocado para *AudioInput* e foi relacionada como "pai" das outras entidades de *Audio* como *AudioCapture* e *AudioAnalyser*.

4.2.1.4 Estabelecimento de dependências funcionais

Além das relações hierárquicas entre as entidades, foram identificadas suas relações de interdependência funcional. Essa fase, em que também foram eliminados nomes e criados outros, foi a que deu mais forma ao diagrama de classes, pois nela foram detectadas quais as reais necessidades de uma entidade ter acesso à outra. Entre as principais idéias surgidas, está a criação de uma entidade agregadora: o núcleo do sistema ViMus que seria responsável por gerenciar as entradas de áudio, vídeo, definições de efeitos do usuário, configurações e a partir disso ordenar a síntese da imagem final (na verdade, uma seqüência de imagens – a cada intervalo mínimo de tempo é sintetizada uma nova imagem resultando em uma animação). O nome dessa entidade é *Mixer* e será discutida em **5.5** *Mixer*.

4.2.2 Arquitetura

Para facilitar a compreensão, nossa arquitetura será apresentada utilizando uma abordagem *top-down*, ou seja, partindo de uma visão global do sistema e chegando a uma visão mais detalhada de cada módulo. Como já foi explicitado em **4.1 Processo de desenvolvimento**, essa não foi exatamente a abordagem utilizada na construção do sistema, porém, para fins didáticos neste relatório, a consideramos mais interessante por ser mais intuitiva e partir da visão do usuário.

4.2.2.1 Visão geral do sistema

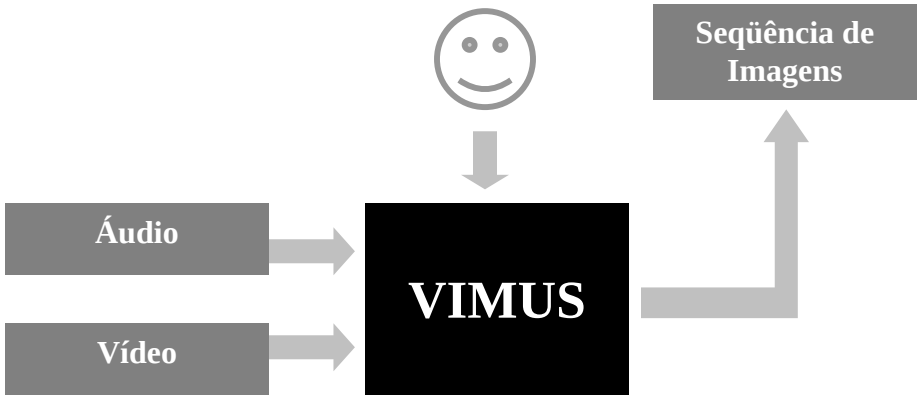

Figura 14. Sistema ViMus na visão do usuário.

Para o usuário, o sistema ViMus é simplesmente uma "máquina" que, a partir do momento que é ligada, fica recebendo como entrada uma *stream* de áudio e uma *stream* de vídeo, parâmetros do usuário, e gerando como saída uma seqüência de imagens. Essa seqüência de imagens é gerada de acordo com o áudio e o vídeo que entram. Já temos então, entre outros, 4 elementos fundamentais de nosso sistema: o usuário, as duas *streams* de áudio e vídeo e o *software* ViMus. Desses, apenas o usuário e o *software* podem executar alguma ação, as *streams* são apenas dados que chamaremos a partir de agora ***AudioInputStream*** e ***VideoInputStream***. A Figura 2 apresenta esta visão global do sistema.

A ação que o usuário executa é dar ordens ao *software* dizendo **como** ele quer que a seqüência de imagens seja gerada. Podemos perceber então que o *software* precisa fornecer uma interface para receber entradas de dados do usuário e exibir a saída – a animação que está sendo gerada. Essa interface assim como outros módulos serão apresentados a seguir.

4.2.2.2 Visão dos módulos

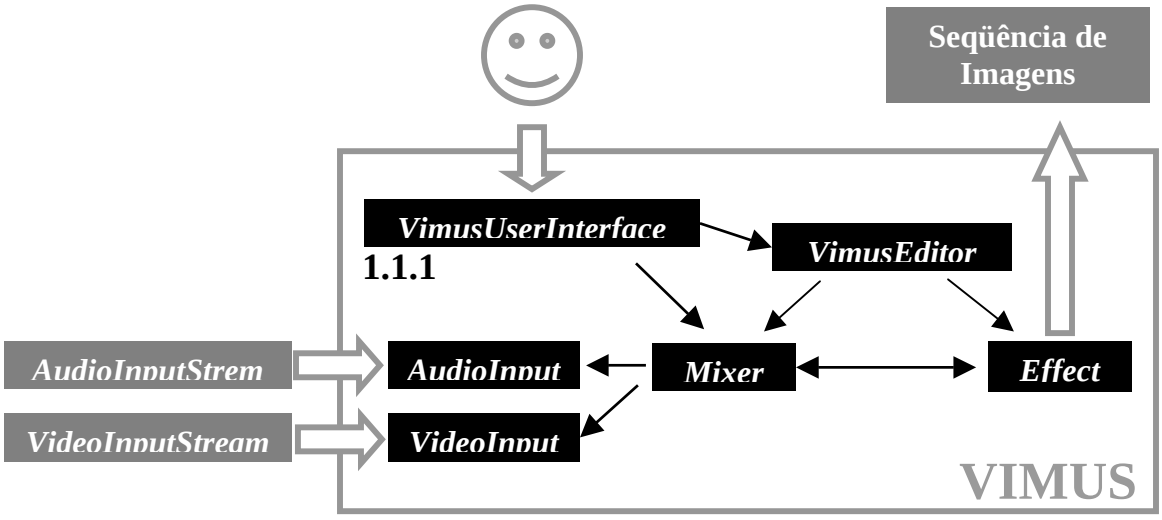

Figura 15. Módulos principais.

Agora que já vimos o ViMus visto de fora, como uma "caixa preta" e sua interação com o mundo externo, vamos abrir a caixa e ver seus principais componentes. Para entender o porquê da existência de cada um desses componentes, depois de respondido a pergunta **o quê** o sistema faz na sessão anterior, vamos começar a responder **o que é necessário** para que isso seja feito e com o detalhamento de cada módulos, gradativamente, entenderemos melhor **como** isso é feito.

O que é necessário para que o usuário ordene ao ViMus que gere uma imagem de uma determinada maneira ou de outra? E como o usuário poderá ver a animação que está sendo gerada? Para isso o sistema precisa de uma interface orientada a eventos, capaz de detectar entradas do usuário (por teclado, *mouse*, etc.). E para a saída, precisamos de uma plataforma gráfica com acesso ao dispositivo de vídeo do computador para gerar as imagens rapidamente ou salvar diretamente em um arquivo. Essas funcionalidades serão agrupadas na entidade ***VimusUserInterface***.

