César Henrique Keiti Kuroiwa
Alexandre Marques Lima

Modelagem 3D de um Pulmão Animado

Monografia de Trabalho de Formatura, apresentado a Escola Politécnica da Universidade de São Paulo para obtenção do título de Engenheiro Mecânico com ênfase em Mecatrônica e Sistemas Mecânicos

São Paulo
2006

César Henrique Keiti Kuroiwa
Alexandre Marques Lima

nota final 10,0
(dez e zero)

Modelagem 3D de um Pulmão Animado

Monografia de Trabalho de Formatura, apresentado a Escola Politécnica da Universidade de São Paulo para obtenção do título de Engenheiro Mecânico com ênfase em Mecatrônica e Sistemas Mecânicos

Área de Concentração:
Engenharia Mecânica com ênfase em Mecatrônica e Sistemas Mecânicos

Orientador:
Prof. Dr. Marcos de Sales Guerra Tsuzuki

São Paulo
2006

FICHA CATALOGRÁFICA

Kuroiwa, Cesar Henrique Keiti Kuroiwa
Lima, Alexandre Marques
Modelagem 3D de um pulmão animado / A.M. Lima, C.H.K. Kuroiwa. -- São Paulo, 2006.
80 p.

Trabalho de Formatura - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos.

1.Pulmão (Modelagem) 2.Terceira dimensão 3.CAD I.Kuroiwa, César Henrique Keiti II.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos III.t.

AGRADECIMENTOS

Agradecemos a Deus por ter nos dado saúde estrutura para podermos chegar onde estamos chegando hoje. Sem Ele com certeza não poderíamos ter chegado tão longe.

Agradecemos também aos nossos familiares que sempre nos deram muito apoio em nossas noites de sono mal dormidas e sempre estavam presentes nos momentos em que mais precisávamos.

Agradecimentos especiais para nosso orientador professor doutor Marcos de Sales Guerra Tsuzuki, que com muita paciência e dedicação nos orientou e ensinou não apenas durante o projeto, mas já anteriormente em nossas aulas. Ao professor doutor Fabio Kawaoka Takase que juntamente com Tsuzuki nos deu ânimo, forças e inúmeros conselhos para que pudéssemos concluir nosso projeto da maneira mais completa possível.

Por fim agradecemos aos nossos amigos, em especial: Carlos Augusto Ferreira Fernandes (Chacal), Ana Eun Lee (Ana Lee), Bruno de Lima Pinheiro (Tato), Leonardo Pereira Monte (França), Rafael José Passalacqua (Fox), e às nossas belas namoradas (Cintia e Leticia) que mesmo não tendo participado diretamente no projeto, participam diretamente em nossas vidas com importância fundamental dando-nos suporte e apoio emocional para atravessarmos os momentos difíceis.

Never send a human to do a machine's job.
(Agent Smith – The Matrix)

RESUMO

Este trabalho consiste na modelagem de um pulmão animado em 3 dimensões. O objetivo principal é a visualização detalhada do pulmão humano em movimento, visto que existe uma grande dificuldade em se obter esta visualização de modo direto por meio de técnicas tradicionais como raios-X e ressonância magnética. Com relação aos raios-X, a sua natureza nociva ao corpo humano impede que ela seja utilizada por um período prolongado. A ressonância magnética é uma técnica lenta permitindo aquisitar apenas animações em 2 dimensões.

A reconstrução e animação do modelo tridimensional serão feitas com base em seqüências de imagens obtidas por ressonância magnética. O procedimento consiste basicamente em duas etapas. Na primeira, serão extraídos os contornos do movimento do pulmão em imagens 2D, sendo que cada seqüência contêm um perfil e um padrão de respiração diferente. De posse desses dados, numa segunda etapa, será então usado um programa, a ser desenvolvido, que será capaz de fazer a reconstrução de um objeto (pulmão) em 3D. Para tornar essa reconstrução possível, é necessário primeiro padronizar todas as fatias 2D, para que possam ser, posteriormente, "encaixadas", formando uma malha 3D.

As informações referentes ao desenho do pulmão em 2 dimensões, tais como coordenadas dos pontos, curvas e respiração, estão armazenadas em arquivos, segundo o formato XML. A interpretação (parsing) desses dados será feita com um programa desenvolvido em C++, com o auxílio da ferramenta Visual C++ 6.0.

Além disso, na parte de programação, será usada também a biblioteca OpenGL para o tratamento de questões relacionadas à renderização de objetos em 3D e animação dos mesmos.

Palavras-chave: Modelagem. 3D. Pulmão. CAD.

ABSTRACT

This work consists of the 3D modeling of an animated lung. The main objective is the detailed visualization of the human lung in movement, because a great difficulty exists in obtaining this visualization in a direct way through traditional techniques such as X-Ray and Magnetic resonance imaging (MRI). Regarding the use X-Ray, its harming nature to the human body impedes that it is used for long period of time. Meanwhile, magnetic resonance is a slow technique which allows the acquisition of sequences of images in only 2 dimensions.

The reconstruction and animation of the three-dimensional model will be made based on sequences of images obtained by magnetic resonance. The procedure consists basically of two stages. In the first, the outline of the lung's movement will be extracted, providing us with a series of 2D images sequences, each of which containing a different patter of breathing. With all that data, in a second stage, a computer program will be developed, which will be capable of doing the reconstruction of a 3D model of the human lung. To make that reconstruction possible, it is first necessary to standardize all the 2D slices, so that they can be, later, assembled, forming a 3D mesh.

The information regarding the drawing of the lung in 2 dimensions, such as coordinates of the points, curves and breathing, are stored in files, in XML format. The interpretation (parsing) of those data it will be done with a program developed in C++, with the aid of the Visual C++ 6.0 development tool.

Besides, in the programming part, it will also be used the library OpenGL for the purpose of handling issues such as the rendering of 3D objects and their animation.

Keywords: Modeling. 3D. Lung. CAD.

LISTA DE FIGURAS

SUMÁRIO

1. INTRODUÇÃO

A motivação para a realização deste trabalho é no campo medicinal, o conhecimento mais aprofundado dos movimentos do pulmão e das características físicas deste como volume, área superficial, divisões internas. O que torna o pulmão um órgão diferente dos demais é o fato de não possuir movimento próprio, sendo que seu movimento é uma conseqüência dos vários outros músculos ao seu redor (coração e diafragma, por exemplo). Isso faz com que o movimento pulmonar não possa ser visto com o tórax do paciente aberto, durante uma cirurgia, sendo assim necessário recorrer a métodos indiretos, como por exemplo, a computação gráfica, que é o caso neste trabalho.

No campo da engenharia, o conhecimento de técnicas de modelagem e de CAD é extremamente importante.

Primeiramente, são utilizados documentos XML para armazenar todas as informações relevantes tiradas a partir de imagens de ressonância magnética. Dentre essas informações, podem-se citar os valores da função de respiração, as coordenadas de diversos pontos e as curvas formadas por eles.

Os dados armazenados são então usados para traçar diversas imagens poligonais das fatias do pulmão, sejam elas sagitais (obtidas de uma vista lateral) ou coronais (obtidas de uma vista frontal). Esses polígonos podem ser então "encaixados" para formar uma malha contendo a silhueta do pulmão humano, a qual posteriormente dará origem a um sólido B-REP.

1.1 INTRODUÇÃO À RESSONÂNCIA MAGNÉTICA

Imagem de ressonância magnética (MRI) é uma técnica de imagem usada principalmente no universo médico para produzir imagens de alta qualidade do interior do corpo humano. MRI está baseada nos princípios de ressonância magnética nuclear (NMR), uma técnica spectroscópica usada por cientistas para obter informações química e física sobre moléculas. A técnica foi chamada de imagem de ressonância magnética ao invés de imagem de ressonância magnética nuclear (NMRI) por causa das conotações negativas associadas com a palavra "nuclear". O termo "nuclear" também não é o mais

correto, uma vez que causa confusão com radioatividade e não há radiação ionizante nesse método.

1.2 HISTÓRICO SOBRE RESSONÂNCIA MAGNÉTICA

Felix Bloch e Edward Purcell foram premiados com o Prêmio de Nobel em 1952, pois descobriram o fenômeno da ressonância magnética em 1946. No período entre 1950 e 1970 foi desenvolvido NMR e utilizado apenas para análises moleculares químicas e físicas.

Em 1971 Raymond Damadian mostrou que o relaxamento magnético nuclear consegue diferenciar tecidos e tumores, motivando assim os cientistas para considerar ressonância magnética para descoberta de doenças. Em 1973 a tomografia computadorizada baseada em raio-x (o CT) foi introduzida por Hounsfield. Esta data é importante à linha do tempo da MRI porque mostrou que hospitais estavam dispostos a gastarem grandes quantias de dinheiro para equipamentos de imagem médica. A Imagem de ressonância magnética foi demonstrada primeiramente em amostras de tubo de teste pequenas no mesmo ano por Paul Lauterbur. Em 1975, Richard Ernst propôs imagem de ressonância magnética que utiliza codificação de fase e freqüência, e transformada de Fourier. Esta técnica é a base das técnicas de MRI atuais. Alguns anos depois, em 1977, Raymond Damadian demonstrou a MRI chamada de campo-focalizado de ressonância magnética nuclear. Neste mesmo ano, Peter Mansfield desenvolveu a técnica chamada de eco-planar imagem (EPI). Esta técnica será melhorada em anos posteriores para produzir imagens a taxas de vídeo (30 ms / imagem).

Edelstein e pesquisadores demonstraram uma imagem do corpo que utiliza a técnica de Ernst, em 1980. Uma única imagem poderia ser adquirida em aproximadamente cinco minutos por esta técnica. Antes das 1986, o tempo de imagem foi reduzido a aproximadamente cinco segundos, sem sacrificar muito a qualidade de imagem. No mesmo ano, estavam desenvolvendo o microscópio de NMR que permitiu aproximadamente 10 micômetros de resolução em amostras de aproximadamente um centímetro. Em 1987 EPI foi

usado para mostrar imagens de em tempo real de um único ciclo cardíaco. Neste mesmo ano Charles Dumoulin estava aperfeiçoando angiografia de ressonância magnética (MRA) que permitiu imagem de sangue corrente sem o uso de agentes de contraste.

