Instituto de Matemática e Estatística
Universidade de São Paulo

MIPRenderer

Uma ferramenta para realizar projeções de máxima intensidade de angiogramas utilizando VTK e Qt.

Hugo Hiroshi Kondo
Otávio J. M. Santana

Orientador: Prof. Marcel P. Jackowski

São Paulo, 2010

Sumário

I Objetiva 4

1 Introdução 5
- 1.1 Motivação . . . 5
- 1.2 Objetivo . . . 6
- 1.3 Problemas . . . 6

2 Tecnologias estudadas 7
- 2.1 Analyze . . . 7
- 2.2 Renderização Volumétrica . . . 8
  - 2.2.1 Volume ray casting . . . 8
  - 2.2.2 Splatting . . . 9
  - 2.2.3 Shear warp . . . 9
  - 2.2.4 Texture mapping . . . 10
  - 2.2.5 Hardware-accelerated volume rendering . . . 10
- 2.3 MIP (*Maximum Intensity Projection)* . . . 11
- 2.4 MinIP (*Minimum Intensity Projection)* . . . 12
- 2.5 AIP (*Average Intensity Projection)* . . . 12
- 2.6 MedIP (*Median Intensity Projection)* . . . 13
- 2.7 LOD (*Level Of Detail)* . . . 14
- 2.8 Animação de Rotação . . . 17
- 2.9 VTK . . . 18
  - 2.9.1 Arquitetura . . . 18
  - 2.9.2 Graphics Model . . . 18
  - 2.9.3 Visualization Model . . . 20
  - 2.9.4 Processamento de imagem . . . 21
  - 2.9.5 Callbacks . . . 22
- 2.10 Qt . . . 23
  - 2.10.1 Signal . . . 23
  - 2.10.2 Slot . . . 24

2.10.3 Conexões com Signal e Slot 24
2.10.4 Meta Object System 26

3 Atividades realizadas **27**

4 Resultados **29**
4.1 Ferramenta 29
4.2 Planos futuros 34

5 Conclusão **35**

6 Referências bibliográficas **36**

II Subjetiva **37**

7 Hugo Hiroshi Kondo **38**
7.1 Desafios e frustações 38
7.2 Disciplinas relevantes 39
7.3 Futuro do projeto 40
7.4 Agradecimentos 40

8 Otávio J. M. Santana **41**
8.1 Desafios e frustações 41
8.2 Lista de disciplinas 42

Parte I

Objetiva

Introdução

1.1 Motivação

Uma das áreas mais importantes da computação é, sem dúvida, a computação gráfica. Muito evidenciada pelos jogos e pelo cinema, ela possui uma importância que fica mais distante do público comum, de fora da ciência da computação, que é a renderização de imagens médicas.

No início do ano estávamos em dúvida sobre o que fazer no Trabalho de Conclusão de Curso e fomos orientados pelo professor Carlos E. Ferreira a procurarmos sobre alguns temas propostos postados no forum da disciplina (PACA). Entre os temas estava um sobre renderização de imagens médicas volumétricas que nos interessou.

O fato de nenhum dos integrantes desse trabalho ter feito matéria de computação gráfica foi um fator importante, pois essa poderia ser uma oportunidade de aprender. Além disso, o fato de o resultado do trabalho poder ser usado como ferramenta para auxiliar pessoas da área de saúde também foi importante.

Esse trabalho tenta mostrar um pouco da área de computação gráfica voltada para a área de saúde com a renderização de imagens médicas volumétricas usando VTK e Qt.

1.2 Objetivo

O objetivo deste trabalho é prover uma ferramenta que a partir de um conjunto de imagens 2D (cortes no eixo x, y, z) renderize uma imagem volumétrica utilizando MIP (*maximum intensity projection*), permitindo ao usuário visualizar a parte da imagem que for de maior interesse.

Além disso, visa criar uma base para o desenvolvimento de um módulo de renderização volumétrica para o programa MedSquare.

1.3 Problemas

Os maiores problemas que tivemos foram:

- ler arquivos no formato *analyze*,
- entender seus campos,
- saber quais seriam necessários para a renderização desejada;
- estudar o funcionameto e arquitetura da ferramenta VTK;
- estudar o funcionamento do *framework* Qt;
- aprender sobre o controle de eventos que é parte importante quando se fala em interfaces gráficas;
- entender o funcionamento do MIP;
- criar uma ferramenta com interface amigável;
- aprender computação gráfica, que apesar de ser deixada para o final deve ser a primeira tarefa a ser realizada.

Tecnologias estudadas

2.1 Analyze

O Analyze é um formato de arquivo criado para armazenar dados de imagens de tomografia computadorizada, esse formato consiste de 2 arquivos: um arquivo header (.hdr) e um arquivo de imagem (.img) que pode estar compactado (.img.gz).

O arquivo header possui instruções que devem ser usadas para a leitura e renderização da imagem. Lá são fornecidos dados como dimensões (altura, largura e profundidade), intensidade máxima e mínima da imagem, sua resolução espacial e o tamanho do ponto para cada uma das três dimensões.

O arquivo da imagem contém as imagens dos cortes nos eixos x, y, z. Ele também contém as intensidades dos voxels.

Encontra-se hoje na versão 7.5 e é produzido pelo grupo Biomedical Resource e pela Mayo Foundation.

2.2 Renderização Volumétrica

É uma técnica que consiste em criar uma projeção 2D de um conjunto de dados 3D, amostrado discretamente. Para realizar esse tipo de renderização é preciso definir uma câmera no espaço relativo à imagem, e também é preciso definir a opacidade e cor de cada *voxel*. Isto é geralmente realizado usando funções de transferência de RGBA que definem valores de RGBA para cada *voxel*.

O arquivo Analyze é um conjunto de dados 3D, sendo assim podemos utilizar a renderização volumétrica para mostrar as imagens armazenadas nele.

Um volume pode ser renderizado diretamente como um bloco. Renderização volumétrica direta é uma tarefa computacionalmente intensa, que pode ser feita de várias maneiras.

- Volume ray casting
- Splatting
- Shear warp
- Texture mapping
- Hardware-accelerated volume rendering

Essa forma de renderização requer que cada amostra seja mapeada para uma opacidade e cor. Uma vez mapeada, o resultado é projetado no pixel correspondente do *frame buffer*. A forma que isso é feito depende da técnica utilizada.

2.2.1 Volume ray casting

Esta técnica produz resultados de alta qualidade, considerada a técnica que provê as melhores qualidades de imagem. *Volume ray casting* é classificada como uma técnica de renderização volumétrica baseada em imagem, uma vez que a computação vem da imagem de saída e não dos dados do volume da entrada como é o caso das técnicas baseadas em objeto.

A técnica consiste em traçar raios que saem do ponto central de visão da câmera e passa pelo pixel da imagem no plano que flutua entre a câmera e o volume a ser renderizado. Então o raio é amostrado em intervalos regulares ou adaptável ao longo do volume. Os dados são interpolados em cada ponto da amostra, a função de transferência é aplicada para formar uma amostra RGBA, a amostra é feita sobre a RGBA acumulado do raio, e o processo é repetido até que o raio saia do volume. A cor RGBA é convertida em uma cor RGB e depositado no pixel correspondente da imagem. O processo é repetido para cada pixel na tela para formar a imagem completa.

2.2.2 Splatting

Essa é uma técnica que troca qualidade por velocidade. Aqui, cada elemento do volume é arremessado na tela de visualização de trás pra frente. Estes *splats* são renderizados como discos cujas propriedades (cor e opacidade) variam de forma diametralmente normal. Discos planos e aqueles que possuem outras formas de distribuição são também usados dependendo da aplicação.

2.2.3 Shear warp

Esta técnica foi desenvolvida por Cameron e Undrill, e popularizada por Philipe Lacroute e Marc Levoy. Nela a visualização é transformada de forma que a face mais próxima do volume fica alinhada com um *image buffer* extern com uma escala fixa de *voxel* para *pixel*. O volume depois é renderizado nesse *buffer* usando o alinhamento de memória e escala fixa mais favoráveis. Uma vez que todos os cortes do volume forem renderizados, o *buffer* é então deformado na orientação desejada e escalado na imagem exibida.

Esta técnica é bastante rápida em *software* pelo custo de ter menos precisão das amostras e potencialmente piores qualidades de imagem comparado com o *volume ray casting*.

Há sobrecarga de memória para armazenar várias cópias do volume, para ter eixos do volume alinhados próximos. Essa sobrecarga pode ser atenuada usando codificação de comprimento.

2.2.4 Texture mapping

Muitos sistemas gráficos 3D utilizam *texture mapping* para aplicar imagens, ou texturas em objetos geométricos. Placas de vídeo comerciais são eficientes em texturização, e eficientemente renderizam cortes de um volume 3D, com capacidades para interação em tempo real. Esta técnica foi descrita primeiramente por Bill Hibbard e David Santek.