O que o usuário poderá mudar através da ***VimusUserInterface*** para que as imagens sejam geradas de forma diferente? Para isso foi criado o conceito de efeitos (***Effect***). Um efeito tem a capacidade de desenhar algo, inclusive alterando um desenho que já foi feito. Este desenho pode ser sintetizado a partir de funções matemáticas de

linhas ou superfícies 3D, assim como a partir de um vídeo vindo de um dispositivo de captura (***VideoInputStream***). Além disso, um efeito é capaz de utilizar informações obtidas a partir da entrada de áudio, como ***AudioInputStream*** como parâmetros para essas funções matemáticas que sintetizem imagens ou que alterem imagens já existentes, provocando, por exemplo, efeitos tradicionais de vídeo como *blur*, detecção de borda, inversão de cores, etc. Para criar, alterar, remover, mudar a ordem dos efeitos, foi concebido o ***VimusEditor***.

Para obter as informações de áudio, é necessária uma infra-estrutura de software capaz de se comunicar com os dispositivos de áudio para capturar da forma mais eficiente as informações sonoras que servirão de parâmetros para os efeitos. Todas as rotinas e entidades envolvidas neste processo estão encapsuladas em ***AudioInput***. Da mesma forma que a infra-estrutura de vídeo está encapsulada em ***VideoInput***.

Está faltando o elemento que intermedia a interação entre todos essas entidades citadas. Este elemento poderia ser a própria ***VimusUserInterface***, porém visando uma maior modularidade e facilidades em futuras implementações para outras plataformas, optou-se pela criação da ***Mixer***. Esta entidade recebe as ordens da ***VimusUserInterface***, como começar a exibir efeitos, trocar de efeito, mudança de parâmetros em efeitos que possibilitem interações com o usuário em tempo de execução – como rotação da câmera no caso de um efeito 3D, por exemplo. Além disso, é responsável por deixar todas as informações de entradas necessárias disponíveis a qualquer efeito que queira utilizá-las.

4.2.2.3 Visão detalhada dos módulos

A Figura abaixo representa esquematicamente um diagrama de classes simplificado do sistema. Em seguida ela será explicada ao discutirmos um pouco do funcionamento de cada um dos módulos no próximo capítulo.

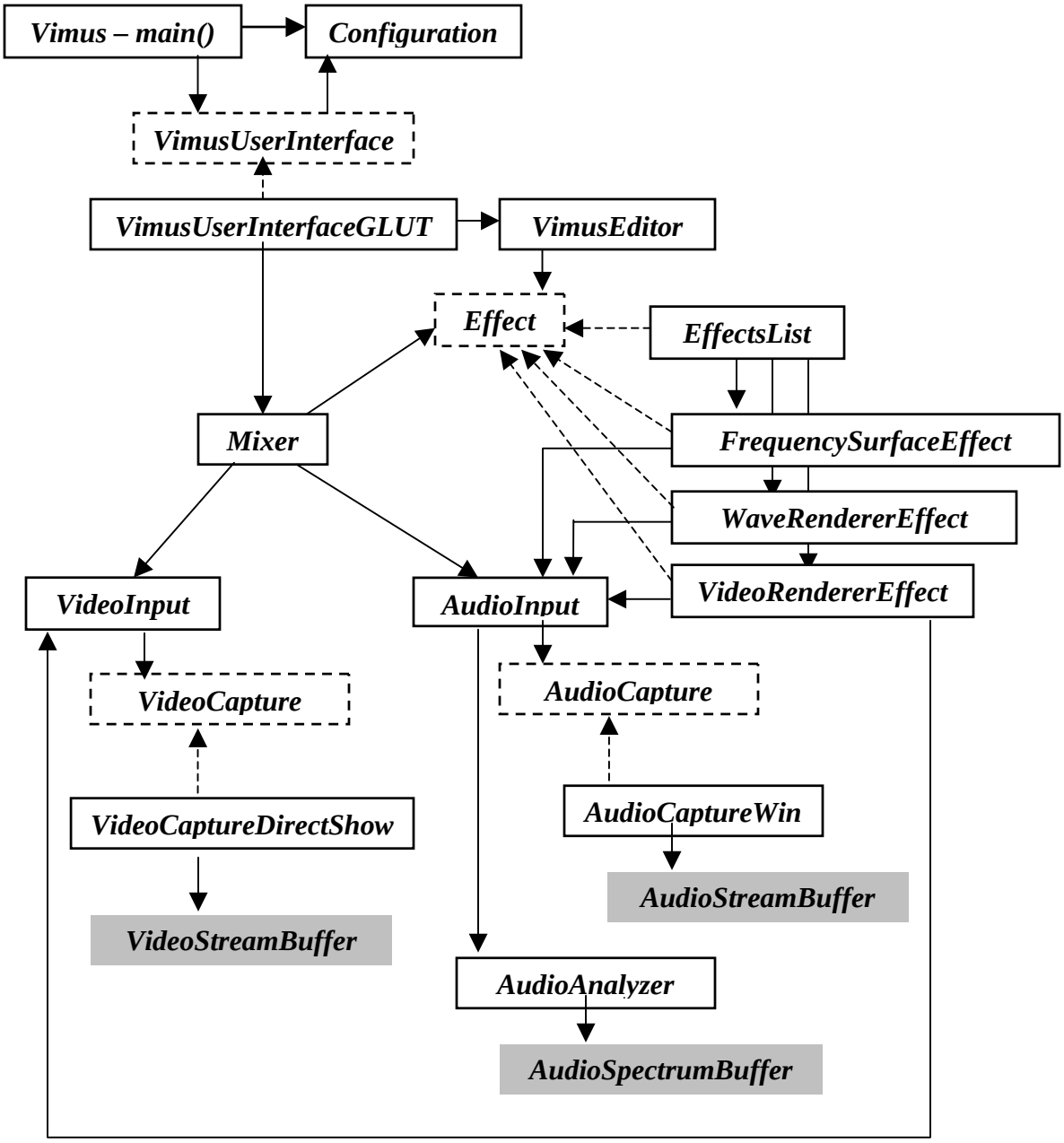

Figura 16. Arquitetura detalhada.

5. Detalhamento dos Módulos

5.1 *VimusUserInterface*

Visando facilidades de reimplementação em outras plataformas e para obter maior modularidade e extensibilidade esta entidade é representada por uma classe abstrata que define métodos comuns a qualquer implementação possível para este módulo. Isso porque, existem várias possibilidades de reimplementação dessa interface. Desde uma interface muito mais simples, pela linha de comando de um sistema operacional sem interface gráfica. Neste caso, as imagens geradas pelo sistema poderiam ser direcionadas a um arquivo, ou a um outro sistema que pudesse exibi-las. Até interfaces remotas, utilizando o paradigma cliente e servidor possibilitando distribuição de processamento. Poderia-se, inclusive, utilizar-se para comunicação neste caso, diferentes tipos de protocolos, até mesmo os já utilizados em música como MIDI (que inclusive é usado para controlar iluminação em concertos e peças de teatro, por exemplo). Além disso, no caso das implementações gráficas, existem diferentes possibilidades como o uso de OpenGL, DirectX para a exibição das imagens e ainda diferentes sistemas de janelas que variam de um sistema operacional para outro.