1.3 PRINCÍPIO BÁSICO DE FUNCIONAMENTO

A técnica fundamenta-se em três etapas: alinhamento, excitação e detecção de radiofreqüência. O alinhamento se refere à propriedade magnética de núcleos de alguns átomos, que tendem a se orientar paralelamente a um campo magnético (como uma bússola em relação ao campo magnético da terra). Por razões físicas e pela abundância, o núcleo de hidrogênio (próton) é o elemento utilizado para produzir imagens de seres biológicos (seres humanos). Assim, para que esses átomos sejam orientados numa certa direção, é necessário um campo magnético intenso – habitualmente cerca de 1,5 Teslas (30 mil vezes mais intenso que o campo magnético da terra). A etapa seguinte é a excitação. Sabe-se que cada núcleo de hidrogênio "vibra" numa determinada freqüência proporcional ao campo magnético em que está localizado. Assim, em 1,5 T, o hidrogênio tem freqüência de 63,8 MHz. O aparelho emite então uma onda eletromagnética nessa mesma freqüência. Existe uma transferência de energia da onda emitida pelo equipamento para os átomos de hidrogênio, fenômeno conhecido como ressonância. Depois desta fase a imagem é produzida. Esta é a terceira etapa: detecção de radiofreqüência. Quando os núcleos de hidrogênio receberam a energia, tornaram-se instáveis. Ao retornar ao estado habitual, eles emitem ondas eletromagnéticas na mesma freqüência (63,8 MHz – faixa de ondas de rádio). Então o equipamento detecta essas ondas e determina a posição no espaço e a intensidade da energia. Essa intensidade é mostrada como "brilho" na imagem, sendo utilizada a nomenclatura "intensidade de sinal". Dependendo da forma e do tempo em que excitamos os átomos, as imagens poderão ser mais sensíveis a diferentes propriedades dos tecidos (Figura 1.1). Por exemplo, temos as imagens T2, nas quais líquidos (liquor), desmielinização e áreas de edema no tecido cerebral se mostram mais claros – alto sinal. Nas imagens T1,

a substância branca é mais clara que a cinzenta e áreas com alto conteúdo protéico e tecido adiposo em geral tem maior sinal - mais claras.

As imagens de RM têm maior capacidade de demonstrar diferentes estruturas no cérebro e têm facilidade em demonstrar mínimas alterações na maioria das doenças. As alterações morfológicas são mais facilmente avaliadas do que na Tomografia Computadorizada (TC), bem como há maior sensibilidade para doenças desmielinizantes e processos infiltrativos. É também possível avaliar estruturas como hipocampos, núcleos da base e cerebelo (o qual é de difícil avaliação na TC) – em alguns casos necessários para pesquisa de transtornos mentais. O aparelho é na verdade um túnel com cerca de 1,5 a 2,5 metros de comprimento e produz um ruído durante a emissão das ondas de radiofreqüência e procedimento de localização do sinal. Esse ambiente é limitante para claustrofobos, contraindicado para pacientes com marca-passo e "clips" de aneurismas (há outras contra-indicações formais).

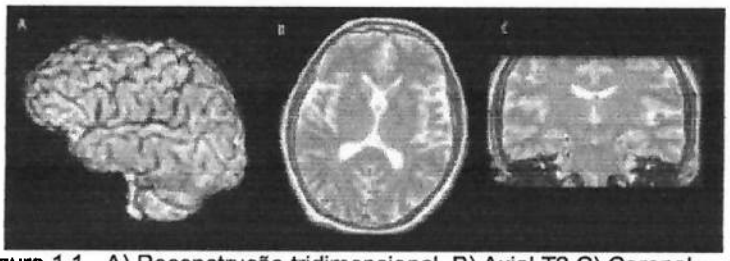

Figura 1.1 - A) Reconstrução tridimensional, B) Axial T2 C) Coronal T2.

1.4 RESSONÂNCIA MAGNÉTICA FUNCIONAL

A técnica de ressonância magnética funcional – RMf – é semelhante a um exame clínico dessa modalidade. As diferenças principais se devem à particularidade de se obter informações relativas à determinada função cerebral. Neste sentido, é necessário que haja uma forma controlada para executar essa função, por exemplo, fluência verbal. Isto se faz necessário devido à característica fundamental de exames de neuroimagem funcional: comparação entre dois (ou mais) "estados cognitivos" do cérebro. Essa comparação é feita por meio de métodos computacionais com técnicas estatísticas complexas para analisar as imagens – o que faz com que o resultado do estudo seja conhecido somente após algumas horas. O princípio da RMf é a oxigenação sangüínea. Em áreas com maior atividade neuronal, há oferta de oxigênio maior que o consumo local. Isto causa um aumento da concentração regional de hemoglobina saturada de oxigênio (oxi-hemoglobina). Essa molécula tem propriedades magnéticas diferentes da hemoglobina não saturada (desoxi-hemoglobina). Assim, utilizando técnicas especiais (seqüências BOLD) podemos observar pequenas variações da intensidade do sinal devidas à ativação cerebral. É possível apresentar estímulos visuais, auditivos, sensitivos e mesmo olfativos e gustativos.

A principal vantagem é a possibilidade de repetir várias vezes cada estudo no mesmo paciente, já que não há radiação ionizante ou necessidade de injeção de contraste. A realização do exame é feita de modo a obter imagens do cérebro durante a execução da atividade que se quer estudar e outras imagens controle, onde essa tarefa não é executada. Desta forma o indivíduo realiza uma série de atividades enquanto o aparelho adquire as imagens, as quais serão analisadas posteriormente. Exemplificando, suponha que o estudo seja para avaliar quais as áreas cerebrais se correlacionam com a tarefa de fluência verbal. Inicialmente, durante 30 segundos, o indivíduo observa letras apresentadas visualmente numa tela. A orientação é gerar palavras que se iniciem com a letra apresentada. Nos 30 segundos seguintes são apresentadas palavras, que devem ser simplesmente lidas (imagens controle). Essas tarefas são repetidas, num total de cinco ciclos, durante os quais são adquiridas cerca de cem imagens de todo o cérebro (uma a cada três segundos).

Uma outra técnica – RMf relacionada a eventos – permite maior resolução temporal e flexibilidade, mas está além do escopo do presente artigo. Após a análise, são mostradas as áreas que apresentaram aumento do sinal de RM no momento de geração das palavras em relação às imagens adquiridas durante o controle (leitura passiva). A Figura 1.2 mostra o resultado desse tipo de exemplo, onde áreas do lobo frontal esquerdo, da porção superior do lobo temporal e do lobo parietal deste lado mostram correlação com a tarefa de fluência verbal. Atualmente, as aplicações são principalmente em pesquisa. A RMf, potencialmente, poderá ser utilizada como dado adicional para planejamento cirúrgico ou para avaliar o impacto de determinado procedimento terapêutico no desempenho do paciente em determinada função cognitiva.

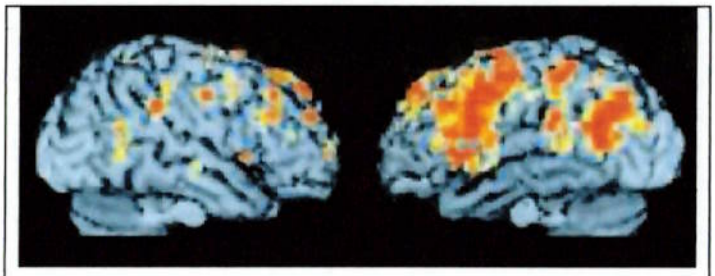

Figura 1.2 - Imagem de Ressonância Magnética Funcional

2. IMAGENS REAIS

As imagens a serem analisadas vieram de exames de ressonâncias magnéticas. Existem dois ângulos diferentes das imagens analisadas:

- Coronal: é a vista de frente do pulmão (figura 2.1).

- Sagital: é a vista de perfil do pulmão (figura 2.2).

É fundamental sabermos a diferença entre estas, uma vez que toda a modelagem se dá em torno destas duas vistas. A partir destas imagens reais, foram mapeados diversos pontos e curvas, bem como uma função de respiração para cada seqüência animada. Estes dados foram então armazenados em documentos no formato XML para cada seqüência diferente. Após isso e tendo a função respiração, foi feita a reconstrução do pulmão 3D com base nos dados armazenados e na função respiração. O algoritmo usado para se fazer esta reconstrução será descrito com mais detalhes mais adiante. A Figura 2.3 ilustra uma imagem coronal, junto com os pontos e curvas que são mapeados no documento XML.

Figura 2.1 – Imagem Sagital

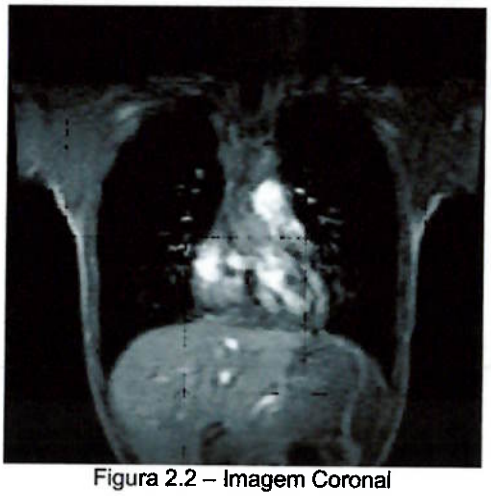

Figura 2.2 – Imagem Coronal

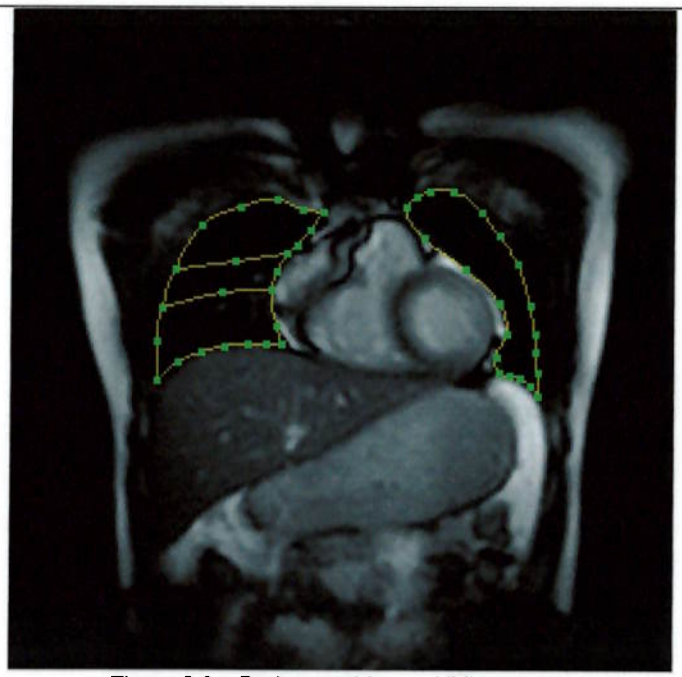

Figura 2.3 – Dados contidos no XML

3. DESCRIÇÃO DOS DADOS EM FORMATO XML

Basicamente, cada arquivo XML contém dois tipos de informações referentes a uma dada seqüência de imagens, que pode ser convertida em uma animação.

