UNIVERSIDADE FEDERAL DE PELOTAS
INSTITUTO DE FÍSICA E MATEMÁTICA
DEPARTAMENTO DE INFORMÁTICA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

FERRAMENTA MULTITHREAD PARA RECONSTRUÇÃO VOLUMÉTRICA DE IMAGENS EM MEDICINA NUCLEAR PARA AUXÍLIO NA DETECÇÃO DE ZONAS EPILEPTOGÊNICAS

CLEBER ROBERTO MILANI

Pelotas, 2007

CLEBER ROBERTO MILANI

FERRAMENTA MULTITHREAD PARA RECONSTRUÇÃO VOLUMÉTRICA DE IMAGENS EM MEDICINA NUCLEAR PARA AUXÍLIO NA DETECÇÃO DE ZONAS EPILEPTOGÊNICAS

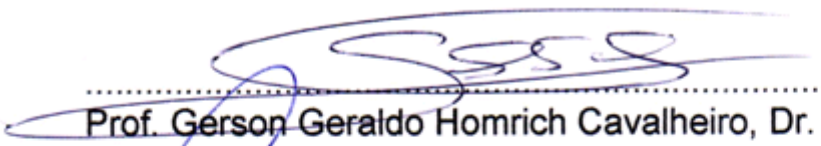
Trabalho acadêmico apresentado ao Curso de Bacharelado em Ciência da Computação do Instituto de Física e Matemática da Universidade Federal de Pelotas, como requisito parcial à obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Gerson Geraldo Homrich Cavalheiro

Co-Orientador: Prof. Dr. Lucas Ferrari de Oliveira

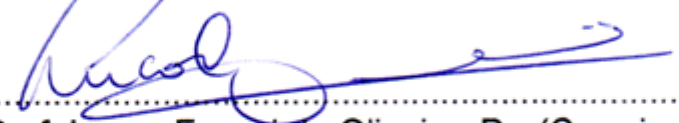
Pelotas, 2007

BANCA EXAMINADORA:



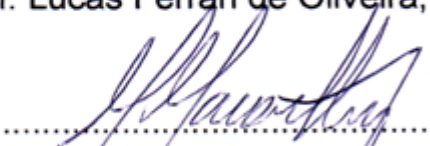
.....

Prof. Gerson Geraldo Homrich Cavalheiro, Dr. (Orientador)



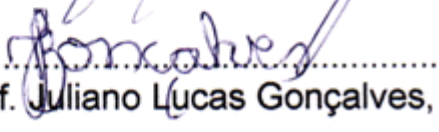
.....

Prof. Lucas Ferrari de Oliveira, Dr. (Co-orientador)



.....

Prof. Marcello da Rocha Macarthy, MSc.



.....

Prof. Juliano Lucas Gonçalves, MSc.

AGRADECIMENTOS

Aos meus pais Inês e Benoni por todo o carinho, apoio e compreensão durante essa caminhada em que, mesmo a 700 KM de distância, estiveram sempre presentes. Ao meu irmão Jeferson, companheiro de todas as horas e um dos grandes responsáveis pelo meu crescimento em todos os sentidos. Sem vocês nada disso seria possível, muito obrigado!

Aos colegas, constantes parceiros nessa jornada, pela amizade e companhia no dia-a-dia, seja para festas ou para virar as madrugadas estudando e fazendo trabalhos. Valeu gurizada!

Aos orientadores Gerson e Lucas por acreditar no meu trabalho e ajudar a enxergar os caminhos. Também pela disposição e ensinamentos, obrigado!

À Vanessa, namorada dedicada e companheira. Sem dúvidas a vida é mais alegre e bonita contigo. Beijão!

A meu grande amigo Papalia pela amizade de tantos anos e por colaborar cedendo arquiteturas para os testes deste trabalho. Valeu!

RESUMO

MILANI, Cleber Roberto. **Ferramenta multithread para reconstrução volumétrica de imagens em medicina nuclear para auxílio na detecção de zonas epileptogênicas.** 2007. 129f**.** Trabalho acadêmico (Graduação em Ciência da Computação) – Instituto de Física e Matemática. Universidade Federal de Pelotas, Pelotas.

A avaliação pré-cirúrgica de pacientes com epilepsia intratável requer uma análise baseada em informações complementares obtidas através de exames de Ressonância Magnética (RM) e SPECT (*Single Photon Emission Computed Tomography*, Tomografia Computadorizada por Emissão de Fóton Único). Entretanto, a comparação visual lado-a-lado, tradicionalmente aplicada, além de ser um processo laborioso, tem resultados por vezes não confiáveis, dado que esses dependem de critérios subjetivos utilizados pelo especialista. Ao longo dos anos, diversas ferramentas vêm sendo desenvolvidas para facilitar esse processo e aumentar a confiabilidade do mesmo. Nesse contexto, este trabalho surge com o objetivo de oferecer uma aplicação implementada através de reconstrução volumétrica (RV) para visualização tri-dimensional de imagens de RM e SPECT alinhadas. Porém, os algoritmos de *Ray-tracing*, responsáveis pela RV, possuem custo computacional bastante elevado. Por outro lado, esses algoritmos apresentam também alto grau de paralelismo potencial. Dessa forma, este trabalho propõe uma implementação concorrente voltada a arquiteturas *multi-core* através da programação *multithread* com VTK (*Visualization ToolKit*) e *threads* POSIX (*Pthreads*) em linguagem C++. A opção por arquiteturas *multi-core* justifica-se pelo constante crescimento da participação dessas no mercado.

Palavras-chave: Programação *multithread*. Arquiteturas *multi-core*. VTK. Visualização Volumétrica.

ABSTRACT

MILANI, Cleber Roberto. **Ferramenta multithread para reconstrução volumétrica de imagens em medicina nuclear para auxílio na detecção de zonas epileptogênicas.** 2007. 129f. Trabalho acadêmico (Graduação em Ciência da Computação) – Instituto de Física e Matemática. Universidade Federal de Pelotas, Pelotas.

Daily pre-surgical evaluation of patients with intractable epilepsy requires analyzing complementary information gotten through Magnetic Resonance (MR) and Single Photon Emission Computed Tomography (SPECT) examinations. However, traditionally applied side-by-side appearance comparison, not only is a laborious process, but may present not trustworthy results, since these depend of subjective criteria used by the specialist. Throughout the years, lots of tools are being developed to facilitate this process and increase the trustworthiness of it. In this context, this work presents with the objective of offering an application implemented through volumetric reconstruction (RV) for three-dimensional visualization of lined up RM and SPECT images. However, ray-tracing algorithms, that are responsible for volumetric reconstruction, have too expensive computational costs. On the other hand, those algorithms are potentially parallelizable too. Thus, this work considers a concurrent implementation destined to multi-core architectures with multithread programming on C++ added by VTK (Visualization ToolKit) and threads POSIX (Pthreads). The choice of multi-core architectures is justified by the constant growth of the participation of these in the market.

Keywords: Multithread programming. Multi-core architectures. VTK. Volumetric Visualization.

LISTA DE FIGURAS

Figura 1 Imagens de SPECT com aplicação de radiofármaco 22
Figura 2 Imagens de Ressonância Magnética nos planos 24
Figura 3 Comparação visual entre Imagens de SPECT 25
Figura 4 Mapeamento dos *pixels* em *voxel*s..... 26
Figura 5 Abstração do algoritmo fundamental utilizado para reconstrução volumétrica - *Ray-tracing*..... 30
Figura 6 Recursividade em *Ray-tracing* 32
Figura 7 Extensão da taxonomia de Flynn sugerida por Johnson 43
Figura 8 Proposta de modelo para desenvolvimento de *software* paralelo..... 48
Figura 9 Trecho de código para execução de três *threads* em paralelo utilizando a classe vtkMultiThreader..... 59
Figura 10 Algoritmo para subtração das imagens de SPECT crítico e intercrítico e alinhamento com a RM 64
Figura 11 Configuração das diretivas para compilação do VTK no *CMake* 67
Figura 12 Arquivo para configuração no *CMake* da ferramenta desenvolvida .. 68
Figura 13 Relação entre as classes desenvolvidas 69
Figura 14 Estruturas utilizadas na manipulação das imagens 71
Figura 15 Estrutura do *pipeline* de leitura dos exames 71
Figura 16 Trecho de código que inclui as classes herdadas pela *View*..... 73
Figura 17 Estrutura do *pipeline* de visualização para um volume 74
Figura 18 Criação das funções de *Ray-casting* no VTK 76
Figura 19 Definição das funções de transferência no VTK 79
Figura 20 Influência das diferentes fontes de luz em uma esfera 80
Figura 21 Trecho de código que instancia e configura um conjunto de propriedades para um volume 81
Figura 22 Trecho de código para criação de um volume e seu *mapper* 82
Figura 23 Quadro das classes adicionadas para implementação da visualização em fatias 2D 83

Figura 24 Trecho do cabeçalho da classe *View* correspondente à Janela para a configuração do número de *threads* 85
Figura 25 Interface da Janela para a configuração do número de *threads* 85
Figura 26 Trecho do cabeçalho da classe *View* correspondente à Janela para a configuração do número desejado de quadros por segundo 86
Figura 27 Interface da Janela para a configuração do número desejado de quadros por segundo 86
Figura 28 Interface da Janela para visualização das fatias em 2D 87
Figura 29 Trecho do cabeçalho da classe *View* correspondente à Janela para visualização das fatias em 2D 88
Figura 30 Trecho do cabeçalho da classe *View* correspondente à Janela para visualização das informações dos volumes 89
Figura 31 Interface da Janela para a visualização dos dados dos volumes 89
Figura 32 Janela principal da ferramenta desenvolvida 90
Figura 33 Quadro de configurações das arquiteturas utilizadas nos testes 93
Figura 34 Comparação entre o volume de fisionomia reconstruído utilizando .. 106
Figura 35 Volumes de fisionomia do paciente reconstruídos 107
Figura 36 Volumes de fisionomia do paciente reconstruídos com parâmetros padrão e variando o tamanho do passo 109
Figura 37 Exame de Ressonância Magnética do paciente reconstruído com um conjunto semelhante de parâmetros 115
Figura 38 Exame de SPECT do paciente reconstruído com um conjunto semelhante de parâmetros 115
Figura 39 Janelas reconstruídas com a ferramenta desenvolvida 116
Figura 40 Visualização volumétrica de uma imagem de TC no VolView 129

LISTA DE TABELAS

Tabela 1 Tempo médio gasto na renderização da janela “Fisionomia” 97
Tabela 2 Tempo médio gasto na renderização da janela “Exames alinhados” 97
Tabela 3 Tempo médio gasto na renderização da janela “Foco Epileptogênico” 97
Tabela 4 Tempo médio gasto na renderização da janela “Fisionomia” em arquiteturas varnuiteturas varnuitetura1tetuçt...

6 AVALIAÇÃO E RESULTADOS 92
6.1 Preparação dos testes 92
6.2 Testes relacionados à estratégia de implementação 95
6.2.1 Teste dos *pipelines* isolados 96
6.2.1.1 Especificação 96
6.2.1.2 Objetivos 96
6.2.1.3 Resultados 96
6.2.1.4 Conclusões 98
6.2.2 Teste de renderização das janelas seqüencialmente 100
6.2.2.1 Especificação 100
6.2.2.2 Objetivos 101
6.2.2.3 Resultados 101
6.2.2.4 Conclusões 102
6.2.3 Teste de renderização das janelas paralelamente 102
6.2.3.1 Especificação 102
6.2.3.2 Objetivos 103
6.2.3.3 Resultados 103
6.2.3.4 Conclusões 104
6.3 Testes relacionados aos parâmetros 104
6.3.1 Testes de interpolação e função de *Ray-casting* 105
6.3.1.1 Especificação 105
6.3.1.2 Objetivos 105
6.3.1.3 Resultados 105
6.3.1.4 Conclusões 107
6.3.2 Teste do tamanho do passo 108
6.3.2.1 Especificação 108
6.3.2.2 Objetivos 108
6.3.2.3 Resultados 109
6.3.2.4 Conclusões 110
6.3.3 Testes de parâmetros avançados 111
6.3.3.1 Especificação 111
6.3.3.2 Objetivos 111
6.3.3.3 Resultados 111
6.3.3.4 Conclusões 113

6.4 Teste de portabilidade 113
6.5 Teste comparativo com o *software* proprietário *VolView* 2.0 114
6.6 Conclusões 116

7 CONCLUSÃO 118
7.1 Trabalhos futuros 120

REFERÊNCIAS BIBLIOGRÁFICAS 121
APÊNDICE A – OUTROS TRABALHOS NA ÁREA 126

1 INTRODUÇÃO

Conhecida na Grécia antiga como "doença sagrada" (por se acreditar na época que seus portadores possuíam características espirituais próprias e poderiam comunicar-se diretamente com os deuses) a epilepsia é uma enfermidade que estima-se fazer parte nos dias de hoje do cotidiano de aproximadamente 50 milhões de pessoas em todo o mundo. No século XIX, Hughlings Jackson estabeleceu as bases da epileptologia ao descrever as crises chamadas de psicomotoras, além de teorizar pioneiramente o papel de descargas elétricas cerebrais na origem dos fenômenos epilépticos. Por volta de 1930, foram descritos os primeiros registros de atividade elétrica *in vivo*, e com a descrição do eletrencefalograma tornou-se possível o estudo das alterações epilépticas funcionalmente (JUNIOR, 2003).

Atualmente, as drogas antiepilépticas obtêm êxito no controle de cerca de 60 a 80% dos casos, porém os demais necessitam intervenção cirúrgica. Essa intervenção requer uma avaliação pré-cirúrgica do paciente, a qual é feita com base em informações complementares de diferentes exames (OLIVEIRA, 2005).

No caso da epilepsia, é realizada, por exemplo, comparação visual entre imagens de SPECT (*Single Photon Emission Computed Tomography*, Tomografia Computadorizada por Emissão de Fóton Único) obtidas no período em que o paciente se encontra em crise e imagens dos períodos entre as crises, chamadas, respectivamente, de imagens críticas e imagens intercríticas. Porém, com a tradicional interpretação visual lado-a-lado pode ser difícil a identificação do foco epilético, particularmente em pacientes com epilepsia intratável extra-temporal ou não localizada. (O' BRIEN et al., 1998 apud OLIVEIRA, 2005). Assim, a busca pelo desenvolvimento de técnicas que permitam localizar e quantificar os focos epiléticos a partir de informações obtidas por fusão e subtração de imagens tem ganhado amplo espaço e se tornado-se constante.

Fusão de imagens é uma técnica cuja idéia principal é agrupar informações complementares obtidas através de diferentes exames e que, normalmente, são visualizadas separadamente. A subtração por sua vez, permite obter a intersecção

existente entre as imagens do paciente em situações normais e de crise, alcançando assim o foco do problema (OLIVEIRA, 2005).

Um método bastante empregado é o SISCOM (*Subtraction Ictal SPECT Co-registered to MRI*, Subtração de SPECT Crítico Co-registrado em imagens de Ressonância Magnética). Essa técnica permite detectar os focos epilépticos a partir da subtração de imagens funcionais de SPECT e localizá-los anatomicamente no exame de Ressonância Magnética (RM), permitindo um melhor estadiamento clínico e o planejamento cirúrgico de pacientes com epilepsia intratável (O' BRIEN et al., 1998 apud OLIVEIRA, 2005).

Oliveira (2005) propôs um método de fusão de imagens em três dimensões. A imagem resultante é formada por três imagens do cérebro do paciente, referentes aos cortes nas posições Coronal (eixo x), Transversal (eixo y) e Sagital (eixo z). Esse método representa avanços na área médica, pois a visualização 3D (tri-dimensional) para a análise qualitativa é de grande utilidade na fase de tomada de decisão.

Motivado pela possibilidade de complementar as técnicas citadas e oferecer uma alternativa às mesmas, este trabalho propõe uma aplicação para visualização de imagens SPECT alinhadas à RM baseada em reconstrução volumétrica (RV). A ferramenta proposta tem como objetivo auxiliar a equipe médica na detecção e avaliação do foco epileptogênico, oferecendo uma visualização muito próxima da real e também a possibilidade de realizar uma navegação virtual através das imagens reconstruídas.

Para que a ferramenta proposta ofereça as facilidades de visualização e navegação desejadas, são utilizadas cores variadas para destacar as zonas epileptogênicas e é dada ao usuário a possibilidade de alterar, dinamicamente, os parâmetros utilizados na reconstrução das estruturas através de uma interface amigável. Essas particularidades foram propostas na tentativa de contribuir com os diversos esforços que, ao longo dos anos, têm sido destinados à exploração de maneiras diferentes de visualizar e interagir com dados volumétricos, justificando o porquê de a ferramenta ser baseada em reconstrução volumétrica.

A reconstrução volumétrica é fundamentada em um algoritmo chamado *Ray-tracing*, ou traçamento de raios. Entretanto, o custo computacional desse algoritmo é bastante elevado, devido principalmente às operações de ponto flutuante utilizadas

na transformação dos objetos 3D, à rasterização dos polígonos transformados e aplicação de *antialiasing* na imagem sintetizada (LIN et al., 2001).

Por se tratar de uma ferramenta de visualização interativa, a aplicação desenvolvida neste trabalho tem de oferecer uma taxa de resposta e repetição das imagens geradas que varie entre 10 e 20 *frames* (quadros) por segundo, pois somente dessa forma é possível uma navegação satisfatória (ROBB, 1999). Esse requisito, adicionado à necessidade de trabalhar com reconstrução volumétrica, faz com que esse tipo de aplicação seja normalmente desenvolvido somente para computadores de grande capacidade computacional, pois seriam esses os únicos capazes de executá-las adequadamente (AHRENS et al., 2000). A exigência de máquinas com grande capacidade de processamento é um fator limitante para a adoção em larga escala das soluções baseadas na reconstrução volumétrica.

Por outro lado, Santos (1994) observa que, embora o custo computacional da aplicação seja alto, existe um alto grau de paralelização potencial, comprovado por trabalhos empregando técnicas de *Ray-tracing* paralelas, grande parte desses voltados a arquiteturas paralelas com memória distribuída, como aglomerados de computadores (*clusters*). Entretanto, por questões técnicas e financeiras, esse tipo de arquitetura não é acessível a usuários convencionais, seu uso fica mais restrito a instituições como centros de pesquisas e universidades. Dado que o ambiente para execução da ferramenta proposta são os computadores pessoais dos especialistas e os disponíveis em seus locais de trabalho, foi necessário basear a implementação em outro tipo de arquitetura paralela.

A alternativa encontrada foi trabalhar com arquiteturas baseadas nos chamados processadores *multi-core*. *Multi-core* é o nome dado aos CPUs (*Central Processing Unit,* Unidade Central de Processamento) com mais de um núcleo de processamento e surgiu como solução encontrada pelas fabricantes para contornar o problema da limitação no crescimento da freqüência de *clock*. No ano de 2005 foi adotada pela Intel como base para seus sistemas de alta performance (ASANOVIC et al., 2006) e desde então vem ganhando espaço mundialmente.

Diferentemente das outras arquiteturas paralelas, como *clusters*, os computadores *multi-core* possuem custo acessível a clínicas, hospitais e mesmo aos médicos em seus consultórios. Além disso, nos dias de hoje processadores *multi-core* estão disponíveis até mesmo em computadores portáteis, o que facilita

consideravelmente a popularização dos sistemas de visualização baseados em reconstrução volumétrica.

Entretanto, os programas seqüenciais tradicionais não se beneficiam do paralelismo oferecido pelas arquiteturas *multi-core*. Para que isso ocorra, as aplicações precisam ser desenvolvidas de maneira paralela, o que, no caso dessas arquiteturas, é feito através da programação com bibliotecas para desenvolvimento *multithread*. A busca de uma solução paralela *multithread* para o problema da reconstrução volumétrica, que explore eficientemente as arquiteturas *multi-core*, constitui a motivação para a realização deste trabalho.

1.1 Objetivos

O objetivo deste trabalho é desenvolver uma ferramenta interativa para visualização 3D de imagens médicas que ofereça facilidades na detecção e avaliação do foco epileptogênico. A utilização, modificação e distribuição da aplicação são permitidas, seguindo os conceitos de *software* livre.

Objetivos complementares são identificar os custos computacionais e as ferramentas adequadas para desenvolvimento de uma implementação da solução baseada em reconstrução volumétrica. Essa solução deve explorar eficientemente o paralelismo em arquiteturas *multi-core* para obter melhor desempenho no processamento de imagens.

1.2 Metodologia

O primeiro passo na realização deste trabalho foi realizar um levantamento sobre os algoritmos para visualização de volumes, buscando identificar qual atende da melhor maneira possível os requisitos propostos. Em paralelo a isso, estudou-se as linguagens e ferramentas de programação disponíveis para verificar quais as mais adequadas para a implementação da solução desenvolvida. Optou-se por trabalhar com a linguagem de programação C++ complementada pelas bibliotecas VTK, FLTK e *Pthreads*.

Feito isso, realizou-se uma implementação seqüencial da ferramenta com o objetivo de identificar os pontos que concentram maior custo computacional e quais

desses contêm paralelismo em potencial. Na seqüência, a solução foi decomposta em partes independentes e implementada de maneira paralela e orientada a objetos.

A avaliação da solução desenvolvida e comparação entre o desempenho das diferentes implementações foram realizadas através da execução em arquiteturas *multi-core* variadas. O alinhamento das imagens utilizadas foi validado por Oliveira (2005), não fazendo, portanto, parte do escopo de avaliação do presente trabalho.

1.3 Resultados

Obtiveram-se resultados satisfatórios nos diferentes aspectos da aplicação desenvolvida. O conjunto de ferramentas adotado permitiu construir uma solução que provê uma GUI (*Graphical User Interface*, Interface Gráfica com o Usuário) amigável e que, ao mesmo, tempo mantém flexível e dinâmica a alteração dos parâmetros utilizados na reconstrução volumétrica. Além disso, a ferramenta desenvolvida apresentou bons resultados no quesito portabilidade: realizou-se a compilação da mesma em ambientes MS *Windows*, *Linux* e *Mac* OS X.

No sentido do desempenho, obtiveram-se resultados interessantes ao dividir o cálculo do *Ray-tracing* em *threads*. Em arquiteturas *multi-core*, a reconstrução do volume de maior custo computacional com 2 *threads* consumiu 32% menos tempo de processador que a mesma com 1 *thread*, o significa um *speedup* de 1.463 e eficiência de 73%.

1.4 Organização do trabalho

Este trabalho está dividido em sete capítulos, sendo o primeiro deles a presente introdução. O Capítulo 2 apresenta conceitos de neuroimagem e visualização científica, contextualizando a importância da aplicação proposta por este trabalho no escopo da visualização.

No Capítulo 3 é feita uma introdução sobre o processamento paralelo, colocando suas características e relevância para a Ciência da Computação. São apresentados os principais modelos e classes de arquiteturas disponíveis para a exploração do paralelismo.

O Capítulo 4 trata sobre a programação paralela especificamente voltada às arquiteturas multiprocessadas, em especial as *multi-core*. São abordadas as diferenças entre processos e *threads*, e, na seqüência as principais ferramentas para programação *multithread*.

O Capítulo 5 apresenta a solução desenvolvida, detalhando a estrutura e metodologias utilizadas. O Capítulo 6 expõe as avaliações as quais a ferramenta foi submetida com seus respectivos resultados.

Por fim, o Capítulo 7 apresenta as conclusões provenientes do trabalho desenvolvido. Na seqüência, são apresentadas as referências bibliográficas utilizadas no decorrer do trabalho e, em anexo, outros trabalhos relacionados.

2 NEUROIMAGEM E A VISUALIZAÇÃO CIENTÍFICA

Este capítulo inicia apresentando conceitos de neuroimagem e introduzindo noções sobre os tipos de imagens utilizados neste trabalho, bem como suas respectivas importâncias para a avaliação das zonas epileptogênicas. Segue-se então uma breve explicação a respeito dos exames que originam cada categoria de imagem, suas relações e também as lacunas existentes na avaliação dos mesmos.

Na seqüência, são abordados alguns conceitos ligados à visualização científica. São apresentadas suas características gerais, além da relação entre a computação gráfica e possíveis soluções para os problemas identificados na neuroimagem, contextualizando o processamento de imagens na área médica.

2.1 Neuroimagem

Segundo Kuzniecky (1995), a partir da década de 70 houve grande aumento na capacidade de avaliação dos pacientes que apresentavam patologias intratáveis, incluindo a epilepsia. Isso ocorreu graças ao desenvolvimento de técnicas de captação de imagens cerebrais, principalmente a Ressonância Magnética e a Tomografia Computadorizada por Emissão de Fóton Único (SPECT).

A neuroimagem subdivide-se em duas categorias principais: imagens anatômicas (neuroimagem estrutural) e imagens funcionais (neuroimagem funcional). Dado o escopo deste trabalho, somente serão abordados os exames de RM, como ferramenta de obtenção das imagens anatômicas, e a técnica de SPECT, como forma de obtenção das imagens funcionais.

2.1.1 Imagens funcionais e exames de SPECT

Imagens funcionais são aquelas que permitem visualizar o fluxo de sangue no cérebro e seu respectivo funcionamento. São utilizadas na avaliação das disfunções metabólicas, facilitando o diagnóstico dos males causados por essas alterações, como por exemplo, as crises epilépticas. De acordo com Filho (2000), as imagens funcionais são as mais utilizadas para estabelecer relações entre a

intensidade de sintomas mentais e as alterações do funcionamento cerebral, uma vez que fornecem imagens dinâmicas do metabolismo cerebral regional.

Segundo Andraus et al. (2002), os métodos funcionais de neuroimagem começaram a ser utilizados a partir da década de 80, sendo os primeiros estudos com SPECT cerebral relatados em 1983. O desenvolvimento da técnica de SPECT tornou possível a análise das características fisiológicas das crises epilépticas, representando de forma confiável o local do foco epileptogênico.

Em seu trabalho, Oliveira (2005) afirma que investigações com PET (*Positrons Emission Tomography*, Tomografia por Emissão de Pósitrons) demonstraram evidências de alterações no metabolismo de glicose em crises epilépticas focais. Durante estas crises, o metabolismo cerebral é aumentado em áreas limitadas, enquanto em fases pós/intercríticas o consumo de glicose é reduzido. Assim sendo, o SPECT é um procedimento apropriado para medir perfusão cerebral.

Para realização do exame de SPECT, o primeiro passo é a administração intravenosa de substância radioativa ao paciente, feito isso é realizado o mapeamento tri-dimensional da distribuição dessa substância no cérebro. Desse modo, é possível obter imagens do fluxo sanguíneo no cérebro representativas do exato momento da crise (OLIVEIRA et al., 1998). A Fig. 1 ilustra o resultado de um exame SPECT.

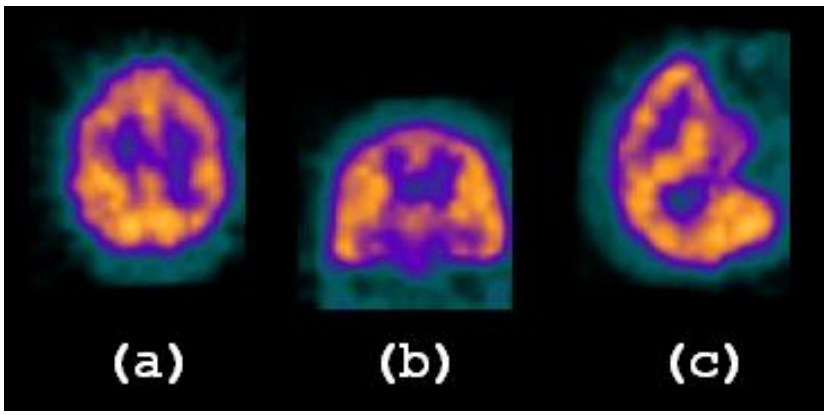


Figura 1 – Imagens de SPECT com aplicação de radiofármaco nos planos a) Axial; b) Coronal; c) Sagital.

Fonte: OLIVEIRA, L. 2005.

2.1.2 Imagens anatômicas e exames de RM

Imagens anatômicas são aquelas que permitem visualizar a estrutura do cérebro, ou de outro órgão em questão. Com elas é possível localizar lesões estruturais e tumores. No caso deste trabalho, essas serão utilizadas como complemento às imagens SPECT, colaborando na localização estrutural do foco epileptogênico. As imagens anatômicas podem ser obtidas por vários tipos de exames, sendo os mais populares Tomografia Computadorizada (TC) e Ressonância Magnética.

O exame por Ressonância Magnética é um dos mais seguros exames por imagem. Não utiliza radiação ionizante e seus agentes de contraste são muito mais seguros se comparados aos utilizados na TC. Muitos dos procedimentos que exigem contraste para realização utilizando TC, podem ser feitos com RM sem o uso de nenhum agente radioativo (JUNIOR, 1999). Além disso, é o método mais eficiente para detectar lesões estruturais grosseiras em pacientes com epilepsia do lobo temporal (OLIVEIRA, 2002).

De acordo com Paiva et al. (1999), exames de RM baseiam-se nas propriedades relativas ao comportamento do núcleo de hidrogênio quando expostas a um campo magnético. A abundância de água nos tecidos torna-os bastante sensíveis à RM. Júnior (1999) explica que as imagens de Ressonância Magnética são formadas submetendo o órgão a ser examinado a um campo magnético uniforme de grande intensidade. Os *spins* dos núcleos de hidrogênio são então alinhados na direção deste campo. Na seqüência, são emitidos pulsos de radiofreqüência que fazem com que os *spins* sofram uma rotação de 90º ou 180º. Ao cessar a emissão da radiofreqüência, os prótons sofrem um relaxamento e tendem a voltar para a sua posição original alinhados com a direção do campo magnético uniforme. Conforme os prótons vão retornando, eles emitem energia e esta é captada pelos sensores para formar a imagem. A RM permite a obtenção das imagens nos planos Sagital, Coronal e Axial (OLIVEIRA, 2005). O resultado de um exame de RM pode ser visto na Fig. 2.

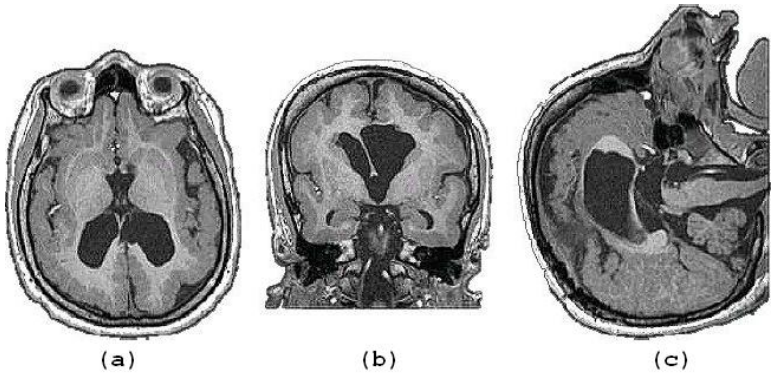


Figura 2 – Imagens de Ressonância Magnética nos planos a) Axial; b) Coronal; c) Sagital.

Fonte: OLIVEIRA, M. 2002.

2.1.3 Neuroimagem e epilepsia

Tradicionalmente, os estudos de neuroimagem são realizados utilizando o método manual de delineamento das regiões de interesse. Entretanto, esse método confere limitações à reprodutibilidade dos estudos e à investigação do volume total do cérebro, pois exige do investigador conhecimentos prévios de neuroanatomia e destreza motora, constituindo-se em um processo muito laborioso e exaustivo (MOREIRA, 2004).

No caso específico da epilepsia, realiza-se uma interpretação visual de imagens lado-a-lado. Imagens SPECT obtidas no período em que o paciente se encontra em crise são comparadas com outras obtidas no período entre as crises, buscando identificar as alterações funcionais que ocorrem durante a crise epiléptica, conforme ilustrado na Fig. 3. Dessa forma, os problemas identificados anteriormente tornam-se uma constante, pois o delineamento das regiões de interesse dependerá basicamente do investigador. Na prática, muitas vezes isso acaba por impedir a identificação de alterações de regiões cerebrais relevantes a uma patologia (VALENTE, 2002 apud MOREIRA, 2004).

Entretanto, os problemas descritos podem ser reduzidos com a automatização dos processos de delimitação e visualização através de técnicas de subtração de imagens. Subtração de imagens é o nome dado ao processo que permite obter a intersecção existente entre imagens do paciente em situações normais e de crise, alcançando assim o foco do problema (OLIVEIRA, 2005).