Estes cortes podem até mesmo ser alinhados com o volume e renderizados em qualquer ângulo para o usuário, ou pode ser alinhado com o plano de visualização e amostrado a partir dos cortes não alinhados pelo volume.

Texturização alinhada pelo volume produz imagens com qualidade razoável, embora muitas vezes há uma transição notável quando o volume for girado.

2.2.5 Hardware-accelerated volume rendering

Devido à natureza extremamente paralela da renderização volumétrica direta, renderização volumétrica para fins especiais era um tópico de pesquisa muito rico antes da renderização volumétrica pelas GPUs se tornar muito rápida. A mais conhecida tecnologia era VolumePro, que usava larga banda de memória e força bruta para renderizar usando o algoritmo *ray casting*.

Uma técnica atualmente explorada para acelerar os algoritmos tradicionais de renderização é o uso de placas gráficas modernas. Começando com *pixel shaders* programáveis, as pessoas reconheceram o poder de operações paralelas em múltiplos *pixels* e começaram a executar computações com propósito geral nos chips gráficos. Os *pixel shaders* são capazes de ler e escrever aleatoriamente da memória de vídeo e realizar alguns cálculos matemáticos e lógicos.

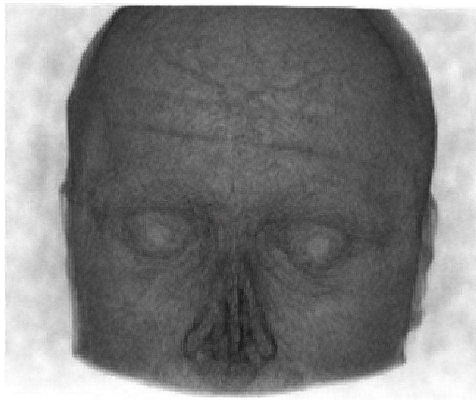
Nas GPUs recentes, os *pixel shaders* são capazes de funcionar como processadores MIMD utilizando até 1GB de memória de textura com formato de ponto flutuante. Com esse poder quase todos os algoritmos que podem ser processados em paralelo podem se executados com tremenda aceleração. Os *pixel shaders* podem ser usados para simular variações nas características de luz, sombra, reflexo, etc.

2.3 MIP (*Maximum Intensity Projection)*

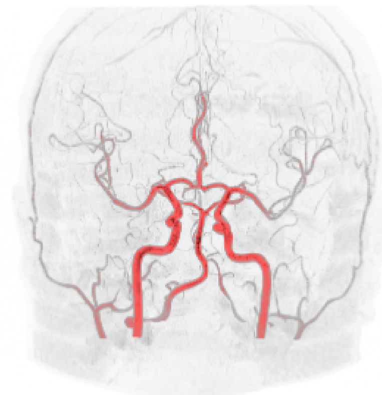
MIP é uma técnica que consiste em projetar a maior intensidade capturada pelos raios perpendiculares ao plano de projeção da imagem. Ela é usada para detecção de nódulos pulmonares em câncer de pulmão nos programas de tomografia computadorizada, pois sua renderização faz com que os nódulos se destaquem dos brônquios e vasos pulmonares.

Para aumentar o senso de tridimensionalidade, animações são, muitas vezes, renderizadas com vários frames de MIP em que os pontos de vista são ligeiramente diferentes, criando uma ilusão de rotação. Uma curiosidade é que, como a projeção é ortogonal, não se pode distinguir entre direita e esquerda, frente ou trás e se a imagem gira no sentido horário ou anti-horário.

MIP foi inventado para uso em Medicina Nuclear por Jerold Wallis em 1988 e posteriormente publicado no IEEE Transactions in Medical Imaging.



(a) Imagem sem MIP



(b) Imagem renderizada com MIP

Figura 2.1: Ambas com as mesmas cores

2.4 MinIP (*Minimum Intensity Projection)*

MinIP, ao contrário do MIP, projeta a menor intensidade capturada pelos raios perpendiculares ao plano de projeção da imagem. Essa técnica é utilizada para visualizar lúmen das vias aéreas centrais, áreas de enfisema, áreas de aprisionamento aéreo e opacidade em vidro fosco.

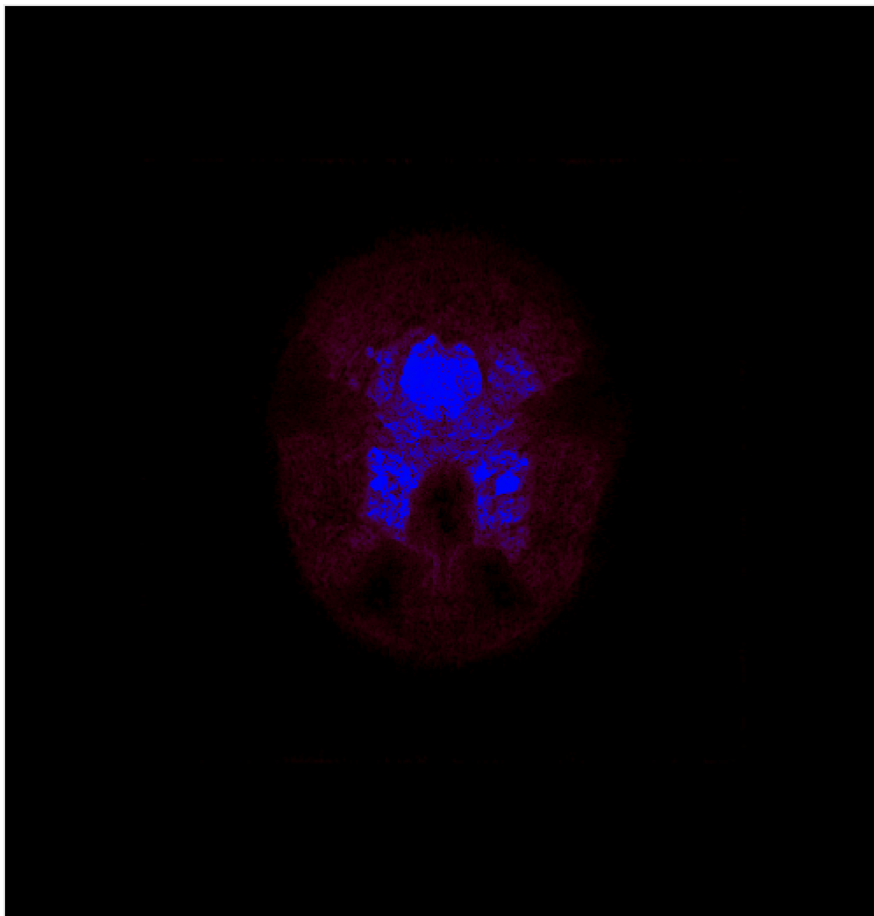


Figura 2.2: Imagem com projeção de intensidade mínima.

2.5 AIP (*Average Intensity Projection)*

A projeção de intensidade média consiste em computar os pontos de intensidade dos raios perpendiculares ao plano de projeção da imagem e calcular a média dessas intensidades. Ela nos dá uma ideia da composição da imagem, sobre a distribuição dos conjuntos de valores de frequência da imagem.

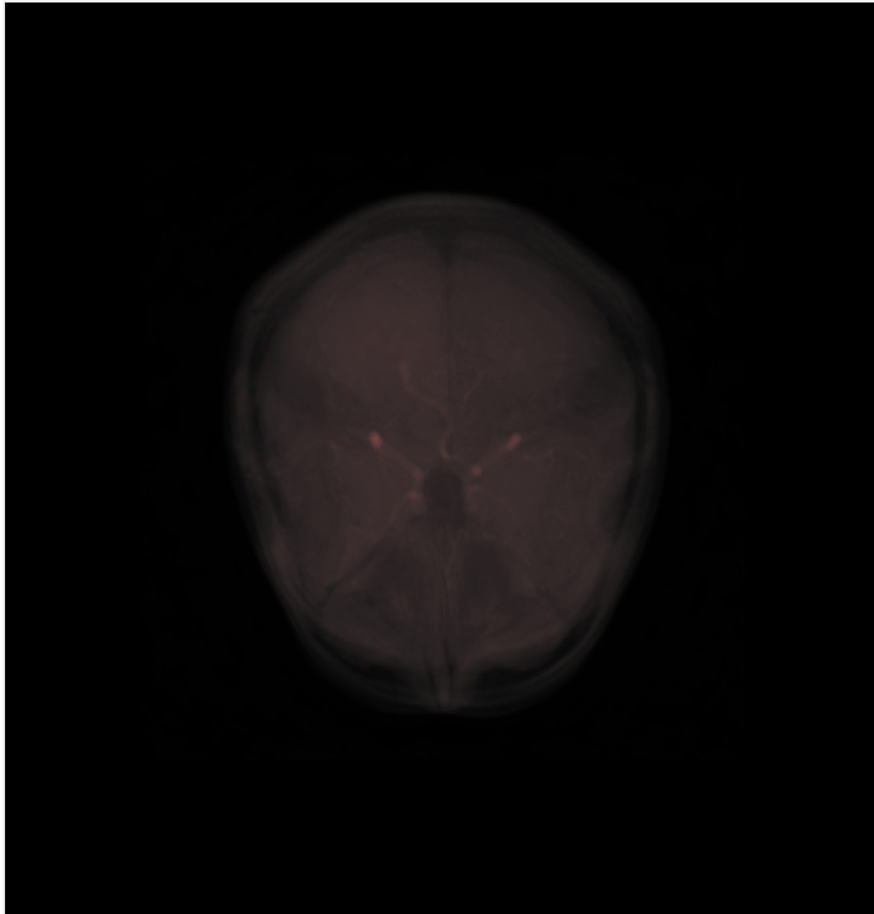


Figura 2.3: Imagem com projeção de intensidade média.