3.1. RESPIRAÇÃO

A primeira informação é a função respiração associada à seqüência em questão. Os dados estão contidos como subelementos de um elemento chamado <basewave>, cada um contendo um valor referente à respiração em um dado instante. Pode-se notar inclusive certa periodicidade nestes valores, o que evidencia uma respiração normal. Abaixo temos um trecho de um elemento <basewave> e alguns valores:

```
<basewave b="48" i="48" e="7" fx="71" fy="119" lx="71" ly="169" w0="36">
        <p value="-21" />
        <p value="-21" />
        <p value="-21" />
        <p value="-21" />
        <p value="-21" />
        <p value="-21" />
        <p value="-21" />
        <p value="-22" />
        <p value="-22" />
        <p value="-22" />
        <p value="-22" />
        <p value="-22" />
        <p value="-22" />
        <p value="-21" />
        <p value="-19" />
        <p value="-15" />
        <p value="-19" />
        <p value="-20" />
        <p value="-20" />
        ....
</basewave>
```

3.2. IMAGEM DO PULMÃO

A segunda informação refere-se à própria imagem do pulmão, representada apenas por um conjunto de pontos que compõem um contorno.

Primeiramente são definidos elementos <areas>. Uma seqüência pode conter duas ou uma áreas, dependendo desta ser coronal (visão frontal do tórax), onde se pode ver ambos os pulmões, ou sagital (visão de perfil do tórax), onde apenas é possível enxergar a lateral de um pulmão.

Cada área é composta por curvas (<curve>). As curvas possuem um atributo "cyclic" que pode assumir apenas os valores "0" ou "1", sendo a curva aberta, começa e termina em pontos distintos, ou fechada, começa e termina no mesmo ponto.

Por fim, cada curva é, logicamente, composta por uma série de pontos. Estes são representados pelos diversos elementos <point>. Por tratar-se de uma animação em 2D, cada elemento <point> possui diversos subelementos denominados <frame>, cada um contendo atributos "x" e "y", representando, portanto, as coordenadas daquele ponto em um dado instante da seqüência animada. Já para o caso de curvas acíclicas, estas possuem pontos inicial e final dentro de outra curva cíclica anteriormente definida. Tal fato é representado pelo mesmo elemento <point>, porém com atributos "ci", para indicar o índice de uma curva, e "pi", indicando um ponto da curva "ci". Abaixo, pode-se ver uma seqüência com 3 curvas, sendo a primeira fechada e as outras duas, abertas:

```
<curve cyclic="1">
        <point>
        <point>
        <point>
        <point>
        <point>
        <point>
        <point>
        <point>
        <point>
        <point>
        <point>
        <point>
        <point>
        <point>
        <point>
        <point>
        <point>
        <point>
```

```
        <point>
        <point>
        <point>
        <point></curve>
<curve cyclic="0">
        <point r="1" ci="0" pi="10" />
        <point>
        <point r="1" ci="0" pi="18" />
</curve>
<curve cyclic="0">
        <point r="1" ci="0" pi="19" />
        <point>
        <point r="1" ci="0" pi="8" />
</curve>
```

Tendo esses dados, é possível então formar uma seqüência respiratória de uma fatia separada do pulmão para cada documento XML. Cada fatia é formada basicamente por 2 ou 3 curvas, sendo uma principal e fechada, representando o contorno externo do pulmão, e 1 ou 2 curvas menores, abertas, que formam as divisões interlobulares do pulmão.

A partir daí, podemos passar para a construção de um modelo 3D do pulmão humano, fazendo-se a junção de diversas fatias (objetos bidimensionais).

4. RECONSTRUÇÃO DO PULMÃO 3D

A reconstrução tridimensional do pulmão é feita da seguinte forma:

Para se reconstruir uma imagem tridimensional do pulmão, é necessário fazer o "encaixe" de diversas fatias do pulmão, sejam elas coronais ou sagitais. Como essas fatias não pertencem necessariamente à mesma respiração, deve-se fazer uma padronização, ou seja, escalamento, de modo a fazer com que todas elas estejam compatíveis umas com as outras para o encaixe.

4.1 ESCALAMENTO DAS FATIAS

Cada ponto da fatia, obtido dos documentos XML, é descrito por suas coordenadas x e y (2D) em cada instante de tempo de uma dada seqüência respiratória. Tendo isso, é possível definir uma direção de movimento para cada ponto e associar uma escala a uma específica posição deste ponto ao longo desta direção. A partir daí, tem-se então uma associação entre uma escala numérica e a forma de uma determinada fatia do pulmão.

Foram definidos dois métodos de encaixe das fatias do pulmão para que se possa fazer a reconstrução do modelo tridimensional. São eles, o escalamento por dimensão de aresta vertical e por coordenada Y inferior. Ambos os métodos, requerem primeiramente que se tenha uma fatia base, sobre a qual todas as outras serão montadas. É importante ressaltar que a fatia base e as demais devem ser de tipos distintos, isto é, sobre uma fatia base sagital são montadas fatias coronais, e vice-versa.

4.1.1 ENCAIXE POR DIMENSÃO DE ARESTA VERTICAL

Neste modo de escalamento, traça-se primeiro uma reta vertical imaginária cortando as fatias, duas a duas, no ponto de intersecção. A partir daí, toma-se uma medida da aresta vertical da fatia base e tentamos encontrar uma escala que faz com que a aresta vertical da fatia a ser encaixada seja

igual à primeira. Por fim, é feita uma translação da fatia secundária para esta cruze a fatia base.

Abaixo, na listagem 4.1, pode-se ver os dados obtidos utlizando-se esse método de encaixe:

Coronal Slices

Slice#	Scale	Translation	EdgeSize	MinY	MaxY
0	0.06	1.75	81.66	0.14	81.80
1	-0.04	-0.25	84.63	5.58	90.21
2	0.04	4.23	87.57	3.92	91.50
3	-0.08	-0.29	90.77	10.51	101.28
4	-0.03	2.28	94.12	9.37	103.50
5	-0.05	1.20	97.64	11.40	109.04
6	-0.09	-1.86	101.40	15.00	116.40
7	-0.05	0.77	105.37	12.66	118.03
8	-0.23	-3.17	109.83	16.67	126.50
9	-0.13	0.62	114.57	12.70	127.27
10	-0.09	0.16	119.02	12.70	131.72
11	-0.08	-1.31	122.89	13.36	136.26
12	-0.05	3.61	126.35	7.30	133.65
13	-0.09	-1.70	129.63	11.06	140.69
14	-0.12	-1.59	132.45	8.99	141.44
15	-0.16	-2.94	134.47	8.01	142.48
16	-0.24	-5.27	135.99	7.66	143.66
17	-0.22	-1.89	137.14	1.32	138.46
18	-0.27	-3.91	137.82	0.02	137.83

Sagittal Slices 1

Slice#	Scale	Translation	EdgeSize	MinY	MaxY
0	0.04	8.08	61.47	-12.55	48.92
1	-0.47	-4.97	70.15	4.71	74.86
2	-0.14	0.04	76.21	2.10	78.31
3	-0.14	0.00	81.66	1.89	83.55
4	-0.26	-3.92	85.67	4.52	90.20
5	-0.29	-5.78	86.61	5.30	91.92

Sagittal Slices 2

Slice#	Scale	Translation	EdgeSize	MinY	MaxY
0	-0.11	6.49	103.38	-10.51	92.88
1	-0.20	4.00	117.41	-5.15	112.26
2	-0.12	1.79	128.18	-2.53	125.65
3	-0.14	0.00	137.82	-3.90	133.92
4	-0.07	1.01	145.04	-8.77	136.26
5	-0.22	-4.92	151.41	-8.27	143.14

Listagem 4.1 – Dados Obtidos por Encaixe por Dimensão de Aresta Vertical

4.1.2 ENCAIXE POR COORDENADA Y INFERIOR

Este método se assemelha muito ao anteriormente descrito, com a diferença de que, ao invés de se igualar a dimensão da aresta vertical, faz-se o mesmo com a coordenada y inferior.

Abaixo, na listagem 4.2, temos os dados referentes a esse método de encaixe:

Coronal Slices

Slice#	Scale	Translation	EdgeSize	MinY	MaxY
0	0.03	0.00	80.70	1.07	81.77
1	-0.01	0.00	85.69	4.47	90.16
2	-0.04	-0.00	84.23	7.26	91.49
3	-0.05	-0.00	91.97	9.29	101.26
4	-0.06	-0.00	92.78	10.70	103.48
5	-0.05	-0.00	97.43	11.61	109.04
6	-0.05	0.00	104.42	12.14	116.55
7	-0.04	-0.00	105.63	12.40	118.03
8	-0.10	0.00	113.92	12.44	126.35
9	-0.11	0.00	115.04	12.22	127.27
10	-0.06	-0.00	120.01	11.72	131.73
11	-0.04	0.00	125.29	10.88	136.17
12	-0.08	-0.00	123.64	9.70	133.34
13	-0.04	-0.00	132.65	8.12	140.77
14	-0.08	0.00	135.43	6.12	141.55
15	-0.08	0.00	138.71	3.77	142.48
16	-0.10	0.00	142.50	1.07	143.57
17	-0.14	-0.00	140.28	-1.92	138.36
18	-0.13	-0.00	143.13	-5.25	137.88

Sagittal Slices 1

Slice#	Scale	Translation	EdgeSize	MinY	MaxY
0	-0.25	-0.00	53.94	-5.22	48.72
1	-0.24	0.00	74.46	-1.02	73.44
2	-0.12	-0.00	77.19	1.37	78.56
3	-0.12	0.00	82.57	1.07	83.64
4	-0.12	-0.00	91.82	-0.30	91.52
5	-0.07	-0.00	94.73	-1.42	93.30

Sagittal Slices 2

Slice#	Scale	Translation	EdgeSize	MinY	MaxY
0	-0.20	0.00	99.54	-5.99	93.55
1	-0.25	0.00	115.26	-2.90	112.36
2	-0.12	-0.00	127.91	-2.27	125.64
3	-0.12	0.00	139.23	-5.25	133.98
4	-0.07	-0.00	145.38	-9.11	136.27

5	-0.11	-0.00	157.35	-14.21	143.14

Listagem 4.2 – Dados Obtidos por Encaixe por Coordenada y Inferior

4.1.3 ANÁLISE DOS RESULTADOS

Abaixo, encontra-se um gráfico (Gráfico 4.1) com uma comparação entre os dois métodos para uma determinada fatia. No eixo das abscissas, temos o instante de tempo, enquanto que nas ordenadas, temos a escala calculada segundo cada um dos métodos.