Para este trabalho de graduação, optamos pelo uso de OpenGL como plataforma gráfica e sua extensão GLUT para o sistemas de janelas e de tratamento de eventos. Os motivos já foram explicitados em **3.3.1 OpenGL**. Porém, caso se queira desenvolver uma versão do sistema utilizando *DirectX*, por exemplo, bastaria implementar a interface ***VimusUserInterface*** utilizando o sistema de eventos e a máquina gráfica do *DirectX* e reimplementar os métodos de desenho dos efeitos. Existe ainda a possibilidade mista utilizando *buffers* de retorno: alguns efeitos utilizando *OpenGL*, outros utilizando *DirectX*.

O diagrama abaixo apresenta as entidades com as quais a ***VimusUserInterface*** (***VUI***) se relaciona. Ela é instanciada pela aplicação principal ***Vimus***, que possui apenas uma função *main* para a carregar a interface e possui acesso ao conjunto de configurações do sistema ***Configuration*** que são recuperadas e salvas em arquivo. Para executar os

efeitos, ou alterá-los, a ***VUI*** tem acesso aos métodos de desenho da ***Mixer***, que por sua vez chama os métodos de desenho dos efeitos.

5.1.1 Problema das funções *callback*

Este caso explicita o fato de que peculiaridades de uma tecnologia escolhida como plataforma pode influenciar na arquitetura do sistema. Como foi visto em **3.4.1 GLUT**, para utilizar o controle de eventos é necessário enviar as funções que se deseje que sejam executadas quando ocorrer cada evento. Por exemplo, para que se execute alguma ação a partir de uma tecla que o usuário pressione, é necessário "instalar" uma função para tratar eventos do teclado, através do comando:

glutKeyboardFunc(funcaoDesejada).

O problema é que esta função de *callback* a ser enviada deve ser estática. Funções estáticas não podem fazer chamadas diretamente a métodos de objetos instanciados. Isso ocorre porque essas funções não têm como acessar esses métodos, pois não podem possuir um ponteiro para objetos estanciados, apenas para classes.

Isso representa um problema de arquitetura para qualquer sistema gráfico dinâmico que utilize GLUT e seus gráficos são objetos dinâmicos, ou são gerados por objetos dinâmicos, que tenham suas propriedades alteradas por alguma tecla do usuário.

No caso de nosso sistema, a função estática instalada através de glutIdleFunc() precisa chamar os métodos de desenho dos objetos efeitos (***Effect***) por exemplo. Porém esses objetos, obviamente, não são estáticos e por tanto, não podem ser chamados pela função instalada (que é estática).

Problema:

VimusUserInterfaceGLUT.h

```
...
class VimusUserInterfaceGLUT : public VimusUserInterface
{
public:
      VimusUserInterfaceGLUT();
      ~VimusUserInterfaceGLUT();

      static void idleFunc()       <----------- nossa idleFunc

private:
      Mixer * mixer;
...
}
```

VimusUserInterfaceGLUT.cpp

```
...
VimusUserInterfaceGLUT :: idleFunc ()
{
      this->mixer->draw();         <------ ERRO de compilação:
                                           idleFunc (estática)
                                           não tem acesso a
                                           mixer (dinâmico).
}
...
```

A solução para esse problema sugerida pelo Prof. Lew Hitchner [Hitchner, 2004] consiste em criar um atributo na classe ***VimusUserInterfaceGLUT*** (nossa implementação GLUT de ***VimusUserInterface***) que aponte para uma classe do tipo ***VimusUserInterfaceGLUT*** e construir um objeto *singleton* eglobal no arquivo de implementação (*.cpp). Agora os métodos representando as funções de *callback* estáticas instaladas no GLUT têm acesso à instância *singleton* de ***VimusUserInterfaceGLUT*** e podem chamar qualquer método deste objeto (inclusive os não estáticos).

Solução:

VimusUserInterfaceGLUT.h

```
...
class VimusUserInterfaceGLUT : public VimusUserInterface
{
public:
      VimusUserInterfaceGLUT();
      ~VimusUserInterfaceGLUT();

      static VimusUserInterfaceGLUT * vimusUIPtr;  <-ponteiro
                                                     estático

      void idleFunc();  <----------- nossa idleFunc não estática

      inline static void idleFuncStatic()   <- essa será instalada
      {
            this->vimusUIPtr->idleFunc() <- chama a não estática
      }                                     através do ponteiro
                                            estático que aponta
                                            para o singleton.
private:
      Mixer * mixer;
...
}
```

VimusUserInterfaceGLUT.cpp

```
      VimusUserInterfaceGLUT * VimusUserInterfaceGLUT::vimusUIPtr = new
VimusUserInterfaceGLUT();          <---- instanciamento do singleton

      ...
      VimusUserInterfaceGLUT :: idleFunc ()
      {
            this->vimusUIPtr->mixer->draw(); <---- agora sem erro!
      }
      ...
```

5.2 AudioInput

Esta entidade é responsável por obter todas as informações necessárias à aplicação relativas à entrada de áudio. Ela encapsula diversas rotinas de captura de áudio e também análise de áudio, como a transformada de Fourier. A forma como está organizada faz com que seja independente do modo como se captura, ou seja, independente da *API* de som que se queira utilizar, e de como se faz o processamento digital de sinal - como Fourier, por exemplo.

Na Figura abaixo temos o trecho do diagrama relativo a ***AudioInput***. ***AudioConfiguration***, que deve ser passada para ***AudioInput*** no momento de sua criação,

definindo as configurações de áudio necessárias para efetuar a captura e a análise espectral, como o número de *buffers*, freqüência, número de canais, etc.

As duas classes coordenadas por ***AudioInput*** para gerar as informações necessárias aos efeitos são ***AudioCapture***, responsável pela captura e ***AudioAnalyzer*** responsável por qualquer tipo de análise que se faça no áudio. Os efeitos podem ter acesso a elas através da ***Mixer***.

O principal método de ***AudioCapture*** utilizado pelos efeitos é ***getSample(int pos)***, que obtém uma amostra em uma dada posição (***pos***) do último *buffer* capturado. Se, por exemplo, desenharmos uma linha ligando vários pontos dos quais a altura é determinada por ***getSample(pos)*** e a abcissa por ***pos*** teremos o desenho da onda "fotografada" no *buffer*.

O principal método de ***AudioAnalyzer*** é o ***getAmp(int freq)***, que obtém a amplitude do som em uma dada amostra de freqüência. Isto pode ser usado para gerar um gráfico do espectro, ou em efeitos, detectando a amplitude do som de entrada para graves, médios e agudos separadamente. Isso possibilita que os efeitos visuais representem melhor um som que esteja sendo capturado.

5.3 *VideoInput*

Esta entidade comanda todas as rotinas relacionadas à entrada de vídeo no sistema, seja de um arquivo AVI, por exemplo, ou algum dispositivo de captura de vídeo em tempo real, como exposto em >>tecnologias. A implementação atual do projeto suporta a captura de vídeo utilizando o sistema de filtros do *DirectShow*.

Como podemos ver no diagrama de classes abaixo, ***VideoInput*** tem acesso à ***VideoCaptureDirectShow*** que por sua vez instancia e utiliza os métodos da ***DirectShowVideoRenderer***. Como foi já foi explicado, a ***DirectShowVideoRenderer***, é uma implementação de um filtro renderizador de vídeo, porém, em vez de renderizar os *frames* na placa de vídeo do computador, ele o faz em um *buffer* que fica acessível a qualquer implementação de efeito que precise utilizá-lo.