2.6 MedIP (*Median Intensity Projection*)

A projeção de intensidade mediana consiste em capturar os pontos de intensidade dos raios perpendiculares ao plano de projeção da imagem e projetar o ponto cuja intensidade esteja no ponto médio do vetor ordenado contendo todas as intensidades desse raio. É a projeção que leva maior tempo para ser executada pois necessita de uma ordenação do vetor para que seja capturada a mediana das intensidades. Assim como a projeção de intensidade média, a mediana nos dá uma ideia sobre a distribuição dos conjuntos de valores de frequência da imagem.

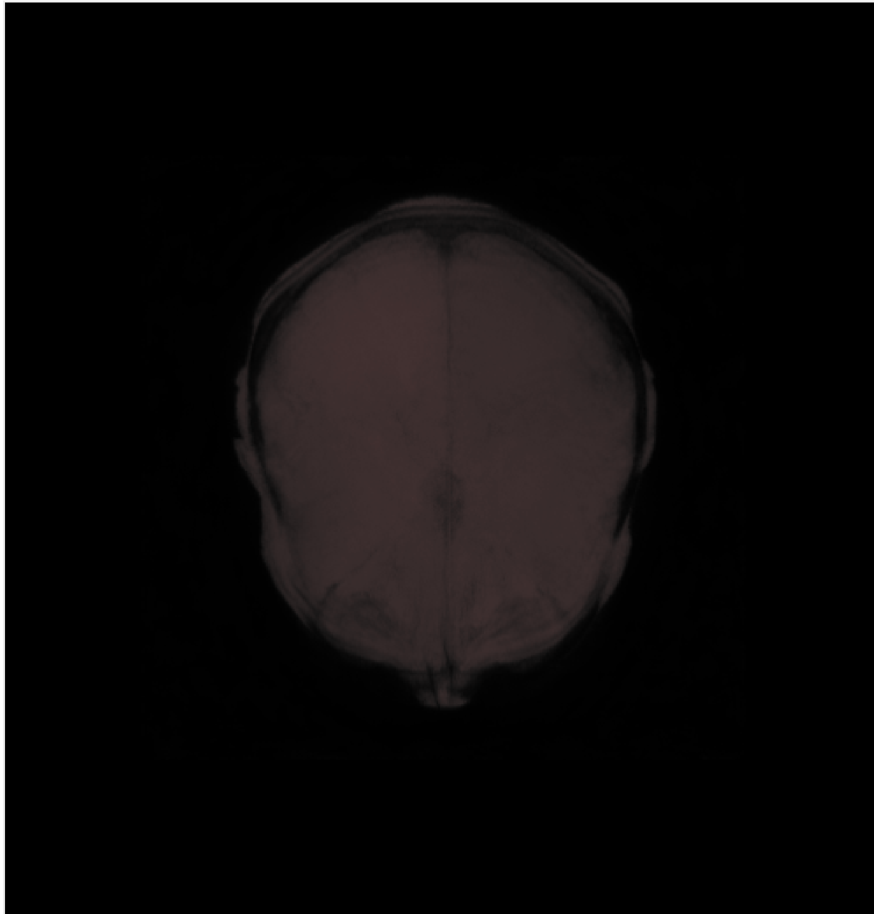


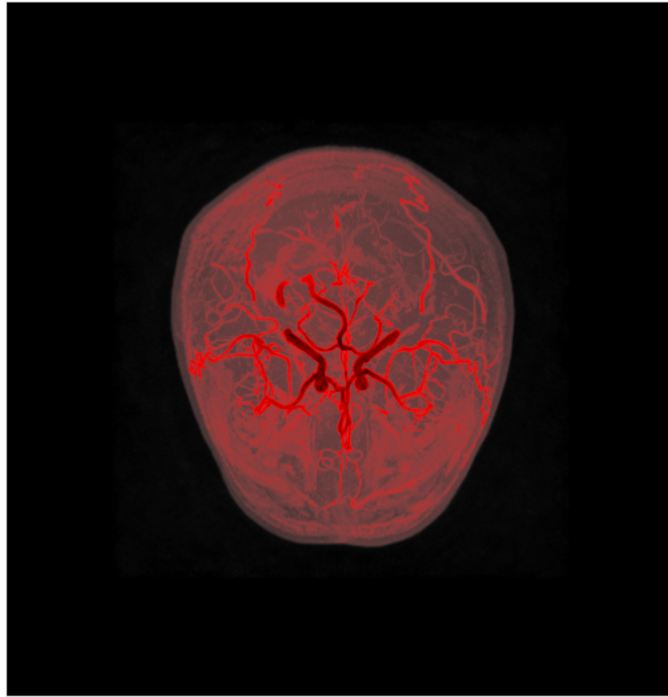
Figura 2.4: Imagem com projeção de intensidade mediana.

2.7 LOD (*Level Of Detail)*

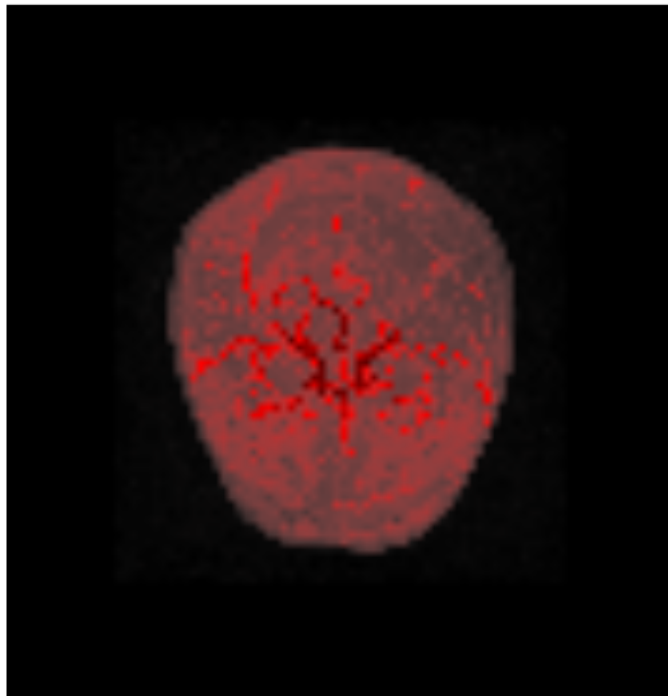
LOD é o nível de detalhe da imagem, quanto maior o LOD, maior será o detalhamento da imagem. Uma consequência disso é que a renderização de imagens com LOD alto demoram mais para serem renderizadas que imagens com LOD baixo.

Essa diferença fica mais evidente quando se usa a animação de rotação com diferentes valores para o LOD. Quando se rotaciona usando o LOD máximo obtém-se uma velocidade de rotação muito inferior à velocidade de rotação com LOD mínimo.

A velocidade de renderização tem relação com o número de frames renderizados por segundo e é esse número que aumenta ou diminui o LOD da imagem. Ou seja, quanto maior o número de frames renderizados por segundo, menor o LOD da imagem renderizada.



(a) Imagem com LOD máximo



(b) Imagem com LOD mínimo

Figura 2.5: Diferenças entre imagens com LOD diferentes.

2.8 Animação de Rotação

Para realizar a rotação deve-se rotacionar a câmera de visualização. Para isso usa-se uma função Azimuth que recebe como parâmetro um double que é o ângulo em graus que você desejar rotacionar. A rotação é feita em relação ao vetor do ponto de vista ao ponto focal e na direção horizontal.

Para criar a animação de rotação, a primeira ideia foi usar um laço para que a câmera girasse até completar um certo ângulo. Isso nos levou a um problema, a ferramenta ficaria travada e não seria possível a utilização de outras funcionalidades pois o processamento ficaria executando o laço. Além do mais não seria possível parar a rotação no meio de sua execução.

Pensamos então em solucionar esse problema abrindo uma thread para a execução da rotação. Para isso usariamos uma flag para dizer se a câmera deveria girar ou não. Quando o usuário iniciasse a animação a flag seria ligada e iniciaria a rotação da imagem e ao parar a animação ela seria desligada, ou seja, o botão apenas mudaria o valor de uma variável e teríamos um função que ficaria num loop infinito checando se poderia ou não rotacionar a imagem. Esse tipo de abordagem leva a um problema que é o consumo de recursos do processador desnecessariamente. Levando-se em conta que a renderização e manuseamento das imagens costuma consumir, quase sempre, cerca de 90% do processador. Uma solução seria usar um sleep na função de rotação da imagem. Isso nos levaria a outro problema pois se o tempo do sleep fosse grande a ação do botão de animação não seria imediata e se fosse muito pequeno ainda continuaríamos com o consumo desnecessário de processamento.