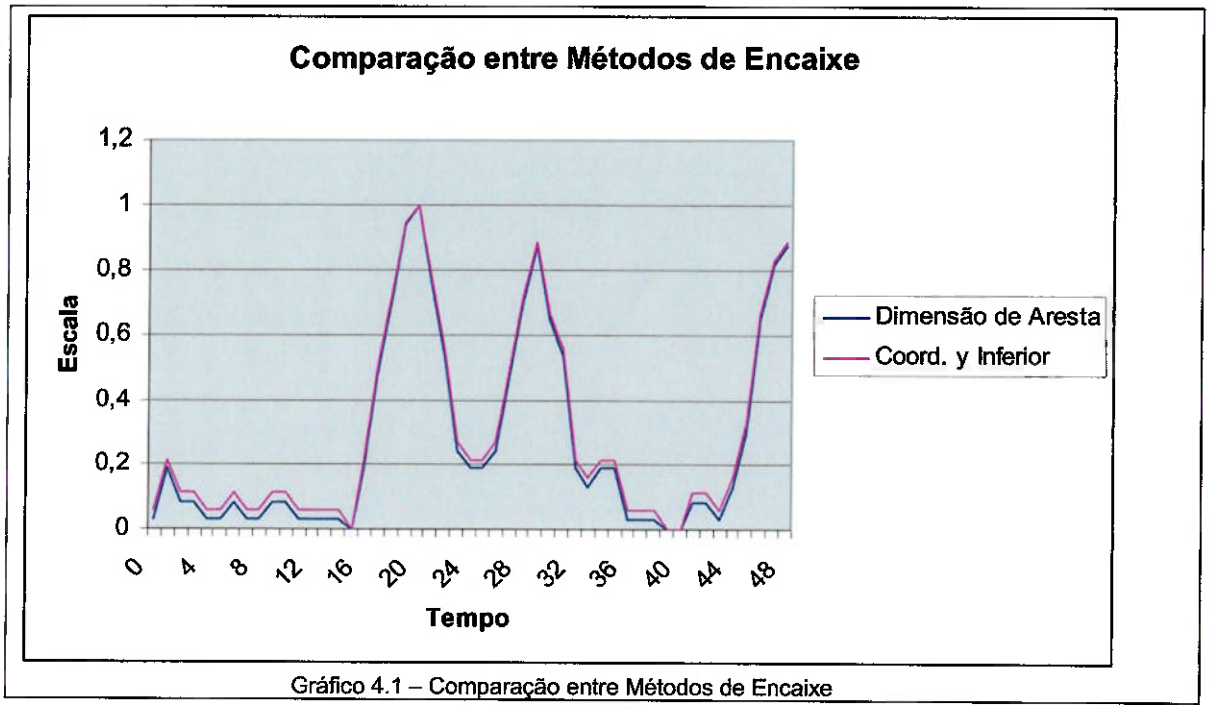

Gráfico 4.1 – Comparação entre Métodos de Encaixe

Conforme pode ser facilmente observado, tantos pelas listagens como pelo gráfico, a diferença na escala entre os dois métodos é muito pequena, podendo então ser utilizado qualquer um sem que o resultado final do modelo fique gravemente prejudicado.

4.2 CONSTRUÇÃO DO MODELO 3D

Antes de iniciar o processo, é preciso que se tenha uma fatia especifica que será usada como base para o encaixe das demais. Podemos chegar assim a mais 2 métodos de montagem do modelo tridimensional, de acordo com o tipo da fatia base.

4.2.1 FATIA BASE SAGITAL

Neste método, o programa parte de uma fatia do tipo sagital para iniciar o encaixe das demais fatias (coronais). Feito isso, o programa seleciona todas as fatias coronais que cortam esta fatia central e estas são enfileiradas, dando inicio a silhueta do pulmão 3D. Para que se tenha uma modelagem suave, os pontos já conhecidos são multiplicados por uma escala, o que torna as dimensões de cada fatia compatíveis entre si, evitando assim diferenças bruscas entre uma fatia e sua vizinha (lembre-se de que as fatias não foram obtidas de uma mesma respiração). Esta escala deve ser calculada por qualquer um dos métodos descritos anteriormente, "dimensão de aresta vertical" ou "coordenada y inferior". A figura 4.1 ilustra essa série de fatias coronais.

Uma vez com as fatias coronais, deve-se fazer um processo semelhante com as fatias sagitais. Ao invés de se utilizar uma fatia central como padrão, desta vez são usadas a primeira e última fatias coronais da lista previamente montada. Para cada uma das duas, repete-se o procedimento de enfileirar as fatias sagitais, encaixando-as devidamente na fatia coronal base através de uma nova escala calculada para cada fatia sagital. Uma diferença importante, no entanto, é a realização de um corte, o qual deixa visível apenas a área externa a fila de fatias coronais. O resultado pode ser observado na figura 4.2.

Figura 4.1 – Fatias Coronais

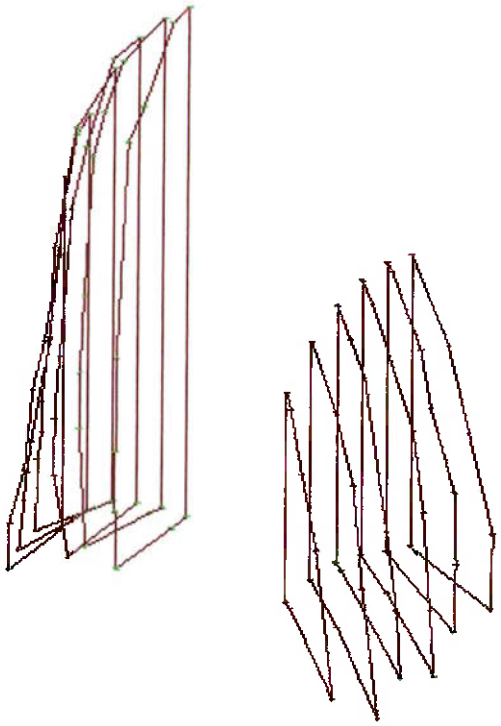

Figura 4.2 – Fatias Sagitais Cortadas

Abaixo, na figura 4.3, podemos ver o resultado em uma ilustração com ambas as fatias coronais e sagitais juntas.

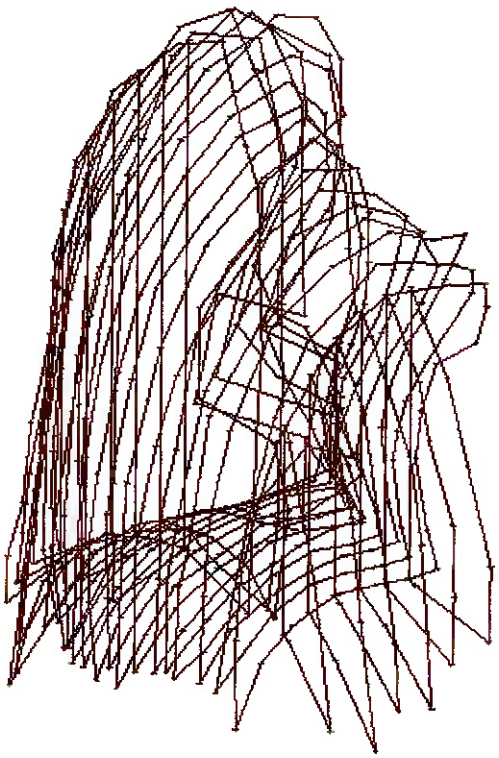

Figura 4.3 – Silhueta Composta do Pulmão

4.2.2 FATIA BASE CORONAL

O processo agora é inverso ao descrito acima, ou seja, usa-se uma fatia base coronal, sobre a qual são enfileiradas as fatias sagitais. Este processo consiste também em usar uma escala, calculada, sobre cada fatia sagital, para que esta se encaixe adequadamente sobre a coronal base.

Em seguida são encaixadas as fatias coronais sobre a primeira e última fatias sagitais, também devidamente escaladas e cortadas, como no método anterior.

As figuras 4.4, 4.5 e 4.6 ajudam a ilustrar esse processo:

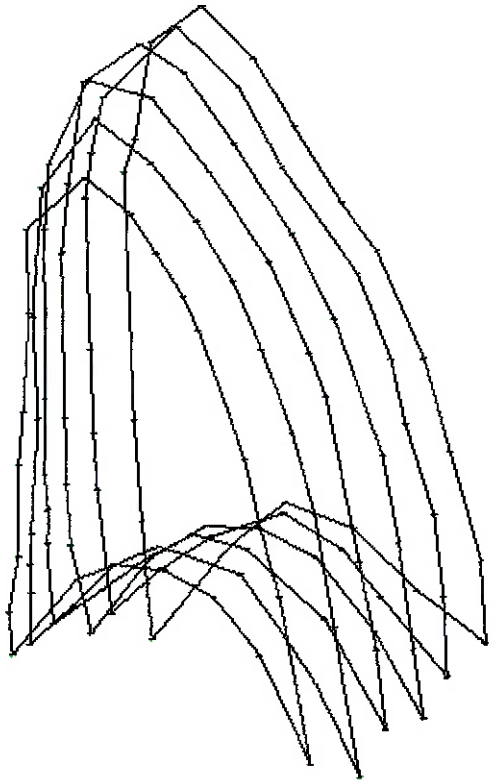

Figura 4.4 – Fatias Sagitais

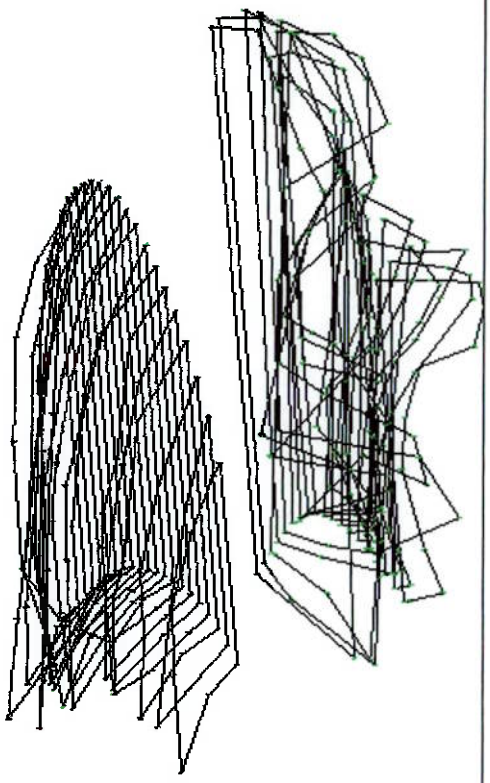

Figura 4.5 – Fatias Coronais Cortadas

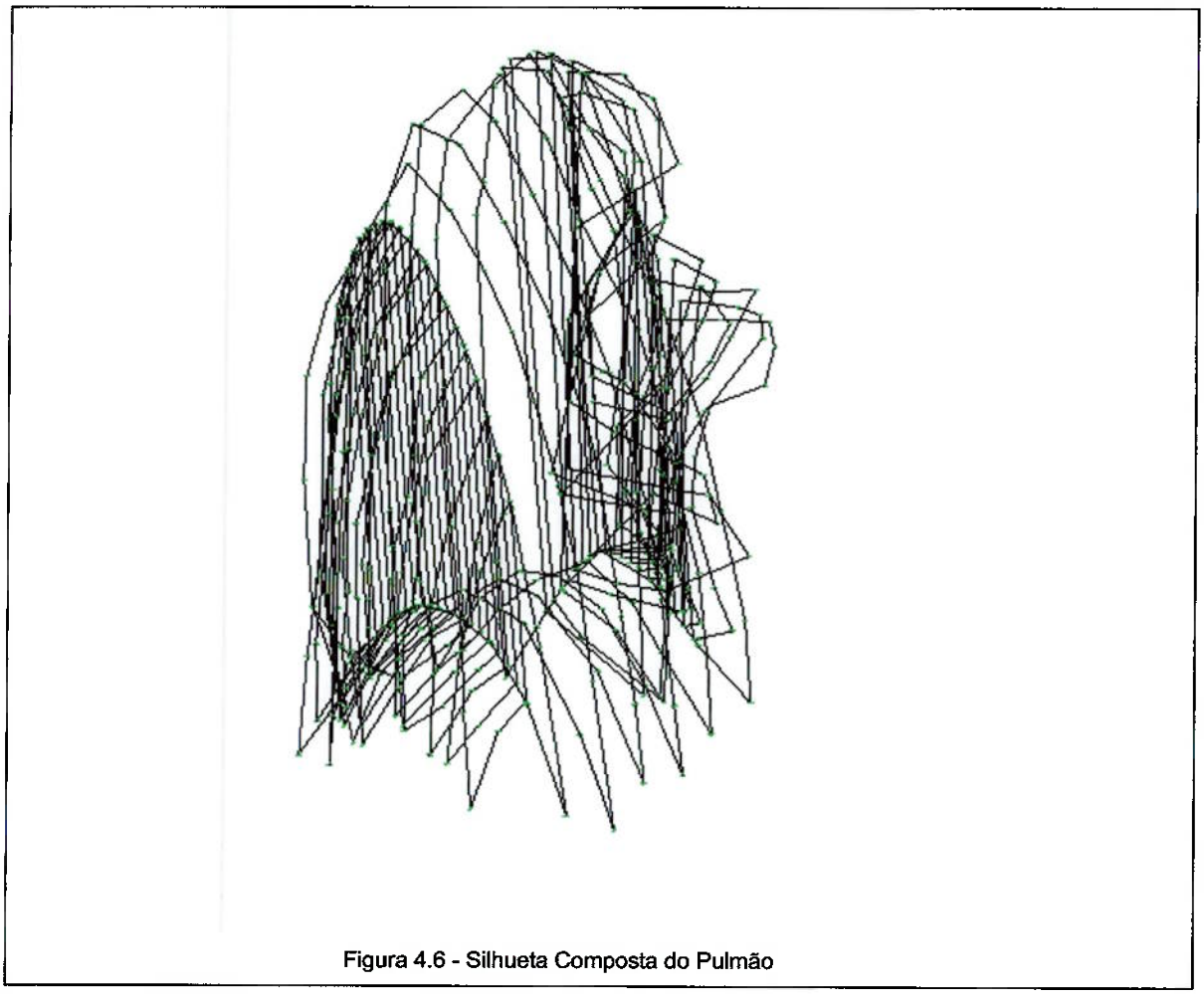

Figura 4.6 - Silhueta Composta do Pulmão

5. DIVISÕES INTERLOBULARES

Os pulmões podem ser ainda subdivididos em estruturas chamados lóbulos, o qual se acredita que possuam um movimento independente entre si. Este é um dos fenômenos que se pretende observar com uma modelagem 3D animada do órgão. O pulmão direito encontra-se dividido em três lóbulos enquanto que o esquerdo em dois.

Assim sendo, além de traçar um contorno da silhueta externa do pulmão, como foi mostrado logo acima, devem ser traçadas também as linhas que formam as divisões entre os lóbulos, isto é, as divisões interlobulares. Uma amostra deste tipo de curva pôde ser vista anteriormente, na figura 3. Porém como pode ser visto na mesma ilustração, trata-se de uma curva aberta e formada por apenas 3 pontos, sendo que dois deles são referências a pontos já pertencentes à curva externa do pulmão. Neste caso, então, esses 3 pontos foram usados como base para se traçar uma curva de Bézier, ao invés de serem simplesmente ligado, como foi feito anteriormente.

Algumas alterações no código tiveram de ser feitas para tornar possível a exibição destas novas curvas. Na classe TState, foi necessário adicionar duas novas estruturas de dados para armazenar os novos polígonos. A listagem a seguir contém a classe TState com suas alterações em destaque:

```
#include "ynupolyg.h"

class TState {
  public:
    TState() {};
    ~TState() {};

    float getTimeL(void) const { return timel; };
    float getTimeF(void) const { return timef; };
    float getMedxL(void) const { return medxL; };
    float getMedyL(void) const { return medyL; };
    float getMedxF(void) const { return medxF; };
    float getMedyF(void) const { return medyF; };
    float getSize(void) const { return size; };

    void setTimeL(float et) { timel = et; };
    void setTimeF(float et) { timef = et; };
    void setMedxL(float exl) { medxL = exl; };
    void setMedyL(float eyl) { medyL = eyl; };
    void setMedxF(float exf) { medxF = exf; };
    void setMedyF(float eyf) { medyF = eyf; };
    void setSize(float es) { size = es; };

    void pushF(TPolygon pp) { lstPolyF.push_back(pp); };
    void pushL1(TPolygon pp) { lstPolyL1.push_back(pp); };
    void pushL2(TPolygon pp) { lstPolyL2.push_back(pp); };

       /**
        * Pushes an interlobular polygon
```

```
         */
        void pushInterLob(TPolygon pp, int curveIndex)
        {
                if (curveIndex == 1) {
                        lstPolyInterLob1.push_back(pp);
                } else if (curveIndex == 2) {
                        lstPolyInterLob2.push_back(pp);
                }
        }

    vector<TPolygon> *getPolyF(void) { return &lstPolyF; };
    vector<TPolygon> *getPolyL1(void) { return &lstPolyL1; };
    vector<TPolygon> *getPolyL2(void) { return &lstPolyL2; };

        /**
         * Gets the list of interlobular polygons
         */
        vector<TPolygon> *getPolyInterLob(int curveIndex)
        {
                if (curveIndex == 1) {
                        return &lstPolyInterLob1;
                } else if (curveIndex == 2) {
                        return &lstPolyInterLob2;
                }
                return NULL;
        }

    TPolygon & getPolyF0(void) { return lstPolyF[0]; };
    TPolygon & getPolyFN(void) { return lstPolyF[lstPolyF.size() - 1]; };

  private:
    float timef, timel, size;
    float medxL, medyL, medxF, medyF;

    vector<TPolygon> lstPolyF;
    vector<TPolygon> lstPolyL1;
    vector<TPolygon> lstPolyL2;

        /**
         * Vector with interlobular polygons
         */
        vector<TPolygon> lstPolyInterLob1;
        vector<TPolygon> lstPolyInterLob2;

};
```

Além disso, no programa principal, a função analyzingFrontalImages, a qual já foi descrita em detalhes anteriormente, também sofreu algumas alterações, como mostrado abaixo:

```
/**
 * Gets the interlobular polygon(s)
 */
TAreasInTime *currentArea = FSlices[k]->getArea(area);
int nInterLobCurves = currentArea->getNumberOfCurves() - 1;
TPolygon interLob[2];
for (int i = 0; i < nInterLobCurves; i++) {
        FSlices[k]->calculateS3DPolygon(area, i + 1, scale4);
        interLob[i] = FSlices[k]->getPolygon(area, i + 1);
        /**
         * Translates and pushes interlobular polygon into state
         */
        interLob[i].translate(0, - maxxF + maxxL);
        state->pushInterLob(interLob[i], i + 1);
}
```

Com isso, o resultado obtido pode ser observado na Figura 5.1, utilizando-se o modelo 3 d com fatia base sagital (divisões sobre as coronais), e

5.2, com fatia base coronal (divisões sobre as sagitais), logo abaixo (note que as imagens das fatias sagitais não foram exibidas para facilitar a visualização das divisões interlobulares, em vermelho):

Figura 5.1 – Divisões Interlobulares (Fatias Coronais)

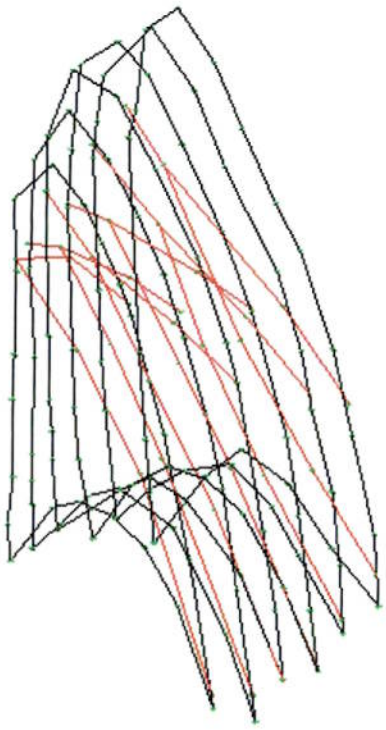

Figura 5.2 – Divisões Interlobulares (Fatias Sagitais)

6. PROGRAMA AUXILIAR PARA VISUALIZAÇÃO DO MOVIMENTO DAS FATIAS (LUNGPLOT)

Foi iniciado também o desenvolvimento de um software auxiliar, com o intuito de permitir a visualização de características de cada fatia, separadamente, para cada valor de escala. Com isso, é possível ver claramente a posição de cada ponto e o formato e tamanho da fatia separada em uma escala desejada. O programa foi desenvolvido utilizando-se a linguagem Java, principalmente devido à facilidade que esta proporciona no desenho de interfaces gráficas ao usuário. Para tal foi utilizada a biblioteca Swing. A este programa, foi dado o nome de LungPlot versão 1.

A figura 6.1 ilustra a interface gráfica deste programa.