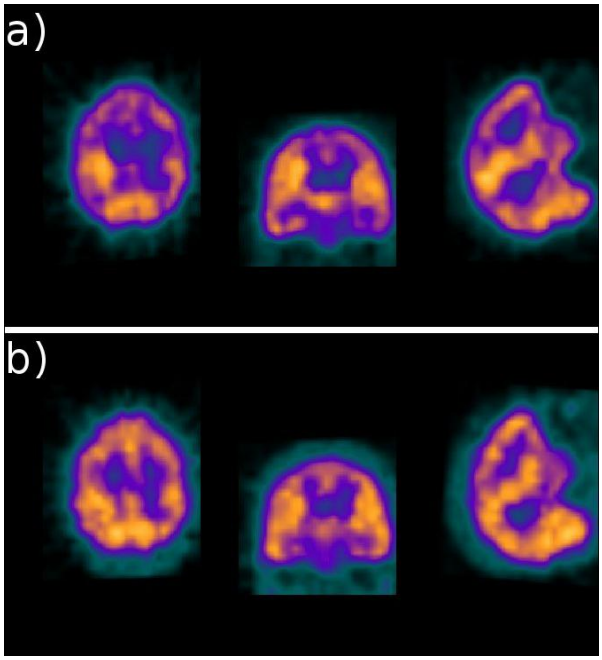


Figura 3 – Comparação visual entre Imagens de SPECT em a) imagem crítica (paciente durante a crise) e em b) intercrítica (paciente fora da crise).

Fonte: OLIVEIRA, L. 2005.

Em geral, o diagnóstico clínico é baseado em informações complementares de diferentes exames. No contexto da epilepsia, a análise complementar pode ser feita utilizando-se imagens SPECT, para obtenção das informações funcionais, compostas com imagens de RM para visualização das informações anatômicas, como ocorre, por exemplo, no método SISCOM. Esse método permite detectar os focos epilépticos a partir da subtração de imagens funcionais de SPECT e localizá-los anatomicamente no exame de RM, permitindo um melhor estadiamento clínico e o planejamento cirúrgico de pacientes com epilepsia intratável (O' BRIEN et al., 1998 apud OLIVEIRA, 2005).

De maneira análoga à comparação por subtração, esse processo pode ser automatizado, utilizando-se o procedimento de fusão de imagens. Fusão de imagens é uma técnica cuja idéia principal é agrupar informações complementares obtidas por exames variados, as quais normalmente são utilizadas de forma isolada. Porém, para realizar a fusão das imagens, estas deverão ser previamente alinhadas geometricamente (OLIVEIRA, 2005).

Maes et al. (1997 apud OLIVEIRA, 2005) afirma que o alinhamento geométrico (ou co-registro) de imagens de várias modalidades é uma tarefa fundamental em numerosas aplicações de processamento de imagens médicas 3D. Segundo Oliveira (2005) isso ocorre porque as imagens para estudos clínicos não são obtidas reproduzindo rigorosamente o posicionamento do paciente nos diferentes exames. Existem ainda outros fatores que tornam as condições ideais muito baixas, como por exemplo: diferenças nas características dos aparelhos (tamanho do *pixel*, espessura de corte e distorções na imagem) e dificuldades de posicionar preventivamente o objeto de correlação com marcadores que serão observados nos dois tipos de imagem com precisão tri-dimensional.

Normalmente, a aquisição dos dados volumétricos na medicina produz imagens compostas por fatias paralelas uniformemente espaçadas, as quais representam cortes transversais ao eixo longitudinal do paciente. O volume é gerado através do empilhamento dessas fatias, mantendo o espaço original entre elas e mapeando os *pixels* em *voxels. Voxel* é o nome dado a cada um dos pontos em um volume, de maneira análoga ao termo *pixel* em uma imagem bi-dimensional. Cada *voxel* tem normalmente associado um número inteiro proporcional ao tom de cinza do *pixel* na imagem correspondente (FALCÃO, 1999). A Fig. 4 ilustra a conversão entre *pixel* e *voxel*.

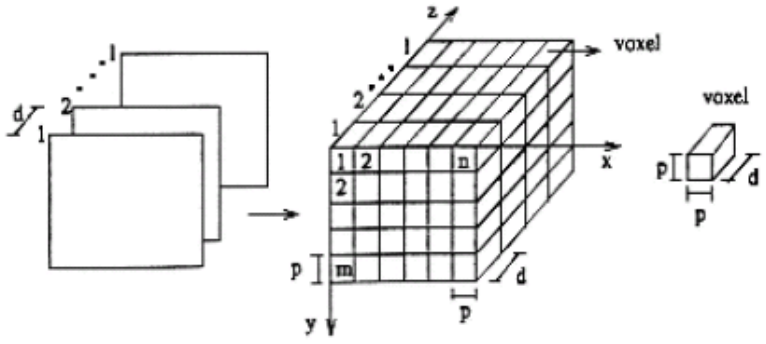


Figura 4 – Mapeamento dos *pixels* em *voxel*s
Fonte: FALCÃO, A. 1999.

2.2 Visualização científica de volumes

Visualização é um termo abrangente e que faz referência a todos os métodos que permitem extrair informações relevantes de conjuntos com dados

complexos. Imagens digitais podem ser consideradas matrizes multidimensionais, cujos índices identificam pontos na imagem. O valor numérico correspondente a cada ponto específico (*pixel*) está associado a uma grandeza física dependente do dispositivo empregado na obtenção da imagem digital (OLIVEIRA, 2002).

A profundidade de uma imagem digital corresponde à quantidade de *bits* utilizada para quantificar os valores dos *pixels*, ou seja, se um *pixel* possui profundidade 1 (um *bit* por *pixel*), este poderá ser pintado com apenas duas cores. A visualização de uma imagem digital em um periférico de saída como, por exemplo, a tela de um monitor é realizada pela conversão do valor numérico de cada *pixel* em luz (FALCÃO, 1999).

O conjunto de técnicas utilizadas na visualização tem como principais objetivos: exibir, explorar e facilitar a compreensão das regiões de interesse (MCCORNICK; BROWN, 1987). Como regra geral, esse conjunto de técnicas deve apresentar boa usabilidade, além de oferecer representações de dados compreensivas, rápidas de serem manipuladas e razoavelmente rápidas de serem renderizadas. Renderização, ou *rendering*, é o processo de conversão da geometria, propriedades, especificações de luz e câmera de um objeto para uma imagem. Seu objetivo é transformar os dados numéricos em primitivas gráficas representáveis pelo computador (LAVORANTE, 2005). Assim, a estratégia de visualização deve permitir ao usuário manipular parâmetros e perceber a imagem resultante instantaneamente (ELVINS, 1992).

Com o passar dos anos, surgiram diversos algoritmos de visualização de superfícies e sólidos. Em relação à visualização de volumes, os algoritmos são divididos em duas categorias: reconstrução por superfície e reconstrução volumétrica. As duas técnicas produzem como resultado final estruturas 3D, cada uma com suas vantagens e desvantagens. A escolha da abordagem mais apropriada a uma determinada aplicação baseia-se nos requisitos específicos a mesma (ROBB, 1999).

2.2.1 Reconstrução por superfície

Também conhecido como *Volume Modeling* (Modelagem de Volume), esta técnica será abordada rapidamente, apenas a título de comparação, dado que não foi utilizada no presente trabalho. A característica básica da reconstrução por

superfície (*surface rendering*) é a necessidade da extração de contornos que definem a superfície da estrutura a ser visualizada. Definidos os valores para a superfície, um algoritmo efetua a construção das mesmas, formando o revestimento do volume para criar a aparência desejada.

A principal vantagem dessa técnica é que, em geral, a quantidade de contorno que constitui uma estrutura a ser reconstruída é pequena, tornando assim o processo de renderização (ou rasterização) desse algoritmo mais rápido (custo computacional menor) se comparado ao tempo e complexidade envolvidos na reconstrução volumétrica. Por outro lado, é necessária a definição exata do contorno da estrutura a ser visualizada. A falta de precisão pode gerar perda na informação a ser visualizada.

Como conseqüência, o processo interativo de visualização torna-se menos dinâmico, visto que a determinação da superfície a ser renderizada deve ser feita especificamente no momento da extração do contorno (ROBB, 1999). Métodos de *surface rendering* apresentam a necessidade do uso de técnicas de *rendering* de polígonos e de reprocessamento do volume para extrair novamente o objeto toda vez que uma alteração da característica do volume a ser visualizado for solicitada. Além disso, essa classe de algoritmos possui a desvantagem de ser incapaz de representar adequadamente fenômenos amorfos, tais como nuvens e fogo. (PAIVA et al., 1999). Alguns exemplos de algoritmos desta classe são o *Contour-Connecting* e *Dividing Cubes*.

Existem métodos mais sofisticados de *surface rendering*, os baseados no reconhecimento e na extração das iso-superfícies. Paiva et al. (1999) define iso-superfície como uma superfície que representa pontos de um valor constante dentro de um espaço de volume e que pode ser renderizada como um único modelo poligonal. Segundo Lorensen e Cline (1987), na criação das iso-superfícies o primeiro passo é definir um valor limite, o qual é chamado valor de *threshold*. Os *pixels* individuais da imagem cujos valores na escala de cinza forem maiores que esse limiar são marcados como parte do objeto e, posteriormente, renderizados para construir a iso-superfície.

Um exemplo de algoritmo baseado em iso-superfícies é o *Marching Cubes*. Criado para gerar uma malha triangular de uma iso-superfície, esse algoritmo gera a malha com base nos 256 modos que uma superfície pode passar por uma célula e reduz por simetria estes 256 casos para 15. Então os triângulos gerados são

passados para o sistema de rasterização, que mostra na tela a superfície reconstruída. A principal desvantagem desta técnica é a dificuldade de selecionar um bom valor de *threshold* para se visualizar aquilo que se está procurando (JUNIOR, 1999).

Porém, esse algoritmo é bastante sensível ao ruído e tem dificuldades de ser adaptado para visualizar superfícies transparentes. Isso faz com que sua utilização, no caso específico das imagens médicas, apresente alguns problemas. O ruído pode, muitas vezes, apresentar falsos negativos, ou seja, casos em que existe, por exemplo, uma pequena lesão e nada é detectado pela má escolha no valor de *threshold* (JUNIOR, 1999).

2.2.2 Reconstrução volumétrica e algoritmos de *Ray-tracing*

Embora muitos algoritmos tenham sido estudados e propostos para implementação da reconstrução volumétrica, nenhum deles é tão simples e poderoso quanto o *Ray-tracing* (SILVA, 1994). A designação *Ray-tracing* aplica-se a um conjunto de algoritmos desenvolvidos a partir do trabalho inicial de Appel publicado em 1968, e não apenas a um algoritmo em particular (LOPES, 2004). Existem quatro abordagens básicas para o problema de reconstrução volumétrica: ordem da imagem (*forward mapping*), ordem dos objetos (*backward mapping*), métodos baseados em mudança de base e métodos baseados em transformações geométricas do volume (PAIVA et al., 1999).

Dentro da divisão citada, o *Ray-tracing* é o representante da primeira abordagem (PAIVA et al., 1999). A idéia básica do algoritmo é muito simples: um observador se senta em frente a uma tela plana transparente, a partir de seus olhos são emitidos diversos “raios visuais” que vão atravessar os pontos da tela e bater nos objetos tri-dimensionais, conforme representado na Fig. 5. Feito isso, o ponto da tela que foi atravessado pelo raio é “pintado” com a cor do objeto atingido pelo mesmo. Já as variações mais avançadas do algoritmo incluem sombras, reflexão e transparência (SILVA, 1994).

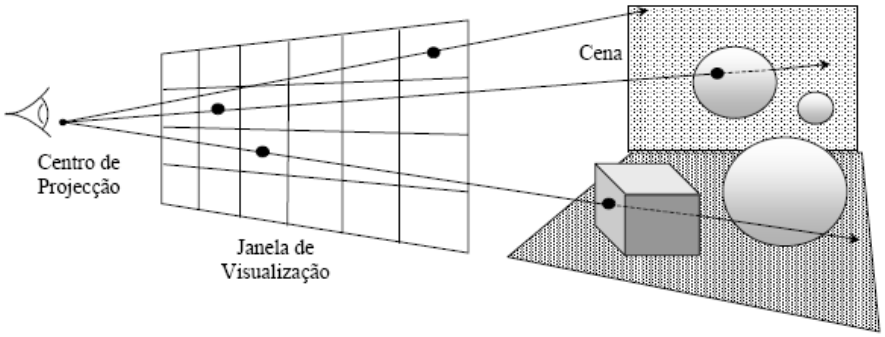


Figura 5 – Abstração do algoritmo fundamental utilizado para reconstrução volumétrica - *Ray-tracing*.

Fonte: SANTOS, E. 1994

Já a abordagem baseada na ordem dos objetos percorre o volume e, para cada *voxel*, encontra os *pixels* que são afetados pela sua contribuição. Um exemplo dessa categoria é o algoritmo *v-buffer*. Os métodos baseados em transformações geométricas do volume têm como exemplo o algoritmo *Slice-shearing*. Essa classe é formada de algoritmos que aplicam transformações geométricas afins ao volume, de modo a gerar uma simplificação das situações anteriores, com o objetivo de otimizar o processo. De maneira semelhante, os métodos baseados em mudança de base realizam uma transformação do volume para outro sistema de coordenadas e, então, empregam uma das duas estratégias iniciais para percorrer o volume e gerar a imagem (PAIVA et al., 1999). Por ser o mais difundido e poderoso dos algoritmos, além de produzir resultados de ótima qualidade e ser passível de paralelização, conforme explicado no decorrer deste trabalho, o algoritmo escolhido para este trabalho foi o de *Ray-tracing*, representante da abordagem “ordem da imagem”.

Utilizando o *Ray-tracing*, nenhuma extração de contorno prévia é necessária. Não é preciso passar por etapas intermediárias de geração de malhas poligonais ou de extração de iso-superfícies, ou seja, não há nenhum modelo geométrico intermediário entre o volume de dados e imagem a ser visualizada (JUNIOR, 1999). Nessa abordagem todo o volume é reconstruído no processo e nenhuma informação da imagem é perdida. Outra vantagem bastante importante é que a imagem renderizada pode ser determinada dinamicamente, através da modificação de parâmetros. Com isso, o conteúdo dos dados é exibido diretamente a partir das

escalas de cinza do volume, o que acarreta maior confiabilidade e precisão (ROBB, 1999).

Em relação aos custos computacionais, Whitted (1980) coloca que o algoritmo de *Ray-tracing* gasta entre 75% e 95% de seu tempo determinando as interseções do raio com os objetos. Logo, a eficiência da rotina de intersecção raio/objetos afeta significativamente a eficiência do algoritmo.

Quando o raio visual atinge um objeto visível, o ponto da tela a ser pintado possui características do ponto do objeto que foi atingido, mas não é necessariamente da cor do objeto. Diversos fatores influem no cálculo da cor do ponto, como a iluminação, por exemplo. Se girarmos um objeto, a impressão visual da cor de um ponto deste muda, ficando mais escuro à medida que a luz que incide sobre ele fica mais perpendicular em relação à normal no ponto. A textura e rugosidade do objeto são fatores que também influem no cálculo da sua cor. A versão simplificada do algoritmo de *Ray-tracing* é conhecida também como *Ray-casting* (SILVA, 1994) e é o mais usado para a visualização de volumes quando necessita-se de imagens de alta qualidade (ELVINS,1992).

Silva (1994) chama atenção para o fato de que os raios lançados no método de *Ray-tracing* seguem o caminho inverso ao da luz. Isso ocorre devido ao fato de que a partir de uma fonte luminosa partem bilhões de fótons, os quais formam os raios luminosos. Porém, apenas uma ínfima parte deles chega aos olhos de um observador. Simular tal mecanismo seria impraticável para o computador, fazendo com que uma simples cena levasse anos para ser reconstruída. Por outro lado, lançando raios a partir do observador, garante-se que serão calculados apenas os raios efetivamente captados pelo observador.

Entretanto, o algoritmo básico de *Ray-tracing* apresentado na Fig. 5 encontra-se incompleto, pois se limita a atribuir a cada *pixel* da imagem a cor do ponto de intersecção do raio primário correspondente ao *pixel* com o objeto mais próximo da origem do raio, sem atender a outras contribuições para a iluminação desse ponto de intersecção além da sua iluminação direta. Para ter em conta todas as outras contribuições para a iluminação é necessário inspecionar recursivamente a cena através da geração de raios secundários (LOPES, 2004). Um exemplo do funcionamento desta abordagem é apresentado na Fig. 6.

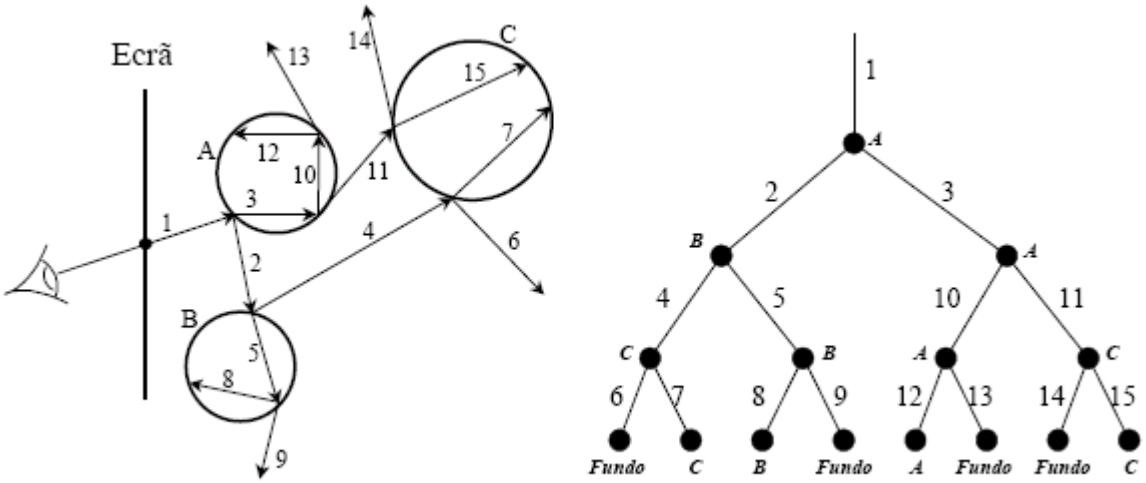


Figura 6 – Recursividade em *Ray-tracing.* Os raios considerados (à esquerda) encontram-se organizados numa estrutura de árvore (à direita).

Fonte: SANTOS, E. 1994.

Uma característica em comum em todas as implementações do *Ray-tracing* é o fato de possuírem custo computacional bastante elevado. Isso se deve principalmente a três componentes do processo: operações de ponto flutuante utilizadas na transformação dos objetos 3D, rasterização dos polígonos transformados e aplicação de *antialiasing* na imagem sintetizada (LIN et al., 2001). A implementação completa recursiva apresenta ainda uma carga computacional adicional, a qual consiste em realizar os cálculos dos raios secundários de reflexão e transmissão (LOPES, 2004).

Em complemento a isso, Silva (1994) explica que o cálculo da intersecção do raio com os objetos da cena é uma operação de grande custo computacional, principalmente porque em geral quando é lançado um raio, este não intercepta nenhum dos objetos da cena, mas mesmo assim são feitos todos os cálculos de intersecção do raio com as superfícies dos objetos. Para evitar essas intersecções desnecessárias, uma alternativa é fazer a intersecção do raio com um sólido limitante, antes de interceptá-lo com a superfície do objeto. A idéia baseia-se na premissa de que se o raio não intercepta o sólido limitante do objeto, então este raio não interceptará o objeto e, portanto, deve ser descartado. Assim, eliminam-se as intersecções desnecessárias.

Toda essa computação intensiva é um aspecto a ser considerado no desenvolvimento de uma ferramenta para visualização interativa. Isso porque, para que ela apresente navegação satisfatória, deve ser capaz de operar com as taxas de geração e repetição de em torno de 10 a 20 *frames* (quadros) por segundo (ROBB, 1999). Por outro lado, Santos (1994) observa que, embora o custo computacional da aplicação seja alto, existe um alto grau de paralelização em potencial, comprovado por outros trabalhos existentes empregando técnicas de *Ray-tracing* paralelas e *Ray-tracing* otimizado. Em complemento, Paiva et al. (1999) afirma que o alto custo computacional desse algoritmo pode ser paralelizado, uma vez que os valores dos *pixels* são determinados através do lançamento de raios independentes entre si. Schroeder et al. (2002) coloca ainda que vários métodos de aceleração do *Ray-casting* são possíveis, desde que para a aplicação em questão não cause problemas sacrificar um pouco a flexibilidade do algoritmo e se disponha de memória adicional.

Neste projeto, optou-se por utilizar a classe de algoritmos de reconstrução volumétrica, e, dentro desta, o algoritmo de *Ray-casting,* combinação esta que atende plenamente as condições e objetivos aos quais o trabalho se propunha. O nicho de possibilidades aberto pela reconstrução volumétrica é bastante amplo, uma vez que é possível a visualização de mais de um objeto contido no volume, bem como de suas porções interiores (PAIVA et al., 1999).

O algoritmo *Ray-casting*, da mesma forma que a grande maioria dos algoritmos de *Ray-tracing*, é de complexidade $O(n^3)$ (SCHROEDER et al., 2002). É um algoritmo capaz de produzir imagens de ótima qualidade para a área médica, pois pode utilizar, sem grandes perdas, um modelo de iluminação mais elaborado com cálculo em tempo de execução. Possui ainda a vantagem de não precisar de *hardware* especial para execução (JUNIOR, 1999).

Outro aspecto importante na utilização de reconstrução volumétrica é a classificação dos dados. Esse passo consiste em mapear os valores escalares dos *voxels* em atributos de cor e opacidade. Esse processo recebe o nome de segmentação ou classificação (SCHROEDER et al., 2002). O mais simples dos métodos de segmentação é o *threshold*, onde, conforme visto anteriormente, estipula-se um valor limiar e somente os *pixels* com valor na escala de cinza maior do que o valor de *threshold* definido são visualizados (LORENSEN; CLINE, 1987).

No caso de imagens médicas a segmentação é uma tarefa mais complexa, pois cada imagem pode apresentar diversos tecidos, cada um com diferentes

propriedades físicas, o que torna difícil a definição dos valores de *threshold*. Uma alternativa é a utilização de parâmetros adicionais com intuito de definir faixas de valores para segmentação. Porém, embora a segmentação por intervalos permita resultados melhores e maior controle sobre esses, ela aumenta consideravelmente a complexidade na escolha dos parâmetros corretos (SCHROEDER et al., 2002).

Do ponto de vista de desempenho, a segmentação também é fundamental. Eficiência e compactação são requisitos importantes não apenas devido à complexidade inerente aos métodos de reconstrução volumétrica, mas também ao tamanho dos conjuntos de dados. Um modelo geométrico formado por um milhão de primitivas é geralmente considerado grande, enquanto um volume com um milhão de *voxels* é usualmente visto como pequeno (SCHROEDER et al., 2002). Com isso, percebe-se claramente que deve ser tomado cuidado especial nas decisões relativas ao armazenamento de informações dos *voxels* e o tempo gasto para processá-los.

2.3 Conclusões

A comparação visual lado-a-lado de imagens médicas é um processo laborioso e com resultados nem sempre confiáveis. Por isso, o uso de ferramentas computacionais que buscam automatizar e facilitar esse procedimento tem ganhado cada vez mais espaço. Nesse contexto, o desenvolvimento de um *software* interativo, baseado na fusão de imagens 3D proposta por Oliveira (2005), surge como alternativa na detecção e visualização intuitiva do foco epileptogênico.

O algoritmo *Ray-casting* demonstrou-se o mais adequado para a implementação deste projeto, pois atende todos os requisitos identificados. Isso porque, além de gerar imagens de ótima qualidade na área médica, o *Ray-casting* permite visualizar mais de um objeto contido no volume, bem como suas porções interiores. Essa possibilidade vai ao encontro da solução aqui proposta onde se deseja reconstruir o foco epileptogênico no interior da estrutura cerebral gerada a partir do exame de Ressonância Magnética.

Além disso, as desvantagens identificadas no *Ray-casting* podem ser amenizadas com a exploração do paralelismo em potencial do algoritmo, visto que, essas são relativas aos custos computacionais e que a estrutura do problema permite sua decomposição em fluxos de execução independentes - *threads*. É necessária também atenção especial para o processo de segmentação das

imagens, dada a quantidade de dados a serem reconstruídos e visualizados. Uma estratégia adequada para segmentação diminui não apenas os custos computacionais, mas também torna a visualização mais limpa e, portanto, eficiente.

3 PROCESSAMENTO PARALELO - CONCEITOS E MODELOS

A perspectiva da exploração do paralelismo nos sistemas computacionais é tão antiga quanto os primeiros computadores. Trabalhos desenvolvidos por von Neumann, na década de 40, já ponderavam a possibilidade de utilizar algoritmos paralelos (YAMIN, 1999). O emprego do paralelismo já passou por fases de extremo otimismo, quando era visto como única e promissora solução para as mais complexas aplicações computacionais, bem como por fases pessimistas, nas quais foi encarado como um nicho de mercado decadente.

Este capítulo apresenta inicialmente aspectos gerais e históricos da aplicação de paralelismo como alternativa para solução de problemas na Ciência da Computação. Feita essa introdução, alguns conceitos importantes para o desenvolvimento de *software* paralelo são colocados.

Na seqüência, são apresentados os principais modelos de arquiteturas para processamento paralelo. Para efeito de discussão, serão abordados apenas os tipos primitivos de cada modelo, embora, via de regra, eles se subdividam em variantes que buscam preencher suas lacunas de desempenho. Atenção especial será dada à classe de arquiteturas MIMD (*Multiple Instruction Streams Multiple Data Streams,* Múltiplos Fluxos de Instruções Múltiplos Fluxos de Dados) devido a sua relevância para este trabalho. Por fim, o capítulo trata sobre modelos para desenvolvimento de *software* paralelo.

3.1 Aspectos gerais e contextualização

Disponível comercialmente há aproximadamente três décadas, o processamento paralelo destaca-se em aplicações numericamente intensivas. Registros apontam que no período de 1944 a 1947 Stibitz e Williams teriam desenvolvido, nos laboratórios da *Bell Telephone*, aquele que é considerado um dos primeiros trabalhos envolvendo paralelismo com implementação em *hardware*, o sistema MODEL V. Esse sistema é formado por dois processadores e três canais de

entrada e saída, possibilitando a execução simultânea de dois programas distintos (YAMIN, 1999).

Skillicorn (1994) observa que um problema central em computação paralela é o desencontro entre as necessidades do *software* e as propriedades disponíveis nas arquiteturas paralelas em que eles serão executados. Porém, essa distância oscila rapidamente, visto que costuma ser grande a diferença entre o tempo de vida útil apresentado pelo *software* (vários anos ou até mesmo décadas, no caso de *softwares* de grande porte) e o apresentado pelo *hardware* (poucos anos).

Os processadores ou CPUs (*Central Processing Unit*, Unidade Central de Processamento) disponíveis atualmente no mercado atingem freqüências de *clock* em torno dos 3 GHz. Esse valor está próximo do limite suportado pelas atuais tecnologias, o que tem limitado o crescimento da velocidade nos novos processadores (ASANOVIC et al., 2006; CAVALHEIRO; SANTOS, 2007).

Hwang (1993) já previa o problema de limitação no crescimento do *clock*, destacando dois aspectos como principais obstáculos do processo: o consumo de energia com sua conseqüente dissipação térmica e a dimensão do processador e seus componentes acessórios. Componentes projetados para *clocks* elevados, além de apresentarem índices de consumo bem mais elevados que os similares projetados para *clocks* menores, têm potencializados os fenômenos de capacitâncias e indutâncias parasitas. Soma-se ainda o fato de que a barreira física dos condutores já está quase em seu limite, tornando o aumento de freqüência um esforço com custos financeiros muito elevados em relação aos ganhos em performance (HSIN MA, 2005).

É importante observar que apenas o crescimento da freqüência do *clock*, isoladamente, não é capaz de proporcionar aumento considerável no desempenho do computador. Outros aspectos, como a eficiência (largura de banda) no acesso à memória, devem acompanhar o crescimento do *clock*. O comportamento conhecido como “Gargalo de von Neumann” caracteriza esse problema, definindo que o poder de processamento disponibilizado para computação de um problema é limitado em função da taxa de transferência de dados e instruções entre a memória e o processador. O uso de paralelismo na arquitetura pode vir a ser uma forma de superar este fator limitante (YAMIN, 1999).

A solução encontrada pelas fabricantes para os problemas acima descritos tem sido empacotar diversos processadores em um único CPU, os chamados

processadores *multi-core.* Entende-se por *multi-core* processadores que possuem dois ou mais *cores*. Essa solução se enquadra na classe das arquiteturas paralelas do tipo SMP (*Symmetric Multiprocessor*, Multiprocessadores Simétricos) e vem se tornado padrão nas arquiteturas para processamento paralelo (ASANOVIC et al., 2006). Entretanto, as aplicações por si só não tiram proveito dos benefícios dessa solução, logo, precisam ser escritas especialmente para este novo tipo de arquitetura (CAVALHEIRO; SANTOS, 2007).

Em complemento a isso, é sabido que os fenômenos naturais são inerentemente paralelos. Portanto, nada mais natural que expressar as computações pertinentes aos mesmos de forma paralela. Em algumas situações a ordem de execução é importante para o melhor entendimento do problema real, já em outras ela é irrelevante (SKILLICORN; TALIA, 1998). Todavia, na programação seqüencial não é possível trabalhar com essa flexibilidade, pois é sempre necessário definir a ordem para a execução das ações. Assim, percebe-se claramente o papel dos paradigmas de *hardware* e *software* para o desenvolvimento paralelo.

Por fim, é importante observar que mesmo com o crescimento da velocidade dos processadores, a computação paralela possibilita muitas vezes uma relação custo/benefício melhor que a obtida ao utilizar equipamentos com um só processador de última geração (YAMIN, 1999). Isso ocorre, em grande parte, devido aos custos. Agrupar em um equipamento paralelo processadores mais antigos provê uma alternativa computacional com custo competitivo e suporte natural a técnicas de tolerância a falhas (característica inerente à redundância de *hardware*). Um exemplo desse tipo de solução são os *clusters Beowulf*. A infra-estrutura de um *cluster Beowulf* é totalmente escalável, formada a partir de arquiteturas de *hardware* populares interligadas em uma rede privada e executando *software* livre em sistema operacional *Linux* (BEOWULF PROJECT OVERVIEW, 2007).

Dentro desse contexto, o paralelismo volta a aparecer como solução promissora, entrando mais uma vez, em uma fase otimista. Porém, dessa vez, a busca por soluções dos problemas em computação de uso geral parecem estar seguindo na direção das arquiteturas paralelas por um caminho irreversível (ASANOVIC et al., 2006).

3.2 Terminologia em computação paralela

Para o correto entendimento das discussões acerca de computação paralela são apresentados, a seguir, os principais termos que compõem este universo. Cabe observar que estes conceitos permanecem válidos em todo o contexto de processamento paralelo, independentemente da arquitetura e/ou modelos de *software* a que estiverem fazendo referência.

3.2.1 Granularidade

Um requisito fundamental para o uso de paralelismo é dividir o problema em atividades que possam ser executadas simultaneamente, independentemente da arquitetura que for utilizada. Granularidade, ou granulosidade, de uma atividade reflete seu volume de cálculo, podendo ser fina ou grossa. No primeiro caso, a sincronização entre os fluxos de execução é freqüente, enquanto na granulosidade grossa a freqüência de sincronização é menor. Quanto maior a granularidade menor serão os custos com criação de processos e com a comunicação entre eles (GRAMA et al., 2003).

É comum existir abundância de paralelismo com baixa granulosidade, o qual, via de regra, não pode ser explorado com eficiência, pois resulta em aumento do volume de comunicação entre os processos e custos de sincronização. O desempenho da arquitetura paralela é obtido através de um equilíbrio entre granulosidade e comunicação (YAMIN, 1999).

Na implementação de uma solução paralela, a opção entre granularidade grossa ou fina depende de vários aspectos. Por exemplo, na programação por passagem de mensagem é interessante se reduzir a comunicação entre processos devido ao tempo gasto na comunicação entre os computadores, o que pode sugerir o aumento do número de instruções por processo.