A solução adotada foi utilizar um timer, ao clicar no botão para iniciar a animação o timer é iniciado e ao clicar no botão novamente, para-se o timer. Usamos a classe QTimer do Qt. Inicialmente conectamos o timer com a função que rotaciona a imagem e quando o timer inicia, ele chama a função de rotação de tempos em tempos. Isso não deixa o botão travado e deixa o código mais claro e fácil de se entender. Além de ser mais fácil implementar usando o timer do que usando threads.

2.9 VTK

VTK é um software para processamento de imagem e visualização para computação gráfica 3D. Sua distribuição é gratuita, possui código aberto e foi todo escrito em C++. Possui vasta documentação disponibilizada no código, páginas do manual e na internet.

O VTK suporta uma larga variedade de algoritmos de visualização como métodos escalares, vetoriais, tensoriais, de textura e volumétricos. Suporta também técnicas avançadas de modelagem como o modelamento implícito, a redução poligonal, a suavização de malhas, o corte, a definição de contornos e a triangulação de Delaunay.

2.9.1 Arquitetura

O VTK consiste de dois subsistemas, um constituído pelas bibliotecas de classes compiladas em C++, e uma interface interpretada que permite a manipuilação dessas classes por linguagens como Java, Tcl e Python.

Ele é um sistema orientado a objeto, e para seu uso ser eficaz é necessário o entendimento de suas classes. Ele possui dois modelos de objeto:

- *Graphics Model*
- *Visualization Model*

2.9.2 Graphics Model

Esse modelo é composto por objetos que realizam a renderização dos dados geométricos. A seguir temos os principais objetos desse modelo:

- vtkActor, vtkActor2D, vtkVolume - subclasses do vtkProp e/ou vtkProp3D
- vtkLight
- vtkCamera
- vtkProperty, vtkProperty2D
- vtkMapper, vtkMapper2D - subclasses de vtkAbstractMapper
- vtkTransform

- vtkLookupTable, vtkColorTransferFunction - subclasses do vtkScalarsToColors
- vtkRenderer
- vtkRenderWindow
- vtkRenderWindowInteractor

Props são a representação das coisas que vemos na cena. Props que são posicionados e manipulados em 3D, são do tipo vtkProp3D (ex. vtkActor, vtkVolume), e os que são posicionados e manipulados em 2D, são do tipo vtkActor2D.

Props utilizam *mappers*, que são responsáveis por representar dados, para representar as suas geometrias. Eles também utilizam objetos de propriedade, que controlam a aparência do prop como cor, efeitos de luz ambiente, representação (*wireframe* ou superfície), entre outros. Atores e volumes ainda tem um objeto interno de transformação (vtkTransform). Esse objeto encapsula uma matriz de transformação 4x4 que controla a posição, orientação e escala do prop.

Luzes (vtkLight) são usadas para representar e manipular a iluminação da cena, elas são usadas apenas em ambiente 3D.

A câmera (vtkCamera) controla como a geometria 3D é projetada na imagem 2D durante o processo de renderização. A câmera possui diversos métodos para posicionar, apontar e orientá-la. Além disso ela controla a projeção perspectiva. Não são necessárias em ambiente 2D.

O *mapper* (vtkMapper), em conjunto com a tabela de pesquisa (vtkLookupTable), é usado para transformar e renderizar a geometria. O *mapper* fornece uma interface entre o *pipeline* de visualização e o *graphics model*. Como o vtkLookupTable é uma subclasse de vtkScalarsToColors, então ela é responsável por mapear valores escalares para cores, que é uma das técnicas de visualização mais importantes.

O vtkColorTransferFunction é usado para a renderização volumétrica, ele também mapeia valores escalares para cores, e a interpolação feita por ele de um valor a outro, que já foram mapeados, é linear.

Render (vtkRender) e *render windows* (vtkRenderWindow) são usados para gerenciar a interface entre a *engine* gráfica e o sistema de janelas do computador. O *render window* é a janela onde o *render* desenha. Mais de um render pode desenhar em um *render window*, assim como mais de um *render window* pode ser utilizado. A área onde um *render* desenha é chamada de *viewport*, da qual podem existir várias em um *render window*.

O VTK possui diversos métodos de interação dentro de uma cena, e um deles é o vtkWindowInteractor, que é uma ferramenta para manipular a câmera, pegar objetos, invocar métodos do usuário, e mudar algumas propriedades dos atores.

Apesar desses serem os principais objetos do *graphics model*, ainda existem diversas subclasses deles que especializam o comportamento dos objetos.

2.9.3 Visualization Model

A função do *pipeline* gráfico é transformar dados gráficos em imagens. A função do *pipeline* de visualização é transformar informação em dados gráficos, ou também podemos definir que o *pipeline* de visualização é responsável por construir a representação geométrica que é renderizada pelo *pipeline* gráfico.

O VTK usa a abordagem de fluxo de dados para transformar informação em dados gráficos. Existem dois tipos básicos de objetos envolvidos nessa abordagem.

- vtkDataObject
- vtkProcessObject

Data objects representam vários tipos de dados. Dados que possuem uma estrutura formal são referidos como *dataset* (class vtkDataSet). *Data objects* consistem de uma estrutura geométrica e topológica (pontos e células), assim como dados de atributos associados como escalares e vetores. Os dados de atributos podem ser associados com os pontos ou células do *dataset*. Células são organizações topológicas de pontos, células formam átomos do *dataset* e são usadas para interpolar informações entre os pontos.

Process objects, também chamados geralmente como *filters*, operam em *data objects* para produzir novos *data objects. Process objects* representam os algoritmos do sistema. *Process* e *data objects* são conectados para formar o *pipeline* de visualização (i.e., redes de fluxo de dados).

Existem importantes questões relativas à construção do *pipeline* de visualização.

Primeiro, *pipeline* topológico é construído usando variações do método

```
aFilter->SetInput( anotherFilter->GetOutput() );
```

que define a entrada do filtro *aFilter* para a saída do filtro *anotherFilter*.

Segundo, nós precisamos ter um mecanismo para controlar a execução do *pipeline*. O que queremos é executar as partições do *pipeline* que são necessárias para atualizar a saída. O VTK usa um esquema de avaliação *lazy* baseada no tempo de modificação interna para cada objeto.

Terceiro, a montagem do *pipeline* requer que apenas aqueles objetos compatíveis entre si possam se encaixar com os métodos SetInput() e GetOutput().

Finalmente, precisamos decidir se armazenamos em *cache*, ou mantemos os *data objects* após a execução do *pipeline*. Isso é importante para o êxito da aplicação de ferramentas de visualização.

VTK oferece métodos para ativar e desativar dados do *cache*, uso de contagem de referência para evitar a cópia de dados, e métodos para fluxo de dados em partes se um *dataset* inteiro não couber na memória.

Existem vários tipos de *process* e *data objects*. *Process objects* varia no tipo dos dados de entrada e saída, e também de acordo com o algoritmo implementado.

2.9.4 Processamento de imagem

VTK suporta um extenso conjunto de funcionalidades de processamento de imagens e renderização volumétrica. Em ambos os casos, 2D (imagem) e 3D (volume), os dados são guardados como vtkImageData. um *dataset* de imagem no VTK é um em que os dados são arranjados em um *array*

regular, com os eixos alinhados. Imagens, *pixmaps*, e *bitmaps* são exemplos de *dataset* de imagens 2D; volumes (uma pilha de imagens 2D) são um *dataset* de imagens 3D

Process objects no *pipeline* de imagem sempre recebem e devolvem *image data objects*. Por causa da regularidade e simplicidade dos dados, o *pipeline* de imagem tem outros importantes recursos. Renderização volumétrica é usado para visualizar vtkImageData 3D, e visualizadores de imagem especiais são usados para visualizar vtkImageData 2D.

Quase todos os *process objects* no *pipeline* de imagem são *multithreaded*, e capazes de transmitir dados em pedaços (para permitir o uso em caso de usuário com memória limitada). *Filters* automaticamente reconhecem o número de processadores disponíveis no sistema e criam esse número de *threads* durante a execução, assim como separam automaticamente os dados em pedaços menores que são transmitidos pelo *pipeline*.

2.9.5 Callbacks

O VTK utiliza-se de callbacks para controle de eventos de interação com o usuário. Callback é uma referência para um código executável que é passado como argumento para uma função. Ele permite uma camada de software de baixo nível chamar uma função ou subrotina definida em uma camada de alto nível.

No VTK usa-se callback usando um Observer e passando para ele o tipo de evento que ele deve observar e o ponteiro para um objeto vtkCommand. Você pode implementar seu próprios callbacks criando classes que tenham como superclasse o vtkCommand.