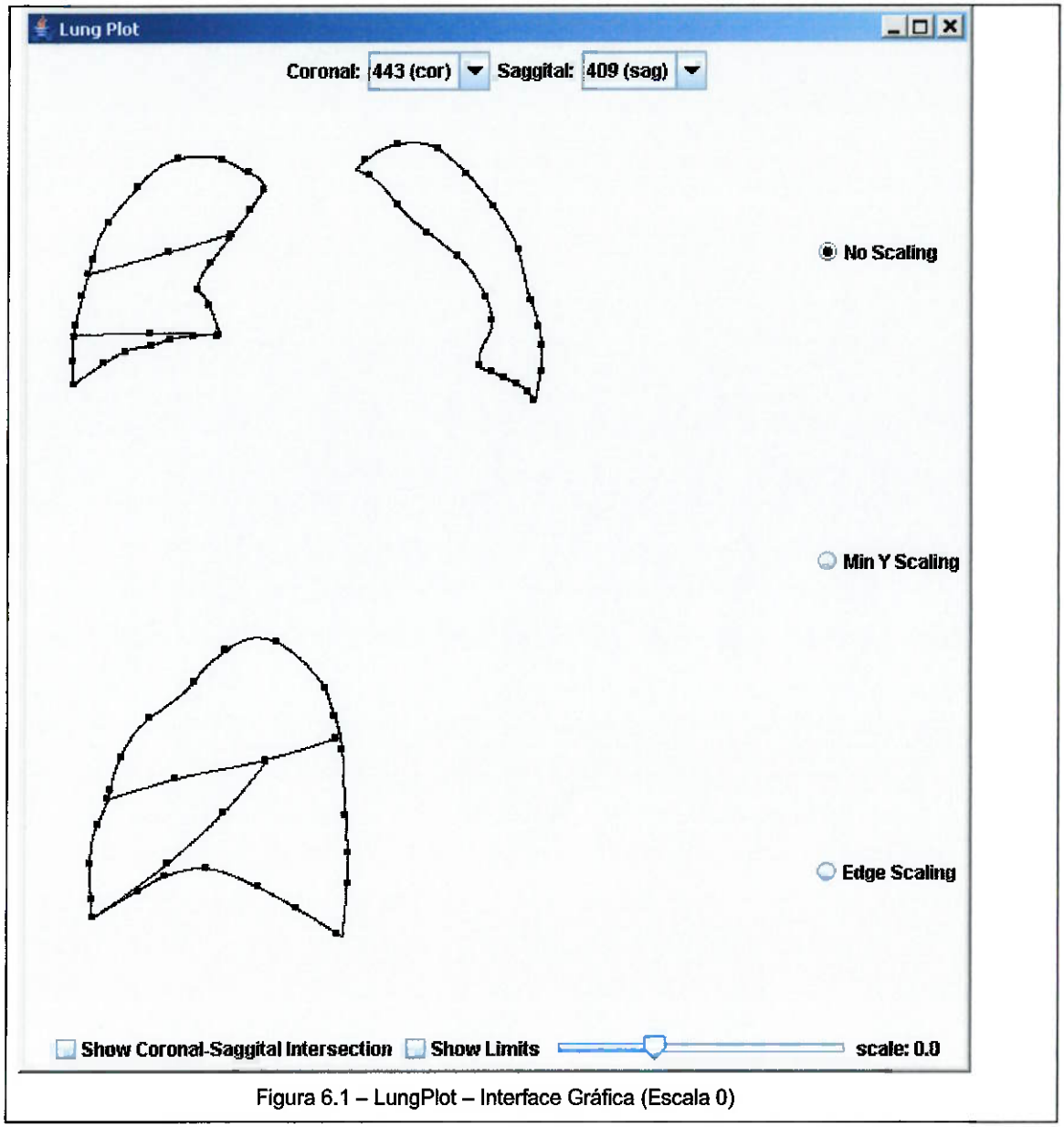

Figura 6.1 – LungPlot – Interface Gráfica (Escala 0)

Conforme pode ser visto, a aplicação consiste basicamente em dois controles do tipo Combo Box, no topo da janela, onde o usuário pode escolher dentre as fatias coronais e sagitais disponíveis para plotagem.

No centro, temos a exibição das imagens selecionadas, sendo as fatias coronais (pulmão esquerdo e direito) na parte superior, e a sagital logo abaixo.

Na parte inferior da janela, há uma barra de rolagem, através da qual se pode variar o fator de escala dos pontos o que afeta imediatamente o desenho mostrado. Na figura 6.1, acima, temos a imagem de uma fatia sagital e coronal,

para uma escala 0, enquanto que logo abaixo, na figura 6.2, pode-se ver a diferença, quando variamos a escala para 1.15.

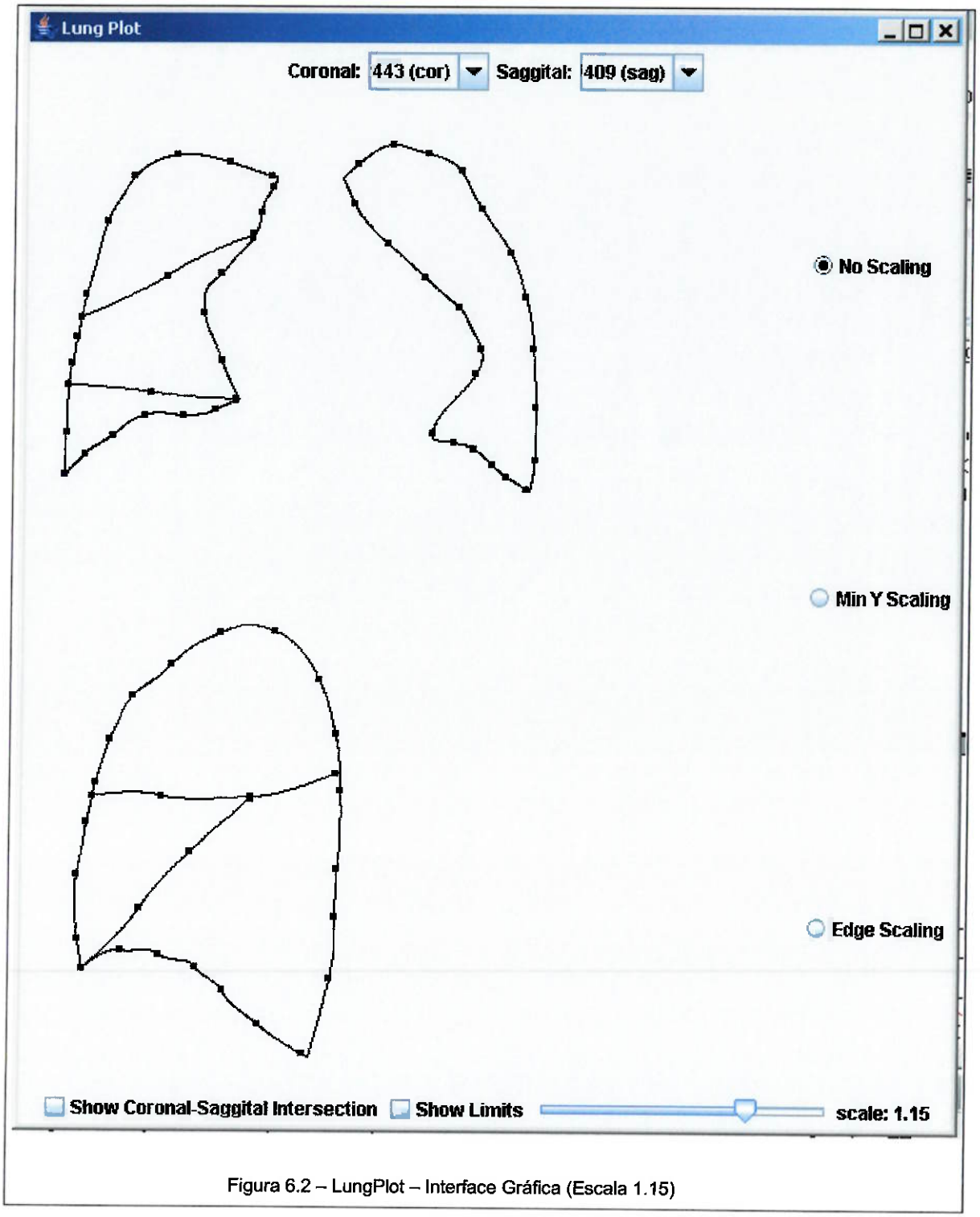

Figura 6.2 – LungPlot – Interface Gráfica (Escala 1.15)

Uma das primeiras contribuições que o programa proporcionou, foi a percepção de que para escalas muito pequenas (menores que -0.5, principalmente), o formato da fatia do pulmão começa a ficar extremamente deformado, perdendo assim o sentido em utilizá-las a partir deste valor. Um exemplo deste fenômeno pode ser visto abaixo, na figura 6.3.

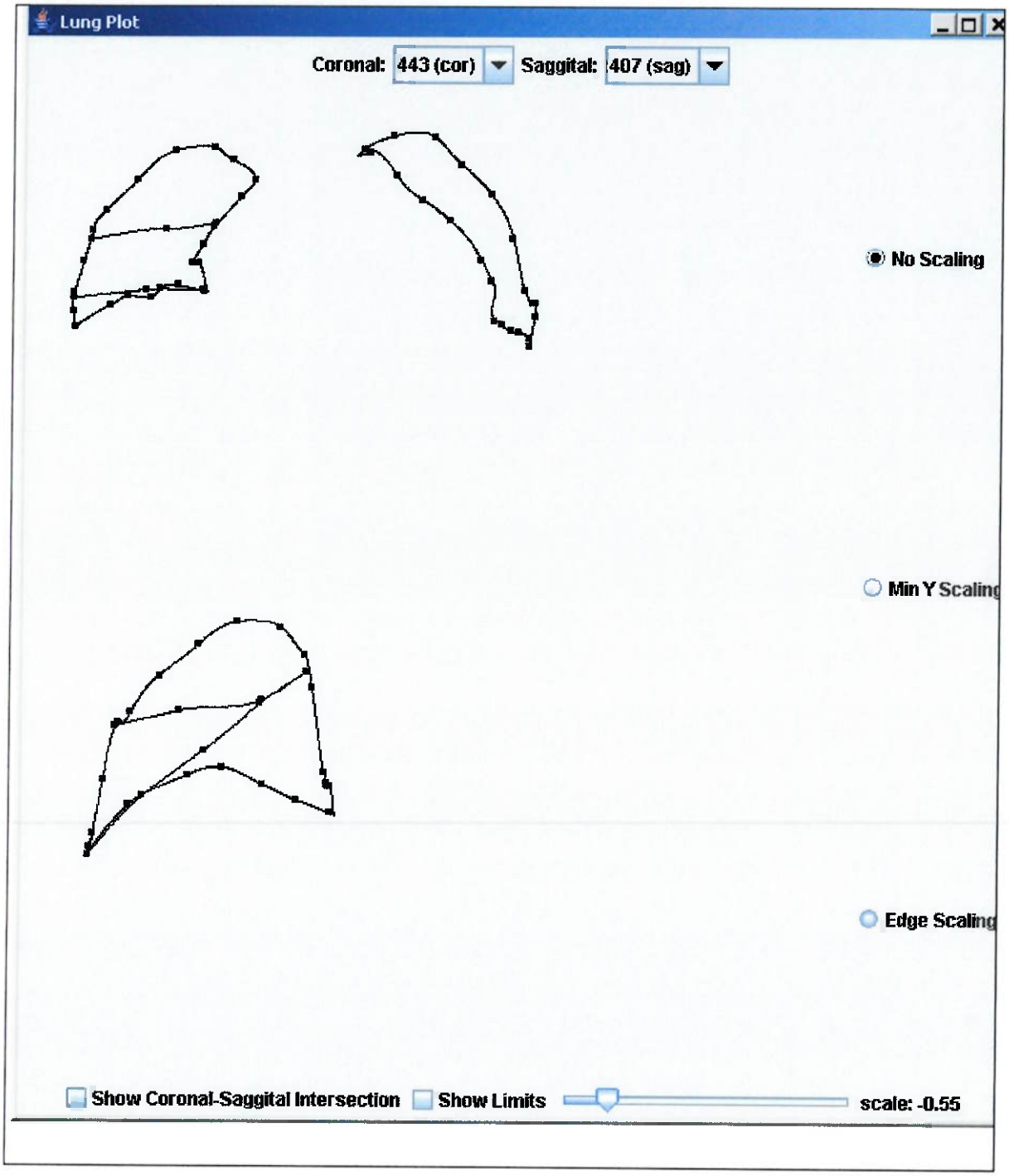

Figura 6.3 – Imagens Deformadas em escala -0.55

Além disso, existem ainda uma série de funcionalidades implementadas nesta aplicação, e que permitem uma análise de alguns detalhes importantes das imagens das fatias do pulmão. A seguir, apresentaremos uma descrição detalhada de cada uma destas funcionalidades.