3.2.2 Medidas de s*peedup* e eficiência

Speedup é a medida relativa entre a performance de um programa seqüencial e seu correspondente paralelo. Em outras palavras, indica o número de

vezes que o programa paralelo é mais rápido que o seqüencial. O cálculo do *speedup* é dado pela Equação 3.1, onde T(s) é o tempo de execução do programa seqüencial e T(p) representa o tempo gasto pelo programa paralelo (QUINN, 1994).

$$Speedup = T(s) / T(p) \quad (3.1)$$

Como regra geral, o aumento do número de processadores implica em aumento do *speedup*. Porém, essa relação é verdadeira apenas até atingir certo ponto, chamado ponto de saturação. Após o ponto de saturação, as perdas com relação à sincronização e comunicação entre os processos paralelos são maiores que os ganhos com a adição de mais processadores. Além disso, a chamada *Lei de Amdahl* define que todo programa paralelo possui uma fração “f” seqüencial que limita seu *speedup.* Assim, o *speedup* máximo é dado pela Fórmula 3.2, onde f é a fração seqüencial do programa e p o número de processadores (QUINN, 1994).

$$Speedup \text{ máximo} = 1 / (f + (1 – f) / p) \quad (3.2)$$

A *Lei de Amdahl* demonstra que o gargalo de uma aplicação paralela é sua parte seqüencial. Conforme a fatia seqüencial do programa aumenta, o *speedup* diminui. Portanto para atingir um bom *speedup*, é necessário tornar a parte seqüencial do programa a menor possível. A *Lei de Amdahl* possui uma extensão que considerado no cálculo do *speedup* máximo não apenas o gargalo dos passos seqüenciais, mas também o *overhead* gerado pelas tarefas de sincronização, comunicação, etc. (PASSOS, 2006).

Eficiência é uma medida utilizada em conjunto com o *speedup*. Demonstra a fração de tempo correspondente ao uso dos processadores durante a computação e é expressa, geralmente, em valores percentuais. É calculada de acordo com a Fórmula 3.3, onde p é o número de processadores disponíveis (TORELLI, 2005).

$$\text{Eficiência} = speedup / p \quad (3.3)$$

De maneira contrária ao *speedup*, o aumento do número de processadores costuma diminuir a eficiência, independentemente da existência de um ponto de saturação (YAMIN, 1999). Isso porque a computação é dividida entre um número maior de processadores e também devido ao aumento da carga de comunicação e

sincronização que, por sua vez, implica no uso menos eficiente de cada processador individualmente.

3.2.3 Custo, escalabilidade e *overhead*

Custo corresponde ao tempo gasto na execução de uma determinada computação. É considerado ótimo para um determinado algoritmo paralelo quando o custo da execução paralela é proporcional ao custo da execução seqüencial, considerando-se o número de processos paralelos (QUINN, 1994).

Escalabilidade, por sua vez, é um termo que pode ser aplicado tanto em relação ao *hardware* como ao *software*. No primeiro caso, é usado para indicar que a arquitetura permite expansão no seu tamanho físico, sendo que, isso faz com que seu desempenho também aumente. No contexto do *software*, uma forma direta de atingir a escalabilidade é dividir o algoritmo em mais tarefas do que o número de processadores disponíveis, pois assim à medida que o número de processadores crescer, o número de tarefas por processador será reduzido sem necessidade de modificar o algoritmo (PASSOS, 2006).

Overhead, é um conceito que faz referência a qualquer processamento ou armazenamento em excesso. Pode ser, por exemplo, utilizado para identificar tempo de computação acima do normal ou tempo de acesso à memória excessivo, dentre outros. No contexto da computação paralela, *overhead* designa o tempo extra utilizado para coordenar as tarefas paralelas, despendido, basicamente, na comunicação entre processos (GRAMA et al., 2003).

3.3 Modelos de arquiteturas para processamento paralelo

A teoria acerca de processamento paralelo foi desenvolvida após a tecnologia e encontra-se, portanto, ainda imatura em alguns aspectos. Deste modo, a teoria nem sempre sugere caminhos e limites para a exploração tecnológica (SKILLICORN; TALIA, 1998). Com relação às arquiteturas, Flynn (1972) propôs uma classificação dos sistemas computacionais baseados na relação entre o fluxo de instruções e o fluxo de dados, os quais podem ser únicos ou paralelos. Paralelismo de instruções (ou funcional) ocorre quando cada tarefa executa cálculos diferentes para resolver um problema, sejam eles sobre um mesmo conjunto de

dados ou sobre dados diferentes. Paralelismo de dados acontece quando um processo executa uma mesma série de operações sobre diferentes dados. O fluxo de instruções e o fluxo de dados são considerados independentes, assim sendo, Flynn (1972) definiu quatro combinações possíveis:

- SISD - *Single Instruction Stream Single Data Stream,* Fluxo Único de Instruções Fluxo Único de Dados;
- SIMD - *Single Instruction Stream Multiple Data Streams,* Fluxo Único de Instruções Múltiplos Fluxos de Dados;
- MISD - *Multiple Instruction Streams Single Data Stream*, Múltiplos Fluxos de Instruções Fluxo Único de Dados;
- MIMD - *Multiple Instruction Streams Multiple Data Streams*, Múltiplos Fluxos de Instruções Múltiplos Fluxos de Dados;

Arquiteturas do tipo SISD correspondem aos computadores seqüenciais convencionais, onde uma única unidade de controle decodifica seqüencialmente as instruções que operam sobre um único conjunto de dados. A classe de arquiteturas SIMD corresponde aos processadores matriciais, nos quais uma única unidade de controle ativa diversos elementos processadores. A unidade de controle está submetida a um único programa e repassa suas instruções para os processadores que por sua vez as executam concorrentemente sobre os dados em suas memórias locais. A divisão de arquiteturas MISD é considerada nos estudos teóricos, porém, não existem computadores reais enquadrados nesta categoria (YAMIN, 1999).

Nas arquiteturas da classe MIMD, existem dois ou mais processadores que executam fluxos de instruções distintos sobre fluxos de dados próprios, ou seja, múltiplas unidades de controle operam simultaneamente de forma independente umas das outras. Na prática, isso significa a possibilidade de executar vários programas diferentes, ou, várias instâncias de um mesmo programa ao mesmo tempo (TORELLI, 2005). É nesta categoria que se enquadra a grande maioria dos sistemas paralelos atuais, incluindo as arquiteturas do tipo SMP que é a classe escolhida para a implementação e execução do presente trabalho.

Johnson (1988), julgando a taxonomia de Flynn incompleta no que tange às arquiteturas MIMD, propôs em seu trabalho uma extensão a taxonomia de Flynn. Em sua proposta, Johnson (1988) subdivide o grupo das arquiteturas MIMD de acordo com a estrutura de memória (global ou distribuída) e com os mecanismos de comunicação/sincronização (variáveis compartilhadas ou passagem de mensagem).

A Fig. 7 ilustra a divisão sugerida, onde a classe das categorias MIMD passa a ser composta pelas seguintes subclasses:

- GMSV (*Global Memory Shared Variables*, Memória Global Variáveis compartilhadas): Engloba as máquinas que contêm vários processadores e que compartilham a mesma memória;
- GMMP (*Global Memory Message Passing,* Memória Global Passagem de Mensagens): Não existe, na prática, um computador com esta classificação;
- DMSV (*Distributed Memory Shared Variables*, Memória Distribuída Variáveis Compartilhadas): Engloba múltiplos computadores interconectados e que compartilham a memória distribuída;
- DMMP (*Distributed Memory Message Passing,* Memória Distribuída Passagem de Mensagens): Engloba múltiplos computadores interconectados e que se comunicam via passagem (troca) de mensagens.

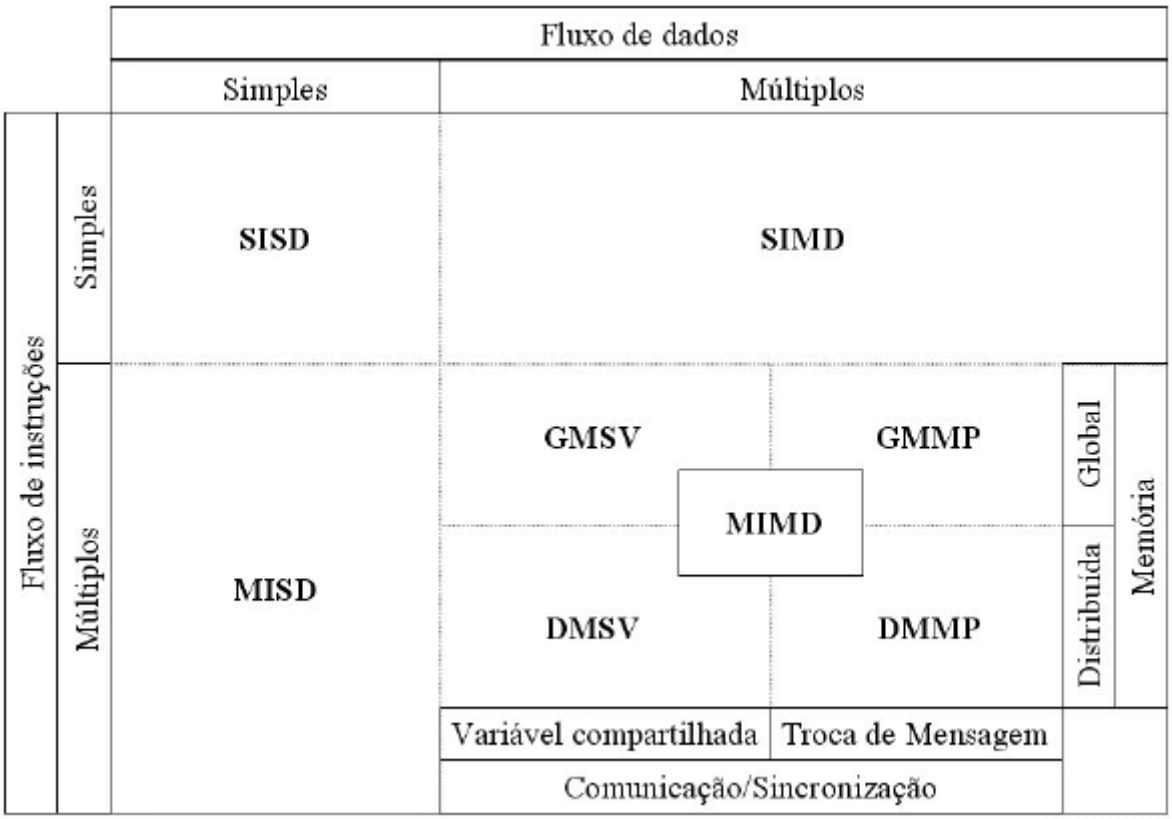


Figura 7 – Extensão da taxonomia de Flynn sugerida por Johnson subdividindo a classe das arquiteturas MIMD.

Fonte: PASSOS, L. 2006.

Em arquiteturas com memória distribuída não ocorre compartilhamento de memória, cada processador tem uma memória própria e os diversos processadores se comunicam trocando mensagens através da rede. Dentre os sistemas MIMD com memória distribuída, os agregados de computadores (ou *clusters*) são os que apresentam maior sucesso atualmente. Um agregado consiste basicamente de uma coleção de computadores independentes e de uma rede que os interliga, criando a ilusão de ser um computador único. Cada computador que compõe um *cluster* é chamado nó (STALLINGS, 2002). Uma vez que este tipo de arquitetura não foi utilizado neste trabalho, ela não será abordada em maiores detalhes.

Nas arquiteturas MIMD com memória compartilhada, existe uma memória comum, acessada de forma uniforme, a todos os processadores, os quais se comunicam lendo e escrevendo nela. Estas arquiteturas são também denominadas de multiprocessadores simétricos (SMP) (MORSE, 1994).

3.3.1 Multiprocessadores simétricos - SMP

Historicamente, a exploração do paralelismo existente entre as instruções de um único *thread* de controle obteve ganhos consideráveis. Para isso, técnicas micro-arquiteturais complexas eram aplicadas, como previsão de desvios, execução fora de ordem, entre outros. Entretanto, no decorrer da década de 90, esse caminho ficou saturado. As técnicas atingiram níveis muito complexos e não apresentavam mais ganhos satisfatórios de desempenho (CAVALHEIRO; SANTOS, 2007). Na busca por soluções alternativas, ao final dos anos 90, pareceu haver um consenso de que a exploração do paralelismo em nível de *threads* era o caminho mais indicado a seguir, iniciando assim a popularização das arquiteturas SMP.

A flexibilidade de uso é uma das principais razões para o sucesso comercial desse tipo de arquitetura. Os processadores em uma arquitetura SMP são todos funcionalmente idênticos, ou seja, capazes de executar as mesmas operações (TORELLI, 2005). Por isso, o processador que a compõe pode ser visto como uma máquina seqüencial tradicional e a existência dos demais processadores, bem como da memória em comum, pode ser abstraída.

Os processadores em uma arquitetura SMP operam sobre o controle de um único sistema operacional responsável pelo escalonamento dos processos (ou *threads*) de usuário e de sistema entre os processadores disponíveis, ou seja,

qualquer processo pode ser executado em qualquer processador (TORELLI, 2005). Em conseqüência dessa particularidade, é possível utilizar nesse modelo de arquiteturas *softwares* seqüenciais já existentes sem nenhuma modificação, e, ainda assim, tirar proveito do paralelismo. Para isso, basta que o sistema operacional escalone os processos relativos aos diversos programas seqüenciais do usuário de forma que executem simultaneamente em diferentes processadores.

Como a memória é um recurso compartilhado, o número de processadores em uma arquitetura SMP não costuma ultrapassar 20, pois dessa forma é possível evitar as complicações previstas pelo Gargalo de von Neumann (YAMIN, 1999). Uma alternativa adotada para aumentar o desempenho é o uso de memória *cache* entre o processador e a memória global.

As arquiteturas SMP possuem uma série de vantagens em relação às demais arquiteturas paralelas. Yamin (1999) destaca como principais aspectos positivos dos multiprocessadores simétricos os seguintes itens:

- Abstração de localidade do processador: o programador pode abstrair a localidade do processador, visto que não será responsabilidade do programador fazer a distribuição dos processos entre eles. A troca de mensagens é feita através de um mecanismo elegante de escrita em variáveis comuns, o que acontece com desempenho bastante elevado, pois a memória é fisicamente compartilhada.
- Semelhança com arquiteturas convencionais: os multiprocessadores de memória compartilhada, por possuírem sistemas operacionais (SOs) e ambientes de programação bastante semelhantes aos das arquiteturas clássicas, permitem que se continue utilizando o *software* seqüencial enquanto soluções paralelas são efetivamente implementadas.
- Facilidade de uso múltiplo: os processadores podem ser alocados tanto individualmente como em grupos, seja para vários programas de vários usuários, vários programas de um único usuário ou programas paralelos.
- Maior compartilhamento dos recursos: a memória comum facilita o compartilhamento de estruturas de dados globais. De forma análoga, os recursos de entrada/saída (E/S) e de memória virtual podem ser aproveitados por todos os nodos processadores.

A popularização das arquiteturas SMP entre usuários domésticos ganhou força em 2004 com o lançamento dos processadores, baseados no conceito de execução *multithreaded,* Pentium 4 *Hyper Threading* (HT) da Intel*.* A tecnologia HT simula a existência de dois processadores, permitindo assim a execução de dois *threads* simultaneamente (INTEL CORPORATION, 2007). Porém, embora as implementações *Hyper Threading* superassem em parte o problema original (exploração do paralelismo), a complexidade dos processadores continuava crescendo (CAVALHEIRO; SANTOS, 2007). Então, em 2005, a indústria da computação mudou seu curso quando a Intel anunciou que daquela data em diante seus sistemas de alta performance seriam baseados em múltiplos processadores dotados de múltiplos *cores* (núcleos) de processamento (ASANOVIC et al., 2006). Além dos fatores relacionados à exploração do paralelismo, questões de projeto e econômicas também contribuíram para essa decisão.

Economicamente, destaca-se o fato de que o desenvolvimento de novos CPUs, sem a necessidade de maiores mudanças no design geral da arquitetura, reduz significativamente o custo e os riscos do projeto, pois um número maior de componentes pode ser reaproveitado (CAVALHEIRO; SANTOS, 2007). Além disso, uma vez que o desenvolvimento de *software* para arquiteturas SMP com vários processadores já era comum, os riscos quanto ao desenvolvimento de *software* para processadores *multi-core* eram pequenos.

O fato de colocar vários *cores* em um único *chip* faz com que os processadores *multi-core* utilizem espaço menor de circuito impresso (*Printed Circuito Board - PCB*) do que as arquiteturas SMP com múltiplos *chips*. Essa proximidade física entre os múltiplos núcleos traz vantagens em relação à velocidade e ao consumo de energia (HSIN MA, 2005; CAVALHEIRO; SANTOS, 2007).

No que tange à velocidade, destaca-se que os circuitos de coerência de memória podem operar em taxas mais altas do que as possíveis quando o sinal viaja fora do *chip*. Isso porque, uma vez que os sinais percorrem distâncias menores, sua degradação é menor, permitindo enviar uma quantidade maior de dados em cada período de tempo. A energia necessária para enviar sinais dentro de um mesmo *chip* é menor do que a necessária para enviar sinais externos, assim, os núcleos podem operar em voltagens menores. Esses fatores levam à diminuição no consumo de energia e também à redução na latência dos processadores, características

importantes em qualquer sistema computacional, mesmo os não paralelos (HSIN MA, 2005; CAVALHEIRO; SANTOS, 2007).

Embora alguns pesquisadores como Asanovic et al. (2006) prevejam o fracasso das arquiteturas *multi-core* em fornecer solução ideal para os problemas envolvendo paralelismo, esse tipo de arquitetura SMP está atualmente presente de forma maciça no mercado e a tendência é que permaneçam assim por vários anos. Como conseqüência dessa popularização dos processadores *multi-core*, os custos financeiros de computadores dotados com os mesmos têm caído drasticamente, criando uma tendência de que esses venham a se tornar *commodity*.

3.4 Modelos de programação para processamento paralelo

A compreensão do inter-relacionamento entre as características de um algoritmo paralelo e as da arquitetura em que ele irá executar é fundamental para a qualidade no desenvolvimento para processamento paralelo. O processamento com desempenho otimizado em arquiteturas paralelas exige que haja equilíbrio entre as características do *hardware* com as do *software*.

As arquiteturas para processamento paralelo estão em constante evolução, portanto, nenhuma arquitetura pode ter a expectativa de se manter como solução ideal por mais do que alguns anos. Assim, buscando contornar o problema da falta de portabilidade e curto tempo de vida útil do *software* paralelo, surgiu a idéia de se utilizar modelos suficientemente abstratos que, ao mesmo tempo em que ocultam os aspectos da arquitetura à medida que eles se alteram, mantém o desempenho da aplicação desenvolvida (YAMIN, 1999).

Um modelo desse tipo pode ser formado construído da seguinte forma: adota-se uma máquina abstrata para a qual o *software* é desenvolvido e a máquina, por sua vez, é emulada pelas diferentes arquiteturas paralelas, conforme ilustrado na Fig. 8. Dessa forma, os aspectos do projeto de *software* podem ser desassociados daqueles de sua implementação (YAMIN, 1999). Skillicorn e Talia (1998) colocam que um modelo ideal tem como necessidades mínimas: facilidade e metodologia para o desenvolvimento de *software*, independência de arquitetura, facilidade de compreensão, garantia de desempenho e medidas de custo.

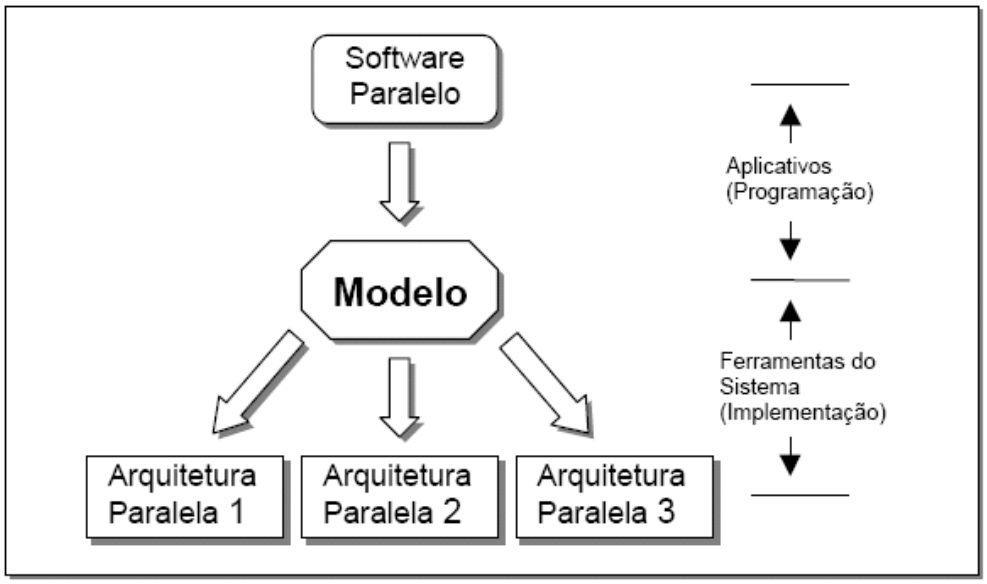


Figura 8 – Proposta de modelo para desenvolvimento de *software* paralelo.
Fonte: YAMIN, A. 1999.

O controle da execução paralela é uma tarefa complexa, portanto, na medida do possível, o modelo deve retirar do programador e passar para o compilador e ambiente de execução essa responsabilidade. Dessa forma é atendido o primeiro requisito do modelo proposto. Já a necessidade de metodologia para desenvolvimento de *software* justifica-se pelo fato de ser inviável utilizar no desenvolvimento paralelo a tradicional metodologia de testes e depuração. Isso ocorre devido a vários fatores, tais como: a equipe de desenvolvimento teria acesso a apenas algumas das arquiteturas paralelas nas quais o *software* poderia vir a ser executado, o não-determinismo no escalonamento dos processadores, entre outros (YAMIN, 1999).

Independência de arquitetura é um requisito naturalmente desejável, e, dado que as arquiteturas paralelas costumam aglutinar diversas tecnologias, todas em constante evolução, é pouco razoável considerar reescrever o *software* cada vez que houver necessidade de troca do *hardware*. A facilidade de ser entendido também aparece como requisito natural em um modelo. Visto que o processamento paralelo tem ampliado sua fatia de mercado, o modelo de sua programação deve poder ser assimilado com facilidade pelos desenvolvedores, oferecendo-lhes uma interface que oculte tanto quanto for possível a complexidade inerente ao paralelismo.

Em relação à garantia de desempenho, Skillicorn e Talia (1998) colocam que somente um modelo com otimização nas comunicações pode oferecer previsibilidade no desempenho. Porém, uma vez que heurísticas complexas para otimização do mapeamento das tarefas paralelas podem introduzir custo computacional elevado, apenas modelos que limitem a freqüência das comunicações poderiam esperar ter garantias mínimas de desempenho em diferentes arquiteturas (YAMIN, 1999).

Uma das premissas da programação paralela é buscar o maior desempenho possível. Nesse contexto, Yamin (1999) coloca que além do desempenho global, a taxa de utilização individual dos processadores e o custo de desenvolvimento são também importantes indicadores na avaliação do *software* paralelo. Na programação paralela, mínimas modificações no programa ou a troca do *hardware* podem mudar completamente o comportamento do programa. Skillicorn e Talia (1998) definem que um modelo oferece medidas de custo se, dado um programa, é possível determinar seu custo a partir de seu código-fonte, de propriedades mínimas da arquitetura paralela para a qual ele destina-se (número de processadores, por exemplo) e das informações a respeito do tamanho de suas entradas.

Skillicorn e Talia (1998) classificam os modelos para desenvolvimento de *software* paralelo utilizando como critério o nível de abstração com que o paralelismo pode ser explorado. As categorias apresentadas são as seguintes:

- Modelos nos quais o paralelismo é explorado de forma totalmente implícita;
- Modelos com assinalamento do paralelismo explícito;
- Modelos com assinalamento e decomposição do paralelismo explícitos;
- Modelos com assinalamento, decomposição e mapeamento do paralelismo explícitos;
- Modelos com assinalamento, decomposição, mapeamento e comunicação explícitos;
- Modelos em que o paralelismo é explorado de forma totalmente explícita;

Essa classificação abrange desde os modelos mais abstratos, que atendem os critérios para desenvolvimento de *software* mas não oferecem medidas de

desempenho, até os modelos de baixo nível, os quais permitem uma previsão de desempenho mas dificultam a construção do *software* (YAMIN, 1999). Diferentes ferramentas de programação implementam diferentes modelos, entretanto, os esforços da comunidade científica nos últimos anos têm sido direcionados para os modelos de mais alto nível que buscam facilitar a programação.

3.5 Conclusões

O paralelismo tem conquistado espaço na Ciência da Computação como solução para os mais diversos problemas. Porém, as arquiteturas para processamento paralelo estão em constante evolução e, portanto, nenhuma arquitetura pode ter a expectativa de se manter como solução ideal por mais do que alguns anos. Assim, a escolha da arquitetura paralela a ser utilizada no desenvolvimento de uma solução computacional deve priorizar atender os requisitos particulares da aplicação em questão. Para a realização do presente trabalho, as arquiteturas SMP baseadas em processadores *multi-core*, subdivisão da classe MIMD, foram identificadas como as mais indicadas.

Um fator importante para a não escolha de arquiteturas MIMD com memória distribuída foi o ambiente de execução ao qual a ferramenta proposta se destina. A aplicação desenvolvida trata-se de um sistema para visualização de imagens médicas que objetiva auxiliar o diagnóstico e a avaliação pré-cirúrgica de pacientes com epilepsia. Logo, é natural que seu ambiente de execução seja composto por computadores pessoais dos especialistas e computadores disponibilizados pelos hospitais e clínicas. Como as arquiteturas de memória distribuída são sistemas de grande porte, não costumam estar presentes nos locais citados, bem como, não são um sistema computacional que um profissional irá adquirir para uso próprio.

As arquiteturas *multi-core*, por outro lado, têm se popularizado bastante nos últimos anos, tendendo a se tornarem *commodity*. Com isso, além de estarem presentes na grande maioria dos novos computadores, elas apresentam custo bastante acessível. A preocupação com o custo neste trabalho justifica-se, novamente, pelo contexto da aplicação. Sendo as arquiteturas acessíveis, tanto médicos como hospitais podem adquirir máquinas capazes de executar satisfatoriamente a ferramenta desenvolvida.

4 PROGRAMAÇÃO PARALELA EM ARQUITETURAS SMP

Desenvolver um programa para arquiteturas SMP requer a decomposição do problema em partes que possam ser executadas independentemente e a distribuição das mesmas dentre os processadores ou *cores*. Na prática, isso pode ser feito construindo um programa composto de múltiplos *threads*, os quais sofrem um processo de escalonamento para ocupação dos CPUs disponíveis.

Inicialmente, este capítulo descreve as características relacionadas ao conceito de *thread*. Na seqüência, é abordado o tema ferramentas para programação *multithread*. Por fim, descreve-se a maneira como o VTK opera em conjunto com bibliotecas *multithread*.

4.1 Processos leves - *Threads*

Processo é um termo muitas vezes confundido com programa, porém é importante diferenciá-los: programa é o código objeto, processo é a execução do mesmo. Um mesmo programa executado por dois usuários ou carregado duas vezes, gera dois processos diferentes. Formalmente, processo é um fluxo de controle seqüencial (parte ativa do processo) e seu espaço de endereçamento (parte passiva) (TANENBAUM, 2001).

O escalonamento de processos em máquinas SMPs, segundo Tanenbaum (2001), pode ser de três formas: *timesharing*, *space sharing* e *gang scheduling*. A estratégia mais comum, *Timesharing*, é utilizada nas versões para multiprocessadores dos SOs *Windows* e *Linux*. Nesse método, cada processo executa em um dos processadores por um intervalo de tempo ou até executar uma operação bloqueante, quanto então, o escalonador seleciona o processo que ocupará o seu lugar.

Thread, por sua vez, é o termo que designa somente o fluxo de controle de um processo, ou seja, uma linha de execução do mesmo. Um processo pode conter vários fluxos de execução, e, portanto, vários *threads*. Nesse caso, os *threads* podem ser vistos como as subdivisões do processo que executam simultaneamente.

Por isso, *threads* são também conhecidos como processos leves (TANEMBAUM, 2001). Em algumas implementações dos serviços de *threads* cabe ao próprio sistema operacional (SO) oferecer suporte aos *threads*, que podem ocorrer em nível de *Kernel* (*Kernel Level Thread* - KLT). Em outros, o suporte é realizado em nível de usuário (*User Level Thread* - ULT).

O escalonamento dos *threads* ocorre de maneira semelhante ao escalonamento dos processos. Os sistemas operacionais oferecem suporte aos *threads* de duas maneiras: através de preempção *multithreading* ou *multithreading* cooperativa. Na primeira maneira o sistema determina quando executar trocas de contexto, enquanto na segunda fica a cargo dos *threads* fazer esse controle.

Atualmente, programas compostos com um único processo subdividido em múltiplos *threads* são mais comuns do que programas compostos por vários processos. Isso ocorre, basicamente, porque o tempo de criação, bem como de sincronização e escalonamento, é menor do que o tempo para realizar as mesmas operações com um processo. Além disso, *threads* podem ser criados sem a necessidade de replicação de todo o código do processo e, como os *threads* compartilham espaço de endereçamento, a comunicação entre eles é mais rápida (TORELLI, 2005).

Grama et al. (2003) destaca algumas vantagens da adoção do desenvolvimento com APIs (*Application Programming Interface,* Interface de Programação de Aplicativos) *multithreaded* em arquiteturas *multi-core*:

- Portabilidade de *software*: os programas podem ser desenvolvidos e executar em máquinas paralelas sem qualquer mudança no código;
- Diminuição na latência do processador: um dos maiores *overheads* em programas, sejam seriais ou paralelos, é causado pela latência no acesso à memória, operações de entrada e saída (E/S) e comunicações. Ao executar vários *threads* no mesmo processador, essa latência é diminuída, pois enquanto um *thread* aguarda a execução de uma operação de E/S ou comunicação, outro pode utilizar o processador;
- Escalonamento e balanceamento de carga: como os programas paralelos escrevem em endereços de memória compartilhados, o programador pode expressar a concorrência de maneira que diminua o *overhead* com interações remotas. Em relação ao balanceamento

de carga, tem-se que enquanto em muitas aplicações estruturadas a tarefa de alocar trabalho igual para os vários processadores é facilmente cumprida, em aplicações não estruturadas e dinâmicas (como jogos ou otimização discreta) esta tarefa é mais difícil. APIs para programação *multithread* permitem especificar um grande número de fluxos concorrentes que serão escalonados dinamicamente para os diversos processadores, livrando assim o programador das tarefas de escalonamento e balanceamento de carga;

- Facilidade de programação e uso disseminado: programas *multithread* são significativamente mais fáceis de serem escritos do que seus equivalentes em outras abordagens de programação paralela. Assim, é menor a necessidade de esforços no treinamento de programadores, o que a torna uma abordagem bastante difundida. Tais características são importantes para que o desenvolvimento da engenharia de *software* possa atender adequadamente a programação paralela.

Considerando os conceitos acima e a decisão por se trabalhar com processadores *multi-core,* optou-se neste projeto por escrever a aplicação proposta através de uma ferramenta de programação *multithread*. Existem diversas ferramentas que suportam este tipo de programação, a próxima seção aborda algumas delas.

4.2 Ferramentas de programação *multithreading*

Na multi-programação leve (*multithread*), os recursos de programação são voltados, basicamente, às operações de controle dos fluxos de execução, como a criação, destruição e suspensão temporária desses. Os programas *multithread* são iniciados da mesma forma que os programas convencionais. A criação, término e sincronização dos diversos fluxos é realizada de forma explícita através da invocação dos respectivos serviços pela biblioteca de *threads* ou pelo SO. Isso ocorre mesmo no caso das ferramentas que oferecem interface de alto nível, como o *OpenMP (Open Multi-Processing*). A diferença é que nesses casos o compilador é

quem traduz a linguagem de alto nível em chamadas de controle (CAVALHEIRO; SANTOS, 2007).