Callback pode ser interessante para reutilização de código nos casos em que você tem tarefas iguais a serem executadas e a diferença seria a função chamada. Com a técnica do callback pode passar a função a ser executada como parâmetro.

2.10 Qt

Qt é um *framework* de aplicações multiplataforma, muito utilizado para criação de interfaces gráficas no desenvolvimento de *software*, também é utilizada no desenvolvimento de programas sem interface gráfica de usuário.

Qt é escrito em C++ e pode ser utilizado com várias outras linguagens de programação. Um caso famoso de utilização do Qt é o KDE, um ambiente gráfico para sistemas UNIX.

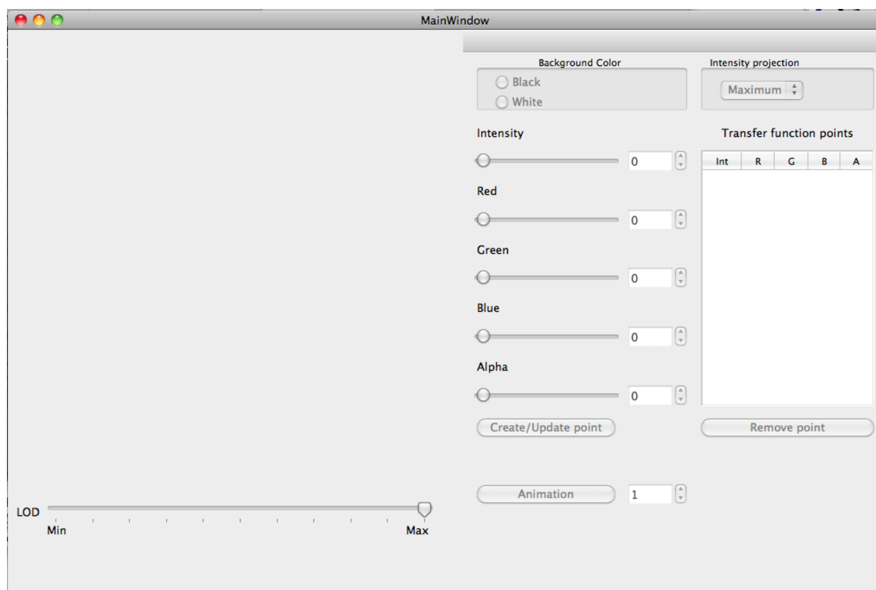


Figura 2.6: Interface criada com o auxílio do Qt.

2.10.1 Signal

Sinais são emitidos por um objeto quando seu estado é alterado. Apenas as classes que definem um sinal ou que herdam dessas classes podem emití-lo. Normalmente, quando um sinal é emitido, os *slots* conectados a ele são chamados como funções normais, sem uma ordem específica, e o código após a emissão do sinal só será executado quando todos os *slots* tiverem terminado suas tarefas. Isso pode ser mudado alterando o tipo de conexão, usando a *"queued connection"*. Os sinais são automaticamente gerados pelo moc (*meta object compiler*) e não precisam ser implementados. Além disso, eles nunca devem ter valor devolvido (devem ser *"void"*).

2.10.2 Slot

O *slot* é chamado quando um sinal conectado a ele é emitido. São funções C++ normais e podem ser chamadas como qualquer outra, porém possuem a capacidade de se conectarem a sinais. Entretanto um *slot* pode se conectar a qualquer componente e ele será invocado independentemente do seu modificador de acesso, ou seja, *slots* do tipo *private* podem ser invocados através dessas conexões mesmo fora de suas classes.

2.10.3 Conexões com Signal e Slot

O Qt possui uma maneira elegante para o controle de eventos e que nos dá uma alternativa interessante ao *callback*. A classe QObject possui um método estático chamado connect() que conecta dois objetos. Esses objetos possuem sinais (para enviar mensagens) e *slots* (para receber mensagens).

Quando se fala em velocidade, *callbacks* levam uma ligeira vantagem sobre o mecanismo do Qt. *Callbacks* são, na média, dez vezes mais rápidos do que sinais e *slots*. Esse é o *overhead* devido ao fato de ser necessário localizar a conexão e percorrer todos os *slots*. Entretanto essa desvantagem é imperceptível em sistemas reais.

Quando se usa o método connect() você estará conectando o sinal de um objeto ao *slot* de outro objeto. Toda classe que herda de QObject ou de suas subclasses podem implementar sinais e *slots*.

Um sinal é emitido toda vez que um evento ocorre, como por exemplo um clique num botão ou uma troca de algum botão do tipo *ratio*. Um *slot* é uma função que é chamada em resposta a um sinal ao qual foi conectado. Pode-se conectar sinais a vários *slots* e até mesmo a outros sinais, o que resultaria em um sinal disparando outro sinal.

Um *slot* também pode ser conectado a vários sinais. Esse tipo de de mecanismo é *type safe* uma vez que a assinatura dos sinais e *slots* deve ser igual a assinatura dos sinais e *slots* implementados e declarados no arquivo .h do QObject correspondente.

Com a compatibilidade da assinatura o compilador ajuda em sua detecção. Além disso, ele possui baixo acoplamento, uma vez que cada sinal não sabe, e nem precisa saber, a qual *slot* está conectado. Da mesma maneira um *slot*

também não sabe se existe algum sinal para ele. Esse mecanismo também garante que se você conecta um sinal a um *slot*, ele será chamado, com os argumentos certos e no momento certo.

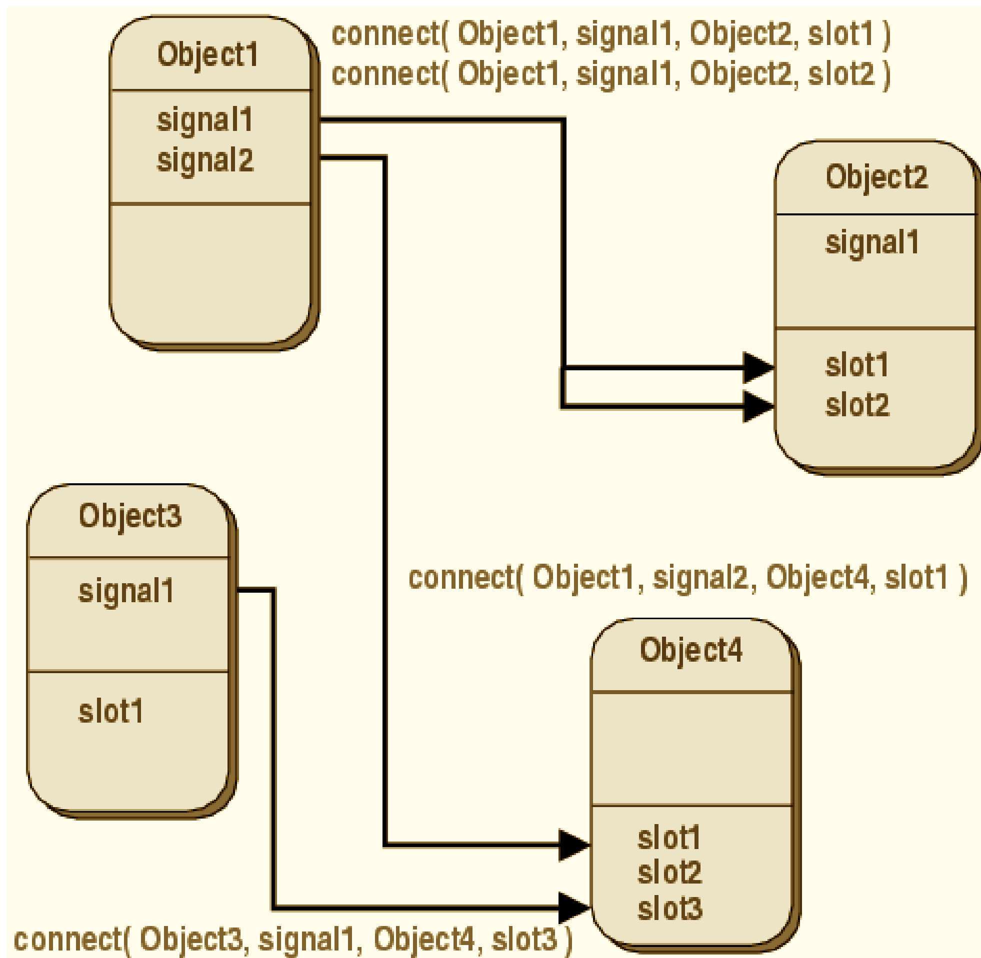


Figura 2.7: Esquema de conexões de sinais e slots do Qt.
Fonte: http://doc.trolltech.com/4.4/signalsandslots.html

No *software* desenvolvido utilizou-se esse mecanismo entre vários elementos como nos dois quantificadores, a caixa (QSpinBox) na qual se escreve o valor e no botão deslizante (QSlider), de cor e opacidade. Há também conexões entre as linhas da tabela de cor e opacidade e a caixa com o valor, além dos botões (QPushButton) de criar e remover cor com o objeto que possui a imagem (QVTKWidget).