6.1 INTERPOLAÇÃO POR CURVAS DE BEZIER

Primeiramente, deve-se ressaltar que todas as curvas plotadas são aproximadas por Bezier. Para se obter uma curva de Bezier entre dois pontos, são necessários certo número de pontos de controle, sendo no mínimo dois, os quais definem o ponto de partida e de chegada da curva. Demais pontos entre os dois primeiros são usados para determinar a direção da curva, portanto não são pontos pertencentes à curva.

A seguir, nas figuras 6.4, 6.5 e 6.6, temos um exemplo rápido de curvas de Bezier que possuem 2, 3 e 4 pontos de controle, respectivamente.

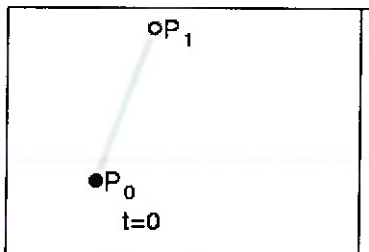

Figura 6.4 – Bezier (2 pontos de controle)

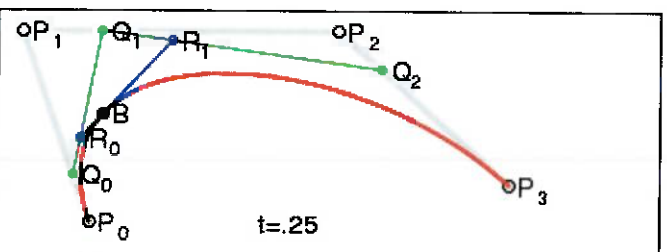

Figura 6.5 – Bezier (3 pontos de controle)

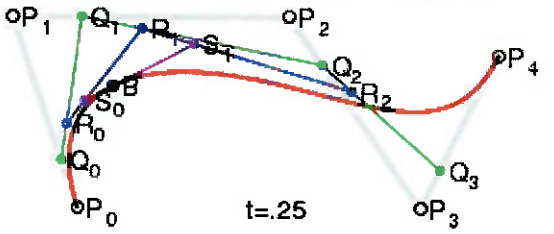

Figura 6.6 – Bezier (4 pontos de controle)

No programa, a cada dois pontos consecutivos, é feita uma curva de Bezier entre os mesmos, porém, para isto, é necessário encontrar pontos de controle entre os dois primeiros. Para conseguir estes pontos de controle, são usados, adicionalmente, o ponto imediatamente anterior ao ponto de partida e o imediatamente posterior ao de chegada, totalizando em quatro pontos, com os quais é possível definir a direção da curva nos pontos inicial e final.

Uma vez obtidos os pontos de controle, basta obter uma série de pontos da curva de Bezier, variando um parâmetro t de 0.0 a 1.0, segundo a seguinte equação:

$$\mathbf{B}(t) = \mathbf{P}_0(1-t)^3 + 3\mathbf{P}_1 t(1-t)^2 + 3\mathbf{P}_2 t^2(1-t) + \mathbf{P}_3 t^3 \ , \ t \in [0,1].$$

O número de pontos a serem obtidos para traçar a curva está claramente definido no programa LungPlot por uma constante de nome BEZIERPOINTS, na classe Slice, como mostrado no trecho logo abaixo:

```
// Number of Points in the Bezier Interpolation
 public static final int BEZIERPOINTS = 50;
```

Atualmente, estamos trabalhando com 50 pontos, o que permite uma curva bastante suave e que não sobrecarregue demais a capacidade de processamento do computador, lembrando que quanto maior o número de pontos, mas custosa será a operação de desenho da curva. Portanto, para computadores mais simples ou mais avançados, talvez seja recomendável variar o valor desta constante entre 10 e 100.

6.2 EXIBIÇÃO DOS LIMITES DE MOVIMENTO

Há também um "checkbox" intitulado de "Show Limits" que quando marcado, irá ativar a exibição dos limites da fatia em questão, que neste caso serão quando a escala tiver o valor de -1.0 e +2.0. Assim pode-se ter uma idéia de toda a amplitude do movimento da fatia numa respiração.

Esta funcionalidade está ilustrada na figura 6.7, logo a seguir. Nesta figura, podemos ver as fatias escaladas com o limite máximo em azul e no limite mínimo na cor rosa. A imagem principal, com a escala atual, ainda permanece no desenho, em preto.

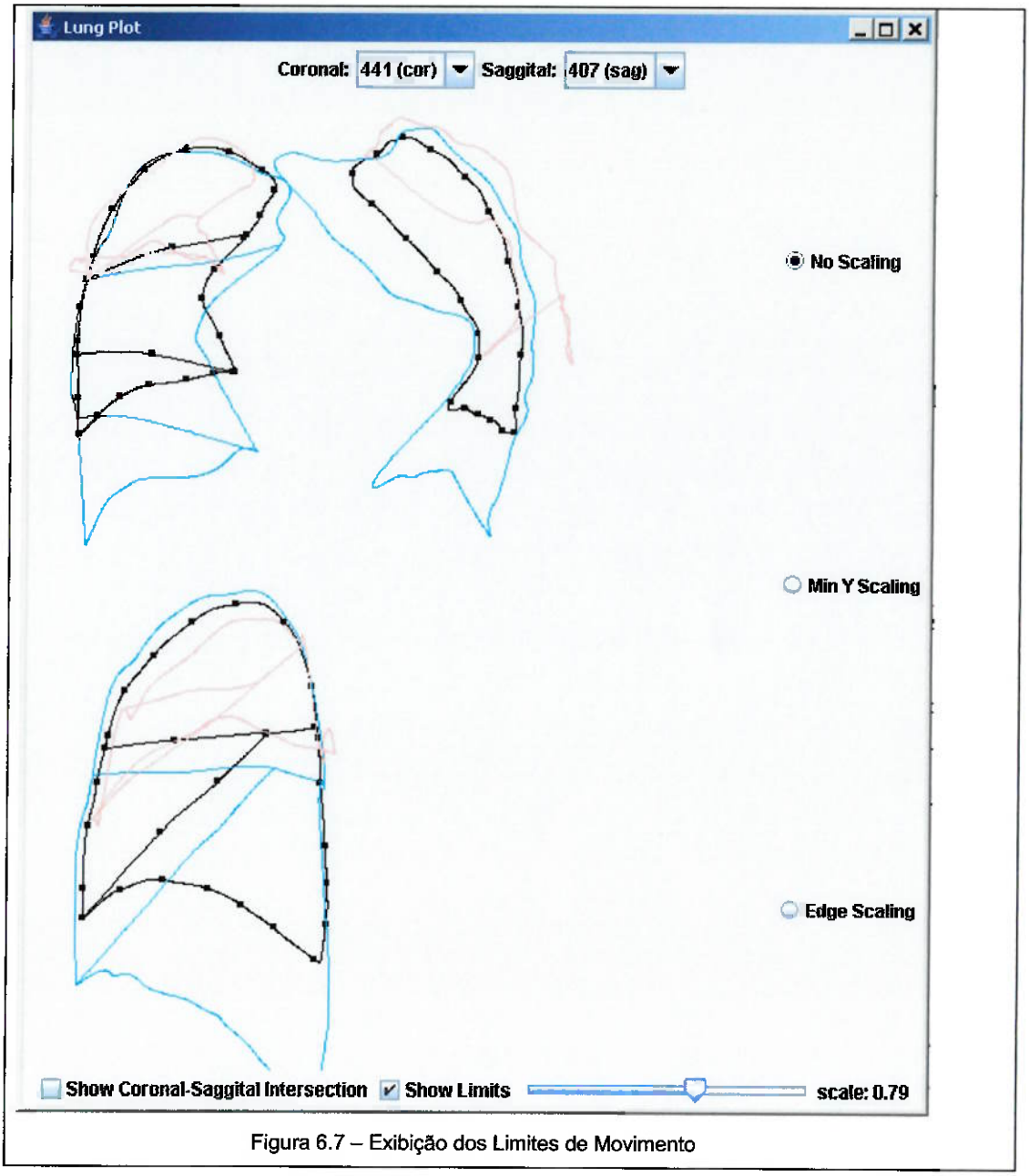

Figura 6.7 – Exibição dos Limites de Movimento

6.3 EXIBIÇÃO DAS INTERSECÇÕES ENTRE SAGITAIS E CORONAIS

As funcionalidades que mais complexas de se implementar e talvez, as que deram resultados mais concretos, foram a exibição das intersecções entre fatias sagitais e coronais, e sua associação aos diferentes métodos de escalamento entre elas, que será descrito na seção 6.4.

Ao lado do "checkbox" "Show Limits", existe um outro "checkbox", intitulado "Show Coronal-Saggital Intersection", o qual, quando clicado, ativa imediatamente a exibição de uma série de pontos coloridos sobre as imagens sagital e coronal que já estiverem plotadas. A figura 6.8 ilustra esse novo modo de exibição.

Na figura, temos uma série de pontos coloridos, cada um com um significado específico. Primeiramente, deve-se observar que todos os pontos podem ser divididos em grupos de quatro ao longo de uma linha vertical, tanto para aqueles sobre a fatia coronal quanto sobre a sagital. Esses grupos de quatro representam cada um, a intersecção de uma fatia do tipo oposto sobre a fatia principal, isto é, na imagem superior, coronal, os 6 grupos de 4 pontos coloridos correspondem à intersecção das 6 fatias sagitais disponíveis, e vice-versa.