As ferramentas para programação *multithread* costumam diferir em relação ao nível de abstração que oferecem para criação, destruição, comunicação e sincronização dos *threads*. De um lado encontram-se as ferramentas como *Pthreads* (POSIX *Threads*), onde o programador deve controlar explicitamente as operações acima citadas. Na outra ponta, estão as ferramentas baseadas em diretivas de compilação, como o *OpenMP*, onde o usuário através de “comentários especiais” (diretivas) delimita os trechos do código que deverão ser paralelizados. Feito isso, o compilador gera automaticamente o código paralelo (TORELLI, 2005).

Programas paralelos, seja através de múltiplos *threads* ou processos, precisam ter cuidado especial no controle ao acesso das chamadas “seções críticas”. Seção crítica é o nome dado a um trecho de código que acessa um ou mais dados na memória compartilhada. Sendo um dado compartilhado, é possível que outro *thread* ou processo executando simultaneamente requisite-o ou tente modificá-lo, gerando então uma inconsistência. Assim, a seção é dita crítica porque é necessário garantir a integridade do dado acessado. Para isso, devem ser utilizados mecanismos que assegurem exclusividade na manipulação do dado a um único *thread* em cada momento. Normalmente, o mecanismo utilizado é o *mutex*, acrônimo para a expressão em inglês *mutual exclusion* (CAVALHEIRO; SANTOS, 2007).

Além da preocupação com a decomposição do problema em fluxos independentes, ao desenvolver um programa paralelo é necessário ter a preocupação de evitar erros que possam levar a falhas como condições de corrida (*data-races*) e situações de impasse (*deadlock*). Condições de corrida são situações que ocorrem devido a implementações inadequadas dos mecanismos para controle do acesso as seções críticas, o que pode ocasionar variação no conteúdo dos dados de acordo com a ordem em que os fluxos forem executados. Como não é possível garantir a ordem com que os *threads* são escalonados, o resultado final da computação pode ser diferente a cada execução. Rotinas que desempenham corretamente a função esperada mesmo quando executadas em ambiente *multithreaded* são chamadas *thread-safe* (OpenMP API, 2005) Situações de impasse, por sua vez, ocorrem quando um *thread* ou processo se mantém bloqueado na espera de um sinal que nunca é enviado. Desta forma, o *thread*

permanece suspenso indefinidamente, impedindo o uso de recursos pelos demais ou comprometendo o término da execução do programa (TANENBAUM, 2001) (CAVALHEIRO; SANTOS, 2007). Assim, na escolha da ferramenta *multithread* que será utilizada, dentre os requisitos analisados, deve ser dado especial cuidado aos mecanismos de controle de seção crítica oferecidos e a maneira como as comunicações são manipuladas.

A ferramenta adotada para realização deste trabalho, o VTK, trabalha com *Pthreads*. A seguir, esse padrão é sumariamente descrito, e, na seqüência, é colocado o funcionamento do VTK enquanto ferramenta *multithread*.

4.2.1 Posix *threads*

O padrão POSIX (*Portable Operating System Interface for uniX*, Interface Portável para Sistema Operacional *Unix*) foi proposto pela IEEE (*Institute of Electrical and Electronics Engineers*) e define um conjunto de interfaces de programação para o sistema *Unix*, sendo o padrão para *threads* (IEEE POSIX 1003.1c de 1995) normalmente referenciado como *Pthreads*. Esse padrão caracteriza a interface de serviços para uso conjunto com programas C/C++, embora existam esforços de padronização para a linguagem Fortran.

O padrão *Pthreads* tem grande vantagem em dois sentidos: como o padrão POSIX está presente em grande parte dos sistemas operacionais atualmente em uso, seu potencial de portabilidade é bastante alto. Já em relação à programação, esse padrão caracteriza-se por oferecer grande liberdade ao programador, o que resulta na possibilidade de empregar diversas estruturas de programação concorrente (CAVALHEIRO; SANTOS, 2007).

A API de *Pthreads* é formada por aproximadamente 60 funções que se destinam a criação, sincronização, controle do escalonamento, suspensão e término dos *threads*. O programa inicia executando um *thread* principal, outros *threads* podem ser então criados explicitamente pelo programador através da função “pthread_create”. A terminação de um *thread* ocorre quando todo o corpo de sua função tiver sido executado ou for realizada uma chamada explícita ao comando “return” ou ainda à função “pthread_exit” da biblioteca (LEWIS; BERG, 1996).

A função que cria um *pthread* recebe quatro argumentos:

- pthread_t *thid: identifica o *thread* para que esse possa ser referenciado futuramente;
- pthread_attr_t *atrib: especifica um conjunto de atributos para a execução do *thread*, como por exemplo, prioridade. Caso não se deseje utilizar nenhum atributo em especial, pode ser passado o valor "*NULL*" como argumento;
- void *(*função) (void *): a função que o novo fluxo executará. Essa função deve ser do tipo "(void*)" e receber um único parâmetro também do tipo "(void*)", que, por sua vez, pode ser um endereço de memória contendo um conjunto de argumentos;
- void *args: o endereço de memória do argumento que deve ser passado para a função que o *thread* irá executar, ou seja, o parâmetro único da função citado no item anterior;

É importante observar que uma vez criado o *thread*, não está definido o momento em que a execução desse iniciará ou, tão pouco, como se dará a sobreposição do tempo de uso do processador pelo novo *thread* com os demais já em execução. Entretanto, o padrão *Pthreads* oferece o a função "pthread_join" para sincronização entre o *thread* que solicitou o serviço e outro que é identificado como parâmetro da função "pthread_join".

O padrão *Pthreads* oferece ainda funções para sincronização e exclusão mútua que implementam os sistemas de *mutex*, variáveis de condição, barreira, semáforo e objetos de eventos. Esse último pode ser visto na realidade como variáveis de condição com outro nome, pois são utilizados para a mesma finalidade, embora suas APIs sejam diferentes (LEWIS; BERG, 1996).

4.2.2 *Threads* no VTK

A implementação da biblioteca VTK oferece programação e execução *multithread* de maneira nativa. Entretanto, para isso, ao compilar o VTK deve-se assinalar de maneira explícita que o mesmo deve ser codificado dessa forma. Isso é feito através da configuração das diretivas de compilação no *CMake*. Porém, por padrão, o *CMake* busca detectar o sistema operacional em uso e a existência de

bibliotecas *multithread*, ou seja, dependendo da configuração do ambiente do programador o suporte a *multithread* será ativado de maneira transparente ao desenvolvedor.

O VTK oferece suporte aos seguintes padrões de *threads*: *Win32Threads* (ambientes *Windows*), *Sprocs* (SGI), *Pthreads* e *HP_Pthreads* (sistemas com suporte ao padrão POSIX). Por isso, a detecção automática realizada pelo *CMake* pode não ser a melhor opção no caso de usuários avançados que desejam utilizar um padrão em específico. O Capítulo 5 aborda de maneira mais detalhada o processo de configuração dessas diretivas.

A programação *multithread* no VTK pode ser explorada de maneira implícita ou explícita, onde o usuário pode controlar o número de *threads* com que uma determinada classe executa, além de criar funções para execução em paralelo. Cabe lembrar que o fato de o VTK trabalhar com *pipelines* de visualização facilita bastante esse trabalho, pois, implica naturalmente em uma decomposição do problema.

Para determinar explicitamente o número de *threads* com que uma determinada classe deverá executar, basta que o programador faça uma chamada ao método “SetNumberOfThreads()” passando o número de *threads* desejado. Em complemento, é oferecido um método “GetNumerOfThreads()” que recupera o número de *threads* em uso para uma dada classe.

Já a definição explícita de funções para execução em paralelo segue uma metodologia mais complexa, onde o paralelismo e a sincronização devem ser também assinalados de maneira explícita. Para isso, deve-se utilizar a classe “vtkMultiThreader”. Essa classe oferece uma abstração de modelo para programação *multithread*, e, sua função de fato, é redefinir as funções oferecidas pelas APIs suportadas no VTK de forma que os algoritmos utilizados por ele sejam implementados para execução concorrente sem preocupação com o tipo de biblioteca a que o código será ligado.

A classe “vtkMultiThreader” está relacionada a praticamente todas as demais do pacote VTK, que, por sua vez, a utilizam como controladora da execução concorrente. De maneira análoga à programação com *Pthreads*, para que uma função possa iniciar um fluxo de execução ela deve ser do tipo ponteiro para *void* e possuir um único argumento também do tipo ponteiro para *void*. Os principais métodos dessa classe são:

- void SingleMethodExecute (): inicia a execução de uma lista de funções de maneira seqüencial;
- void MultipleMethodExecute (): inicia a execução de uma lista de funções em paralelo;
- void SetSingleMethod (vtkThreadFunctionType, void *data): adiciona uma função a fila para execução seqüencial, onde "*data*" representa o argumento da função;
- void SetMultipleMethod (int index, vtkThreadFunctionType, void *data): cria uma lista de funções para serem executadas em paralelo, onde "*data*" representa o argumento de cada função;
- int SpawnThread (vtkThreadFunctionType, void *data): cria um novo *thread* para a função passada como parâmetro, onde "*data*" representa seu argumento;
- void TerminateThread (int thread_id): termina um fluxo de execução;
- virtual void SetNumberOfThreads (int): configura o número de *threads* para execução simultânea;
- virtual int GetNumberOfThreads (): recupera o número de *threads* para execução simultânea;
- static vtkMultiThreader * New (): cria uma nova instância da classe;
- static vtkMultiThreaderIDType GetCurrentThreadID (): recupera o identificador de um *thread*;
- static void SetGlobalMaximumNumberOfThreads (int val): configura o número máximo de *threads* para execução em simultâneo globalmente;
- static int GetGlobalMaximumNumberOfThreads (): recupera o número máximo de *threads* para execução em simultâneo globalmente;
- static void SetGlobalDefaultNumberOfThreads (int val): modifica o número padrão de *threads* para execução em simultâneo globalmente;
- static int GetGlobalDefaultNumberOfThreads (): recupera o número padrão de *threads* para execução em simultâneo globalmente;

- static int ThreadsEqual (vtkMultiThreaderIDType t1, vtkMultiThreaderIDType t2): verifica se dois identificadores de *threads* apontam para um mesmo fluxo;

A Fig. 9 ilustra uma abstração em alto-nível de um trecho de código para a execução de três *threads* em paralelo utilizando a classe “vtkMultiThreader”. Primeiramente, são colocados os protótipos das funções que executaram nos novos fluxos, a seguir são definidos os parâmetros para uma instância da classe “vtkMultiThreader”, a qual inicia então a execução.

```
#include <vtkMultiThreader.h>

...

void* ThreadRM(void* imagem);
void* ThreadSPECT(void* imagem);
void* ThreadSISCOM(void* imagem);

...

vtkMultiThreader* controller = vtkMultiThreader::New();
  controller->SetNumberOfThreads(3);
  controller->SetMultipleMethod(0, ThreadRM, cast);
  controller->SetMultipleMethod(1, ThreadSPECT, cast);
  controller->SetMultipleMethod(2, ThreadSISCOM, cast);
  controller->MultipleMethodExecute();
```

Figura 9 – Trecho de código para execução de três *threads* em paralelo utilizando a classe vtkMultiThreader.

O VTK oferece ainda outras classes para exploração do paralelismo: em nível de *threads*, por exemplo, as classes “vtkSimpleMutexLock” ou "vtkMutexLock" são utilizadas para o controle do acesso às seções críticas. Já para a programação voltada a sistemas de memória distribuída, existe a classe “vtkMultiProcessController” que tem função análoga a “vtkMultiThreader”, porém com outro foco.

4.3 Conclusões

Em relação aos modelos para desenvolvimento de *software* classificados por Skillicorn e Talia (1998), o utilizado pelo VTK não se enquadra totalmente em nenhuma das classificações. Esse pode ser visto como um intermediário entre os "Modelos nos quais o paralelismo é explorado de forma totalmente implícita" e os "Modelos com assinalamento, decomposição, mapeamento e comunicação explícitos". Como o VTK trabalha baseado em *pipelines* de visualização, a correta programação implica naturalmente em decomposição do problema.

Entretanto, as implicações da decomposição são tratadas de forma transparente ao programador: uma vez dividido o problema, a atribuição das partes aos processadores e a forma como essas vão se comunicar e sincronizar não precisam ser explicitadas no programa. Além disso, os algoritmos da biblioteca VTK referenciados pelo desenvolvedor são codificados de maneira a permitir execução *multithread* mesmo por aqueles que desconhecem a multi-programação.

Porém, caso o programador julgue necessário, é permitido que esse manipule o paralelismo de maneira explícita controlando a criação dos *threads* e o momento em que os mesmos serão executados. O VTK possui ainda métodos para alterar o número de *threads* utilizados em seus algoritmos básicos e também classes para programação voltada a sistemas distribuídos em que a comunicação, sincronização e mapeamento também precisam ser controlados. Por isso, embora o VTK permita o paralelismo totalmente implícito, não pode ser dito um modelo onde o paralelismo é explorado de maneira totalmente implícita.

Além disso, o suporte a variadas bibliotecas *multithread* facilita fortemente sua utilização em SOs diversos sem a necessidade de sacrificar a execução concorrente. Esse fator é importante, pois, conforme colocado no capítulo anterior, cada vez mais se tem buscado atingir portabilidade no desenvolvimento de *software*, inclusive do *software* paralelo.

Em complemento a isso, tem-se que o padrão POSIX, suportado no VTK, atualmente está presente em grande número de sistemas operacionais. Dessa forma, uma abordagem que combina VTK e POSIX *Threads* além de oferecer grande portabilidade, aumenta a flexibilidade e autonomia no desenvolvimento

paralelo, pois permite tirar proveito das facilidades oferecidas no VTK ao mesmo tempo em que se tem o controle maior inerente a programação *Pthreads*. Assim, essa é uma combinação que atende todos os níveis de desenvolvedores.

Por fim, observa-se que o VTK é a ferramenta ideal para o desenvolvimento deste projeto. Isso porque, além das vantagens relacionadas à programação paralela, esse oferece suporte aos algoritmos de computação gráfica e processamento de imagens necessários. Um exemplo disso é o *Ray-casting*, que encontra-se nativamente implementado no VTK para execução *multithread*. Assim, optou-se por utilizar neste projeto o VTK compilado para execução paralela baseada em *Pthreads*.

5 SOLUÇÃO DESENVOLVIDA – MATERIAIS E MÉTODOS

O desenvolvimento da ferramenta proposta neste projeto tem como objetivo oferecer uma aplicação de visualização interativa complementar ao método de fusão de imagens em três dimensões proposto por Oliveira (2005) e ao SISCOM. Nos capítulos anteriores foram apresentados conceitos que serviram como base para este trabalho. A utilização de tais conceitos no desenvolvimento deste projeto é colocada no presente capítulo, bem como as ferramentas utilizadas para o desenvolvimento da solução proposta.

5.1 Ferramentas utilizadas

Por requerer grande esforço de implementação, diversas ferramentas de apoio foram utilizadas neste projeto. Essas ferramentas encontram-se descritas e caracterizadas na seqüência.

5.1.1 Ambiente de desenvolvimento

O trabalho foi desenvolvido em ambiente de programação *Microsoft Visual Studio* 2005 *Express* na linguagem orientada a objetos C++ executando sobre o sistema operacional *Microsoft Windows* XP. A opção pela linguagem C++ foi motivada por diversos fatores, dentre os quais se destaca o fato de que esta linguagem é largamente utilizada no desenvolvimento de aplicações científicas. Soma-se ainda que C++ apresenta grande flexibilidade na descrição de modelos de programas, mantendo-se robusta e eficiente. Outra característica relevante dessa linguagem é permitir fácil integração com as demais ferramentas utilizadas para o desenvolvimento deste projeto (VTK, FLTK e *Pthreads*-win32). Ressalta-se que, embora essas estejam disponíveis para outras linguagens, as implementações nativas dessas ferramentas destinam-se e são baseadas em linguagem C++.

O ambiente de programação MS *Visual Studio* 2005 *Express* foi escolhido por oferecer facilidades de codificação e testes, além de ser disponibilizado

gratuitamente. Em conjunto com a API foi necessário utilizar a plataforma *Microsoft* SDK (*Software Development Kit, Kit* para Desenvolvimento de *Software*). Essa plataforma também possui versões paga e gratuita, sendo que a versão utilizada foi a gratuita, disponibilizada no site do fabricante.

Em complemento ao ambiente e plataforma acima citados, foi utilizado o *software CMake* como auxiliar no processo de compilação. O *CMake* é um sistema para geração da árvore de *build* para diversas plataformas e plataformas cruzadas. O projetista precisa apenas especificar, sem se preocupar com a plataforma, as bibliotecas externas e arquivos de código-fonte que deverão ser ligados na compilação de seu projeto (CMAKE DOCUMENTATION, 2007).

5.1.2 Imagens de RM e SPECT

No Capítulo 2 foi discutida a importância das imagens funcionais e das imagens anatômicas na detecção e localização do foco epileptogênico, bem como a obtenção das mesmas através de exames de SPECT e RM, respectivamente. Entretanto, a razão principal para utilização desses formatos de exames no presente trabalho, é a premissa de desenvolver uma ferramenta complementar ao método proposto por Oliveira (2005) e ao SISCOM, ambos baseados na combinação de SPECT e RM.

Cabe lembrar que a utilização de imagens de diferentes modalidades, como é o caso de RM e SPECT, requer a realização de alinhamento geométrico (ou co-registro) das mesmas. Entretanto, essa tarefa não foi desenvolvida por este projeto, dado que este utiliza o método proposto por Oliveira (2005). Uma abstração em alto nível do processamento efetuado nas imagens é descrito na Fig. 10.

Os exames de RM e SPECT utilizados neste trabalho encontram-se no formato *Analyze* 7.5, um dos mais populares para armazenamento de imagens médicas. Nesse formato, uma imagem é armazenada com auxílio de dois arquivos, um de cabeçalho contendo uma descrição da imagem e outro com o conteúdo da imagem propriamente dita. Os arquivos possuem o mesmo nome e são distinguidos por suas extensões, “hdr” e “img“ respectivamente.

Passo 1.	Alinhamento do SPECT crítico com a RM;
Passo 2.	Alinhamento do SPECT crítico com o SPECT intercrítico;
Passo 3.	Eliminação de artefatos de espalhamento e de sinais produzidos por regiões não interessantes (como exemplo, artefatos de atividade radioativa residual no couro cabeludo) produzidos pelos SPECTs crítico e intercrítico;
Passo 4.	Normalização do nível de captação do radiofármaco;
Passo 5.	Subtração SPECT crítico – intercrítico;
Passo 6.	Eliminação de ruídos;
Passo 7.	Posicionamento com a RM.

Figura 10 – Algoritmo para subtração das imagens de SPECT crítico e intercrítico e alinhamento com a RM

O arquivo da imagem contém os valores dos *pixels* sem nenhuma compressão. O formato do cabeçalho é flexível e pode ser estendido para um novo formato definido pelo usuário. Usualmente, é representado como uma estrutura de dados em C que descreve o próprio cabeçalho, as dimensões da imagem e, opcionalmente, o histórico dos dados dos *pixels* (CLINIC, 2007).

O suporte a imagens *Analyze* não é oferecido de maneira nativa por nenhuma das ferramentas utilizadas neste projeto. Foi então utilizada uma classe desenvolvida especificamente para a leitura desse formato, “vtkAnalyzeReader” (GOMES et al., 2007). Essa classe, implementada com base em uma de função análoga, “itkAnalyzeImageIO” disponibilizada pela biblioteca ITK (*Insight Segmentation and Registration ToolKit*), é subclasse de “vtkImageReader2” do VTK.

5.1.3 Bibliotecas para visualização e programação *multithread*

Para implementação dos algoritmos de visualização utilizou-se o VTK (VTK, 2007). O VTK é uma biblioteca gratuita, de código aberto e *multithread*, para computação gráfica 3D e processamento de imagens. Essa biblioteca, além de oferecer um conjunto de classes C++ para utilização em programas escritos nessa linguagem, oferece também *wrappers* (interpretadores) que permitem utilizar os

recursos da mesma nas linguagens *Java, Python* e TCL. O VTK pode, portanto, ser utilizado em qualquer uma das 4 linguagens citadas.

Dentre as vantagens oferecidas pelo VTK que culminaram em sua escolha como ferramenta para implementação da aplicação proposta, destacam-se:

- Portabilidade: além de poder ser utilizado em diversas linguagens de programação, o VTK é uma biblioteca validada em todas as versões do sistema operacional *Unix*, versões 98/ME/NT/2000/XP do MS *Windows* e *Mac* OS X *Jaguar* ou posteriores;
- Implementação baseada em um *pipeline* de visualização: o fluxo de operações no VTK, bem como nos *softwares* desenvolvidos a partir dele, dá-se através de um *pipeline* de visualização. Dessa maneira, o VTK facilita e estimula a decomposição do problema em partes, que é um requisito no desenvolvimento em computação paralela;
- Implementação *multithread*: o VTK oferece suporte a *threads* de maneira implícita e explícita. O controle dos fluxos de execução é feito pela própria ferramenta, porém, são oferecidos métodos e classes específicos que permitem ao usuário fazê-lo. Os seguintes padrões são suportados: *Win32Threads* (ambientes *Windows*), *Sprocs* (SGI), *Pthreads* e HP_*Pthreads* (sistemas com suporte ao padrão POSIX). A opção por utilizar um desses padrões deve ser assinalada explicitamente no momento da compilação do VTK;
- Poderoso suporte à reconstrução volumétrica: o VTK oferece suporte completo à visualização de volumes, desde a leitura até a reconstrução e visualização 3D. O algoritmo *Ray-casting*, cuja escolha foi justificada no Capítulo 2, é implementado com alto nível de detalhamento e flexibilidade;
- *Software* livre: o VTK é distribuído como *software* livre. Permite adequação às necessidades particulares do desenvolvedor e possui uma enorme comunidade trabalhando em seu aperfeiçoamento, além de uma lista de e-mails com mais de 1900 usuários registrados, os quais contribuem mutuamente entre si.

Conforme citado no Capítulo 4, este projeto optou por utilizar *Posix Threads* (*Pthreads*) como abordagem para programação *multithread*. Porém, o MS *Windows*

não oferece nativamente suporte para compilação de programas com *Pthreads*. Dessa forma, para a compilação do VTK baseada em *threads* POSIX e do *software* desenvolvido, foi necessário obter uma implementação de *Pthreads* de acordo com o padrão POSIX 1003.1-2001, permitindo assim o uso do VTK como ferramenta *multithread* em ambiente MS *Windows.*

A implementação adotada, *Open Source* POSIX *Threads for Win32* (POSIX THREADS FOR WIN32, 2007), é disponibilizada na forma de *software* livre, sob licença *GNU Lesser General Public License* (LGPL) e está atualmente na versão 2.8.0 (liberada em 22 de dezembro de 2006). A distribuição é composta por diversos arquivos, sendo que cada um deles é uma versão alternativa da biblioteca. A implementação é feita dessa forma para oferecer suporte a diversos compiladores e esquemas de controle de exceções variados, o que não seria possível se utilizada uma compilação única. Neste projeto foi utilizada a biblioteca “pthreadVC2”.

5.1.4 Biblioteca para construção de interface

Embora o VTK seja um poderosíssimo instrumento para visualização científica em geral, seu potencial de interatividade com o usuário pode ser melhorado com a utilização da biblioteca FLTK (*Fast Light ToolKit*). O FLTK é uma biblioteca para desenvolvimento de interface independente de plataforma, suportada em *Unix*/*Linux*, MS *Windows* e *Mac* OS X. É distribuído como *software* livre sobre os termos da GNU *Library Public License* e também possui facilidades de execução *multithread* (FAST LIGHT TOOLKIT, 2007).

A interação entre o VTK e FLTK apresenta alguns problemas de sincronização. Por isso, a integração entre as bibliotecas foi realizada com auxílio de uma classe externa, “vtkFlRenderWindowInteractor”. Essa classe é implementada como subclasse do VTK, independente de plataforma e distribuída na forma de *software* livre (VTKFLRENDERWINDOWINTERACTOR, 2007).

5.2 Metodologia e ferramenta desenvolvida

O desejo de desenvolver uma ferramenta portável para diversas arquiteturas implicou na necessidade de sistematizar os processos de desenvolvimento adotados. Nesta seção são documentadas as estratégias aplicadas.

5.2.1 Compilação das ferramentas *multithread*

A tarefa inicial para implementação da solução foi compilar o VTK em plataforma *Windows* com suporte à execução *multithread*. O primeiro passo do processo de compilação é configurar o *CMake* de acordo com as opções desejadas. Para isso, um arquivo denominado "CMakeLists.txt" deve ser criado no mesmo diretório do projeto. Nesse arquivo define-se então parâmetros prévios para o *CMake*, como por exemplo, as bibliotecas que serão ligadas ao projeto.

No caso do VTK e FLTK, os respectivos arquivos "CMakeLists.txt" já acompanham o código-fonte, portanto, não precisam ser criados pelo usuário. Ao iniciar o *CMake* indica-se qual compilador será utilizado no projeto, na seqüência é exibida uma janela com as diretivas de compilação (Fig. 11). Neste momento devem ser definidos itens como: se compilação será voltada para abordagem *multithread*, se deseja-se gerar bibliotecas para ligação estática ou dinâmica, entre outros.

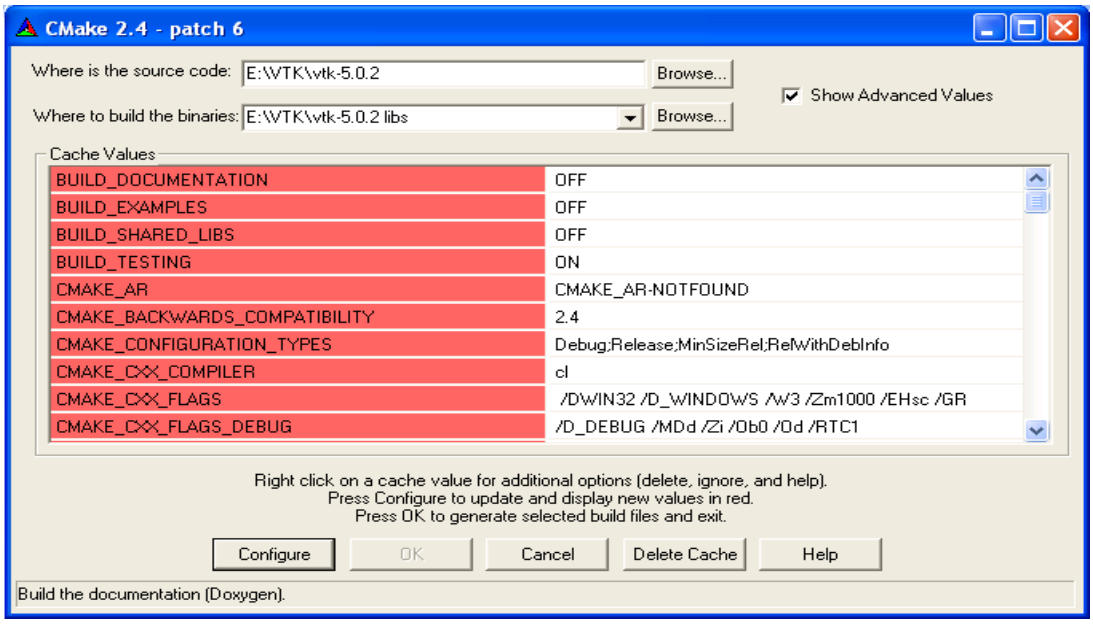


Figura 11 – Configuração das diretivas para compilação do VTK no *CMake*.

Os parâmetros relativos ao uso de *threads*, por ordem de exibição, são os seguintes:

- CMAKE_HP_THREADS;
- CMAKE_THREAD_LIBS;
- CMAKE_USE_PTHREADS;
- CMAKE_USE_SPROCS;
- CMAKE_USE_WIN32_THREADS.

O segundo parâmetro da lista acima indica a biblioteca que deverá ser ligada para o suporte a *multithread*, no caso deste projeto, *pthreadVC2*, uma implementação para *Windows* de *Pthreads*. Os campos recebem os valores *booleanos* "*on*" ou "*off*" para indicar o tipo de *thread* que será utilizado.

Terminada a configuração das diretivas, o *CMake* cria um projeto (ou arquivo *makefile*, conforme o compilador indicado) com todas as dependências resolvidas para compilação. A compilação do FLTK *multithread,* bem como dos programas do usuário, é feita de maneira análoga: define-se um "CMakeLists.txt" indicando os arquivos fonte do projeto e suas dependências. Através do *CMake* gera-se um projeto para abertura no compilador desejado. A Fig. 12 ilustra o arquivo de configuração deste trabalho, onde "paralelo" é o nome com o qual o projeto será gerado e "paralelo.cxx" é o arquivo que implementa a função "*main*()".

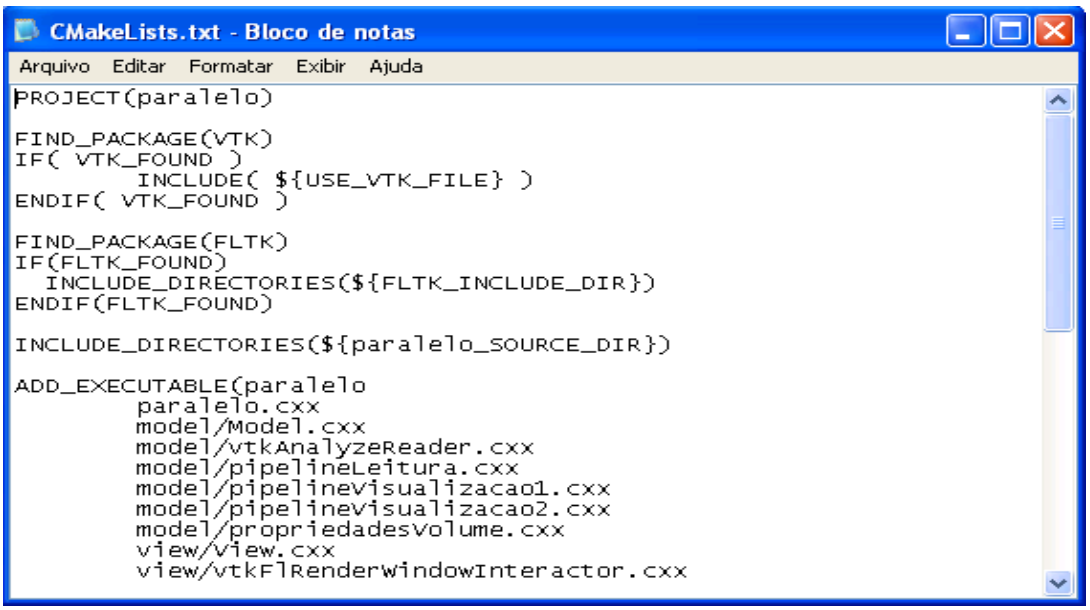


Figura 12 – Arquivo para configuração no *CMake* da ferramenta desenvolvida.

5.2.2 Classes desenvolvidas

A estrutura de classes do programa foi desenvolvida baseada no padrão de arquitetura de s*oftware Model View Controller* (MVC). Esse modelo divide o *software* em três camadas: *Model, View* e *Controller* (MVC, 2007). A Fig. 13 apresenta a relação entre essas camadas e os arquivos de código-fonte que as compõem.