2.10.4 Meta Object System

Meta-object é qualquer entidade que manipula, cria, descreve ou implementa outros objetos. O moc (*meta-object compiler*) faz um *parse* da classe de declaração C++ e gera um código C++ que inicializa o *meta-object*. Esse *meta-object* contém todos os nomes de sinais e *slots*, assim como os ponteiros para essas funções.

O *meta-object system* é baseado em 3 coisas:

- a classe QObject provê uma base para objetos poderem se beneficiar do *meta object system*;
- uma macro QObject dentro de uma seção privada da classe é usada para habilitar funcionalidades *meta-object*, como sinais e *slots*;
- o moc fornece para cada subclasse do QObject o código necessário para implementar funcionalidades *meta-object*.

A ferramenta moc lê arquivos *header* em C++, encontra as classes que contém a macro QObject e produz o código fonte contendo o código do *meta-object* dessas classes. Esse código é necessário para o mecanismo de sinais e *slots*. O código gerado pelo moc deve ser compilado e *linkado* junto com a classe de implementação.

Atividades realizadas

Entre as tarefas que deveriam ser cumpridas, estudar computação gráfica era sem dúvida a primeira a ser realizada, uma vez que não cursamos a matéria de computação gráfica. Esse fato talvez tenha sido um dos fatores para que o começo do trabalho tenha sido pouco produtivo, uma vez que não conseguíamos avaliar se estava sendo feito algum progresso, pois não havia nenhum *software* sendo desenvolvido. Após os estudos fomos procurar pelo professor Marcel que nos orientou a criar o *software* de visualização.

Para auxílio nessa tarefa foi utilizado o VTK e o Qt. Para utilizarmos o VTK e aproveitarmos algumas de suas funcionalidades, como o suporte para Qt, baixamos e compilamos o código fonte, podendo assim configurá-lo da maneira mais conveniente. Entre outras coisas pode-se escolher se sua compilação deve gerar bibliotecas para desenvolvimento em Python, Tcl e Java. Escolhendo, por exemplo, Java o código é empacotado para se gerar o vtk.jar. Para o desenvolvimento do software não foi preciso escolher nenhum tipo de empacotamento pois ficou definido que o código seria escrito em C++.

Para a compilação do VTK é necessária a instalação do programa CMake (*cross plataform make*), um sistema para geração automatizada. Ao rodar o *cmake* para o VTK ele usa o arquivo CMakeLists.txt para algumas configurações e gera o arquivo CMakeCache.txt. Nesse arquivo são feitas todas as opções de funcionalidades desejadas. Ao configurar o CMakeCache.txt com as opções desejadas, deve-se rodar novamente o *cmake* para que seja gerado o *makefile* do VTK. Após isso deve-se rodar o comando *make* e em seguida *make install.* Todo esse processo demora aproximadamente 2 horas.

A instalação do Qt é um pouco mais fácil, porém é mais demorada. O processo demorou cerca de 5 horas. O processo foi parecido: baixar, descompactar e compilar o código fonte para gerar as bibliotecas. O primeiro passo para a instalação é rodar um *script shell* chamado configure e escolher algumas opções como criar as bibliotecas como dinâmicas ou estáticas. Após isso

deve-se rodar o *make* e em seguida o *make install*. Ao final deve-se configurar o caminho dos binários gerados do Qt na variável de ambiente $PATH.

Após a instalação das ferramentas utilizadas e todo o estudo sobre elas pudemos construir o *software* desejado.

Primeiramente nos comprometemos a renderizar a imagem da maneira que deveria ser feita utilizando apenas o VTK. Para isso começamos a escrever o código para ler o formato *analyze*.

Cumprida a tarefa de ler o formato *analyze*, começamos a tentar renderizar a imagem utilizando o VTK e deixamos um pouco de lado os dados do arquivo. Queríamos primeiro entender como renderizar a imagem para depois utilizar esses dados. Nesse momento intensificamos nossos estudos para o VTK, seja no livro, seja lendo códigos dos *headers* das bibliotecas.

Tinhamos então dois módulos que precisavam ser integrados, o que não foi complicado devido a maneira como foi desenvolvido o código.

Faltava então criar a interface gráfica com o auxílio do Qt. Nesse ponto paramos para estudar um pouco sobre o mecanismo de controle de eventos do Qt. Tivemos uma certa dificuldade na integração do mecanismo de abrir os arquivos com o nosso leitor de arquivos, uma vez que Qt, VTK e o leitor usam tipos distintos de strings. Resolvido isso chegamos a ferramenta para visualização.

Resultados

4.1 Ferramenta

Foi criado um menu (*Open*) para abrir arquivos em que o usuário poderá navegar pelos seus arquivos, mas somente poderá selecionar os arquivos com extensão .hdr ou .img.

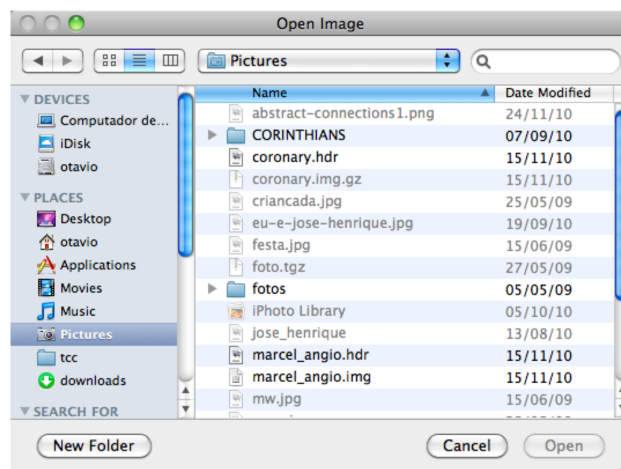


Figura 4.1: Menu para navegação e escolha de arquivos. Apenas no formato *analyze* são aceitas.

O programa apresenta uma interface simples onde o usuário pode criar um ponto de RGBA para uma determindada intensidade, onde RGB determina a cor do ponto (0 -> completamente escuro e 255 -> completamente intenso) e A a opacidade (0 -> totalmente transparente e 100 -> totalmente opaco). Ao escolher um arquivo a imagem é renderizada com apenas dois pontos de cor, um deles é para a menor intensidade com RGBA = (0, 0, 0, 0) e outro para a maior intensidade com RGBA = (255, 255, 255, 100).

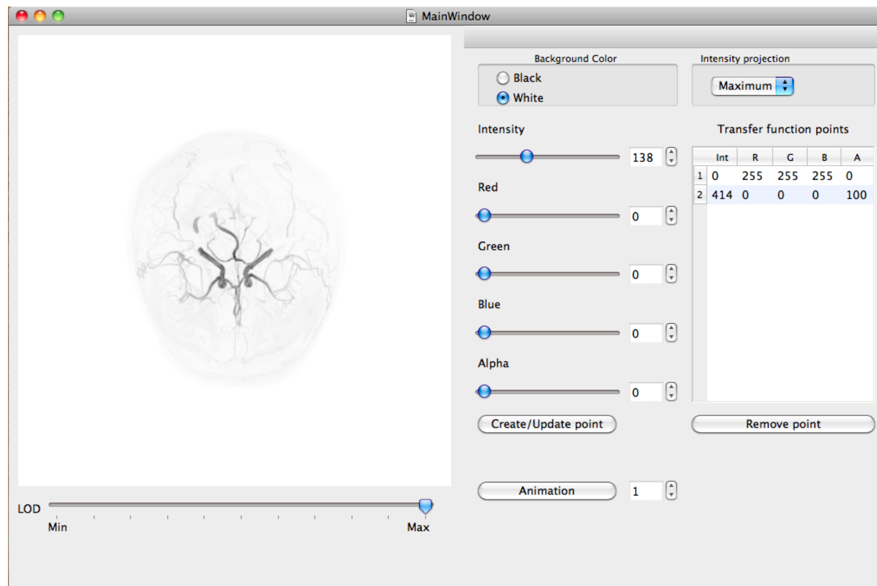


Figura 4.2: Visualização da imagem ao ser aberta.

Em seguida o usuário poderá inserir pontos RGBA e trocar a cor do plano de fundo podendo escolher entre duas cores: preto ou branco.