5.4 *Effect*

A arquitetura de efeitos do ViMus segue uma idéia já bastante sedimentada em *softwares* de visualização, que consiste em encadear efeitos seqüencialmente, ou seja, um efeito é aplicado a uma imagem gerada por outro efeito e assim em diante. Porém, possui peculiaridades inspiradas no *plugin AVS* do *Winamp*, como a idéia de lista de efeitos, que podem ser elementos de outra lista. Este é um dos principais motivos do *AVS* ser considerado o sistema de visualização mais flexível já produzido. Vejamos como essa idéia foi adaptada ao ViMus.

Podemos ver o esquema de listas do *AVS* como uma árvore ordenada, já imaginando a estrutura de dados que poderia representá-la. Essa estrutura de dados foi implementada fazendo-se uso de artifícios da orientação a objeto, como a herança e abstração. O primeiro passo foi definir o conceito de ***Effect***: um efeito geral que tem a capacidade de gerar uma imagem. No código, ***Effect*** é uma classe abstrata que possui o método abstrato ***draw()***.

A principal implementação de ***Effect*** é a ***EffectsList***, que possui um *array* de ***Effect***, possibilitando a recursão da estrutura e, assim a formação da árvore. O método ***draw()*** de ***EffectsList*** consiste em desenhar cada um dos elementos do *array* na ordem em que foram adicionados.

A função ***draw()*** de uma implementação qualquer de ***Effect*** pode ser usada basicamente para duas ações: **síntese**, gerando um novo desenho independente de qualquer imagem já existente através de primitivas gráficas como linhas, superfícies, etc., e **transformação**, gerando um novo desenho baseado em algum outro anteriormente exibido, reutilizando um *buffer* contendo a imagem, efetuando operações que alterem esse *buffer*. Eventualmente, um efeito pode sintetizar e transformar ao mesmo tempo.

As sínteses e transformações podem ser parametrizadas de diversas formas. A principal delas é, obviamente, a parametrização de acordo com a entrada de áudio. Um exemplo seria, aumentar a intensidade de uma cor na imagem de acordo com a intensidade do som de entrada. Mas, além disso, podemos e devemos oferecer ao usuário a possibilidade alterar a forma como as imagens são sintetizadas ou transformadas. Esse comando de alteração deve vir, naturalmente, da ***VimusUserInterface***. E tornar o efeito

capaz de responder à essa solicitação, basta implementar as funções abstratas de ***Effect*** relativas a resposta de eventos disparados pela interface. Exemplos dessas funções são: ***keyPressed()*** e ***windowReshaped()*** (chamado quando o usuário redimensiona a janela). Para intermediar essa e outros tipos de interações entre o usuário, os dispositivos de entrada e os efeitos, foi idealizada a entidade ***Mixer***.

5.5 *Mixer*

Esta entidade está no núcleo do sistema Vimus, sendo a responsável por executar as solicitações de ***VimusUserInterface***, e fornecer a infra-estrutura necessária aos efeitos como, informações do usuário, informações de áudio e de vídeo, e armazenar *buffers* de imagens (para serem usados por efeitos transformadores, por exemplo).

Para gerenciar a interação entre a interface e os efeitos, o ***Mixer*** faz simplesmente chama os métodos dos efeitos que desejem ter algum de seus parâmetros alterados a partir de alguma tecla que o usuário aperte, ou qualquer outro tipo de evento que ocorra e seja detectável pelo sistema de janelas utilizado (GLUT no caso da implementação atual). Um exemplo seria, ao usuário pressionar uma tecla, o método ***keyPressed()*** do efeito faz com que ocorra o aumento da intensidade de um efeito, ou o aumento da influência da música naquele efeito, mudança de cores, enfim, parâmetros que modifiquem de alguma forma o modo como o efeito está sintetizando ou transformando imagens.

Outra função primordial do ***Mixer*** é o gerenciamento de recursos como ***AudioInput*** e ***VideoInput***. É através desta classe que as capturas, processamentos de sinal são iniciados, interrompidos e têm suas configurações alteradas. Esta centralização é necessária por uma razão muito simples: seria menos eficiente se cada efeito instanciasse seu ***AudioInput*** e ***VideoInput***, pois teríamos mais de um sistema de *buffers* de áudio (um para cada efeito, que quisesse utilizar), por exemplo. Outra possibilidade de arquitetura seria a própria ***VimusUserInterface*** passar as referências de ***AudioInput*** e ***VideoInput*** para os efeitos. A desvantagem disso é que, além de sobrecarregar a classe ***VimusUserInterface*** (que antes teria apenas a função de intermediar a entrada e saída do Vimus com o usuário), tornaria o código menos modularizado, dificultando reimplementações para outras plataformas, por exemplo.

E, finalmente, a principal tarefa do ***Mixer***: controlar a árvore de efeitos, iniciando, alterando ou interrompendo a síntese da seqüência de imagens que constituem a saída do sistema. A Figura abaixo nos ajuda a compreender como esse controle é feito. Observe que o ***Mixer*** possui um *array* de árvores de efeitos disponíveis. Uma dessas árvores é a que está sendo processada no momento, ou seja, o método ***draw()*** dessa árvore está sendo chamado a todo momento, gerando a saída, enquanto as outras estão desligadas e inativas (não estão carregadas na memória). Ao comando do usuário, enviado por ***VimusUserInterface***, outra árvore pode ser escolhida, o ***Mixer*** então descarrega a árvore atual e carrega a escolhida. Este *array* de árvores pode ser alterado pelo ***VimusEditor***.

5.6 *VimusEditor*

Como vimos na sessão sobre ***Effect***, existe uma organização para uma árvore de efeitos que pode ser bem simples, apenas um efeito, ou bem complexa, com várias listas de efeitos dentro de outras listas. Para construir e alterar essas árvores, foi criada a classe ***VimusEditor***. Trata-se de uma classe comandada pela ***VimusUserInterface*** e que sua ação principal é enviar árvores de efeitos construídas, ou alteradas à classe ***Mixer***, alimentando seu *array* de árvores desta última, como descrito na sessão anterior.

Na implementação atual do projeto, esta classe não possui ainda interface gráfica. Ela funciona como um compositor automático de alguns efeitos com a finalidade de demonstrar o potencial do sistema, pois o escopo do projeto, como já ficou claro, foi estudar as tecnologias de domínio e desenvolver uma infra-estrutura de *software* básica.

6. Exemplos de efeitos

Neste capítulo apresentaremos alguns efeitos implementados utilizando a plataforma **ViMus**.

6.1 *WaveEffectRenderer*

Este efeito consiste em desenhar a onda capturada. Isso é feito desenhando-se linhas ligando os pontos de altura determinada por ***getSample(pos)*** e a abcissa por um fator multiplicado por ***pos***. Este fator multiplicado pelo número de *samples* por ***buffers*** é igual à largura da tela, isso garante que, independente do número de *samples* a onda sempre seja exibida dentro da tela.