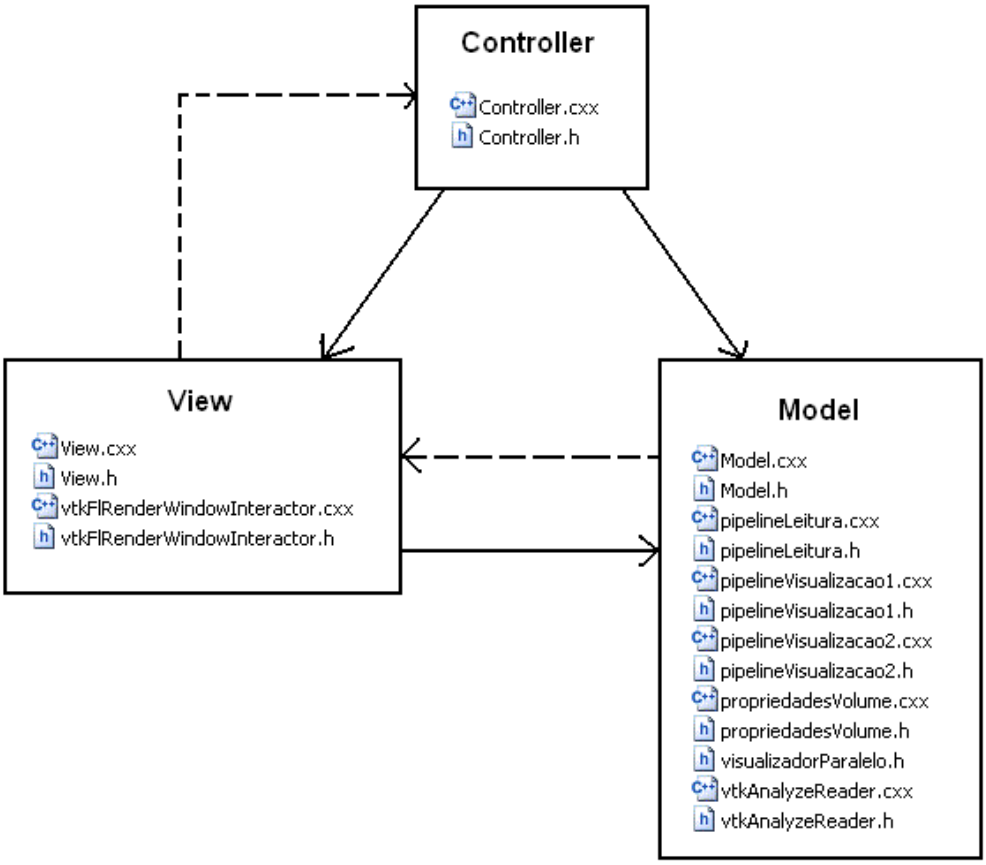


Figura 13 – Relação entre as classes desenvolvidas. As setas sólidas indicam associação direta e as pontilhadas representam associação indireta.

5.2.2.1 Camada *Controller*

A camada *Controller* processa e responde a eventos e invoca alterações no *Model*, se necessário. Em geral, seus métodos atendem requisições feitas pelas funções de *callback* da interface com o usuário. Além dessas funcionalidades, o C*ontroller* disponibiliza também os seguintes métodos:

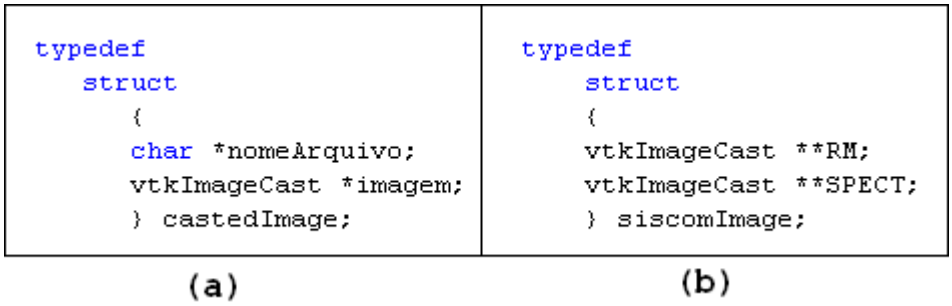
- void ativaInterface(void): exibe os botões para manipulação da GUI após a renderização dos volumes;
- void exibeFatias2D(void): é o responsável pela exibição dos exames no formato de fatias em 2D (duas dimensões). Ao ser chamado pela V*iew*, faz requisições para que o *Model* gere as fatias, solicita a criação de uma nova janela de visualização na classe V*iew* e então envia para essa as informações a serem exibidas;
- void exibeJanela(int numero): efetua a abertura da janela passada como argumento;
- void fechaJanela(int numero): responsável pelo fechamento da janela passada como parâmetro;
- int selecionaImagens(void): cria as janelas para seleção dos arquivos dos exames e associa então os arquivos indicados aos respectivos *pipelines* de visualização na classe *Model*.

5.2.2.2 Camada *Model*

A camada *Model* é a responsável pela lógica do programa. Efetua as operações de leitura e manipulação dos exames. O arquivo principal da classe é o "Model.cxx". Nos arquivos "pipelineLeitura.cxx", "pipelineVisualizacao1.cxx" e "pipelineVisualizacao2.cxx" são implementados os *pipelines* para leitura e visualização dos exames, respectivamente. A diferença entre os dois *pipelines* de visualização definidos é o número de volumes que eles exibem. O *pipeline* 1, utilizado na renderização das janelas "Fisionomia" e "Foco Epileptogênico", reconstrói apenas 1 volume, enquanto o *pipeline* 2, utilizado na renderização da janela "Exames alinhados" efetua a reconstrução de 2 volumes. Já o arquivo "propriedadesVolume.cxx" contém a coleção de propriedades para os volumes que é oferecida na interface com o usuário. Por fim, "vtkAnalyzeReader.cxx" implementa a leitura das imagens *Analyze.*

Ao ser criada uma instância de *Model*, seu método construtor efetua uma chamada ao método "inicializa" em "propriedadesVolume" para que seja gerada a coleção de propriedades. Ainda no construtor, é feita a alocação das estruturas que armazenam o resultado da leitura dos exames, sendo duas estruturas do tipo

“castedImage”, uma para a imagem de RM e outra para o SPECT intercrítico subtraído de SPECT crítico, e uma estrutura do tipo “siscomImage” para as imagens alinhadas. Essas estruturas são definidas no arquivo de cabeçalho da classe e são colocadas na Fig. 14.



```
typedef
   struct
      {
      char *nomeArquivo;
      vtkImageCast *imagem;
      } castedImage;
```

(a)

```
typedef
    struct
    {
    vtkImageCast **RM;
    vtkImageCast **SPECT;
    } siscomImage;
```

(b)

Figura 14 – Estruturas utilizadas na manipulação das imagens. Em a) estrutura para o *pipeline* simples e em b) estrutura operada no *pipeline* composto.

No momento em que o usuário seleciona os arquivos relativos aos exames, o *Controller* associa os nomes dos mesmos ao parâmetro “nomeArquivo” da estrutura “castedImage”. Então, o método “carregaImagens()” do *Model* cria duas instâncias do *pipeline* de leitura, uma para cada estrutura “castedImage”. A Fig. 15 apresenta a estrutura do *pipeline* de leitura.

```
#include "pipelineLeitura.h"

void* pipelineLeitura(void* Imagem) {
   castedImage* arquivo = (castedImage*) Imagem;
   vtkAnalyzeReader *leitorImagem = vtkAnalyzeReader::New();
   leitorImagem->SetHeaderFileName(arquivo->nomeArquivo);
   leitorImagem->Update();
   arquivo->imagem->SetInputConnection(leitorImagem->GetOutputPort());
   arquivo->imagem->SetOutputScalarTypeToUnsignedChar();
   pthread_exit(NULL);
   return NULL;
}
```

Figura 15 – Estrutura do *pipeline* de leitura dos exames

O *pipeline* de leitura recebe como parâmetro a estrutura “castedImage” em que a imagem será alocada e faz então uma chamada à classe “vtkAnalyzeReader”

para que essa efetue a leitura das imagens indicadas pelo usuário. A saída do leitor *Analyze* data é um objeto “vtkImageData” formado por valores do tipo *short*.

Entretanto, as classes para reconstrução volumétrica utilizadas neste projeto operam apenas com valores *unsigned short* ou *unsigned char*. Como o VTK trabalha com *pipelines* onde os dados passados de uma classe para a outra devem ser obrigatoriamente de mesmo tipo, é necessário aplicar uma operação de *type casting* no objeto de imagem criado pelo leitor de modo a transformá-lo em um “vtkImageData” com escalares do tipo *unsigned char*. Esse processo é realizado pela classe “vtkImageCast” (Fig. 15). Terminado esse passo, são atribuídos os endereços das imagens SPECT e RM para os respectivos parâmetros na estrutura “siscomImage”.

Os outros dois principais métodos da camada *Model* são o “carregaPipelines()” e o “carregaFatias2D()”. O primeiro cria e inicializa os três *pipelines* para reconstrução volumétrica - um para cada janela (Fisionomia, Exames Alinhados e Foco Epileptogênico). O segundo executa o *pipeline* para visualização das fatias em 2D. A implementação desses *pipelines* é colocada nas seções 6.2.3 e 6.2.4, respectivamente.

5.2.2.3 Camada *View*

A camada *View* implementa a interface com o usuário. É na *View* que são incluídas as classes do FLTK e a classe “vtkFlRenderWindowInteractor” (Fig. 16).

São cinco as janelas implementadas na classe *View*. Essas janelas foram desenvolvidas para diferentes finalidades, a Seção 5.2.5 apresenta as mesmas de maneira detalhada.

```
// Bibliotecas do FLTK necessárias para construção da interface
#include <FL/Fl.H>
#include <FL/Fl_Button.H>
#include <FL/Fl_Check_Button.H>
#include <FL/Fl_Double_Window.H>
#include <FL/Fl_Group.H>
#include <FL/Fl_Input.H>
#include <FL/Fl_Input_Choice.H>
#include <FL/Fl_Menu_Bar.H>
#include <FL/Fl_Output.H>
#include <FL/Fl_Round_Button.H>
#include <FL/Fl_Slider.H>
#include <FL/Fl_Spinner.H>
#include <FL/Fl_Tabs.H>
#include <FL/Fl_Text_Display.H>
#include <FL/Fl_Value_Input.H>

// Classes complementares do visualizador
#include "../controller/Controller.h"
#include "../model/Model.h"
#include "../util/Observer.h"

class Controller;

// Bibliotecas do VTK necessárias para renderização
#include "vtkRenderer.h"
#include "vtkRenderWindow.h"
#include "vtkInteractorStyleImage.h"
#include "vtkFlRenderWindowInteractor.h"
```

Figura 16 – Trecho de código que inclui as classes herdadas pela *View*.

5.2.3 *Pipeline* de reconstrução volumétrica

Foram desenvolvidos dois *pipelines* de visualização para a implementação da reconstrução volumétrica. A diferença entre esses é que em um deles apenas um volume é reconstruído (pipelineVisualizacao1.cxx), enquanto no outro, o resultado final é uma composição de dois volumes (pipelineVisualizacao2.cxx). Ambos são complementados pelas definições contidas no arquivo “propriedadesVolume.cxx”, pois é nele que são definidas as funções de transferência.

Os *pipelines* de visualização definem as propriedades dos volumes e do algoritmo de *Ray-casting*. A Fig. 17 representa a estrutura do *pipeline* para visualização de um volume. No *pipeline* para visualização composta, a estrutura básica se mantém a mesma, porém com os atributos duplicados.

```
class pipelineVisualizacao1 {
  public:
  static pipelineVisualizacao1 *New(void);
  void Delete(void);
  void Inicializa();
  vtkRenderer* GetOutput(void);
  void defineEntrada(castedImage* imagemEntrada);
  void defineFuncaoRaio(int classe, int opcao);
  void definePropriedadesVolume(vtkVolumeProperty *propriedadesVolume);
  void movimentaCamera(int direcao, int quantidadeMovimento);

  vtkVolumeRayCastMapper *raycastMapper;
  vtkRenderer *renderer;
  vtkVolumeProperty *propriedades;
  vtkVolume *volume;

  private:
  vtkFiniteDifferenceGradientEstimator *gradest;
  vtkVolumeRayCastCompositeFunction *compositeFunction;
  vtkVolumeRayCastMIPFunction *MIPFunction;

};
```

Figura 17 – Estrutura do *pipeline* de visualização para um volume.

Os métodos definidos pelo *pipeline* têm os seguintes propósitos:

- New(): instancia um novo *pipeline* de visualização;
- Delete(): libera a memória alocada pelo *pipeline*;
- Inicializa(): aloca o espaço em memória necessário para as partes do *pipeline* e inicializa os ponteiros;
- GetOutput(): retorna o *renderer* que contém o resultado da execução do *pipeline* para ser exibido na janela de visualização;
- defineEntrada(): define a imagem de entrada do *pipeline*;
- defineFuncaoRaio(): altera a função de *Ray-casting* que será utilizada para reconstruir o volume;
- definePropriedadesVolume(): relaciona um conjunto de propriedades de iluminação, cor, entre outros, ao volume;
- movimentaCamera(): movimenta a câmera no *renderer* do *pipeline* de acordo com os parâmetros recebidos.

Os dois passos principais do algoritmo *Ray-casting* são determinar os valores encontrados no raio e então processá-los de acordo com a função de raio

indicada. Em geral, inclusive no caso do VTK, essas etapas são implementadas de forma combinada. Entretanto, elas serão abordadas separadamente para discutir de maneira mais detalhada a implementação. A seguir são discutidos os componentes do *pipeline* de execução da reconstrução volumétrica.

5.2.3.1 Funções de raio

Dentre as funções de raio oferecidas pela biblioteca VTK foram escolhidas duas: *Maximum Intensity Projection* (MIP) e *Composite*. A primeira é uma função que realiza operações básicas nos valores escalares, enquanto a segunda faz uso de uma técnica de composição *alpha*, tratando os valores do raio como amostras de opacidade acumulada por unidade de distância. A função MIP é uma maneira mais simples de visualizar dados volumétricos, pois produz uma visão intuitiva de dados subjacentes. Entretanto, tem como inconveniente a dificuldade de localização espacial dos dados.

As funções para reconstrução volumétrica *Composite* e MIP são implementadas pelas classes do VTK “vtkVolumeRayCastCompositeFunction” e “vtkVolumeRayCastMIPFunction”, respectivamente. A criação do objeto função é feita invocando-se o método “New()” da classe desejada. Na seqüência, deve-se executar um método que indica qual das abordagens da função será utilizada:

- Realizar primeiro a classificação dos *voxels* no volume ou a interpolação, no caso de funções *Composite*;
- Usar o método de maximização baseado nos escalares ou na opacidade, para as funções do tipo MIP.

No caso das funções *composite*, em geral, classificar os *voxels* primeiro gera resultados mais precisos. O processo de criação das funções de raio é ilustrado na Fig. 18, onde são declarados as 4 combinações possíveis de funções MIP e *Composite*.

```
// Definição das funções de Ray-casting possíveis
// Função do tipo Composite "ClassifyFirst"
vtkVolumeRayCastCompositeFunction *compositeFunction1 = vtkVolumeRayCastCompositeFunction::New();
   compositeFunction1->SetCompositeMethodToClassifyFirst();
// Função do tipo Composite "InterpolateFirst"
vtkVolumeRayCastCompositeFunction *compositeFunction2 = vtkVolumeRayCastCompositeFunction::New();
   compositeFunction2->SetCompositeMethodToInterpolateFirst();
// Função do tipo MIP "MaximizeMethodToScalarValue"
vtkVolumeRayCastMIPFunction *MIPFunction1 = vtkVolumeRayCastMIPFunction::New();
   MIPFunction1->SetMaximizeMethodToScalarValue();
// Função do tipo MIP "MaximizeMethodToOpacity"
vtkVolumeRayCastMIPFunction *MIPFunction2 = vtkVolumeRayCastMIPFunction::New();
   MIPFunction2->SetMaximizeMethodToOpacity();
```

Figura 18 – Criação das funções de *Ray-casting* no VTK.

Funções que produzem volumes coloridos e com transparência possuem mais informações, entretanto, a dificuldade de interpretação ou a facilidade de interpretação equivocada de algum dado também são maiores. Portanto, é interessante oferecer a possibilidade de se utilizar mais de uma técnica ao visualizar o volume. Partindo desse princípio, optou-se na aplicação desenvolvida por oferecer ao usuário a opção de escolher em tempo de execução a função a ser utilizada, através da interface da ferramenta. Dessa forma, cada usuário pode visualizar diferentes imagens da maneira que julgar mais conveniente.

5.2.3.2 Funções de interpolação

Na representação de um volume, os valores escalares dos *voxels* são definidos como pontos de uma matriz 3D regular. Entretanto, o algoritmo de *Ray-casting* freqüentemente precisa obter amostras de valores em locais arbitrários do volume. Para que isso seja possível, é necessário que sejam definidas funções de interpolação que possam retornar valores escalares para qualquer espaço intermediário localizado entre quaisquer pontos do volume.

Para abordar esse problema, foram utilizadas duas funções de interpolação oferecidas pelo VTK no escopo da reconstrução volumétrica: *linear interpolation* (interpolação linear) e *nearest neighbor interpolation* (interpolação por vizinho mais próximo), também conhecida como interpolação constante ou ainda interpolação de ordem zero.

A interpolação constante recupera o valor do ponto mais próximo e o utiliza no ponto cuja informação está sendo gerada. É a função de interpolação mais simples e que possui o custo computacional menor. Já a interpolação linear gera um valor para o ponto desejado baseando-se na distância entre esse e os pontos mais próximos em cada um dos três eixos. Embora seu custo computacional seja maior se comparado ao da interpolação constante, o resultado oferecido é também de maior qualidade.

Adotou-se como padrão neste projeto a interpolação linear devido a sua qualidade. Porém, dado que qualidade interfere na velocidade, a opção pela interpolação desejada torna-se uma escolha subjetiva que varia de acordo com cada caso. Por isso, foi mantida na interface com o usuário a possibilidade de o mesmo alterar o tipo de interpolação utilizado em cada imagem.

5.2.3.3 Amostragem dos valores e percurso do raio

A passagem do raio pelo volume pode ser implementada de duas maneiras: percorrendo intervalos uniformes ou fazendo uma travessia em uma representação discreta do raio através do volume, examinando cada *voxel* encontrado. A abordagem a ser adotada depende de vários fatores como a técnica de interpolação utilizada, a função de raio e a relação desejada entre qualidade e velocidade.

Uma dificuldade encontrada ao trabalhar com intervalos uniformes é determinar um bom tamanho para o intervalo (passo). De maneira geral, se optarmos por um passo muito grande, dados são perdidos e a qualidade é reduzida. Por outro lado, ao escolher um passo muito pequeno, aumentará significativamente o tempo de processamento. Assim, em alguns casos é mais adequado examinar cada *voxel* ao longo do raio ao invés de intervalos, como ocorre quando utilizado o método de interpolação constante.

Constatou-se empiricamente (vide seção 6.3.2) que a abordagem ideal para este projeto é percorrer intervalos uniformes com tamanho de passo igual a 1 (um), que é o padrão no VTK. Portanto, foi definido, em nível de codificação, a manutenção do tamanho do passo, sem que o usuário possa alterá-lo.

5.2.3.4 Classificação dos *voxels* – funções de transferência

Nas imagens de RM, bem como nas de SPECT, a classificação é necessária para selecionar quais partes dos exames desejamos analisar, ou seja, para definir as regiões de interesse. Um exemplo desse procedimento é definir valores correspondentes aos dados que representam ossos e valores que correspondem ao ar, para que então os primeiros sejam exibidos e os segundos ignorados. Em reconstrução volumétrica, as funções que realizam a classificação são chamadas funções de transferência. Tais funções são responsáveis por mapear a informação de um *voxel* em valores diferentes de opacidade e cor para indicar os tecidos.

As imagens dos exames utilizadas neste projeto são compostas por múltiplos materiais (tecido ósseo, tecido epitelial, etc.), o que implica na necessidade de utilizar funções para classificação mais complexas que permitam delimitar as áreas de interesse através de intervalos de transparência e opacidade, ao invés da maneira binária com valores de *threshold*. Isso porque, o volume, embora único, deve ser processado como uma composição de vários materiais a fim de permitir a navegação por seus dados de maneira intuitiva, amigável e realista.

Como a proposta deste projeto é empregar cores para facilitar a detecção do foco epileptogênico, fez-se necessário utilizar também funções para mapeamento dos escalares em cores no padrão RGB, além das funções para transferência de escalares para opacidade. As classes do VTK que implementam transferência de escalar em opacidade e transferência para cores são, respectivamente, “vtkPiecewiseFunction” e “vtkColorTransferFunction”.

Além disso, classificar um volume baseado em valores escalares apenas pode não ser muitas vezes capaz de isolar um objeto de interesse. Em complemento, o VTK implementa a técnica de magnitude do gradiente às funções de transferência. Com essa técnica, é possível especificar objetos no volume baseados em uma combinação entre valores escalares e a magnitude do gradiente. Essa implementação também é realizada pela classe “vtkPiecewiseFunction”. Por razões matemáticas relacionadas ao processo de interpolação, é aconselhável que, em conjunto a técnica que mapeia magnitude do gradiente em opacidade, seja utilizada uma técnica de estimativa de diferenças finitas. O VTK oferece suporte a essa técnica complementar através da classe “vtkFiniteDifferenceGradientEstimator”.

Um exemplo do processo de criação das funções de transferência é colocado no trecho de código representado pela Fig. 19, onde são declaradas funções para transformação de escalares em opacidade, escalares em níveis de cinza, escalares em cores e, por fim, uma função para transferência da magnitude do gradiente em opacidade.

```
/* Definição das funções de transferência responsáveis por mapear
escalares em opacidade */
vtkPiecewiseFunction *oTFun1 = vtkPiecewiseFunction::New();
   oTFun1->AddSegment(0, 0.0, 75, 0.0);
   oTFun1->AddSegment(75, 0.0, 86, 0.09);
   oTFun1->AddSegment(86, 0.09, 105, 0.11);
   oTFun1->AddSegment(105, 0.11, 255, 0.05);

/* Definição das funções de transferência responsáveis por mapear
valores escalares em valores na escala de cinza */
vtkPiecewiseFunction *gTFun1 = vtkPiecewiseFunction::New();
   gTFun1->AddSegment(0, 0.0, 75, 0.0);
   gTFun1->AddSegment(75, 0.0, 100, 1.0);
   gTFun1->AddSegment(100, 1.0, 120, 0.0);
   gTFun1->AddSegment(120, 0.0, 255, 1.0);
vtkColorTransferFunction *cTFun = vtkColorTransferFunction::New();
   cTFun->AddRGBPoint(   0, 1.0, 0.0, 0.0 );
   cTFun->AddRGBPoint(  64, 1.0, 1.0, 0.0 );
   cTFun->AddRGBPoint( 128, 0.0, 1.0, 0.0 );
   cTFun->AddRGBPoint( 192, 0.0, 1.0, 1.0 );
   cTFun->AddRGBPoint( 255, 0.0, 0.0, 1.0 );

/* Definição da função de transferência responsável por mapear
a magnitude do gradiente em opacidade */
vtkPiecewiseFunction *goTFun = vtkPiecewiseFunction::New();
   goTFun->AddPoint(   0, 0.0 );
   goTFun->AddPoint(  30, 0.0 );
   goTFun->AddPoint(  40, 1.0 );
   goTFun->AddPoint( 255, 1.0 );
```

Figura 19 – Definição das funções de transferência no VTK.

5.2.3.5 Iluminação do volume

A utilização de técnicas avançadas de iluminação no algoritmo de *Ray-casting* introduz uma complexidade maior na implementação do mesmo. Contudo, há grandes vantagens em relação ao resultado final para visualização, o que justifica essa decisão. Assim, em relação à iluminação, optou-se por reconstruir os volumes

com valores pré-definidos, porém, permitindo ao usuário manipulá-los através da interface.

O VTK oferece um modelo bastante elaborado de iluminação, composto por três tipos de luz: ambiente, difusa e especular, além de um parâmetro para utilização de *shading* (sombreamento). Esse modelo foi desenvolvido através de técnicas matemáticas e computacionais altamente complexas para permitir, assim, ao programador tirar proveito dos benefícios desse sistema elaborado de iluminação sem que a performance do *software* seja comprometida. Maiores detalhes com relação à implementação podem ser obtidos em Schroeder et al. (2002). No Capítulo 6 os custos computacionais desse modelo são analisados.

Existem classes específicas do VTK para manipulação da iluminação, como a “vtkLight” e “vtkLightKit”. Entretanto, na reconstrução volumétrica neste projeto essas classes não foram utilizadas de maneira direta. Isso é possível pelo fato de o VTK oferecer métodos para a manipulação das luzes em um volume através da simples passagem de parâmetros para o objeto que contém as propriedades do volume, o qual será abordado na próxima seção.

A influência dos diferentes tipos de luz é ilustrada na Fig. 20. A primeira esfera é o objeto original, onde as três fontes de luz (ambiente, difusa e especular) encontram-se desligadas. Em (b), a luz ambiente encontra-se com intensidade total, as demais desligadas. Em (c), a esfera é iluminada apenas pela luz difusa com intensidade total. Já em (d), apenas a luz especular está ligada, a intensidade é máxima e o parâmetro “*power*” (poder) com valor mínimo. Por fim, em (e), somente a luz especular encontra-se ligada, porém, neste caso com o “*power*” em seu valor máximo.

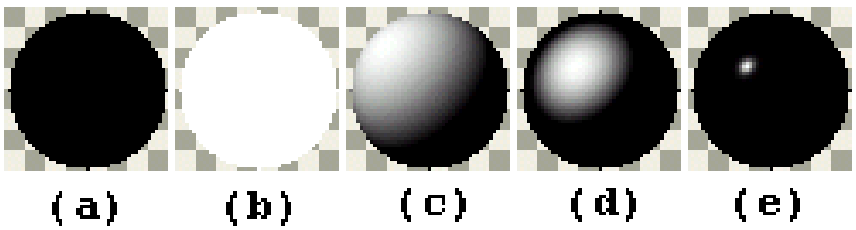


Figura 20 – Influência das diferentes fontes de luz em uma esfera. a) sem iluminação; b) luz ambiente; c) luz difusa; d) luz especular com poder mínimo; e) luz especular com poder máximo.

Os métodos que configuram as luzes ambiente, difusa e especular na reconstrução volumétrica são, respectivamente: “SetAmbient()”, “SetDiffuse()” e “SetSpecular()”. Já o parâmetro “*power*” da luz especular é modificado através da função “SetSpecularPower()”. Todos esses métodos recebem como parâmetro um valor do tipo *double*, o qual representa a intensidade desejada. Existem ainda os métodos “GetAmbient()”, “GetDiffuse()”, “GetSpecular()” e “GetSpecularPower()” que recuperam os valores em uso num dado momento.

5.2.3.6 Propriedades do volume

Até o presente momento, colocou-se a forma como os conceitos atrelados a reconstrução volumétrica foram abordados. Para a integração dos mesmos, o VTK utiliza uma classe que agrupa as propriedades escolhidas em um objeto de propriedades, a “vtkVolumeProperty”. É nela que são definidos os parâmetros de iluminação, interpolação, a relação entre o volume e uma dada função de transferência, entre outros. A Fig. 21 representa a criação e configuração de uma instância de “vtkVolumeProperty”.

```
// Define volume's properties
vtkVolumeProperty *propriedades = vtkVolumeProperty::New();
   propriedades->SetShade(1);
   propriedades->SetAmbient(0.3);
   propriedades->SetDiffuse(1.0);
   propriedades->SetSpecular(0.4);
   propriedades->SetSpecularPower(8.0);
   propriedades->SetScalarOpacity(oTFun);
   propriedades->SetGradientOpacity(goTFun);
   propriedades->SetColor(gTFun);
   propriedades->SetInterpolationTypeToLinear();
```

Figura 21 – Trecho de código que instancia e configura um conjunto de propriedades para um volume.

No exemplo acima, as cinco primeiras linhas definem o sistema de iluminação que será utilizado na reconstrução do volume. Já as três subseqüentes relacionam o volume com as funções de transferência anteriormente definidas (Fig. 19). A última define o tipo de interpolação.

5.2.3.7 Estágio final – criação da cena

Após devidamente configuradas todas as funções de transferência e raio, entre outros, chega-se ao estágio final do *pipeline* de reconstrução volumétrica. Nessa fase, o volume é efetivamente criado e ligado às propriedades definidas. Na seqüência, ele é reconstruído e relacionado a uma dada cena em uma janela de visualização. Ou seja, é nesse momento que o resultado é de fato gerado. A Fig. 22 ilustra, de maneira simplificada, a criação de um volume e seu *mapper*.

```
// Create RayCast mapper
vtkFiniteDifferenceGradientEstimator *gradEst = vtkFiniteDifferenceGradientEstimator::New();
vtkVolumeRayCastMapper *raycastMapper = vtkVolumeRayCastMapper::New();
   raycastMapper->SetInput((*(imagemEntrada->SPECT))->GetOutput());
   raycastMapper->SetVolumeRayCastFunction(compositeFunction);
   raycastMapper->SetGradientEstimator(gradEst);
   raycastMapper->SetNumberOfThreads(1);
   raycastMapper->SetSampleDistance(1);

// Create Volume
vtkVolume *volume = vtkVolume::New();
   volume->SetProperty(propriedades);
   volume->SetMapper(raycastMapper);
```

Figura 22 – Trecho de código para criação de um volume e seu *mapper*.

No *pipeline* do VTK, todos os objetos possuem um *mapper*, que é o responsável por mapear os dados em primitivas gráficas. No caso de volumes a ser percorridos pelo *Ray-casting,* o *mapper* é uma instância da classe "vtkVolumeRayCastMapper". Os métodos do *mapper* na Fig. 22 definem, respectivamente, sua entrada, ou seja, o objeto a reconstruir que, nesse caso, é a saída do leitor *Analyze*, a função de raio que será utilizada, a classe para o cálculo do gradiente, o número de *threads* em que o *Ray-casting* deve ser dividido e o tamanho do passo que o mesmo percorre.

Na seqüência, é criado efetivamente o objeto volume, que é uma instância da classe “vtkVolume”. Os métodos invocados para um volume são o “SetProperty()” e o “SetMapper()”. No primeiro, é passado um parâmetro do tipo “vtkVolumeProperty” que contém o conjunto das propriedades, conforme visto na seção anterior. Já o segundo é o que relaciona um volume a seu *mapper*.

O próximo passo do *pipeline* é adicionar o volume a um *renderer*, objeto que instancia a classe “vtkRenderer”. Isso é feito invocando-se o método “AddVolume()” do *renderer* e passando como parâmetro o volume. Fazem parte ainda do estágio final do *pipeline* de visualização a janela de renderização, instância da classe “vtkRenderWindow” e o *interactor*, objeto do tipo “vtkFlRenderWindowInteractor”.

A janela de renderização é quem exibe a cena criada, resultado da saída do *renderer*. Já o *interactor* é quem permite a interação entre o observador e a janela de renderização. A execução desse *pipeline* tem como resultado final os volumes reconstruídos. Um exemplo desse resultado é colocado na Seção 5.2.5.5, onde a interface da janela principal da ferramenta é discutida.

5.2.4 *Pipeline* de exibição das fatias 2D

A proposta inicial deste projeto não previa a funcionalidade exibir as imagens no formato de fatias 2D. No entanto, esta apresentação permite complementar a visualização, tendo sido adicionada à ferramenta desenvolvida por esta razão. A implementação do seu *pipeline*, de maneira geral, segue os mesmos passos e princípios apresentados para a reconstrução volumétrica, diferindo com relação às classes utilizadas.

Nome da classe	**Função:**
vtkImageActor	Equivalente à classe “vtkVolume” na reconstrução volumétrica;
vtkImageMapToColors	Recebe uma imagem como entrada e a transforma de acordo com uma lookup table (vtkLookupTable);
vtkImagePermute	Reordena os eixos da imagem no espaço;
vtkInteractorStyleImage	Define um interactor com características especiais para interação com uma imagem;
vtkLookupTable	Mapeia escalares em cores RGB ou vice-e-versa.

Figura 23 - Quadro das classes adicionadas para implementação da visualização em fatias 2D.

A leitura dos arquivos *Analyze* não é necessária no momento de geração das fatias 2D, pois, uma vez lidos os dados para reconstrução dos volumes, eles permanecem na memória até o término da execução da ferramenta. Já a necessidade da criação de *renderes* e *render windows* se mantém para as fatias 2D, pois as janelas em que essas serão exibidas não são as mesmas que os volumes. No quadro da Fig. 23 são rapidamente descritas as classes extras utilizadas para a visualização 2D.

5.2.5 Interface com o usuário

5.2.5.1 Janela para configuração do número de *threads*

Oferece ao usuário a opção de modificar o número de *threads* utilizado em cada um dos *Ray-castings*. No momento que uma alteração é realizada, o método "SetNumberOfThreads()" do referido *mapper* é invocado recebendo como argumento o valor digitado. Sua interface é apresentada na Fig. 25 e o trecho do cabeçalho da classe relativo a essa janela na Fig. 24.

5.2.5.2 Janela para configuração do número de *frames* por segundo

Conforme visto no Capítulo 1, em aplicações interativas é desejável uma taxa repetição em torno de em torno de 20 *frames* por segundo. Para facilitar essa meta, o VTK implementa em seus renderizadores métodos onde é possível definir uma quantidade esperada de quadros por segundo. Feita essa definição, o VTK opera suas classes relacionadas ao parâmetro "LOD" (*level of details*) para aproximar o máximo possível da taxa de *frames* almejada. Exemplos dessas classes são a "vtkLODActor" e "vtkLODProp3D ".