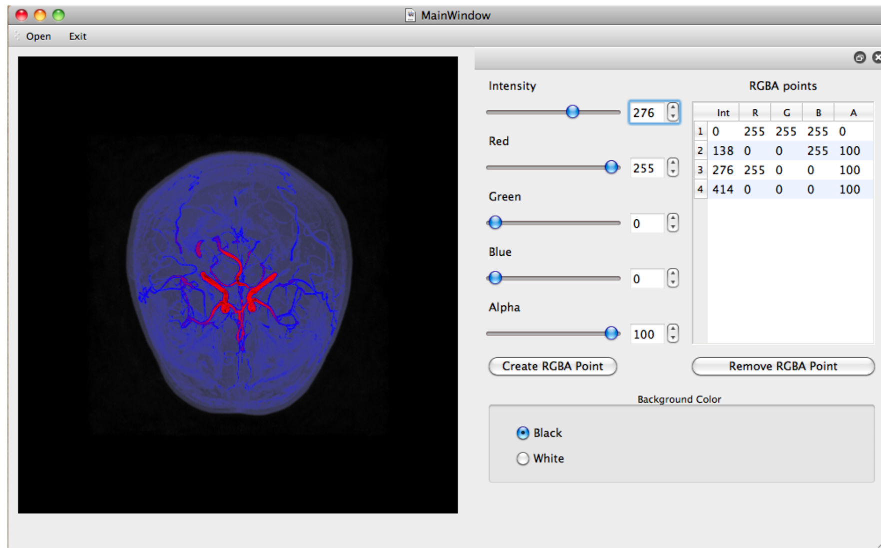
Para inserir um novo ponto o usuário deve escolher um valor de intensidade e um valor de RGBA. Esses pontos são adicionados à tabela a direita.

Pode-se também remover pontos da tabela, bastando apenas escolher a intensidade do ponto que se deseja remover e clicar no botão "Remove RGBA Point", ou ainda pode-se clicar em algum ponto da tabela, ele será automaticamente passado para as caixas de RGBA, e clicar no botão "Remove RGBA Point".

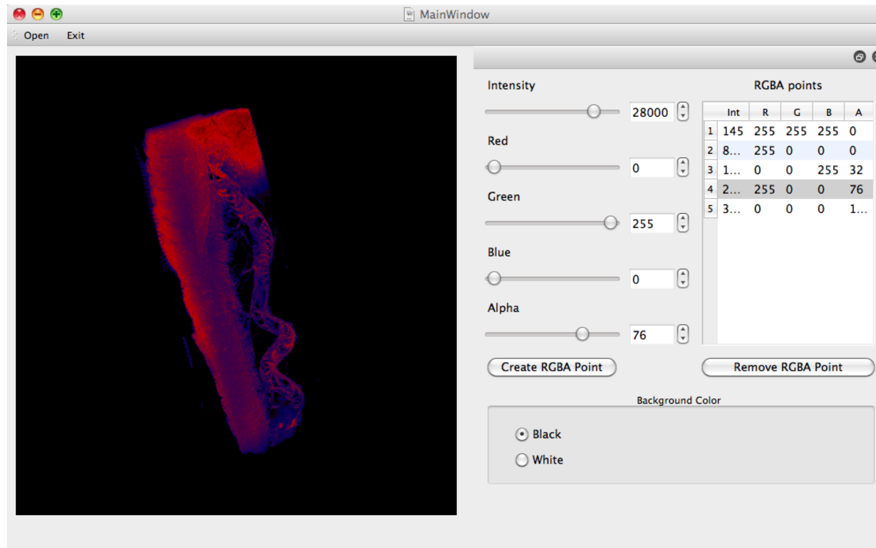
Para pontos intermediários a dois pontos adjacentes adicionados à imagem, a variação da opacidade é uma função de primeiro grau que tem como coordenadas inicias (Io, Ao) e finais (If, Af) onde I e A são, respectivamente, a intensidade e a opacidade do ponto. O mesmo ocorre com o sistema de cores. A esse tipo de variação se dá o nome de interpolação linear.

A ferramenta permite também que se rotacione a imagem através do botão "Animation" e com o número de graus determinado pela campo ao lado do botão. Pode-se escolher ângulos de 1 até 360 graus.

Além da projeção MIP, pode-se também ver a imagem com outros tipos de projeções, como a mediana, a média e a mínima. A escolha pode ser feita através do combo em cima da tabela de cores.



(a) Imagem de um angiograma de um cérebro humano visto de cima.

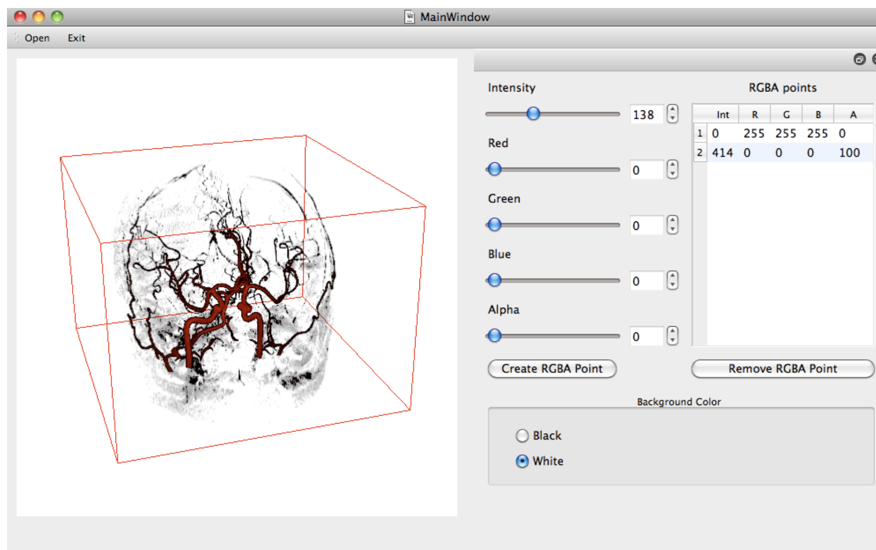


(b) Imagem de RM ilustrando segmento de coronária de um roedor (ex vivo). Cortesia de *Harvard Medical School, Boston, MA - USA*

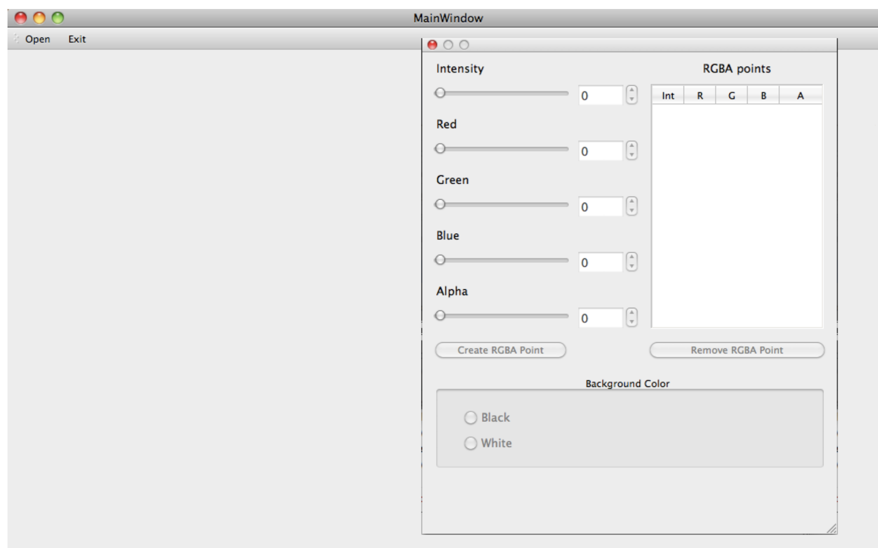
Figura 4.3: Ambas imagens visualizadas na ferramenta desenvolvida com o auxílio do Qt e VTK

O usuário pode ainda rotacionar a imagem, dar zoom e deslocar a imagem dentro do limites da tela de visualização.

Para ajudar na visualização e dar alguma noção ao usuário sobre qual posição atual da imagem, existe a opção de mostrar um box em volta dela, que é ativada apertando-se a tecla “p” do teclado. É possível também separar o navegador lateral da tela principal do programa facilitando a visualização e manuseio da imagem.



(a) Imagem com o box envolta da imagem volumétrica



(b) Imagem com o menu separado da tela principal

4.2 Planos futuros

Com as dificuldades iniciais que tivemos nos estudos não tivemos tempo para fazermos tudo o que queríamos, também com o progresso do trabalho começaram a surgir novas ideias para incrementarmos a nossa ferramenta, que não foram concluídas.

Uma das ideias que surgiram foi a criação a do navegador lateral que pode ser destacado e reintegrado à janela principal, porém quando o fechamos não é possível reabrí-lo. Outra ideia que acabamos não implementando foi a de dar a opção de mudar o tipo de projeção, uma com MIP e a outra sem.

Tinhamos a ideia inicial de integrar a nossa ferramenta ao MedSquare, porém não foi possível testar a compatibilidade.

Conclusão

Ainda que a primeira tarefa realizada no trabalho não tivesse surtido nenhum efeito imediato, como produção de *software*, ela foi importante para o resto do projeto. Sem dúvida alguma o processo seria muito mais demorado sem uma base em computação gráfica. Entender os processos envolvidos na renderização de *software* apenas olhando para o VTK seria extremamente complicado. Se tivéssemos cursado a matéria de computação gráfica o resultado obtido, provavelmente, teria sido mais rápido e menos doloroso.