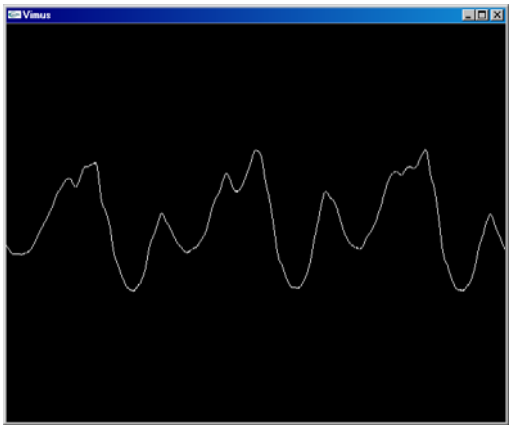

Figura 17. *WaveEffectRenderer*.

6.2 FrequencySurfaceEffectRenderer

Este efeito desenha uma superfície paramétrica (NURBS) para a qual um dos eixos tem seus pontos definidos pelo ***audioSpectrumBuffer*** (*buffer* resultante da transformada de Fourier). Assim, seu desenho muda de acordo com o quão agudo ou grave é o som.

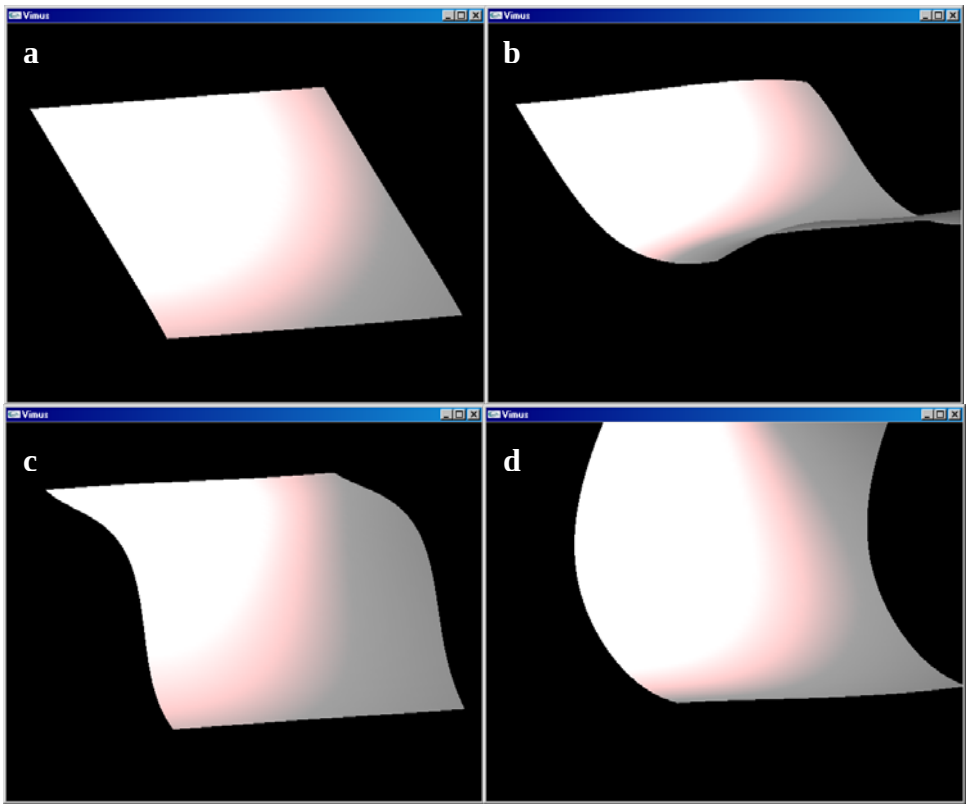

Figura 18. Superfície de freqüência – a) sem som; b) graves; c) médios; d) agudos.

6.3 *VideoCaptureRenderer*

Este efeito faz um acesso ao ***videoSreamBuffer*** e preenche em um *bitmap* apontado por uma textura *OpenGL* o último *frame* capturado. Esta textura é aplicada a um quadrado de *OpenGL* construído para ocupar toda a tela de forma que a largura e altura do vídeo é igual a largura e altura da tela, como se o mesmo fosse uma imagem de fundo.

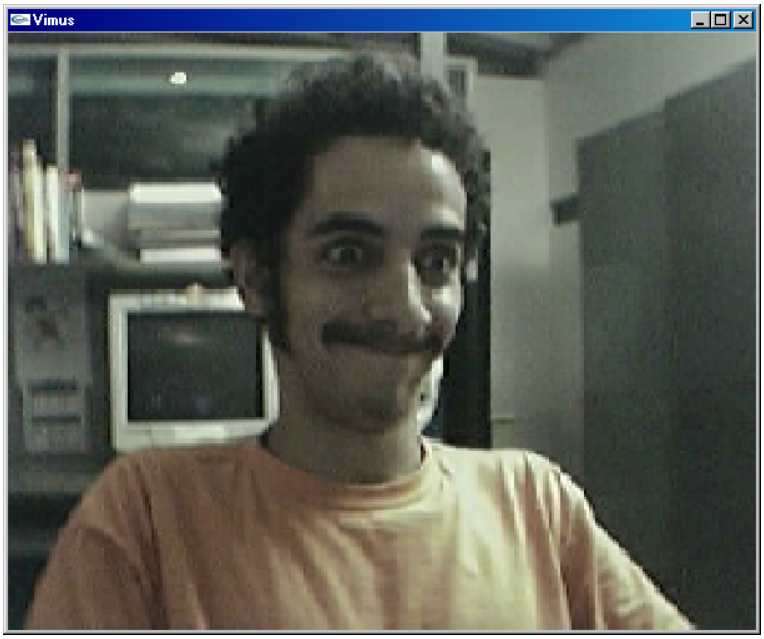

Figura 19. Vídeo capturado da *webcam* e exibido pelo ViMus como uma textura gerada em tempo real para cada *frame* capturado.

Como se trata de uma textura *OpenGL*, esta pode ser naturalmente mapeada em qualquer superfície criada. É o que mostram a figura abaixo.

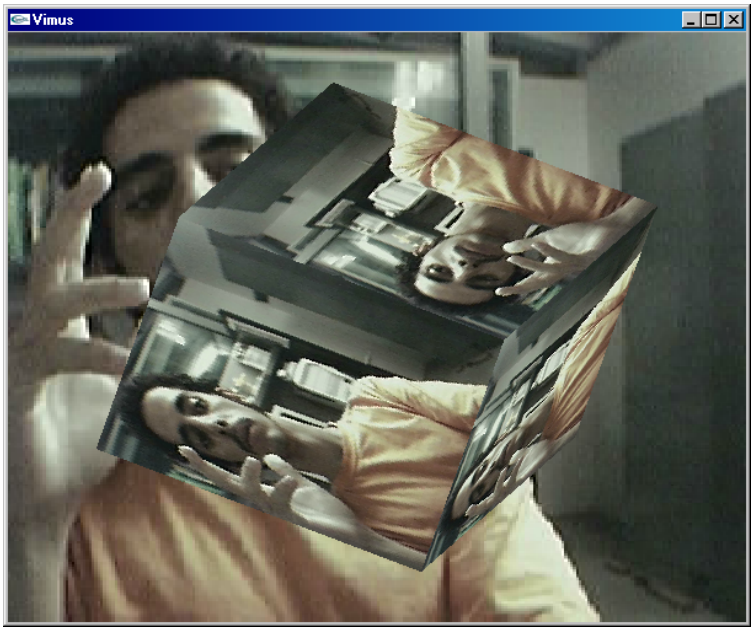

Figura 20. *Frame* capturado em textura aplicada às faces de um cubo.