De fato, o que acontece, é que no momento em que o usuário determina um número de *frames* desejado, quando a cena ou um objeto da mesma precisar ser reconstruído por diversas vezes para interação, isso será feito utilizando um nível de detalhamento menor do mesmo. No momento que a interação termina, o objeto é então renderizado na sua qualidade normal. A Fig. 27 apresenta a interface da janela. Na Fig. 26 é colocado o trecho do cabeçalho relativo a essa.

```
//---------------------------------------------------------------
// Janela para configuração do número de threads
//---------------------------------------------------------------

// Itens da Janela
public:
Fl_Double_Window* geraJanelaThreads();
Fl_Double_Window *janelaThreads;

private:
Fl_Button *cancelarThreadsButton;
Fl_Button *reconThreadsButton;
Fl_Spinner *threadsAlinhados;
Fl_Spinner *threadsFisionomia;
Fl_Spinner *threadsFoco;


// Callbacks
void cb_janelaThreads_i(Fl_Double_Window*, void*);
static void cb_janelaThreads(Fl_Double_Window*, void*);
void cb_reconThreadsButton_i(Fl_Button*, void*);
static void cb_reconThreadsButton(Fl_Button*, void*);
void cb_cancelarThreadsButton_i(Fl_Button*, void*);
static void cb_cancelarThreadsButton(Fl_Button*, void*);
```

Figura 24 – Trecho do cabeçalho da classe *View* correspondente à Janela para a configuração do número de *threads*.

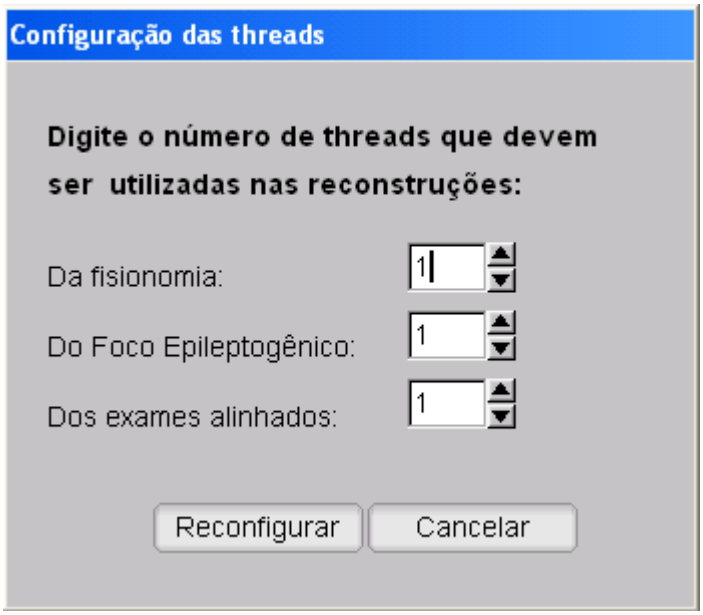


Figura 25 – Interface da Janela para a configuração do número de *threads*.

```
//---------------------------------------------------------------------
// Janela para configuração da taxa de quadros pro segundo
//---------------------------------------------------------------------

// Itens da Janela
public:
Fl_Double_Window* geraJanelaQuadros();
Fl_Double_Window *janelaQuadros;

private:
Fl_Button *cancelarQuadrosButton;
Fl_Button *reconQuadrosButton;
Fl_Spinner *quadrosTaxa;


// Callbacks
void cb_janelaQuadros_i(Fl_Double_Window*, void*);
static void cb_janelaQuadros(Fl_Double_Window*, void*);
void cb_reconQuadrosButton_i(Fl_Button*, void*);
static void cb_reconQuadrosButton(Fl_Button*, void*);
void cb_cancelarQuadrosButton_i(Fl_Button*, void*);
static void cb_cancelarQuadrosButton(Fl_Button*, void*);
```

Figura 26 – Trecho do cabeçalho da classe *View* correspondente à Janela para a configuração do número desejado de quadros por segundo.

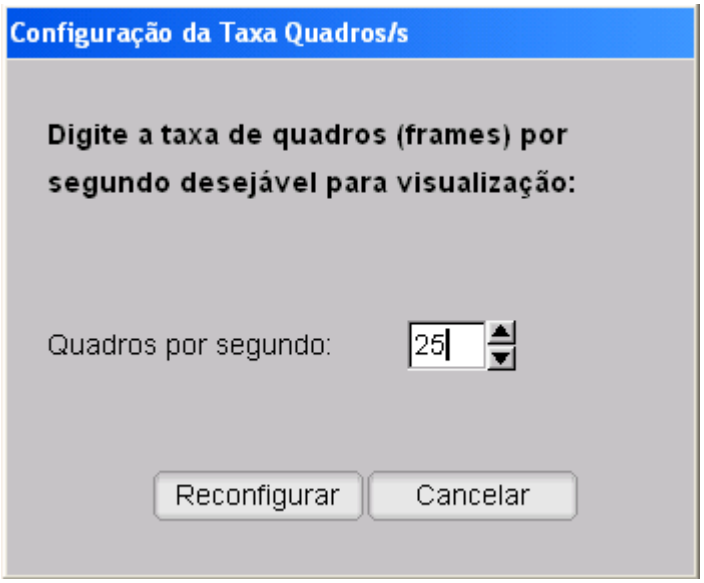


Figura 27 – Interface da Janela para a configuração do número desejado de quadros por segundo.

5.2.5.3 Janela para exibição das fatias em 2D

Essa janela implementa uma funcionalidade extra adicionada ao projeto: a possibilidade de visualizar as imagens alinhadas no formato de fatias bi-dimensionais. Essa abordagem é complementar a idéia inicial do trabalho, e pode ser útil para uma visualização mais próxima da tradicional dos exames. O objetivo é fazer uso do alinhamento de imagens, mas mantendo a visualização com a qual muitos especialistas estão habituados.

A Fig. 29 coloca o trecho do cabeçalho correspondente a essa janela. Na Fig. 28 é apresentada a interface da mesma. Abaixo de cada janela encontra-se um *slider* para navegação pelas fatias e, abaixo dessas, o número da fatia em exibição e o número total de fatias naquele eixo.

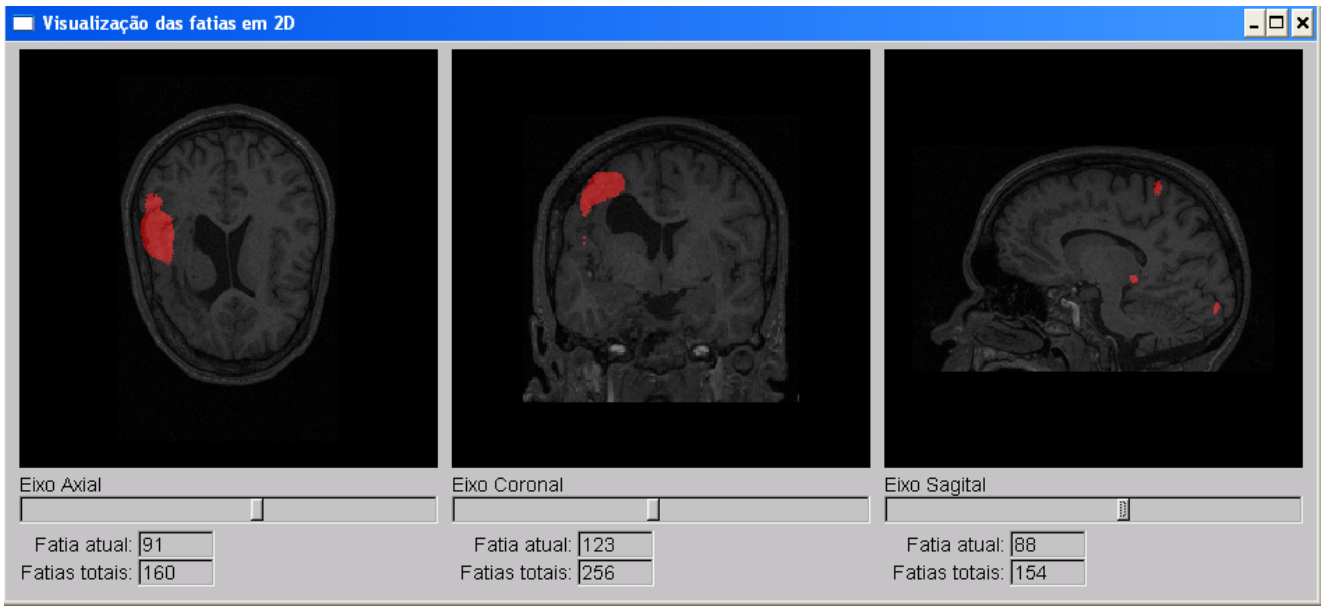


Figura 28 – Interface da Janela para visualização das fatias em 2D nos eixos Axial, Coronal e Sagital, respectivamente. As nuvens em vermelho na Ressonância Magnética correspondem às regiões onde o foco epileptogênico está localizado.

```
//---------------------------------------------------------------
// Janela para visualização das fatias 2D
//---------------------------------------------------------------

// Itens da Janela
public:
Fl_Double_Window* geraJanelaFatias();
Fl_Double_Window *janelaFatias;

private:
Fl_Output *fatiaAxial;
Fl_Output *fatiaCoronal;
Fl_Output *fatiaSagital;
Fl_Output *totalFatiasAxial;
Fl_Output *totalFatiasCoronal;
Fl_Output *totalFatiasSagital;
Fl_Slider *sliderAxial;
Fl_Slider *sliderCoronal;
Fl_Slider *sliderSagital;
vtkFlRenderWindowInteractor *flrwiAxial;
vtkFlRenderWindowInteractor *flrwiCoronal;
vtkFlRenderWindowInteractor *flrwiSagital;
vtkRenderer *rendererAxial;
vtkRenderer *rendererCoronal;
vtkRenderer *rendererSagital;
vtkRenderWindow *rwAxial;
vtkRenderWindow *rwCoronal;
vtkRenderWindow *rwSagital;

// Callbacks
void cb_janelaFatias_i(Fl_Double_Window*, void*);
static void cb_janelaFatias(Fl_Double_Window*, void*);
void cb_sliderAxial_i(Fl_Slider*, void*);
static void cb_sliderAxial(Fl_Slider*, void*);
void cb_sliderCoronal_i(Fl_Slider*, void*);
static void cb_sliderCoronal(Fl_Slider*, void*);
void cb_sliderSagital_i(Fl_Slider*, void*);
static void cb_sliderSagital(Fl_Slider*, void*);
```

Figura 29 – Trecho do cabeçalho da classe *View* correspondente à Janela para visualização das fatias em 2D.

5.2.5.4 Janela para exibição dos dados dos volumes

A finalidade dessa janela é fornecer informações quantitativas dos volumes que estão sendo visualizados. Esses dados são relacionados basicamente ao tamanho do volume e os valores de seus escalares. Na Fig. 30 é colocado o trecho do cabeçalho correspondente a essa janela, e a Fig. 31 ilustra a interface da mesma.

```
//---------------------------------------------------------------
// Janela para visualização das informações dos volumes
//---------------------------------------------------------------

// Itens da Janela
public:
Fl_Double_Window* geraJanelaInfos();
Fl_Double_Window *janelaInfos;
Fl_Text_Display *maximoRM;
Fl_Text_Display *maximoSPECT;
Fl_Text_Display *minimoRM;
Fl_Text_Display *minimoSPECT;
Fl_Text_Display *medioRM;
Fl_Text_Display *medioSPECT;
Fl_Text_Display *voxelsRM;
Fl_Text_Display *voxelsSPECT;
Fl_Text_Display *origemRM;
Fl_Text_Display *origemSPECT;
Fl_Text_Display *espacamentoRM;
Fl_Text_Display *espacamentoSPECT;
```

Figura 30 – Trecho do cabeçalho da classe *View* correspondente à Janela para visualização das informações dos volumes.

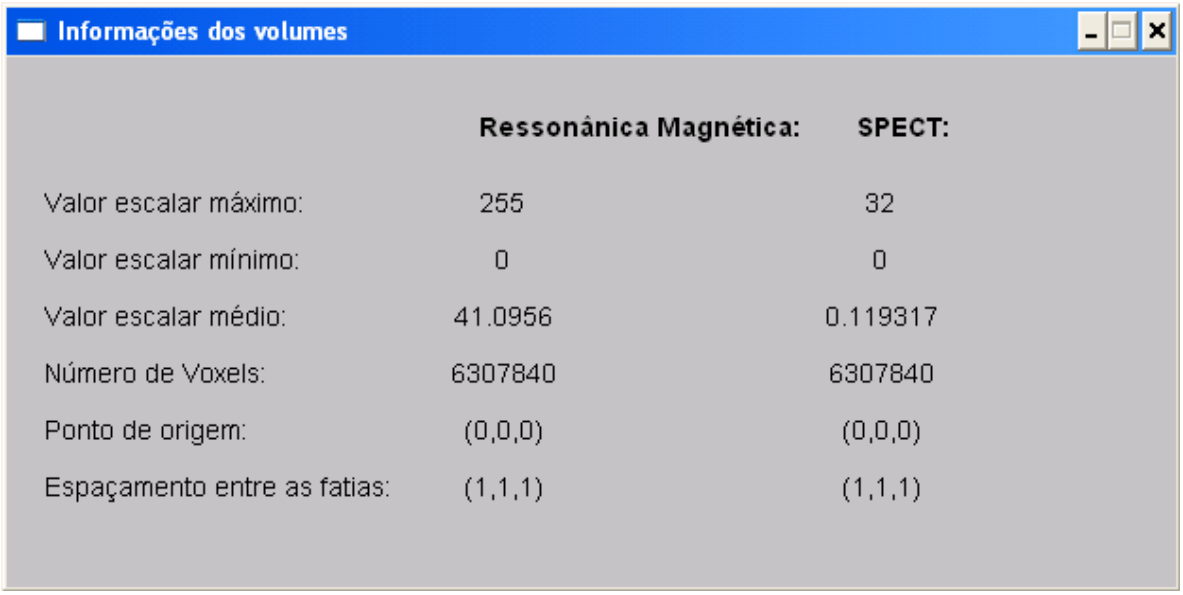


Figura 31 – Interface da Janela para a visualização dos dados dos volumes.

5.2.5.5 Janela principal

A janela principal é aquela na qual o usuário interage com os dados dos exames. Ela encontra-se dividida em duas partes, uma superior formada pelas cenas, ou janelas de exibição dos volumes, e a outra, inferior, onde se encontram os botões para a interação do usuário, conforme ilustrado na Fig. 32.

Figura 32 – Janela principal da ferramenta desenvolvida.

No quadrante inferior esquerdo encontra-se um grupo de abas onde o usuário tem a opção de alterar os parâmetros do *Ray-casting* anteriormente discutidos. As opções foram agrupadas por assunto, visando manter a interface intuitiva. Na parte central inferior, encontram-se dois grupos de botões, um para aplicação de *zoom* e outro para rotação simultânea dos volumes. Por fim, no canto direito inferior, encontra-se uma lista com as opções onde os parâmetros alterados nas abas devem ser aplicados.

É importante observar que a interação com os volumes relacionadas a transformações geométricas, como rotação e deslocamento, podem ser feitas através de eventos do *mouse* diretamente na janela desejada. Por exemplo, para

girar os três volumes o usuário opera os botões da interface, mas, caso deseje rotacionar apenas um dos três, pode fazer isso clicando com o *mouse* em sua respectiva janela.

Com relação aos menus, também estão agrupados por assunto. Em “arquivo” estão as operações de abrir os exames e fechar a ferramenta. Ao solicitar a abertura de imagens, o usuário será automaticamente solicitado a indicar os dois arquivos, de SPECT e RM. O menu “desempenho” oferece acesso às janelas para configuração do número de *threads* e do número desejado de quadros por segundo. Já no através “exibir” é feito o acesso à visualização das fatias bi-dimensionais e aos dados dos volumes.

6 AVALIAÇÃO E RESULTADOS

A avaliação dos resultados obtidos no presente trabalho se deu pela análise do desempenho da ferramenta construída em diferentes configurações de arquiteturas *single* e *multi-core*. Em particular observou-se a variação de desempenho considerando a variação no número de *threads*. Também foi avaliada a qualidade dos resultados obtidos através de uma comparação das imagens geradas pela ferramenta proposta com aquelas apresentadas pelo *software* proprietário *VolView*. Realizou-se ainda a compilação da aplicação em SOs variados. O objetivo dos experimentos em diferentes ambientes de *hardware* e *software* é contabilizar o tempo de processador gasto nas situações diversas e a portabilidade da solução desenvolvida. Já as análises do resultado visual buscam comparar o produto final deste projeto com o *software* proprietário e a influência que os parâmetros utilizados para reconstrução volumétrica exercem na visualização.

Este capítulo descreve os experimentos realizados, apresentando, para cada um desses os objetivos, justificativas e resultados, além das conclusões oriundas dos mesmos. Inicialmente são apresentadas as análises quantitativas (de desempenho) realizadas. Na seqüência são apresentados os experimentos tratando as questões de portabilidade e uma análise qualitativa comparando as imagens geradas com aquelas produzidas pelo *software* proprietário *VolView*.

6.1 Preparação dos testes

Para a realização do procedimento de avaliação do *software* em diferentes arquiteturas, o primeiro passo foi isolar e desabilitar os trechos de código responsáveis pela interface com o usuário de forma a eliminar possíveis interferências de eventos provindos da interação com o usuário nos tempos de execução. Dessa maneira, pode-se também automatizar a execução da ferramenta e, com isso, realizar um número maior de verificações obtendo uma amostra mais confiável dos resultados.

Terminada essa primeira fase, foram desenvolvidos programas em linguagem C++ que recebem como parâmetro, em linha de comando, o número de vezes que um teste deve ser realizado e, na seqüência, iniciam a execução da ferramenta. A estratégia adotada em relação aos *pipelines*, bem como de que forma os respectivos custos computacionais foram apurados, varia de teste para teste, e, por isso, é explicitada em detalhes nas seções correspondentes a cada caso.

Configuração	**Processador**	**Cache**	**Memória RAM**	**Placa de Vídeo**	**Classe**
Modelo 1	AMD Semprom 2600+ @ 1.84 GHz	256 KB L2	1 x 512 MB 333 MHz	Cirrus Logic 2 MB	Desktop
Modelo 2	AMD Atlhon XP 2000+ @ 1.67 GHz	256 KB L2	2 x 256 MB 266 MHz	GeForce 5200 128 MB	Desktop
Modelo 3	Intel Pentium 4 HT @ 3.00 GHz	1 MB L2	1 x 480 MB 333 MHz	Onboard 32 MB compartilhados	Desktop
Modelo 4	Intel Pentium D 805 @ 2.66 GHz	2 x 1 MB L2	1 x 496 MB 333 MHz	Onboard 16 MB compartilhados	Desktop
Modelo 5	Intel Core™2 Duo E6300 @ 1.86 GHz	2 MB L2	2 x 1 GB 667 MHz	GeForce 7600 GS 256 MB	Desktop
Modelo 6	Intel Core Duo T2300 @ 1.66 GHz	2 MB L2	2 x 448 MB 667 MHz	Onboard 128 MB compart.	Portátil
Modelo 7	AMD Turion 64 X2 TL-50 @ 1.60 GHz	2 x 256 KB L2	1 x 896 MB 667 MHz	Onboard 128 MB compart.	Portátil
Modelo 8	PowerPC G4 @ 1.5 GHz	512 KB L2	2 x 512 MB 333 MHz	ATI Mob.Radeon	Portátil
Modelo 9	2 x Xeon Dual Core 3070 @ 3,20 GHz	4 MB L2	2 x 1 GB 800 MHz	Onboard 2 MB compartilhados.	Servidor

Figura 33 - Quadro de configurações das arquiteturas utilizadas nos testes.

Dado que este projeto teve como foco o desenvolvimento de uma solução computacional otimizada para arquiteturas SMP MIMD GMSV, é naturalmente desejável sua avaliação em arquiteturas *multi-core* variadas, além da execução em computadores com processadores *single-core* no intuito de identificar possíveis ganhos advindos da exploração do paralelismo. Por isso, a avaliação da ferramenta desenvolvida deu-se em computadores de mesa, portáteis e servidores, conforme colocado no quadro da Fig. 33. A arquitetura "modelo 8" é representante da classe RISC (*Reduced Instruction Set Computer*, Computador com Conjunto Reduzido de Instruções), enquanto as demais encaixam-se na categoria CISC (*Complex Instruction Set Computer,* Computador com Conjunto Complexo de Instruções).

No momento de realização dos experimentos, tais arquiteturas executavam os seguintes sistemas operacionais:

- Modelos de 1 a 7: *Microsoft Windows* XP *Professional* com *Service Pack* 2 (SP2);
- Arquitetura de número 8: *Mac* OS X 10.4.10;
- Configuração número 9: *Microsoft Windows Small Business Server* 2003.

O número de amostras utilizado nos testes foi 100. Isso significa que, a cada alteração em algum parâmetro, a cena foi reconstruída e exibida cem vezes. Obtém-se o resultado final somando o tempo gasto em cada execução e dividindo pelo número de testes. Esse valor foi escolhido por representar um tamanho de amostra significativo, o que implica em um resultado de maior confiabilidade. Tamanhos de amostra pequenos podem não ser capazes de eliminar vieses relacionados às atividades do sistema operacional, as quais não podem ser controladas.

Outro item importante a ser observado é que todas as medições de tempo foram realizadas através de novas execuções do programa. Isso porque, o VTK é otimizado de maneira a reconstruir o volume por completo somente na primeira renderização, nas demais, apenas são aplicadas modificações em cima do volume original. Dessa forma, para obter os valores corretos de tempo, é necessário re-executar a ferramenta em cada momento.

Em todos os experimentos, os tempos foram contabilizados reconstruindo os exames de SPECT e RM de um mesmo paciente diagnosticado como portador de epilepsia. Isso porque, diferentes pacientes divergem nas estruturas físicas e

químicas, o que altera os valores dos *voxels* na imagem e, portanto, influencia diretamente o custo computacional da renderização.

Tanto as imagens de RM como as de SPECT são compostas por 160 x 256 x 154 fatias com espaçamento uniforme de 1 ponto entre elas. Os volumes em ambos os exames possuem 6.307.840 *voxels*. Os valores escalares dos *voxels* no volume de RM variam entre 0 e 255, a média entre eles é 41.09 com desvio padrão de 41.25. No segundo volume (SPECT), o intervalo de escalares varia entre 0 e 32, sendo a média deles 0.12 e o desvio padrão 1.51.

Recursos próprios à biblioteca VTK foram utilizados para contabilizar os tempos de reconstrução. A classe “vtkVolumeRayCastMapper” possui o método “GetTimeToDraw()” que, ao ser invocado, retorna um valor do tipo *double* representando o tempo em segundos gasto no percurso do raio. Já a classe “vtkRenderer” implementa o método “GetLastRenderTimeInSeconds()” que, de maneira análoga ao anterior, também retorna um valor em segundos. Nesse caso, o retorno é o tempo necessário para a renderização da cena completa contida na janela.

Neste projeto, em particular, os dois métodos geram valores bastante semelhantes. Isso porque, os únicos objetos contidos em uma cena são os volumes. Logo, o tempo total de renderização da cena é o gasto na reconstrução do volume acrescido do necessário para operação de saída da imagem para a tela.

Entretanto, apenas esses dois métodos não são suficientes para a realização dos testes, pois não permitem contabilizar, por exemplo, o tempo total consumido por um dado trecho de código que efetue leitura de arquivos e alocação de memória. No intuito de preencher essa lacuna, foi utilizada uma classe específica para medição de tempo, “vtkTimerLog”. Com ela é possível criar, de maneira simples e intuitiva, objetos que funcionam como cronômetros: basta acionar “StartTimer()” para iniciar a contagem e “StopTimer()” para pará-la. O intervalo percorrido é recuperado invocando-se o método “GetElapsedTime()”.

6.2 Testes relacionados à estratégia de implementação

O primeiro conjunto de experimentos é composto por aqueles que visam medir e comparar os tempos de execução da ferramenta de acordo com as diferentes abordagens que podem ser adotadas na implementação de seus

pipelines. Foram realizados três testes nessa categoria, a seguir são detalhadas suas especificações e os resultados dos mesmos.

6.2.1 Teste dos *pipelines* isolados

6.2.1.1 Especificação

O primeiro teste realizado consiste na medição do custo computacional para reconstrução de cada cena, isoladamente, com seus parâmetros *default*. Para isso, cada uma das três janelas foi renderizada n vezes, sem a presença das demais e utilizando como parâmetros de cor, interpolação etc., aqueles valores que lhe são atribuídos por padrão no *software* desenvolvido.

6.2.1.2 Objetivos

O objetivo deste experimento é determinar o custo computacional dos *pipelines* de visualização, para, com isso, verificar a influência do número de *threads* nos mesmos, quantificar o tempo de processador consumido pelo *Ray-casting* e o gasto com as operações de E/S em cada janela. Por fim, deseja-se comparar os custos dos *pipelines* das cenas para determinar qual desses consome maior tempo de processador.

6.2.1.3 Resultados

As tabelas 1, 2 e 3 indicam, respectivamente, os resultados do teste para a janela de exibição da fisionomia, exames alinhados e foco epileptogênico, os quais consideram apenas o tempo de *Ray-casting*. As colunas A1, A2, etc., fazem referência às configurações de arquiteturas definidas no quadro da Fig. 33.

O tempo total gasto no percurso do *pipeline* de exibição da fisionomia, exames alinhados e foco epileptogênico, incluindo desde a leitura dos exames até a execução do *Ray-casting* e exibição na tela, é colocado nas tabelas 4, 5 e 6, respectivamente. As colunas A1, A2 etc., referenciam as configurações de arquiteturas definidas na Fig. 33.

Tabela 1 – Tempo médio gasto na renderização da janela “Fisionomia” em arquiteturas variadas

Nº. de *Threads*	Tempo total de renderização em segundos								
	A1	A2	A3	A4	A5	A6	A7	A8	A9
1	2.199	2.416	1.633	1.933	0.923	1.162	1.429	6.514	1.049
2	2.218	2.439	1.557	1.321	0.761	0.952	1.113	6.552	0.860
4	2.223	2.452	1.550	1.311	0.773	0.967	1.138	6.584	0.761
8	2.229	2.452	1.542	1.306	0.778	0.970	1.136	6.573	0.753

Tabela 2 – Tempo médio gasto na renderização da janela “Exames alinhados” em arquiteturas variadas

Nº. de *Threads*	Tempo total de renderização em segundos								
	A1	A2	A3	A4	A5	A6	A7	A8	A9
1	2.127	2.325	1.818	1.818	0.902	1.330	1.869	7.778	2.265
2	2.139	2.351	1.657	1.234	0.599	0.846	1.206	7.827	1.163
4	2.135	2.364	1.659	1.214	0.624	0.880	1.236	7.848	0.831
8	2.138	2.373	1.661	1.221	0.616	0.875	1.221	7.866	0.828

Tabela 3 – Tempo médio gasto na renderização da janela “Foco Epileptogênico” em arquiteturas variadas

Nº. de *Threads*	Tempo total de renderização em segundos								
	A1	A2	A3	A4	A5	A6	A7	A8	A9
1	1.506	1.638	1.207	1.128	0.599	0.857	1.195	5.456	0.859
2	1.512	1.651	1.174	0.880	0.419	0.604	0.856	5.465	0.687
4	1.516	1.654	1.194	0.878	0.433	0.621	0.870	5.478	0.659
8	1.517	1.663	1.203	0.884	0.428	0.621	0.857	5.484	0.592

Tabela 4 – Tempo médio gasto na renderização da janela “Fisionomia” em arquiteturas variadas, incluindo operações de E/S

Nº. de *Threads*	Tempo total de renderização em segundos								
	A1	A2	A3	A4	A5	A6	A7	A8	A9
1	2.518	2.815	2.001	2.306	1.208	1.447	1.787	7.191	1.464
2	2.534	2.839	1.916	1.691	1.034	1.231	1.467	7.262	1.272
4	2.536	2.851	1.911	1.681	1.044	1.244	1.486	7.308	1.175
8	2.540	2.852	1.897	1.676	1.047	1.241	1.480	7.296	1.172

Tabela 5 – Tempo médio gasto na renderização da janela “Exames alinhados” em arquiteturas variadas, incluindo operações de E/S

Nº. de *Threads*	Tempo total de renderização em segundos								
	A1	A2	A3	A4	A5	A6	A7	A8	A9
1	2.702	3.114	2.523	2.558	1.455	1.865	2.527	9.179	2.127
2	2.719	3.140	2.372	1.971	1.109	1.377	1.865	9.236	2.056
4	2.712	3.154	2.360	1.953	1.133	1.416	1.892	9.258	1.642
8	2.714	3.165	2.361	1.962	1.125	1.410	1.882	9.271	1.642

Tabela 6 – Tempo médio gasto na renderização da janela “Foco Epileptogênico” em arquiteturas variadas, incluindo operações de E/S

Nº. de *Threads*	Tempo total de renderização em segundos								
	A1	A2	A3	A4	A5	A6	A7	A8	A9
1	1.797	2.035	1.574	1.500	0.866	1.131	1.544	6.172	1.273
2	1.809	2.053	1.539	1.254	0.684	0.873	1.204	6.183	1.095
4	1.811	2.060	1.555	1.249	0.699	0.892	1.218	6.198	1.295
8	1.810	2.065	1.556	1.256	0.693	0.893	1.204	6.219	1.016

6.2.1.4 Conclusões

A análise dos resultados mostra que em computadores com um único núcleo de processamento o aumento no número de *threads* fez crescer o tempo de processador necessário para a execução. Esse é um resultado esperado, pois a

divisão em fluxos independentes introduz *overhead* para o controle dos mesmos. Como não há processadores ou *cores* extras para o escalonamento dos múltiplos *threads*, esses não executam em paralelo. Logo, a reconstrução continua a executar seqüencialmente, porém, com o custo extra para controle dos fluxos.

Observando ainda os resultados da execução nas arquiteturas *single-core* (modelos 1 e 2), verifica-se que a presença de placa de vídeo com aceleração 3D não altera o desempenho. Essa constatação é também esperada, pois, conforme visto na Seção 2.2.2, os algoritmos de *Ray-tracing* não utilizam *hardware* especial para processamento gráfico e a implementação realizada privilegiou o desenvolvimento de código portável ao invés de explorar características específicas de *hardware* disponíveis.

Já nas arquiteturas *multi-core* e na configuração com suporte a HT, observa-se ganho de desempenho ao comparar a execução seqüencial (1 *thread*) com a *multithread*. Em geral, as arquiteturas *dual-core* apresentaram o melhor resultado na reconstrução com dois *threads*, o que é natural considerando-se o argumento de que, de fato, somente 2 *threads* são escalonados para execução em simultâneo. Na arquitetura modelo 5, por exemplo, a reconstrução da janela que exibe a fisionomia do paciente utilizando 2 *threads* apresentou em relação à seqüencial tempo de processamento 32% menor. Isso significa que se obteve *speedup* de 1.463 e eficiência de 74%[1].

Outro dado apresentado neste teste é que a janela que consome maior tempo de processamento dentre as três é responsável por exibir a fisionomia do paciente. Conforme visto na Seção 6.1, o valor médio entre os escalares do volume de RM testado é consideravelmente maior do que o calculado para o exame de SPECT. A média no caso do SPECT aponta grande concentração de *voxels* em torno do valor zero. Na RM, pelo contrário, os valores encontram-se distribuídos entre as faixas mapeadas pelas funções de transferência como regiões de interesse. Dado que o tempo necessário para a reconstrução volumétrica é diretamente proporcional ao número de pontos opacos do volume, é natural que a fisionomia demande maior tempo de processamento que o foco epileptogênico. Já a janela que

[1] *Speedup*: ganho de desempenho, relação entre o tempo de processamento seqüencial e o tempo de processamento paralelo. Eficiência: utilização efetiva dos recursos de processamento disponíveis, relação entre o *speedup* obtido e o número de processadores disponíveis.

exibe os exames alinhados, embora composta pelos dois volumes, também possui custo computacional menor. Isso porque, essa reconstrói a RM de maneira transparente, o que implica em um número menor de *voxels* mapeados nas funções de transferência. Além disso, mesmo os mapeados apresentam opacidade menor do que os considerados para reconstrução da janela de fisionomia, explicando assim o porquê de, apesar de reconstruir dois volumes, essa cena consumir menos tempo de processador que a primeira.