O desenvolvimento desse trabalho nos deu uma bagagem grande sobre o controle de eventos, *callbacks* e sinais + *slots*, um tópico importante em desenvolvimento de interfaces gráficas e muito utilizado para desenvolvimento web.

Olhando para a arquitetura do Qt e VTK percebemos como um código bem estruturado e com comentários facilita no entendimento da ferramenta, uma vez que muito do que foi aprendido e utilizado dessas ferramentas veio da leitura de códigos dos arquivos de *header*.

Outro grande aprendizado é sobre trabalho em grupo num projeto grande, no qual você acaba passando um bom tempo com outra pessoa e aprende a trabalhar em grupo.

Aprendemos também a importância do processamento de imagens médicas do tipo tomografia computadorizada e ressonância magnética, pelo fato dos médicos poderem identificar se existem problemas apenas alterando a cor e a opacidade de alguns pontos com intensidades específicas.

Referências bibliográficas

VTK

http://www.vtk.org/
http://www.macresearch.org/installing_vtk_on_mac_os_x
Livro - VTK User's Guide (VTK 4.2) - Kitware

Qt

http://doc.qt.nokia.com/qt-maemo-4.7/install-mac.html
http://qt.nokia.com/
http://doc.trolltech.com/4.4/signalsandslots.html
http://doc.trolltech.com/4.4/tutorials-tutorial.html
http://doc.trolltech.com/4.4/metaobjects.html

Analyze

http://www.grahamwideman.com/gw/brain/analyze/formatdoc.htm

Volume Rendering

http://en.wikipedia.org/wiki/Volume_rendering

MIP

http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=41482

C++

http://www.cplusplus.com/doc/tutorial/

Parte II

Subjetiva

Hugo Hiroshi Kondo

7.1 Desafios e frustações

O primeiro desafio que tivemos foi o aprendizado de computação gráfica, uma vez que nenhum dos integrantes havia feito a disciplina de introdução à computação gráfica durante o curso de graduação.

Outro grande desafio foi aprender a mexer com as ferramentas do VTK, que possui muitos exemplos e boa documentação, porém por ter muitas funcionalidades demanda tempo para aprendê-las. Ainda com o VTK tivemos problemas iniciais para conseguir instalá-lo corretamente assim como conseguir configurar as variáveis de ambiente necessárias, pois começamos vendo os exemplos que existiam em Tcl, que tinha configurações diferentes de C++.

Quando conseguimos rodar os exemplos em Tcl usavamos imagens que eram fornecidas com o exemplo, porém quando tentamos inserir a nossa imagem, no caso o angiograma do cérebro do professor Marcel, não conseguimos renderizá-la de primeira. Olhando as imagens do exemplo vimos que existia um *header* dentro da imagem que era interpretado pelo VTK e fizemos o mesmo com a nossa imagem. Conseguimos renderizar mas não obtivemos uma imagem nítida, tivemos novamente que descobrir qual era o problema e conseguimos encontrar a solução.

Após conseguirmos com sucesso renderizar o angiograma o professor Marcel nos deu outro desafio, que era o de criar uma interface gráfica simples e fácil de utilizar para interação com o usuário, então ele sugeriu usarmos o *framework* Qt, que é naturalmente compatível com o VTK.

Na instalação do Qt tivemos outros problemas, primeiro a instalação que é demorada, segundo apesar do Qt e do VTK serem naturalmente compatíveis, era necessário *linkar* as duas ferramentas o que nos tomou algum tempo.

Construida a interface gráfica e a renderização sendo realizada com sucesso, faltava interpretar e fornecer a imagem de forma correta, pois não estávamos interpretando o arquivo *header* (.hdr) da imagem, mas sim mantendo as informações sobre a imagem de fora *hardcoded* no arquivo .img. Para consertarmos isso tivemos que pesquisar a forma correta de se ler e armazenar as informações contidas no *header*.

Com uma documentação não tão boa quanto a do VTK e Qt, tivemos mais dificuldades em conseguir decifrar as informações contidas no *header* e selecionar aquelas que de fato iríamos utilizar, porém depois de muito pesquisar na *internet*, conseguimos retirar as informações necessárias para podermos ler de forma correta a imagem.

Tivemos algumas frustrações no desenvolvimento da interface gráfica, como por exemplo o fato de não termos conseguido reabrir o navegador lateral após destacá-lo e fechá-lo, e também não conseguimos redimensionar a janela principal para o tamanho do visualizador quando descávamos o navegador lateral.

Outra frustração que tive foi o fato de termos começado tarde o desenvolvimento do *software* e com isso as tarefas ficaram muito apertadas e outras ideias que estavam fora da proposta não puderam ser implementadas.

7.2 Disciplinas relevantes

MAC0110 - Introdução à Computação

Como eu nunca havia mexido com programação e não tinha nem idéia do que era, esse foi o início do meu interesse pela área de ciências da computação.

MAC0122 - Princípios de Desenvolvimento de Algoritmos

Disciplina que dá início ao estudo de uma grande preocupação de um cientista da computação, a eficiencia. Nesta disciplina descobri diversas formas de se realizar uma mesma tarefa podendo assim tomar a melhor decisão e tentar fazer do *software* desenvolvido o melhor possível.

MAC0323 - Estruturas de Dados

Nesta disciplina pude aprender como armazenar os dados e também como a forma de armazená-los faz toda a diferença na hora de consultá-los.

MAC0211 - Laboratório de Programação I

É nesta disciplina que aprendemos a trabalhar em grupo e temos o contato com um projeto maior. Ela nos ajuda a entender a importância de um programa bem escrito e também é nela que temos o primeiro contato com controladores de versão, como o subversion, e com o LaTeXpara fazermos relatórios.

MAC0441 - Programação Orientada a Objetos

Com os conceitos aprendidos nessa disciplina, pudemos entender melhor as classes do VTK e Qt, bem como mantivemos um código limpo e encapsulado.

7.3 Futuro do projeto

Conforme apresentado na seção 4.2 o principal objetivo futuro é a integração dessa ferramenta com o *software* MedSquare.

7.4 Agradecimentos

Agradeço ao professor Marcel por ter nos apoiado e guiado nessa empreitada, sem os conselhos e orientações dele esse trabalho não teria sido concuido. Agradeço também aos professores do BCC que durante os anos de graduação me ensinaram muito, e por último mas não menos importante agradeço à minha família por ter me apoiado durante toda a minha vida.

Otávio J. M. Santana

8.1 Desafios e frustações

Com certeza um dos maiores desafios encontrados foi aprender computação gráfica. Por não ter feito a matéria isso foi um dos fatores que mais atrasou o desenvolvimento da tarefa. Como não sabíamos muita coisa sobre computação gráfica, as reuniões com o professor Marcel foram deixadas mais para o final. Quando essas reuniões se tornaram mais frequentes, visivelmente o trabalho foi executado mais rapidamente e com muita qualidade.

Outra dificuldade foi aprendizado de VTK que apesar de muito bem escrito e bem documentado, tem cerca de 1800 arquivos de header e demanda bastante tempo para se encontrar o que deseja. Uma das grandes fontes de informação que usei foram os comentários no código.

A escolha de se fazer o programa em C++ nos trouxe um pequeno problema com relação ao tempo, uma vez que a cada alteração no código era necessário compilar. Compilar o código era relativamente rápido, algo em torno de 5 a 10 segundos, mas quando se faz necessário compilar seguidas vezes por pequenos erros esse tempo acaba sendo alto. As vezes faziamos alterações que demoravam menos tempo do que o tempo de se compilar. Talvez a escolha por Tcl tivesse sido mais eficiente nesse sentido, porém teríamos que aprender outra linguagem. Apesar de nunca ter programado em C++ antes, eu já havia programado em C, o que pesou na escolha da linguagem.

A grande frustação fica por conta de eu não ter feito computação gráfica na graduação, pois acredito que teríamos um salto de qualidade na execução as tarefas.

8.2 Lista de disciplinas

MAC0110 - Introdução à computação

Matéria que começa o fascínio pela programação e que se inicia o desenvolvimento de lógica de programação.

MAC0122 - Princípios de desenvolvimento de algoritmos

Primeiro contato com estruturas de dados simples e onde se começam as preocupações com consumo de memória e consumo de tempo de funções.

MAC0211 - Laboratório de programação I e
MAC0242 - Laboratório de programação II

Primeiro contato com projetos grandes e aprendizado sobre linkagem de bibliotecas, como fazer makefiles, configuração de ambientes.

MAC0316 - Conceitos de linguagens de programação

Aprendizado sobre funções e tipos de funções. Funções que recebem funções como argumento. Tudo isso foi essencial para um melhor entendimento sobre callbacks.

MAC0422 - Sistemas operacionais

Essa matéria foi importante primeiro para ajudar na instalação das ferramentas utilizadas. Outro ponto importante é que nessa matéria aprendemos muito sobre sinais e isso facilitou no entendimento do controle de eventos do VTK e Qt.

MAC0441 - Programação orientada a objetos

Os conceitos ensinados sobre modularização e orientação a objetos foram essenciais para manter um código limpo e organizado.