6.4 Efeitos de vídeo baseados na entrada de áudio

O seguinte exemplo apresenta uma possibilidade de efeito no vídeo capturado utilizando-se como parâmetro a intensidade do som em uma determinada freqüência.

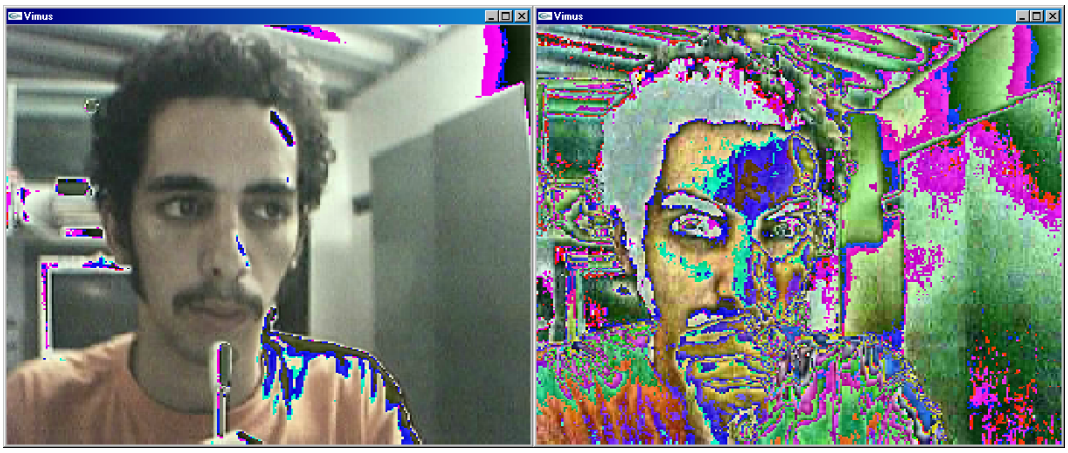

Figura 21. Intensidade de som capturado do microfone como parâmetro de efeito: esquerda - som de volume baixo; direita - som de volume muito alto.

6.5 Exemplos de efeitos compostos

Graças à flexibilidade de combinação de efeitos de diversas formas, as possibilidades de efeitos não são numeráveis. Apresentemos apenas alguns exemplos entre as várias combinações experimentadas.

6.5.1 Vídeo + *Blur* + Estouro de Cores (áudio) + Wave

Figura 22. Bactérias sonoro-psico-desintegradoras.

O efeito abaixo combina o efeito composto anterior com a superfície de freqüências. Importante lembrar que todas essas imagens são geradas em tempo real, incluindo, logicamente, a renderização da superfície iluminada e texturizada com a imagem do vídeo em tempo real após o processamento gráfico que gerou os efeitos.

Figura 23. Combinação pesada: renderização de superfície *NURBS* (som grave), iluminada, com geração de sombra e texturizada com vídeo capturado e processado.

6.5.2 Pseudo-fractais

A exibição do vídeo capturado por uma câmera em tempo real possibilita um efeito popularmente conhecido como *feedback* [London, 1995] e consiste em apontar a câmera para o próprio monitor no qual sua imagem está sendo exibida. Com o ViMus, porém, esse efeito é potencializado com a flexibilidade de criação de objetos 3D e texturização de suas faces com o vídeo em tempo-real. As figuras seguintes apresentam alguns **pseudo-fractais** criados utilizando-se a técnica de *feedback* em objetos 3D.

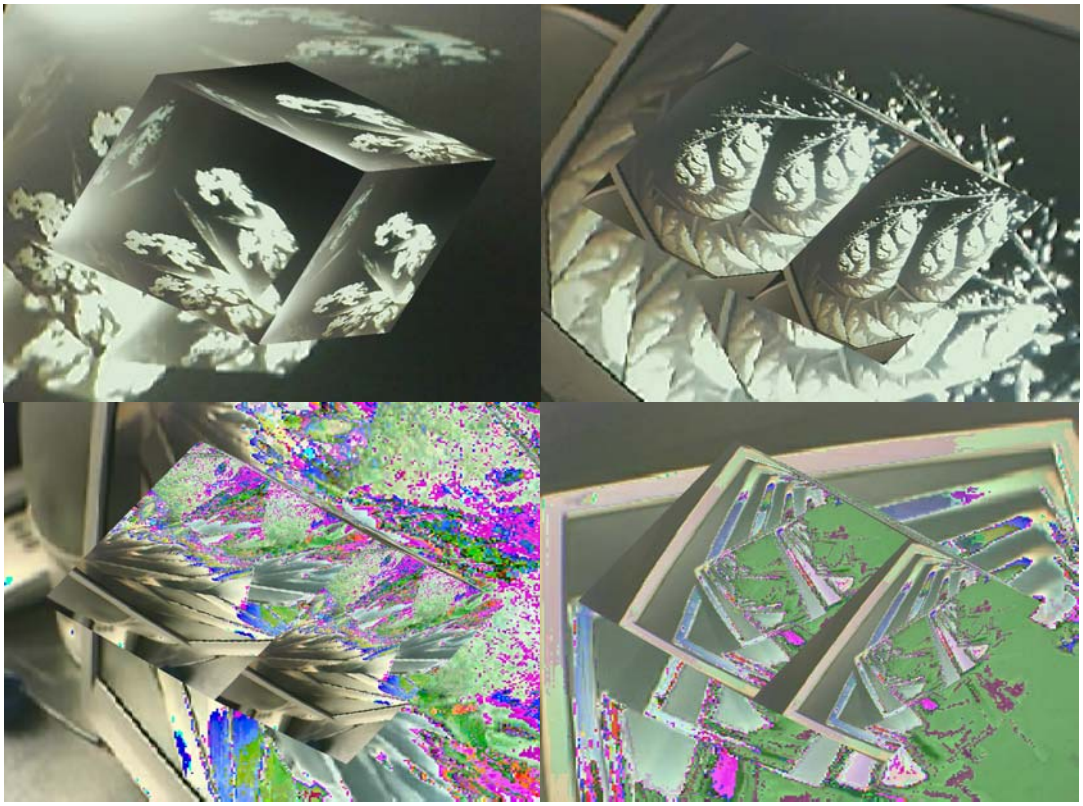

Figura 24. Pseudo-fractais obtidos sintetizados pelo ViMus utilizando-se *feedback* e objeto 3D. As duas imagens de baixo foram alteradas pelo áudio de entrada também.