Com relação ao tempo necessário para as operações de E/S, observa-se que essas consomem em torno de 0.3 segundos. Considerando-se esse tempo, a janela com maior custo computacional passa a ser a que exibe os exames alinhados. Isso porque essa tem de ler dois arquivos, enquanto as demais lêem apenas um. No entanto, esse resultado não reflete a realidade da ferramenta desenvolvida uma vez que ela faz a leitura das imagens desvinculada das cenas, ou seja, lê cada arquivo apenas uma vez. Já para a contabilização do tempo de E/S neste teste, cada janela efetuou a leitura do seu respectivo arquivo.

Por fim, verificou-se que processadores *PowerPC* G4 rodando *Mac* OS X (configuração 8) são, dentre os experimentados, os menos indicados para a execução da ferramenta desenvolvida. A arquitetura que apresentou melhores resultados de forma geral foi a número 5 e, por isso, será a utilizada para realização dos testes da influência dos parâmetros.

6.2.2 Teste de renderização das janelas seqüencialmente

6.2.2.1 Especificação

A segunda avaliação realizada consiste em medir o tempo total necessário para reconstruir as três janelas da ferramenta com implementação que utilize vários *threads* em cada uma delas, porém, as reconstrua em seqüência, uma após a outra. A contabilização inclui o tempo gasto com as operações de E/S, tais como leitura dos arquivos, alocação de memória, exibição na tela etc. Isso porque, embora algumas dessas operações não sejam constantemente realizadas durante a operação da aplicação (como a leitura dos arquivos, por exemplo) todas elas são obrigatoriamente executadas na inicialização do *software*.

Os volumes de fisionomia, exames alinhados e foco epileptogênico foram reconstruídos, nessa ordem, cem vezes. Utilizaram-se como parâmetros de cor, interpolação etc., os valores *default* de cada cena.

6.2.2.2 Objetivos

Este teste objetiva determinar o tempo que, efetivamente, o usuário teria de aguardar ao solicitar o carregamento dos exames para visualização na ferramenta caso esta seja implementada da maneira acima especificada.

6.2.2.3 Resultados

A tabela 7 indica os resultados do teste em diferentes arquiteturas, mantendo os parâmetros do *Ray-casting* constantes e variando o número de *threads* aplicados na reconstrução de cada janela. As colunas J1, J2 e J3 representam o número de *threads* utilizados na renderização das janelas de fisionomia, exames alinhados e foco epileptogênico, respectivamente. As colunas A1, A2, etc., se referem às configurações de arquiteturas definidas no quadro da Fig. 33.

Tabela 7 – Tempo médio gasto na renderização das três janelas em seqüência em arquiteturas variadas, incluindo operações de E/S

Threads			Tempo total de renderização em segundos								
J1	J2	J3	A1	A2	A3	A4	A5	A6	A7	A8	A9
1	1	1	6.557	7.182	5.475	5.308	3.037	3.919	5.237	21.201	4.211
1	1	2	6.550	7.196	5.447	5.052	2.861	3.662	4.897	21.298	4.058
1	2	1	6.528	7.220	5.316	4.721	2.689	3.426	4.600	21.363	3.811
1	2	2	6.534	7.224	5.284	4.467	2.510	3.175	4.248	21.336	3.631
2	1	1	6.524	7.231	5.378	5.030	2.891	3.709	4.911	21.355	4.031
2	1	2	6.542	7.239	5.356	4.785	2.707	3.457	4.592	21.345	3.864
2	2	1	6.554	7.278	5.232	4.452	2.536	3.222	4.287	21.357	4.194
2	2	2	6.553	7.310	5.191	4.205	2.361	2.980	3.965	21.003	3.439
4	4	4	6.580	7.314	5.206	4.187	2.413	3.052	4.038	21.061	3.022
8	8	8	6.586	7.439	5.200	4.203	2.408	3.050	4.008	21.226	3.003

6.2.2.4 Conclusões

Nas arquiteturas *multi-core*, em geral o resultado mais satisfatório foi obtido ao reconstruir as três janelas com 2 *threads* cada. Apenas as arquiteturas 4 e 9 não seguiram o padrão. No caso da configuração 9, a melhor abordagem foi aquela em que aplicou-se 8 *threads* na reconstrução de cada janela e houve uma redução aproximada de 29% no tempo de processamento com *speedup* de 1.40 e eficiência de 35%. Esse resultado, em conjunto com os obtidos no teste anterior, aponta no sentido de que se obtém uma exploração mais adequada de arquiteturas desse porte utilizando-se um número maior de fluxos de execução.

Já a configuração 4 obteve resultados melhores com o uso de 4 *threads* por janela. A redução aproximada do tempo de processamento nesse caso foi de 21% e o *speedup* de 1.27. Entretanto, observa-se que os ganhos com o uso de 2 *threads* por janela foram bastante semelhantes. A arquitetura 8 manteve-se com o desempenho menos satisfatório do grupo. Ao comparar o menor tempo de processador gasto por uma arquitetura *dual-core* (configuração 5 com 2 *threads* por cena) com o tempo médio necessário para a reconstrução na arquitetura *single-core* A1, verifica-se redução no tempo de processamento de 64%.

Os resultados obtidos em computadores portáteis *dual-core* são também satisfatórios. Na configuração 6, a redução no tempo de processamento comparando-se a execução seqüencial com a *multithread* foi de 24% com *speedup* igual a 1.31 e eficiência de 65%. A arquitetura 7, por sua vez, apresentou redução de 25% no tempo de processamento, *speedup* de 1.32 e eficiência igual a 66%.

6.2.3 Teste de renderização das janelas paralelamente

6.2.3.1 Especificação

Este teste é complementar e bastante semelhante ao apresentado na seção anterior, a diferença entre eles é a maneira como a construção das janelas é abordada. No anterior, as cenas eram renderizadas em seqüência, enquanto neste, as três reconstruções iniciam simultaneamente. Os *pipelines* de exibição das cenas são ativados em paralelo com seus parâmetros *default* e, durante o experimento,

variou-se o número de *threads* em uma ou mais janelas. Da mesma forma que no experimento anterior, considerou-se o tempo relativo às operações de E/S.

6.2.3.2 Objetivos

O objetivo neste caso foi determinar o tempo que efetivamente o usuário teria de aguardar ao realizar a abertura dos exames para visualização na ferramenta com esta fazendo a reconstrução simultânea das janelas. Assim, pode-se determinar qual das duas abordagens é a mais satisfatória, esta ou a colocada no teste anterior.

6.2.3.3 Resultados

A tabela 8 indica os resultados do teste em diferentes arquiteturas. As colunas J1, J2 e J3 representam o número de *threads* utilizados na reconstrução das janelas de fisionomia, exames alinhados e foco epileptogênico, respectivamente. As colunas A1, A2, etc., referem-se às configurações de arquiteturas definidas no quadro da Fig. 33.

Tabela 8 – Tempo médio gasto na renderização das três janelas em paralelo em arquiteturas variadas, incluindo operações de E/S

Threads			Tempo total de renderização em segundos								
J1	J2	J3	A1	A2	A3	A4	A5	A6	A7	A8	A9
1	1	1	6.902	7.696	5.147	4.177	2.484	3.042	4.250	22.021	3.228
1	1	2	6.918	7.712	5.206	4.212	2.633	3.055	4.219	22.348	3.269
1	2	1	6.918	7.732	5.110	4.240	2.601	3.066	3.844	22.403	3.530
1	2	2	6.945	7.742	5.162	4.266	2.429	3.072	3.843	22.376	3.059
2	1	1	6.925	7.783	5.172	4.280	3.297	3.110	4.266	22.355	3.219
2	1	2	6.944	7.752	5.228	4.344	3.187	3.140	4.468	22.312	3.255
2	2	1	6.936	7.774	5.103	4.093	3.023	2.962	3.875	22.319	2.995
2	2	2	6.951	7.776	5.116	4.129	3.148	2.964	3.844	23.109	3.017
4	4	4	6.964	7.801	5.150	4.152	3.070	2.976	4.235	23.154	2.989
8	8	8	6.979	7.831	5.159	4.188	3.062	2.998	4.296	23.218	2.956

6.2.3.4 Conclusões

Neste teste, mantiveram-se as tendências detectadas no anterior. A arquitetura 8 continuou com o desempenho mais baixo e o modelo 9 novamente apresentou seu melhor resultado ao utilizar 8 *threads* para a reconstrução de cada cena. No entanto, a eficiência da arquitetura 9 aumentou de 35% no experimento anterior para 55% neste, o que fortalece o argumento de que quanto maior o número de fluxos de execução, melhor a exploração dessa arquitetura.

O desempenho dos processadores *single-core* foi pior se comparado ao obtido com a abordagem anterior. Esse resultado é previsível, dado que são iniciados diversos fluxos de execução para escalonamento em apenas 1 núcleo. Já a arquitetura com suporte a HT e as *multi-core* apresentaram resultados melhores com a abordagem deste experimento. Dado que o desenvolvimento deste projeto é voltado para arquiteturas *multi-core* e, além disso, a variação de desempenho nos computadores *single-core* foi baixa, tem-se que a estratégia de renderizar as três janelas simultaneamente em paralelo é a mais indicada.

6.3 Testes relacionados aos parâmetros

Até o presente momento, as avaliações realizadas consideraram apenas os valores de entrada definidos como padrão nesta ferramenta. Porém, a escolha desses como *default* não se deu ao acaso, mas, justamente, porque são aqueles identificados como os mais adequados na relação entre desempenho e qualidade no escopo da aplicação desenvolvida. Os testes a seguir visam verificar a influência de diferentes conjuntos de parâmetros no resultado visual final (volumes reconstruídos) e no custo computacional.

Dado que a arquitetura “Modelo 5” descrita na Fig. 33 foi a que, de modo geral, apresentou melhor desempenho dentre os computadores pessoais, os experimentos a seguir foram realizados nessa arquitetura. O volume utilizado para os experimentos a seguir foi o de RM correspondente à fisionomia, conforme colocado na Seção 6.2.1.4.

6.3.1 Testes de interpolação e função de *Ray-casting*

6.3.1.1 Especificação

No Capítulo 5 foram expostas as particularidades de cada interpolação e função de raio. Este teste destina-se a comprovar tais peculiaridades, verificando as diferenças visuais geradas de acordo com a interpolação e função de raio aplicada, além de medir o tempo gasto para a reconstrução em cada caso. As contagens de tempo foram realizadas de duas formas: incluindo o tempo relativo às operações de E/S e também sem considerá-lo.

6.3.1.2 Objetivos

O objetivo deste teste é comparar a qualidade das imagens geradas com cada tipo de raio e interpolação, buscando assim definir quais apresentam melhores resultados na relação custo computacional X qualidade. Assim, o objetivo final desse experimento é validar a escolha dos parâmetros adotados como padrão no desenvolvimento desta ferramenta.

6.3.1.3 Resultados

A Fig. 34 ilustra a imagem de fisionomia reconstruída utilizando os parâmetros padrão da ferramenta com interpolação linear e constante, respectivamente. Observa-se que o volume gerado com interpolação constante apresenta “pontos claros” que sugerem ruído na imagem, diminuindo assim a qualidade da mesma.

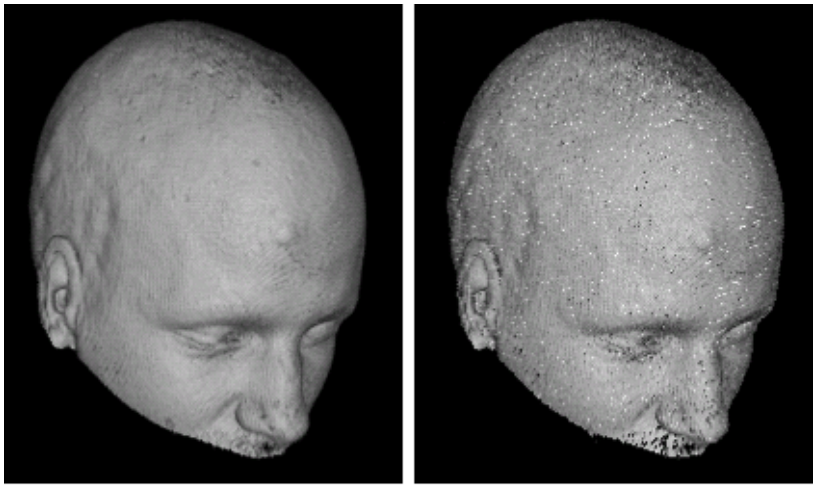


Figura 34 – Comparação entre o volume de fisionomia reconstruído utilizando em a) interpolação linear; b) interpolação constante.

Os volumes resultantes da reconstrução volumétrica através das diferentes funções de raio oferecidas pela aplicação são ilustrados na Fig. 35. Em todas as renderizações a interpolação utilizada foi a linear, devido à sua qualidade superior. Entretanto, na geração dessa figura em específico, as funções para transferência de escalares em cor e opacidade aplicadas não foram as padrões da ferramenta. Isso porque, o conjunto de parâmetros que gera bons resultados visuais para funções MIP não apresenta necessariamente a mesma qualidade em funções *Composite*. Assim, foi escolhido um grupo de valores que apresentasse resultados próximos do padrão nas quatro situações.

Na tabela 9 são apresentados os tempos necessários para renderização da janela de fisionomia utilizando, respectivamente, as funções de *Ray-casting Composite Interpolate First*, *Composite Classify First*, MIP *Maximize Method to Scalar Value* e MIP *Maximize Method to Opacity*. São colocados os resultados sem considerar as operações de E/S e também considerando-as. Para a medição do tempo, manteve-se os parâmetros *default* do volume e variou-se o número de *threads* e a interpolação utilizados na reconstrução.

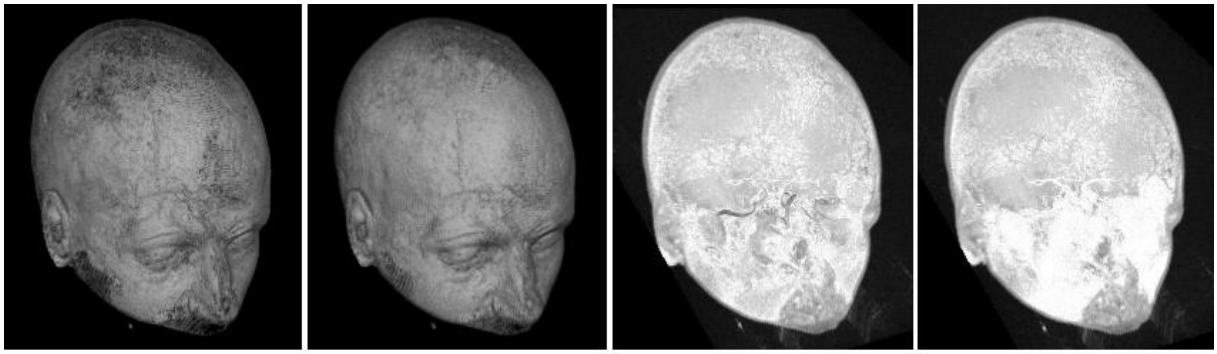


Figura 35 – Volumes de fisionomia do paciente reconstruídos utilizando em a) Função *Composite Interpolate First*; b) Função *Composite Classify First;* c) Função MIP *Maximize Method to Scalar Value;* d) Função MIP *Maximize Method to Opacity.*

Tabela 9 – Tempo médio para renderização do volume de fisionomia com diferentes funções de *Ray-casting* e de interpolação

		Função de *Ray-casting*							
		Tempo de *Ray-casting*				Tempo incluindo E/S			
Threads	Interpolação	I. F.	C. F.	M. S.	M. O.	I. F.	C. F.	M. S.	M. O.
1	Linear	0.950	0.912	0.912	0.990	1.217	1.181	1.184	1.257
2	Linear	0.787	0.763	0.765	0.806	1.055	1.033	1.035	1.077
4	Linear	0.801	0.777	0.777	0.808	1.068	1.045	1.048	1.075
8	Linear	0.803	0.773	0.776	0.808	1.072	1.041	1.043	1.076
1	Constante	0.731	0.724	0.683	0.764	1.002	0.992	0.950	1.031
2	Constante	0.700	0.676	0.647	0.690	0.968	0.946	0.917	0.961
4	Constante	0.686	0.683	0.650	0.692	0.951	0.953	0.919	0.962
8	Constante	0.686	0.683	0.650	0.689	0.953	0.951	0.918	0.953

6.3.1.4 Conclusões

Observa-se com este teste que a interpolação linear possui maior custo computacional quando comparada com a constante. Entretanto, analisando os resultados apresentados na Fig. 34 percebe-se que é superior a qualidade do volume renderizado aplicando interpolação linear. A imagem reconstruída com interpolação constante apresenta uma série de pontos claros ou luminosos, os quais

podem ser vistos como ruído e artefatos indesejados na imagem. Assim, apesar de o tempo de processamento para reconstrução com interpolação constante ser em torno de 25% menor, a perda de qualidade no resultado final desencoraja tal economia. Além disso, considerando-se as quatro funções de *Ray-casting* analisadas, o *speedup* médio obtido ao executar com dois *threads* foi de 1.20 para interpolação linear e de apenas 1.05 para interpolação constante. Isso sugere que a aplicação do paralelismo, por apresentar ganhos maiores na reconstrução com interpolação linear, pode vir a compensar a diferença de custo computacional entre as duas.

Em relação às funções de raio, a análise qualitativa dos volumes comprova que a função *Composite Classify First* apresenta qualidade superior em comparação com a *Composite Interpolate First*. Além disso, o tempo de processador gasto pela primeira é menor, logo, em geral sua utilização é mais indicada. Já as funções do tipo MIP apresentaram resultados bastante semelhantes entre si na aplicação desenvolvida.

6.3.2 Teste do tamanho do passo

6.3.2.1 Especificação

O tamanho do passo percorrido nas funções de *Ray-casting* afeta diretamente o resultado final, tanto em termos de qualidade como em relação ao custo computacional (vide seção 6.2.3.3). Este teste executou a função de *Ray-casting* padrão para o volume de fisionomia variando o tamanho do passo e número de *threads*, mantendo para os demais parâmetros seus valores *default*. Foram capturados os volumes gerados para cada tamanho de passo e medidos os tempos médios necessários para a reconstrução dos mesmos.

6.3.2.2 Objetivos

A finalidade deste teste é gerar volumes com diferentes tamanhos de passo para comparar, qualitativamente, os resultados visuais de cada caso e, quantitativamente, os respectivos custos computacionais. Espera-se assim validar o valor escolhido como padrão durante o desenvolvimento desta aplicação.

6.3.2.3 Resultados

A Fig. 36 ilustra os volumes gerados com tamanho de passo igual a 0.1, 0.5, 1.0, 2.0, 3.0 e 5.0. Tamanho de passo maior que 5.0 não foram considerados na análise qualitativa, pois a reconstrução obtida é excessivamente degradada.

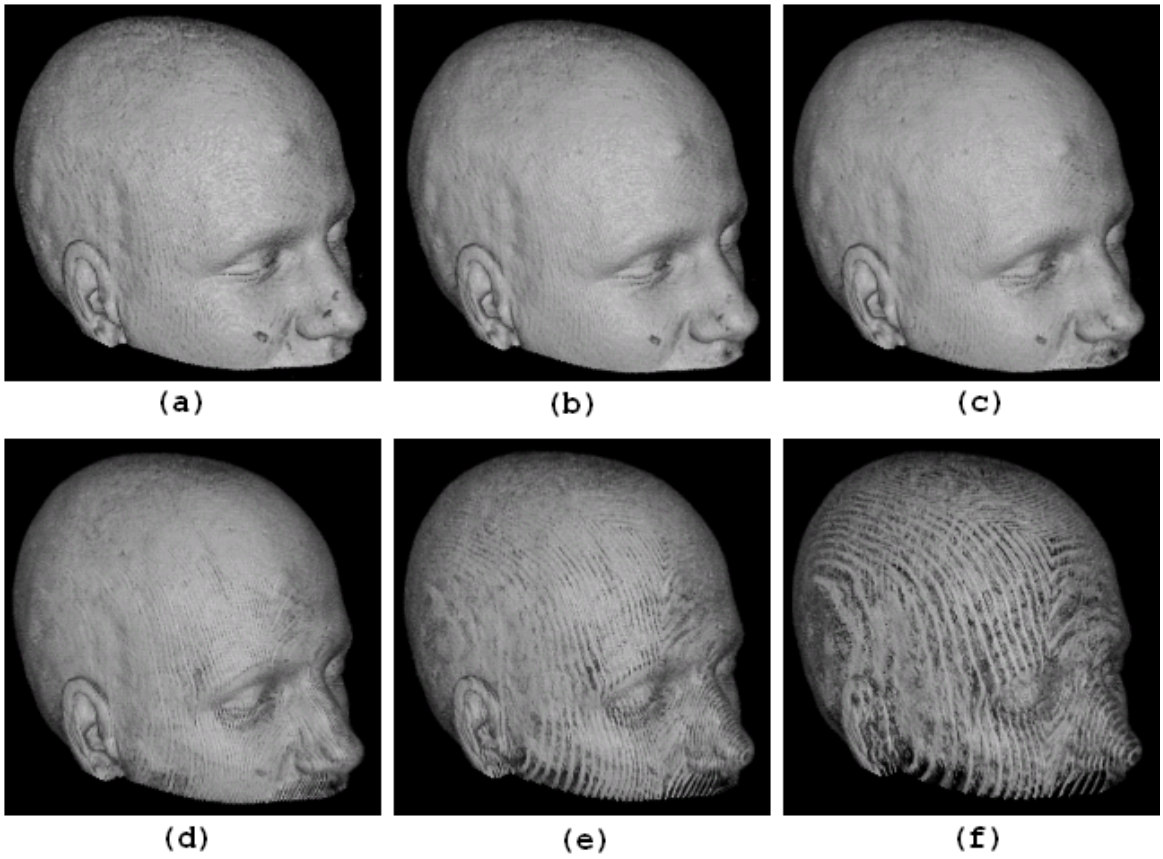


Figura 36 – Volumes de fisionomia do paciente reconstruídos com parâmetros padrão e variando o tamanho do passo. Tamanho em a) 0.1; b) 0.5; c) 1; d) 2; e) 3; f) 5.

A tabela 10 apresenta os tempos médios necessários para renderização dos volumes com diferentes tamanhos de passo na função de *Ray-casting*. Mantiveram-se os parâmetros padrão do volume e variou-se o número de *threads* utilizado na reconstrução.

Tabela 10 – Tempo médio para renderização do volume de fisionomia com diferentes funções de *Ray-casting* e tamanhos de passo variados

	Tamanho do passo na função							
	Tempo de *Ray-casting*				Tempo incluindo E/S			
Threads	0.1	0.5	1	10	0.1	0.5	1	10
1	1.762	1.047	0.923	0.681	2.031	1.303	1.208	0.944
2	1.194	0.825	0.761	0.653	1.459	1.091	1.034	0.909
4	1.222	0.838	0.773	0.647	1.485	1.100	1.044	0.913
8	1.212	0.828	0.778	0.653	1.481	1.091	1.047	0.919

6.3.2.4 Conclusões

O tamanho de passo ideal a ser aplicado para a função de *Ray-casting* da aplicação desenvolvida é 1. Tal decisão justifica-se pelo fato de, na análise qualitativa das imagens geradas, observar-se que passos menores do que 1 resultam em volumes muito semelhantes ao reconstruído com valor 1, porém, implicam em custo computacional maior, conforme observado na tabela 10.

Por outro lado, tamanhos de passo maiores resultam em volumes degradados, nos quais a qualidade é inversamente proporcional ao tamanho utilizado, enquanto o ganho de desempenho identificado não é proporcional. Observa-se que aumentando o tamanho de passo de 1 para 10, o custo computacional é reduzido em 26%. Entretanto, a diferença de qualidade entre os volumes reconstruídos com tamanho de passo 1 e 3 já é claramente percebida.

Em complemento a isso, tem-se que o espaçamento entre as fatias que compõem os exames de RM e SPECT é, por padrão, de 1 ponto. Logo, ao percorrer passos de tamanhos maiores, a função acaba por ignorar algumas das fatias, o que contribui fortemente para a perda de qualidade. Por outro lado, ao percorrer intervalos menores, será necessário aplicar com maior freqüência operações de interpolação, o que além de afetar diretamente o desempenho pode acabar por criar artefatos que não correspondem exatamente ao objeto real.

6.3.3 Testes de parâmetros avançados

6.3.3.1 Especificação

Anteriormente neste trabalho, foi colocado que implementações complexas do algoritmo *Ray-tracing* permitem a manipulação de parâmetros avançados como, por exemplo, a iluminação do volume. No capítulo anterior, verificou-se que as funções de *Ray-casting* oferecidas pelo VTK se enquadram nessa situação e foram expostas quais as classes utilizadas neste trabalho para manipular tais fatores. Assim, esse teste visa justamente verificar os diferentes resultados gerados quando da alteração de um ou mais desses parâmetros.

Durante o experimento variou-se a intensidade das fontes de luz - especular, ambiente e difusa, as cores utilizadas no volume e a aplicação da opção de *shading*. Foram então capturadas as imagens reconstruídas e medidos os tempos médios de processamento para reconstrução em cada caso.

6.3.3.2 Objetivos

Este teste tem por objetivo comparar a qualidade dos volumes gerados utilizando-se intensidades de luz e funções de transferência diversas e variando também a opção de *shading*. Como no experimento anterior, tais comparações objetivam verificar de que forma esses parâmetros afetam o resultado final e com isso quais as combinações mais indicadas para a ferramenta desenvolvida.

6.3.3.3 Resultados

A tabela 11 apresenta os tempos medidos para renderização da fisionomia utilizando, respectivamente, as funções transferência de escalares em cores “Tons de Cinza 1” e “Cores RGB 1” disponíveis na interface da ferramenta, com a opção de “shading” variando entre habilitada e desabilitada em cada um dos casos. As colunas T. R. e T. E/S indicam o tempo médio considerando, respectivamente, apenas o custo do *Ray-casting* e considerando também as operações de E/S. Os

demais parâmetros foram mantidos com seus valores padrão e variou-se o número de *threads* entre 1 e 8.

Tabela 11 – Tempo médio de renderização do volume de fisionomia considerando diferentes valores para o parâmetro "*shading*" e diferentes funções para transferência de escalares em cores

Nº. de *Threads*	Tons de Cinza 1				Cores RGB 1			
	Shading On		*Shading Off*		*Shading On*		*Shading Off*	
	T. R.	T. E/S	T. R.	T. E/S	T. R.	T. E/S	T. R.	T. E/S
1	0.918	1.185	0.898	1.168	2.075	2.346	0.902	1.170
2	0.771	1.036	0.754	1.020	1.356	1.624	0.763	1.033
4	0.781	1.051	0.763	1.030	1.384	1.652	0.772	1.042
8	0.780	1.049	0.766	1.034	1.381	1.649	0.771	1.040

Os custos computacionais relacionados à iluminação são colocados na tabela 12. As medições foram realizadas com o parâmetro *shading* desabilitado para que a influência do fator iluminação fosse efetivamente isolada. Variou-se a função para transferência de escalares em cor entre as opções "Tons de Cinza 1" e "Cores RGB 1" da GUI. Os demais parâmetros foram mantidos com seus respectivos valores padrão.

Tabela 12 – Tempo médio de renderização do volume de fisionomia considerando diferentes configurações de iluminação e diferentes funções para transferência de escalares em cores

Iluminação			Tempo médio de renderização em segundos			
			Tons de Cinza 1		Cores RGB 1	
Ambiente	Difusa	Especular	1 *thread*	2 *threads*	1 *thread*	2 *threads*
0.8	0.6	0.1	0.893	0.751	0.895	0.752
0.1	1.0	0.1	0.893	0.750	0.895	0.753
0.1	0.6	0.5	0.899	0.757	0.895	0.758
1.0	1.0	1.0	0.899	0.757	0.902	0.760

6.3.3.4 Conclusões

A comparação entre o tempo consumido para renderizar o volume utilizando cores RGB ou somente tons de cinza demonstra que apenas o fator cores isoladamente não exerce grande influência no custo computacional. Porém, quando operado em conjunto com o parâmetro de sombreamento (*shading*) a diferença no tempo de processamento é bastante grande.

A reconstrução seqüencial do volume com a função de transferência “Cores RGB 1” e *shading* desativado ocupa 43% menos tempo de processador que a mesma reconstrução com o sombreamento ativado. Já utilizando-se a função “Tons de Cinza 1”, essa diferença cai para apenas 3%. Por outro lado, o *speedup* obtido ao executar com dois *threads* a reconstrução mantendo o *shading* ligado e a função “Cores RGB 1” é de 1.54, enquanto nos demais casos é em média 1.19.

Já a variação dos parâmetros de iluminação não produziu diferenças significativas no custo computacional. Esse resultado é bastante conveniente no sentido que, durante o processo interativo de visualização, a aplicação e variação de fontes de iluminação diversas é um poderoso recurso para o auxiliar na melhor visualização dos dados.

6.4 Teste de portabilidade

Visto que a ferramenta desenvolvida foi baseada em bibliotecas de programação portáveis, outro passo natural na fase de testes é verificar se a portabilidade é efetivamente comprovada no produto final do desenvolvimento. Por isso, compilou-se o *software* desenvolvido em sistemas operacionais variados.

Tanto o VTK, como o FLTK e o *CMake* operam em MS *Windows*, *Linux* e *Mac* OS X. Já o padrão *Pthreads,* embora não seja nativo no MS *Windows,* o é nos demais e pode ser utilizado no *Windows,* bastando para isso fazer o *download* de uma implementação do padrão para esse SO. Portanto, a ferramenta foi compilada nos seguintes sistemas operacionais:

- MS *Windows* XP PRO com SP2 na máquina “modelo 1” da Fig. 33;
- *Debian Linux* na máquina “modelo 8” da Fig. 33;
- *Mac* OS X 10.4.10 na máquina “modelo 9” da Fig. 33;

A portabilidade do *software* desenvolvido foi de fato comprovada. Logrou-se êxito na compilação com as três configurações acima citadas sem ter de, para isso, alterar o código-fonte. A única modificação necessária é no arquivo CMakeLists.txt para compilação em MS *Windows*, pois deve-se indicar a biblioteca utilizada para implementação do padrão *Pthreads*. Isso é feito adicionando-se o nome da mesma à lista de valores do parâmetro “TARGET_LINK_LIBRARIES”, *pthreadVC2.lib* no caso deste projeto.

6.5 Teste comparativo com o *software* proprietário *VolView* 2.0

Para efeitos de avaliação da qualidade dos resultados obtidos com a ferramenta desenvolvida, reconstruiu-se os exames de um determinado paciente com a ferramenta desenvolvida e com o *software* proprietário *VolView* 2.0 (VOLVIEW, 2007). Esses resultados foram então comparados de maneira qualitativa.

Uma vez que o resultado da reconstrução volumétrica apresenta diferenças em decorrência das variações de cor, iluminação etc., os parâmetros utilizados para realização deste teste são aqueles definidos como *default* na ferramenta desenvolvida. No entanto, devido à implementação do *VolView* e também ao fato de os experimentos com esse *software* terem sido realizados utilizando uma licença *free*, alguns parâmetros não puderam ser configurados no mesmo.

Além disso, o *VolView,* apesar de desenvolvido pelos criadores do VTK, implementa de maneira própria as funções de *Ray-casting*. Logo, embora haja grande proximidade, não se pode dizer que a equivalência entre o conjunto de argumentos utilizados nas duas reconstruções é total. As Fig. 37 e Fig. 38 ilustram, respectivamente, os exames de Ressonância Magnética e SPECT reconstruídos com o *VolView* e com a ferramenta aqui apresentada.

Figura 37 – Exame de Ressonância Magnética do paciente reconstruído com um conjunto semelhante de parâmetros em a) no *software* proprietário *VolView* 2.0; b) na ferramenta desenvolvida neste projeto.

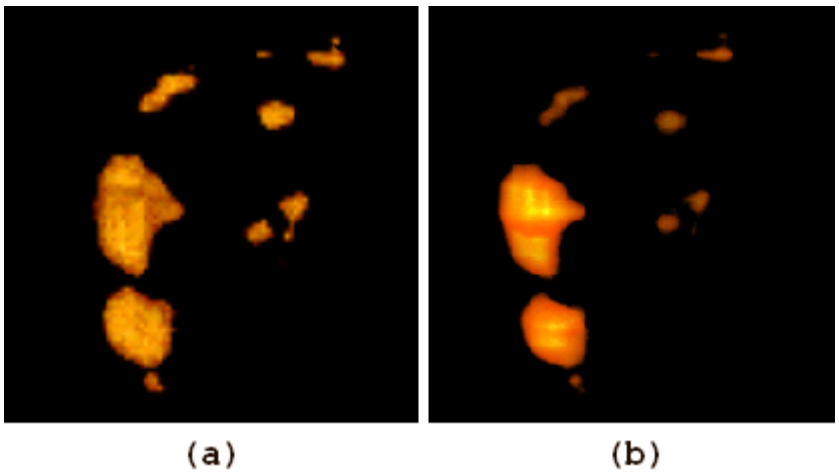


Figura 38 – Exame de SPECT do paciente reconstruído com um conjunto semelhante de parâmetros em a) no *software* proprietário *VolView* 2.0; b) na ferramenta desenvolvida neste projeto.

Os volumes reconstruídos nos dois casos são bem próximos, o que indica resultados bastante satisfatórios em ambas as situações. Percebe-se que as imagens geradas com o *VolView* aparentam superfícies levemente rugosas, enquanto os volumes gerados pela ferramenta desenvolvida possuem aparência lisa, suavizada. Essa variação é causada por divergências nas funções de *Ray-casting*. Outra diferença é que o volume de SPECT reconstruídos com a ferramenta

desenvolvida aparenta estruturas levemente menores. Isso porque não foi possível reproduzir com precisão absoluta a posição da câmera e as funções de transferência nos dois *softwares*. Por isso, cabe ao usuário variar as funções de transferência e o ângulo de visão através da GUI, para assim encontrar a solução que lhe é ideal.

Por fim, cabe destacar que a validação do alinhamento das imagens não foi necessária neste projeto. Isso porque, os métodos e algoritmos para essa operação não foram desenvolvidos neste trabalho, mas sim em Oliveira (2005), onde os resultados do co-registro foram devidamente avaliados e validados. A Fig. 39 ilustra o resultado final obtido neste projeto: a visualização na ferramenta das imagens alinhadas e reconstruídas volumetricamente.

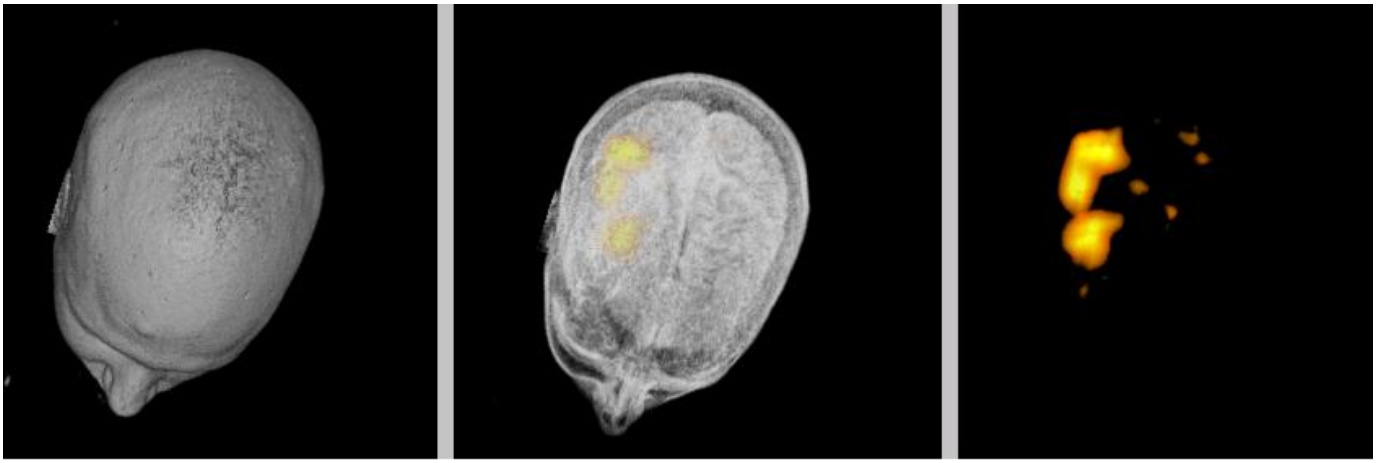


Figura 39 – Janelas reconstruídas com a ferramenta desenvolvida. Em a) a fisionomia do paciente; b) os exames alinhados; c) o foco epileptogênico.

6.6 Conclusões

Os testes de desempenho demonstraram que houve ganho com a exploração do paralelismo potencial da ferramenta. A execução se mostrou satisfatória em modelos variados de computadores pessoais portáteis e de mesa, mesmo naqueles não dotados de recursos especiais para processamento gráfico.

No quesito portabilidade, os resultados foram também satisfatórios uma vez que logrou-se êxito ao compilar/executar a ferramenta nas arquiteturas testadas. Já a comparação do produto final desenvolvido com o *software* proprietário *VolView* 2.0

apontou para o fato de que, do ponto de vista da qualidade das imagens reconstruídas, a ferramenta desenvolvida atendeu as expectativas.

7 CONCLUSÃO

Neste trabalho foi apresentada uma ferramenta para visualização 3D interativa de imagens médicas. Essa ferramenta permite a manipulação e análise de imagens complementares de diferentes exames para auxiliar na detecção e avaliação do foco epileptogênico. Uma questão tratada na implementação da ferramenta foi o custo computacional associado ao algoritmo de reconstrução das imagens. A solução adotada utilizou o processamento *multithread* em arquiteturas *multi-core*.

Dentro desse contexto, o algoritmo *Ray-casting* foi identificado como ideal para a construção da ferramenta. As imagens geradas com o mesmo são de ótima qualidade para a aplicação em questão e suas lacunas de desempenho puderam ser amenizadas com as estratégias de programação aqui adotadas.

Com a realização deste trabalho, verificou-se que o conjunto de estratégias acima descrito aplicado a imagens de Ressonância Magnética alinhadas ao SPECT crítico subtraído do intercrítico é, de fato, uma poderosa solução para o problema em questão. A ferramenta desenvolvida apresentou resultados satisfatórios quando analisada de diversos pontos de vista, tais quais portabilidade, desempenho, escalabilidade, usabilidade e, principalmente, do resultado visual gerado.

Em relação à portabilidade de *software*, a aplicação desenvolvida foi compilada com sucesso nos sistemas operacionais MS *Windows* XP, *Debian Linux* e *Mac* OS X 10.4.10. Já a portabilidade de *hardware* foi comprovada com a execução da ferramenta em arquiteturas *single-core* e *multi-core* de fabricantes diversas. Nessa amostra encontram-se processadores que, além de divergir em relação ao número de núcleos, possuem diferentes conjuntos de instruções. Foram testados ainda processadores de 32 e 64 *bits*, logrou-se êxito nos dois casos.

Os experimentos nas diferentes configurações de *hardware* apontaram no sentido de que a solução desenvolvida é melhor explorada em arquiteturas do tipo CISC. Porém, cabe colocar que a amostra de arquiteturas RISC utilizada foi pequena, o que sugere que esse resultado deve ser melhor analisado.

Do ponto de vista do desempenho e exploração do paralelismo, os resultados foram também satisfatórios. A exploração do paralelismo inerente ao algoritmo de *Ray-casting,* dividindo-o em fluxos independentes de execução, diminuiu em torno de 30% seu custo computacional nas arquiteturas *multi-core*. A taxa de *frames* por segundo colocada como adequada para aplicações interativas foi alcançada na ferramenta desenvolvida. Isso foi possível adicionando-se à exploração do paralelismo técnicas que permitem reconstruir o volume com nível menor de detalhamento durante a interação. A taxa de quadros por segundo pode ainda ser alterada em tempo de execução através da interface com o usuário.

A escalabilidade, por sua vez, é também uma característica presente na ferramenta desenvolvida. Trata-se de escalabilidade de *software*. A abordagem aplicada para obter-se tal propriedade foi a de desenvolver a aplicação de maneira que se permita manipular, dinamicamente, em tempo de execução o número de *threads*. Logo, havendo aumento do número de processadores ou *cores* na arquitetura, pode-se configurar o *software* para explorá-los eficientemente através da divisão em um número maior de fluxos de execução.

A concepção da interface considerou facilidades de interação oferecidas por ferramentas populares com o objetivo de oferecer uma interface amigável. Esse esforço é importante no sentido de que uma interface intuitiva reduz a curva de aprendizado necessário para operação do *software*. Essa redução, por sua vez, é um fator que pode vir a contribuir para a popularização da aplicação.

Em comparação com a interface do *software* proprietário *VolView*, alguns fatores sugerem que, no caso de usuários leigos em processamento de imagens, a interface da aplicação desenvolvida é mais simples de ser manipulada. Um exemplo desses fatores é a manipulação das funções de transferência. No *VolView,* a manipulação é feita definindo-se pontos no histograma da imagem, enquanto no *software* desenvolvido existe um conjunto pré-configurado de funções. Nesse caso, basta o usuário selecionar a opção desejada. Cabe, porém, uma observação importante: essa abordagem mais abstrata da manipulação das funções de transferência implica em perda de flexibilidade, o que é um efeito colateral natural, visto que as opções oferecidas restringem-se àquelas pré-definidas. A definição de parâmetros adequados para tais funções foi uma das fases que apresentaram maior dificuldade no desenvolvimento deste trabalho.

Por fim, o resultado final do projeto – a reconstrução das imagens - foi também satisfatório. Essa conclusão advém da comparação com o *software* proprietário *VolView*, cujo foco é também a visualização de dados através da reconstrução volumétrica. Além dos exames reconstruídos apresentarem qualidade bastante próxima, pôde-se oferecer na aplicação desenvolvida as principais funcionalidades do *VolView*, tais como a modificação em tempo de execução dos parâmetros da reconstrução e a visualização dos exames em forma de volumes 3D ou fatias 2D.

O objetivo de desenvolver um método de visualização complementar ao SISCOM e à fusão de imagens em 3D foi, portanto, atingido de maneira satisfatória. Com a conclusão deste projeto, encontra-se disponível uma ferramenta computacional que implementa eficientemente essa nova abordagem de visualização para auxílio detecção do foco epileptogênico.

7.1 Trabalhos futuros

Durante a realização deste trabalho, algumas funcionalidades foram identificadas como interessantes para a continuação e aperfeiçoamento do mesmo. São elas:

- Aperfeiçoar a interação do usuário para com as funções de transferência, visando aumentar a flexibilidade, porém, ao mesmo tempo, mantendo a interface amigável. Uma abordagem possível para isso é utilizar *sliders* que simulem níveis percentuais de cor e opacidade;
- Adicionar a funcionalidade de marcação de volumes de interesse (sub-volumes) para visualização isolada em tempo de execução;
- Incluir operações de *cropping*, de maneira análoga às oferecidas no *VolView*;
- Adicionar a possibilidade de o usuário aplicar janelamento na visualização das fatias 2D;
- Incluir a opção de aplicar filtros de suavização e aguçamento, na visualização das fatias 2D e também dos volumes.

REFERÊNCIAS BIBLIOGRÁFICAS

AHRENS, J.; Law, C.; Schroeder, W.; Martin, K.; Papka, M. A Parallel Approach for Efficiently Visualizing Extremely Large, Time-Varying Datasets. **Los Alamos National Laboratory Technical Report #LAUR-00-1620**. 2000.

ALVES, Matheus Borges; SOUZA, Rafael Landim de; RODRIGUES, Luciene Cavalcanti. Visualização de Imagens Médicas com Recursos em 3D. **IX Congresso Brasileiro de Informática em Saúde**, São José do Rio Preto, São Paulo 2004.

CLINIC, M. ANALYZE^{TM} 7.5 File Format, **Biomedical Imaging Resource**, Mayo Foundation. http://www.mayo.edu/bir/PDF/ANALYZE75.pdf, Acesso em 02 de jul. 2007.

ANDRAUS, M.; COSENZA, C; FONSECA, L.; ANDRAUS, C.; D'ANDREA, I; ALVES-LEON, S. SPECT Cerebral Interictal em Pacientes com Epilepsia do Lobo Temporal de Difícil Controle. **Arquivos de Neuro-Psiquiatria**, v.60 nº. 3B. 2002.

ASANOVIC, Krste; BODIK, Ras; CATANZARO, Bryan C.; GEBIS, Joseph J.; HUSBANDS, Parry; KEUTZER, Kurt; PATTERSON, David A.; PLISHKER, William L.; SHALF, John; WILLIAMS, Samuel W.; YELICK, Katherine A. The Landscape of Parallel Computing Research: A View from Berkeley. **Electrical Engineering and Computer Sciences Department, University of California, Berkeley. UCB/EECS-2006**. p183. 2006.

Beowulf Project Overview.
Disponível em: <http://www.beowulf.org/overview/> Acesso em 07 de jun. 2007.

CAVALHEIRO, Gerson G. H.; SANTOS, Rafael R. dos; **Atualizações em Informática 2007**. Capítulo de Livro. Capítulo 8 - Multiprogramação leve em arquiteturas multi-core. 1ª ed. SBC. Porto Alegre, 2007.

CMake documentation.
Disponível em <http://www.cmake.org/HTML/Documentation.html> Acesso em 03 de jun. 2007.

ELVINS, T. T. A. Survey of Algorithms for Volume Visualization. **Computer Graphics**, 1992.

FALCÃO, Alexandre X. **Visualização de Volumes Aplicada à Área Médica**. 1999. Dissertação (Mestrado em Engenharia Elétrica) - Instituto de Computação, Universidade Estadual de Campinas, Campinas.

Fast Light Toolkit (FLTK).
Disponível em <http://www.fltk.org/> Acesso em 02 mai. 2007.

FILHO, G. B. A anatomia estrutural e funcional da esquizofrenia: achados de neuropatologia e neuroimagem, **Revista Brasileira de Psiquiatria**, São Paulo, v.22, mai. 2000.

FLYNN, M. J. Some Computer Organizations and their Effectiveness. **IEEE Transactions on Computers**, vol C-21, p. 948-960, Setembro, 1972.

GOMES, Vinícius; ROSA, Leandro Zanetti Paiva da; FONSECA, Carolina Marques. **vtkAnalyzeReader**. Disponível em <http://www.ufpel.edu.br/~vgomes.ifm>. Acesso em 06 jan. 2007.

GRAMA, Ananth; GUPTA, Anshul; KARYPIS, George; KUMAR, Vipin; **Introduction to Parallel Computing**. 2ª ed. 856p. Addison Wesley. 2003. ISBN: 0-201-64865-2

HSIN MA, Josué T. **Multicore**. Instituto de Computação, Universidade de Campinas, Campinas. 2005.

HWANG, Kai. **Advanced Computer Architeture: Parallelism, Scalability, Programmability**. 1993. 771p. McGraw-Hill, Inc.

Intel Corporation. Tecnologia Hyper-Threading. Disponível em <http://www.intel.com/portugues/products/ht/hyperthreading_more.htm> Acesso em 11 jun. 2007.

JOHNSON, Eric E. Completing an MIMD multiprocessor taxonomy. **ACM Press**, v.16, n.3, p.44-47, 1988.

JÚNIOR, David Araújo. **Contribuição das medidas volumétricas das estruturas temporais mesiais e neocorticais ao tratamento cirúrgico da epilepsia do lobo temporal.** 2003. 114f. Tese (Doutorado em Neurologia) - Faculdade de Medicina de Ribeirão Preto, Universidade de São Paulo, Ribeirão Preto.

JUNIOR, Pedro P. de M. O. **Exames Virtuais Utilizando um Algoritmo de Ray Casting Acelerado**. 1999. 75f. Dissertação (Mestrado em Ciências em Informática) - Departamento de Informática, PUC-Rio, Rio de Janeiro.

KUZNIECKY, R.; JACKSON, G. Neuroimaging in epilepsy Magnetic Resonance in Epilepsy. **Raven Press,** New York. p.27-48. 1995.

LAVORANTE, Luca Pallozi. **Técnicas de Modelagem 3D aplicada a dados paleobatimétricos das bacias de Santos e Campos e à simulação deformacional de objetos geológicos.** 2005. 220f. Dissertação (Mestrado) - Instituto de Geociências e Ciências Exatas, Universidade Estadual Paulista, Rio Claro.

LEWIS, Bil; BERG, Daniel J. PThreads Primer. **A Guide to Multithreaded Programming.** SunSoft Press, A Prentice Hall Title. 370p. 1996. ISBN 0-13-443698-9.

LIN, W., Lau, R. W. H., Hwang, K., Lin, X., Cheung, P. Y. S. Adaptive Parallel Rendering on Multiprocessors and Workstation Clusters. **IEEE Transactions on Parallel and Distributed Systems,** v.12, n.3, p.241-258, 2001.

LOPES, J. M. B. Computação Gráfica. **Raytracing**. Capítulo de Livro. 2004.

LORENSEN, W. E.; CLINE, H. E., 'Marching Cubes: A High Resolution 3D Surface Construction Algorithm', **Computer Graphics**, p.163-170, 1987.

MAES, F.; Collignon, A.; Vandermeulen D. Multimodality Image Registration by Maximization of Mutual Information. **IEEE Transactions on Medical Imaging,** 16(2):187-198, 1997.

MCCORNICK, B. D.; Brown, M. Visualization in Scientific Computing. **Computer Graphics**, 1987.

MOREIRA, Daniel F. **Jogo patológico: análise por neuroimagem, neuropsicológica e de personalidade**. 2004. 143f. Tese (Doutorado em Ciências) - Faculdade de Medicina, Universidade de São Paulo, São Paulo.

MORSE, H. Stephen. **Pratical Parallel Computing.** Academic Press, Inc. 1994. 401p.

MVC. Disponível em < http://pt.wikipedia.org/wiki/MVC> Acesso em 01 ago. 2007.

O'BRIEN, T., So, E., Mullan, B., Hause, M., Brinkmann, B., Bohnen, N., Hanson, D., Cascino, G., Jack, C. J., Sharbrough, F. Subtraction Ictal SPECT co-registered to MRI improves clinical usefulness of SPECT in localizing the surgical seizure focus. **Neurology**, v.50, n.2, p.445-454, 1998.

OLIVEIRA, A.; HILÁRIO, L.; ANSELMI, O. SPECT Cerebral: Princípios e Aspectos Técnicos. **Fundamentos Neurobiológicos das Epilepsias: Aspectos Clínicos e Cirúrgicos**, São Paulo, v.1, p.659-671, 1998.

OLIVEIRA, Lucas F. **Fusão de Imagens 3D para Localização e Quantificação de Zona Epileptogênica**. 2005. 151f. Tese (Doutorado em Clínica Médica) – Faculdade de Medicina de Ribeirão Preto, Universidade de São Paulo, Ribeirão Preto.

OLIVEIRA, Marcelo C. **Visualização de Volumes em Morfometria de Estruturas Cerebrais a partir de Imagens de Ressonância Magnética Nuclear**. 2002. 263f. Dissertação (Mestrado em Física Aplicada à Medicina e Biologia) - Faculdade de Filosofia, Ciências e Letras, Universidade de São Paulo, Ribeirão Preto.

OpenMP API. **OpenMP Architecture Review Board.** Version 2.5 May 2005.

OPENMP Specifications.
Disponível em <http://www.openmp.org/drupal/node/view/8> Acesso em 06 de fev. 2007.

PAIVA, A., SEIXAS, R. e GATTAS, M. **Introdução à Visualização Volumétrica**. 1999. 107f. Trabalho acadêmico (Graduação em Ciência da Computação) - Departamento de Ciência da Computação, PUC-Rio, Rio de Janeiro.

PASSOS, Lorena B. C. **Avaliação de desempenho de método para a resolução da evolução temporal de sistemas autogravitantes em dois paradigmas de programação paralela: troca de mensagens e memória compartilhada**. 2006. Dissertação (Mestrado em Ciência da Computação) – Departamento de Ciência da Computação, Universidade de Brasília, Brasília.

POSIX Threads (pthreads) for Win32.
Disponível em <http://sourceware.org/pthreads-win32/> Acesso em 20 de abr. 2007.

QUINN, M. J. **Parallel Computing: Theory and Practice**. 2ª ed. Nova Iorque. McGraw-Hill, 1994.

ROBB, R. A. Visualization in biomedical computing. **Parallel Computing** v.25 p.2067-2110, 1999.

SANTOS, Eduardo T. **Avaliação do algoritmo de Ray Tracing em Multicomputadores**. 1994. 181p. Dissertação (Mestrado) – Escola Politécnica da Universidade de São Paulo, Departamento de Engenharia de Computação e Sistemas II.

SCHROEDER, Will; MARTIN, Ken; LORENSEN, Bill. **The Visualization Toolkit - An Object Oriented Approach to 3D Graphics**. 3ª ed. Kitware Inc, 2002.

SILVA, Fernando W. S. V. da. Introdução ao Ray Tracing. **CPU-Informática**, ano 2, no. 3, 4 and 5, Bônus Editora, 1994.

SKILLICORN, David B. Fundations of Parallel Programming. **Cambridge University Press**. 197f. Jan, 1994.

SKILLICORN, David B.; TALIA, Domenico. Models and Languages for Parallel Computation. **ACM Computing Surveys**, New York, v.30, n.2, p.123-169, junho 1998.

STALLINGS, W. **Arquitetura e organização de computadores: projeto para o desempenho.** 5 ed. São Paulo. Prentice Hall, 2002.

TANENBAUM, A. S. **Modern Operating Systems**. 2ª ed. Upper Saddle River, N.J.: Prentice Hall, 2001.

TORELLI, Julio C. **Implementação paralela da transformada de distância euclidiana exata**. 2005. 99f. Dissertação (Mestrado em Ciência da Computação e Matemática Computacional) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos.

VALENTE, A. C. Jr. **Investigação de anormalidades volumétricas regionais cerebrais em pacientes com transtorno obsessivo-compulsivo (TOC): um estudo utilizando ressonância magnética nuclear**. 2002. Dissertação (mestrado), Departamento de Psiquiatria da Universidade de São Paulo, São Paulo,

VOLVIEW - Interactive and Intuitive Volume Visualization.
Disponível em <http://www.kitware.com/products/volview/> Acesso em 10 de jun. 2007.

VTK - The Visualization Tool Kit.
Disponível em <http:// public.kitware.com/VTK> Acesso em 31 de jan. 2007.

vtkFlRenderWindowInteractor.
Disponível em <http://cpbotha.net/software/vtkflrenderwindowinteractor> Acesso em 01 de jun. 2007.

WHITTED, T., **An Improved Illumination Model for Shaded Display**, CACM, 23(6), 343-349. 1980.

YAMIN, Adenauer C. **Um Estudo das Potencialidades e Limites na Exploração do Paralelismo**. 1999. 80f. Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

APÊNDICE A – Outros trabalhos na área

A seguir são apresentados trabalhos e ferramentas comerciais disponíveis que tratam dos mesmos temas e buscam soluções para problemas semelhantes àqueles abordados pelo presente trabalho. Anteriormente, foi colocada a importância da visualização científica como facilitadora dos diagnósticos em medicina, citando como exemplo os métodos SISCOM e a fusão de imagens em três dimensões proposta por Oliveira (2005), os quais esse trabalho se propôs a complementar utilizando reconstrução volumétrica e computação paralela. Além de trabalhos de processamento de imagens aplicado à medicina, serão vistos estudos relacionados ao algoritmo de *Ray-tracing* e relacionados à aplicação da computação paralela como solução no processamento de dados com alto custo computacional.

Os trabalhos que serão apresentados são os seguintes:

- Exames Virtuais Utilizando um Algoritmo de *Ray-casting* Acelerado (JUNIOR, 1999);
- Visualização de Volumes em Morfometria de Estruturas Cerebrais a partir de Imagens de Ressonância Magnética Nuclear (OLIVEIRA, 2002);
- Visualização de Imagens Médicas com Recursos em 3D (ALVES et al., 2004);
- *VolView - Interactive and Intuitive Volume Visualization* (VOLVIEW, 2007).

Exames virtuais utilizando um algoritmo de *Ray-casting* acelerado

Junior (1999), autor da dissertação, define seu trabalho como “um estudo sobre exames virtuais, que são técnicas de visualização do interior do corpo humano para fins de avaliação clínica utilizando técnicas de visualização volumétrica”. O trabalho expõe ainda um estudo comparativo entre os exames virtuais e seus análogos convencionais, colocando as vantagens e desvantagens em cada caso.

O trabalho de Junior (1999) é motivador no sentido de estudar aplicação da visualização científica em medicina e biologia, pois identifica uma série de vantagens

dos exames virtuais sobre os exames invasivos tradicionais. Os benefícios colocados são os seguintes:

- Exames virtuais são capazes de produzir vistas impossíveis de serem obtidas por meios convencionais;
- Por não serem invasivos, os exames virtuais não exigem nenhum tipo de contato com o interior do corpo do paciente;
- Diferentemente dos exames reais, exames virtuais não utilizam medicamentos para sedação, os quais podem produzir uma série de efeitos colaterais e necessitam de grande tempo para recuperação do paciente, já que a vida média de atuação de alguns destes medicamentos é bastante alta e podem ocorrer recaídas;
- O exame virtual facilita saber em que local exatamente se está. Uma dificuldade encontrada pelos médicos nos processos reais que é a de saber o ponto exato onde foi encontrada a patologia, como ocorre, por exemplo, na endoscopia ou colonoscopia.

Com relação à visualização volumétrica, Junior (1999) apresenta uma coletânea de procedimentos que visam acelerar o *Ray-casting* e propõe seu próprio algoritmo para tal. A solução, implementada pelo autor em linguagem C++, recebeu o nome de Algoritmo da Transparência em Multiresolução e apresentou ganho considerável em tempo de processamento se comparado aos algoritmos tradicionais e também a outros métodos de aceleração. Desta forma, o trabalho de Junior (1999) estimula a pesquisa e desenvolvimento das técnicas de exames virtuais baseados em reconstrução volumétrica, demonstrando ser possível obter eficiência na execução dos mesmos.

Visualização de volumes em morfometria de estruturas cerebrais a partir de imagens de Ressonância Magnética Nuclear

Oliveira (2002) apresenta uma ferramenta computacional desenvolvida para visualização de volumes anatômicos visando auxiliar o diagnóstico e avaliação clínica da epilepsia do lobo temporal. Porém, embora a visualização nesta aplicação seja volumétrica, a reconstrução é por superfície, realizada através do algoritmo *Marching Cubes*. Entretanto, outros algoritmos para reconstrução de superfície são

abordados de forma comparativa, o que é bastante importante para o correto entendimento de suas particularidades.

O trabalho atenta para o fato das aplicações de visualização aplicadas à medicina terem como requisitos fundamentais desempenho e interface amigável. A implementação utiliza as bibliotecas VTK para visualização e FLTK para a construção da interface, as mesmas utilizadas pelo presente projeto.

Além de validar mais uma vez a aplicação de visualização volumétrica na medicina, o trabalho de Oliveira (2002) chama atenção para os benefícios em relação ao custo trazidos por soluções desenvolvidas de acordo com as necessidades/possibilidades locais e com utilização de bibliotecas de *software* livre.

Visualização de imagens médicas com recursos em 3D

Também no contexto da visualização volumétrica aplicada no auxílio em exames e diagnósticos em medicina, encontra-se o trabalho de Alves et al. (2004). Alves et al. (2004) apresenta uma ferramenta alternativa para ajuda no diagnóstico de traumatismo craniano baseada em reconstrução volumétrica de imagens de Tomografia Computadorizada em formato *DICOM*.

A linguagem de programação utilizada foi C++ em conjunto com a biblioteca VTK. O suporte a imagens de TC no formato *DICOM* é nativo no VTK, portanto não são necessários esforços extras nesse sentido. Na reconstrução dos volumes foi utilizado o algoritmo de *Ray-casting*, também implementado nativamente pelo VTK.

VolView* 2.0 - *Interactive and intuitive volume visualization

O *VolView* (nome derivado de *Volumetric View*) é uma poderosa ferramenta comercial para visualização volumétrica desenvolvida pelos criadores do VTK. Atualmente em sua versão 2.02, o campo de aplicação do *VolView* não se restringe apenas a medicina, mas a visualização científica em geral, sendo utilizada na biologia, meteorologia, engenharias, etc. Além da visualização, é possível fazer diversas outras operações, tais como marcar áreas de interesse, capturar áreas específicas, imprimir e processar volumes ou dados, dentre outras.

O *VolView* foi desenvolvido utilizando a combinação VTK + ITK *(Insight Segmentation and Registration ToolKit*). A visualização de um volume depende da

função de reconstrução volumétrica utilizada, que pode ser: *Maximum Intensity Projection, Composite Function* ou *Isosurface*. A Fig. 40 ilustra uma imagem de Tomografia Computadorizada renderizada por cada um dos tipos possíveis.

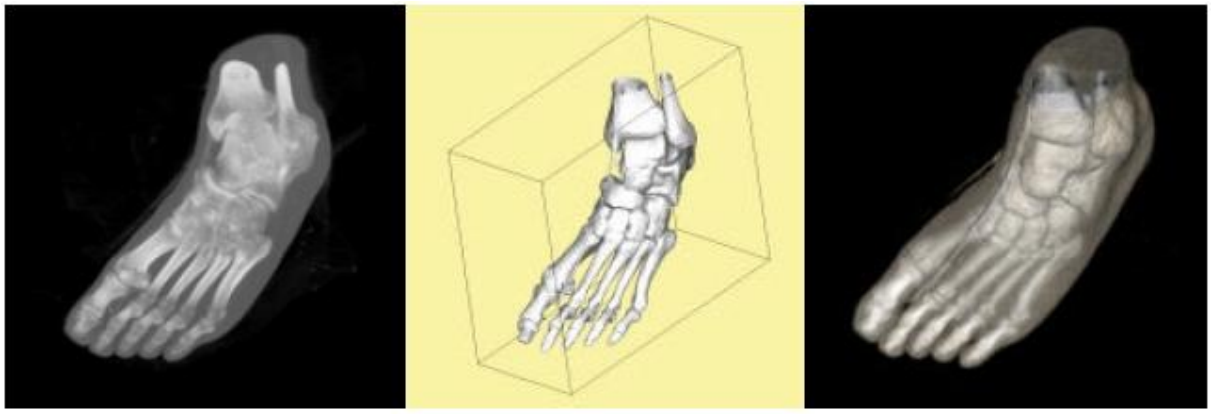


Figura 40 - Visualização volumétrica de uma imagem de TC no *VolView* utilizando em a) função de *composite*; b) iso-superfície (*isosurface*); c) MIP (projeção de máxima intensidade)

Fonte: VOLVIEW. 2007.

Entretanto, por ser uma ferramenta comercial o *VolView* apresenta como desvantagem seu custo elevado: US$ 2.500,00 (dois mil e quinhentos dólares) por licença, embora existam licenças acadêmicas de menor custo e gratuitas de uso limitado. Outra desvantagem advinda do fato de ser um *software* comercial, é que seu código-fonte, embora baseado no VTK e ITK, não é aberto. Logo, não pode ser adequado para atender necessidades locais específicas.

Conclusões

Os trabalhos acima listados indicam que os exames computadorizados não invasivos são vantajosos se comparados aos métodos tradicionais. Essas experiências relatam casos em que a combinação VTK + FLTK + C++ + *Ray-casting* foram aplicadas com sucesso como ferramentas para auxílio nos exames virtuais, reforçando assim as escolhas deste projeto. Colocam ainda estudos sobre os custos computacionais dos algoritmos de *Ray-tracing,* o que justifica sua implementação *multithread*.

Dados de catalogação na fonte:
Ubirajara Buddin Cruz – CRB-10/901
Biblioteca de Ciência & Tecnologia - UFPel

M637f Milani, Cleber Roberto
Ferramenta multithread para reconstrução volumétrica de imagens em medicina nuclear para auxílio na detecção de zonas epileptogênicas / Cleber Roberto Milani ; orientador Gerson Geraldo Homrich Cavalheiro ; co-orientador Lucas Ferrari de Oliveira. – Pelotas, 2007. – 129f. : fig. - Monografia (Conclusão de curso). Curso de Bacharelado em Ciência da Computação. Departamento de Informática. Instituto de Física e Matemática. Universidade Federal de Pelotas. Pelotas, 2007.

1.Informática. 2.Programação multihread. 3.Arquiteturas multi-core. 4.VTK. 5.Visualização volumétrica. I.Cavalheiro, Gerson Geraldo Homrich. II.Oliveira, Lucas Ferrari de. III.Título.

CDD: 006.6