7. Análise dos resultados obtidos

Através da leitura de revisões de *softwares* encontradas em diferentes sítios especializados (como os já citados VJCentral.com, VJForums, PIXnMix, etc.) direcionados às comunidades de *VJs*, foram levantados alguns critérios a serem utilizados para análise do produto desenvolvido. Porém, a maioria das características avaliadas por estes artigos, estão relacionadas à interface gráfica com usuário, como por exemplo, uma boa curva de aprendizado e usabilidade. Estes critérios não se aplicam à avaliação da implementação atual do sistema, pois, como foi dito, a interface gráfica atual foi implementada apenas para possibilitar os primeiros testes da infra-estrutura gráfica da aplicação. Entretanto, os seguintes critérios são fundamentais para a avaliação da infra-estrutura de um *software* do tipo: **número de quadros por segundo (*fps*)**, **tempo de resposta ao estímulo sonoro.**

7.1 Número de quadros por segundo (*fps*)

A saída do ViMus é uma animação e por isso deve ter uma **fluência** mínima necessária para as pessoas a perceberem como tal e não como uma seqüência de imagens estáticas sendo exibidas uma após a outra. Como foi discutido em **2.3 Processamento Gráfico e Vídeo Digital**, a quantidade mínima "satisfatória" de *fps* para sistemas de computador é difícil de se estabelecer pois para o usuário quanto maior, melhor. Porém, baseando-se em diversas conversas com usuários de jogos de computador experientes (estes desenvolvem muita sensibilidade à quantidade de *fps*) e através de demonstrações a *VJs*, chegamos à seguinte tabela.

fps	0-9	10-14	15-20	21-30	31-50	> 50
avaliação	Péssimo	Ruim	Razoável	Bom	Ótimo	Excelente

Tabela 2. Classificação de aplicações de acordo com *fps*.

Foi observada a combinação de efeitos altamente complexa aqui apresentada na sessão **6.5.1** em que ocorre renderização de NURBS iluminada. Observou-se que, quando

nenhum som é produzido, a animação é de aproximadamente 32 ***fps*** em uma máquina de processador 2.0 GHz de *clock*. Porém, quando algum som é produzido, os valores que estão sendo operados para se calcular as posições da NURBS e o estouro de cores para o processamento do vídeo aumentam muito, e essa taxa cai para 22, 23 ***fps***. Ou seja, obtivemos um desempenho entre Bom e Ótimo.

7.2 Tempo de resposta ao estímulo sonoro

Este critério é muito importante para avaliação, pois um dos objetivos principais do sistema ViMus é utilizar a execução musical como parâmetro para síntese de imagens em tempo real. Assim, a resposta do sintetizador a esses estímulos deve ser a mais rápida possível. Esta resposta depende de uma variável: o número de *buffers* utilizados para captura de áudio. Após alguns testes, foi constatado que o número mínimo de *buffers* de áudio necessários para que a onda seja capturada sem que o *buffer* não tenha sido enchido ainda é de 3. Aumentando-se esse número gradativamente, percebeu-se que o atraso de resposta ao estímulo sonoro ia aumentando. Porém, com o número de *buffers* entre 3 e 8, este atraso ainda é imperceptível para o usuário. Ou seja, novamente o ViMus obteve um desempenho satisfatório.

8. Conclusões e trabalhos futuros

Neste capítulo se faz uma avaliação crítica de todo o trabalho de pesquisa e desenvolvimento apresentado neste documento. Além disso, são apresentadas propostas para melhoramentos e novas funcionalidades do sistema ViMus.

8.1 Sobre as tecnologias pesquisadas e utilizadas

OpenGL mostrou-se uma plataforma gráfica adequada, porém talvez seja interessante implementar uma outra interface utilizando DirectX para comparar os resultados e finalmente fazer um estudo sério de comparação de desempenho para uma mesma tarefa em OpenGL e DirectX. Este estudo será muito importante e depois de divulgado, provavelmente será muito consultado, pois é uma dúvida constante em fóruns de desenvolvedores, esta a respeito de qual o melhor desempenho entre as duas tecnologias.

Outro próximo passo, em relação às tecnologias utilizadas será o estudo de *APIs* de captura de som e vídeo no Linux e, por que não, Macintosh, para iniciar o processo de uma nova implementação do sistema para outras plataformas.

8.2 Sobre a pesquisa de trabalhos relacionados

Apesar de suficiente para o desenvolvimento do ViMus em seu escopo atual, estudos mais sérios de trabalhos já realizados na área deverão ser realizados. Estes estudos deverão incluir testes sistemáticos de desempenho e análise de usabilidade em produtos semelhantes. Ao fim deste estudo teremos tabelas comparativas e isso nos dará condições de avaliar melhor nosso trabalho.

8.3 Sobre o processo

Para o desenvolvimento desta primeira versão do sistema, o processo mostrou-se apropriado, pois em apenas uma semana e meia todo o sistema foi modelado implementado. Isso por que a implementação consistiu em criar todos as classes definidas na modelagem e preencher grande parte de seus métodos com funções que já haviam sido implementadas para o desenvolvimento dos protótipos de estudo que estavam funcionando separadamente.

Na próxima fase desse projeto, porém, é provável que o número de desenvolvedores aumente (para este trabalho, eu fui o único desenvolvedor) e isso implica na necessidade de uma adoção de um sistema de controle de versões, além de uma documentação mais séria.

Outro ponto crucial em relação ao processo de desenvolvimento é que, o próximo passo deste projeto sem dúvidas é a implementação de uma interface gráfica totalmente voltada para as necessidades dos potenciais usuários. Será necessário então um levantamento sério e sistemático de requisitos, através de, por exemplo, entrevistas com *VJs* ou qualquer tipo de potencial usuário do sistema. E além de testes de desempenho, serão feitos ao final da implementação vários testes de usabilidade, avaliando critérios como curva de aprendizado para enfim avaliar o sistema.

8.4 Sobre o produto

Considero o produto desenvolvido uma vitória do projeto, pois superou minhas expectativas, e as de vários outros amigos e colegas que acompanharam o desenvolvimento e alguns consideraram, com certa razão, que o desempenho seria um problema muito sério e provavelmente não fosse razoavelmente superável em tão pouco tempo restante para implementação (nessa época estava a um mês do prazo de entrega). Entretanto os resultados podem ser lidos e vistos neste relatório escrito, porém muito melhores quando vistos em execução em várias cores funcionando, em tempo-real. Apesar disso, o sistema está distante de ser um bom e completo *software* para *VJs* pois não possui uma interface gráfica bem feita ainda. Este será o próximo objetivo do

projeto: construir uma interface gráfica inspirada tanto em sistemas orientados a efeitos, nos quais se pode montar e alterar livremente as árvores de efeitos e sistemas orientados a amostras, nos quais se podem disparar efeitos e amostras de vídeo de arquivo a partir do teclado. A seguir, algumas outras melhorias futuras pensadas para o ViMus.

8.4.1 Otimizações

Existem várias possibilidades de otimizações do código a serem exploradas ainda. Uma delas é a reimplementação de algumas rotinas em linguagem Assembly, como é o caso de uma função utilizada no sistema para inverter a cor vermelha, pela azul em cada pixel capturado da placa de vídeo que vêm em ordem contrária: em vez de RGB, vêm BGR. Segue abaixo o código desta função, implementada por Jeff Molofee (mais conhecido como NeHe) [Molofee, 2004] na lição 35 de seu tutorial de OpenGL:

```
void VideoCaptureRenderer :: flipIt(void* buffer)
{
      void* b = buffer;
      __asm                         // Assembler Code To Follow
      {
            mov ecx, VIDEO_WIDTH*VIDEO_HEIGHT //Dimensions of Mem Block
            mov ebx, b              // Points ebx To Our Data (b)
            label:                  // Label Used For Looping
                  mov al,[ebx+0]   // Loads Value At ebx Into al
                  mov ah,[ebx+2]   // Loads Value At ebx+2 Into ah
                  mov [ebx+2],al   // Stores Value In al At ebx+2
                  mov [ebx+0],ah   // Stores Value In ah At ebx

                  add ebx,3         // Moves Through The Data By 3 Bytes
                  dec ecx           // Decreases Our Loop Counter
                  jnz label         // If Not Zero Jump Back To Label
      }
}
```

Da mesma forma outras funções que alterem o *bitmap* do *frame* atual escrito no bloco de memória da textura, como os efeitos de *blur* e de estouro de cor, podem ser reescritas em Assembly, utilizando técnicas de otimização mais sofisticadas.

8.4.2 Possibilitar utilização de placas *DSPs*

Placas *DSP* (*Digital Signal Processor*) estão presentes nas melhores placas de som e podem ser adquiridas separadamente. É possível utilizá-las para processamento de som, como execução de transformada de Fourier, evitando que este processamento seja feito pela CPU, passando a ser feito pela placa *DSP* paralelamente. Uma das formas de se fazer isso é utilizando a API código aberto de processamento de som chamada [PureData].

8.4.3 Entrada MIDI

Outra melhoria muito importante seria aceitar como entrada, além de áudio, mensagens MIDI. Isso ampliaria as possibilidades de integração do *software* com diversos controladores MIDI, vários tipos de *hardwares* e de *softwares* que gerem saída MIDI.

8.4.4 Análise Musical e Instrumentos Visuais Harmônicos

Utilizando-se técnicas de IA como redes neurais [machine learning], poderemos efetuar análises muito mais complexas que simplesmente detectar se o som está predominantemente agudo ou grave, por exemplo. Redes neurais podem ser construídas e treinadas para detecção de peculiaridades de execução de instrumentos específicos e ao detectar essas peculiaridades disparar efeitos visuais específicos escolhidos para estas.

Indo mais longe, podemos imaginar interpretação harmônica, como detectar se o tom da música está maior ou menor, criando um novo conceito o qual poderíamos chamar de **Instumentos Visuais Harmônicos** (**IVH**). Desta forma, se o executante mudasse a tonalidade da música, o **IVH** mudaria as cores de forma a expressar aquela mudança de “clima” da música, naquele instante.

9. Referências Bibliográficas

[ATI, 2005] ATI (2005). Gamer Products URL: http://www.ati.com/products/gamer.html Consultado em março de 2005.

[AVS, 2005] AVS (2005). WINAMP.COM | NSDN | Winamp | Writing Plugins | AVS. URL: http://www.winamp.com/nsdn/winamp/plugins/avs. Consultado em janeiro de 2005.

[Brand, 2001] Brand D. (2001). Human Eye Frames Per Second. URL: http://www.techspot.com/vb/archive/index.php/t-14907.html. Consultado em março de 2005.

[Davies, 2003] Davies (2005). Vídeo Capture. URL: http://www.gdcl.co.uk/vidcap.htm. Consultado em março de 2005.

[DirectShow, 2005] Microsoft (2005). Introduction to DirectShow. URL: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directshow/htm/introductiontodirectshow.asp. Consultado em março 2005.

[Embree, 1998] Embree, Paul M., Danieli D. (1998). C++ Algorithms for Digital Signal Processing (2nd Edition), Prentice Hall PTR; 2 edition (November 13, 1998)

[FolhaOnLine, 2005] FolhaOnline (2005). URL: http:://www.folha.com.br. Consultado em março de 2005.

[Hitchner, 2004] Hitchner, L. (2004). C++ Notes for Java Programmers. URL: http://www.csc.calpoly.edu/~hitchner/CSC471/lab.1.html. Consultado em março de 2005.

[Kilgard, 1996] Kilgard J. M. (1996). The OpenGL Utility Toolkit (GLUT) Programming Interface API Version 3. Silicon Graphics, Inc. URL: http://www.opengl.org/ documentation/specs/glut/spec3/spec3.html. Consultado em março de 2005.

[London, 1995] London B. (1995). Time as Medium: Five Artists' Video Installations. URL: http://www.experimentaltvcenter.org/history/people/ptext.php3?id=55.

[MediaSana, 2005] Media Sana (2005). Media Sana. URL: http://www.mediasana.org. Consultado em março de 2005.

[Mendes, 2002] Mendes A. (2002). Arquitetura de Software - Desenvolvimento orientado para arquitetura. Editora Campus.

[Meso, 2005] Meso (2005). meso | digital media systems design. URL: http://www.meso.net. Consultado em março de 2005.

[Molofee, 2004] Molofee J. (2004). NeHe Productions: OpenGL Lesson #35. URL: http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=35. Consultado em mar/2005.

[Nvidia, 2005] Nvidia (2005)Nvidia Home URL: http://www.nvidia.com/page/home.html Consultado em março de 2005.

[PIXnMIX, 2005] PIXnMIX (2005). PIXnMIX Home. URL: http://www.channel4.com/learning/microsites/I/ideasfactory/pixnmix. Consultado em março de 2005.

[Presse, 2005] Presse F. (2005). FolhaOnLine URL: http://www1.folha.uol.com.br/folha/ciencia/ult306u13024.shtml. Consultado em março de 2005.

[Re:combo, 2005] Re:combo (2005). Re:combo URL: http://www.recombo.art.br. Consultado em março de 2005.

[SGI, 2005] Silicon Graphics (2005). Welcome to SGI. URL: http://www.sgi.com. Consultado em março de 2005.

[*VfW*, 2005] Microsoft (2005). Vídeo For Windows URL: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/multimed/htm/_win32_video_for_windows.asp. Consultado em março de 2005.

[*VirtualDub*, 2005] VirtualDub (2005). Welcome to virtualdub.org! URL: http://www.virtualdub.org. Consultado em março de 2005.

[Visual Jockey, 2005] Visual Jockey (2005). visualJockey - Realtime Interactive Animation Software. URL: http://www.visualjockey.com. Consultado em mar/2005.

[VJForums, 2005] VJForums (2005). VJForums.com - The VJ Community Message Board. URL: http://www.vjforums.com. Consultado em março de 2005.

[VVVV, 2005] Meso (2005). VVVV – A Multipurpose Toolkit. URL: http://vvvv.meso.net. Consultado em março de 2005.

[Wikipedia, 2005] Wikipedia (2005). Som – Wikipédia. URL : http://pt.wikipedia.org/wiki/Som. Consultado em março de 2005.

[Windows Media API, 2005] Microsoft (2005). Windows Media SDK. URL: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/multimed/htm/_win32_windows_multimedia_start_page.asp. Consultado em março 2005.

Assinaturas

Prof.º Sílvio Melo

Jarbas Jácome de Oliveira Júnior