Com relação às cores, azul, verde e amarelo, a função delas é simplesmente diferenciar entre as curvas a qual se referem (curva externa, interlobular superior e interlobular inferior). Já os grupos de 4 pontos em vermelho, indicam os pontos de intersecção entre as duas fatias selecionadas para exibição, ou seja, na figura 6.8, estão selecionadas a fatia coronal 441 e sagital 407, portanto, os pontos vermelhos sobre a imagem coronal representam a intersecção com a fatia sagital 407, enquanto que aqueles sobre a imagem sagital, são a intersecção com a coronal 441.

Um fato importante de se observar aqui, é que para fatores de escala próximos de 0.0, os pontos das intersecções se encontram muito próximos da própria curva plotada, principalmente para o polígono principal, que representa a silhueta externa do pulmão. Porém conforme se varia o fator de escala, tanto para cima como para baixo, vemos que essas intersecções assumem uma posição aleatória, não mais semelhante à curva plotada. Este fenônemo pode ser percebido na figura 6.9, onde mostramos as mesmas fatias da figura 6.8, porém com uma escala muito maior.

No entanto, este tipo de situação é facilmente resolvido com o escalamento auxiliar das fatias secundárias, como já foi explicado anteriormente nas seções 4.1.1 e 4.1.2, aplicado ao projeto em Visual C++ em 3D. A seção 6.4 ilustra este mesmo processo de escalamente, porém como uma funcionalidade do software LungPlot.

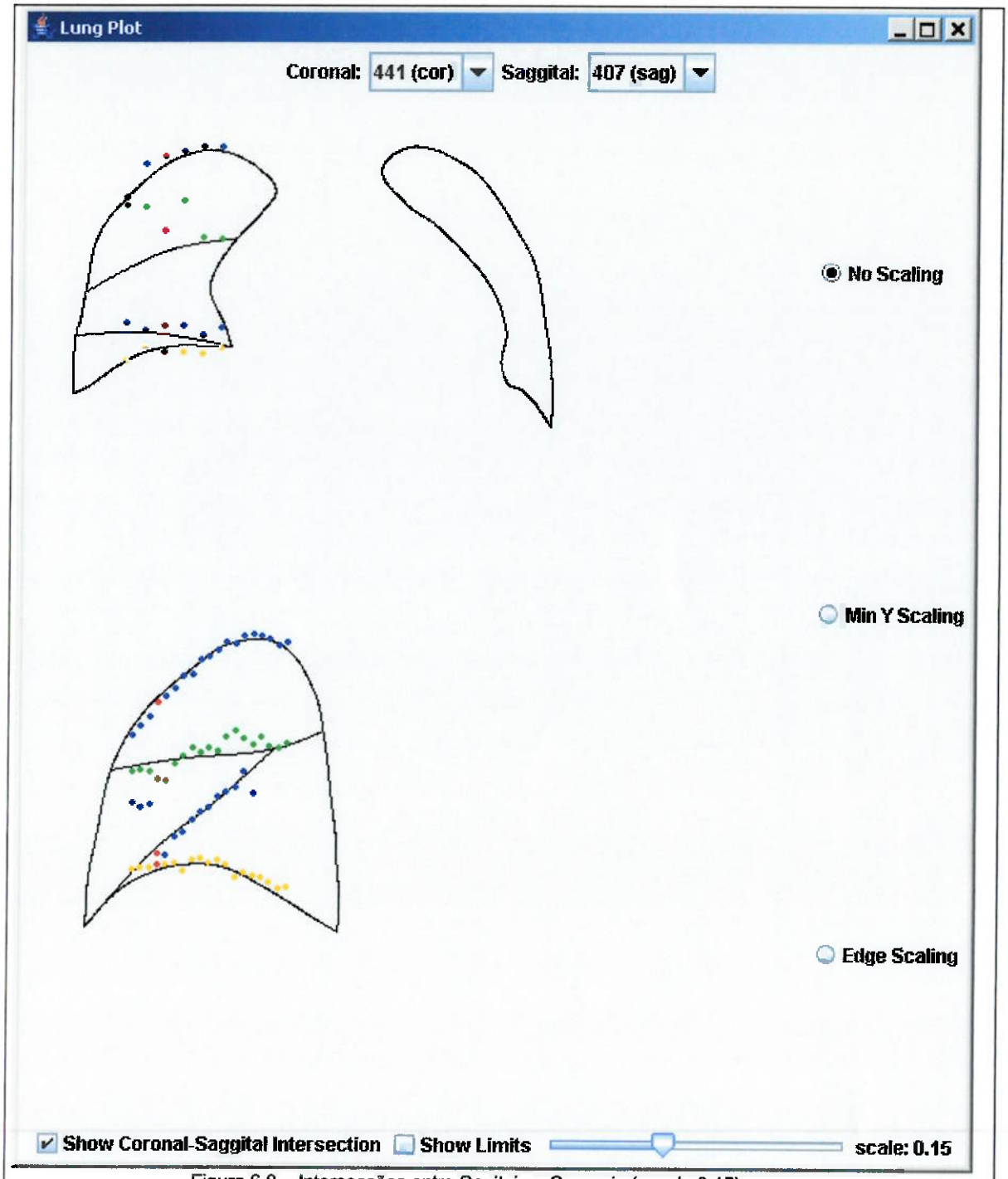

Figura 6.8 – Intersecções entre Sagitais e Coronais (escala 0.15)

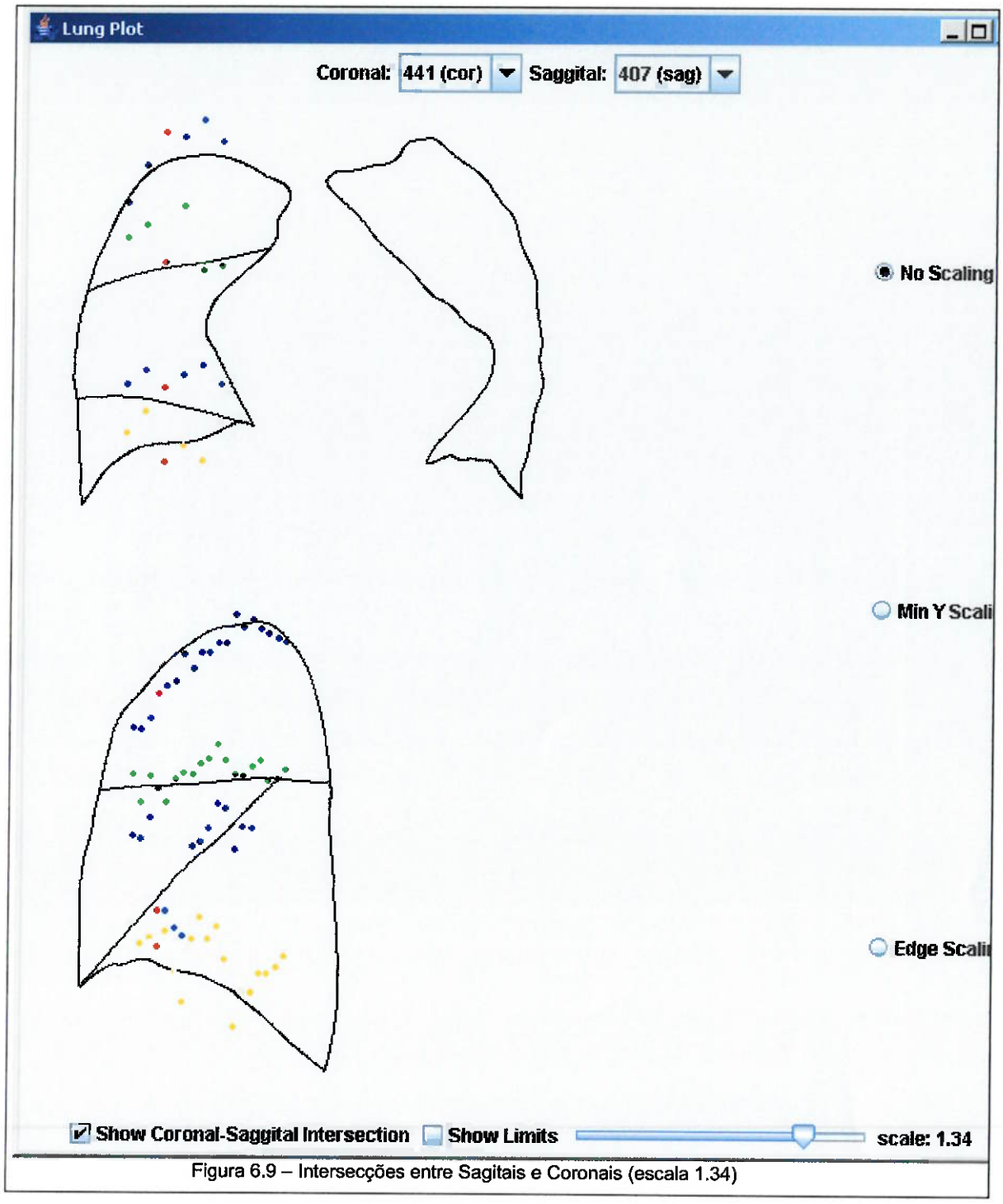

Figura 6.9 – Intersecções entre Sagitais e Coronais (escala 1.34)

6.4 DIFERENTES MÉTODOS DE ESCALAMENTO

Conforme visto na seção anterior, quando se estão sendo exibidos os pontos de intersecção entre coronais e sagitais, há um alto grau de aleatoriedade entre os pontos dos dois tipos de fatias (coronal e sagital) quando o fator de escala fica muito distante de 0.0. Também já foi comentado que isto se deve ao fato de que as fatias foram obtidas por seqüências de ressonância magnética independentes umas das outras, o que faz com que cada uma se comporte diferentemente para um valor de escala. Por isso, foram introduzidos dois métodos de encaixe descritos nas seções 4.1.1 e 4.1.2, que visam determinar uma escala diferente da escala principal para as fatias secundárias, de maneira que se consiga um melhor encaixe no polígono principal. Assim sendo, temos os botões de rádio no canto direito da janela do LungPlot, onde o usuário pode escolher entre os métodos de escalamento e visualizar o resultado imediatamente na figura plotada. Até o momento, estávamos trabalhando no modo "No Scaling", ou seja, sem escalamento nenhum, o que faz com que todas as curvas sejam escaladas pelo fator de escala principal, escolhido com a barra de rolagem, no canto inferior da janela.

Quando mudamos o modo de escalamento para "Edge Scaling", obtemos o escalamento por dimensão de aresta vertical, já descrito na seção 4.1.1. O resultado deste modo pode ser visualizado na figura 6.10.

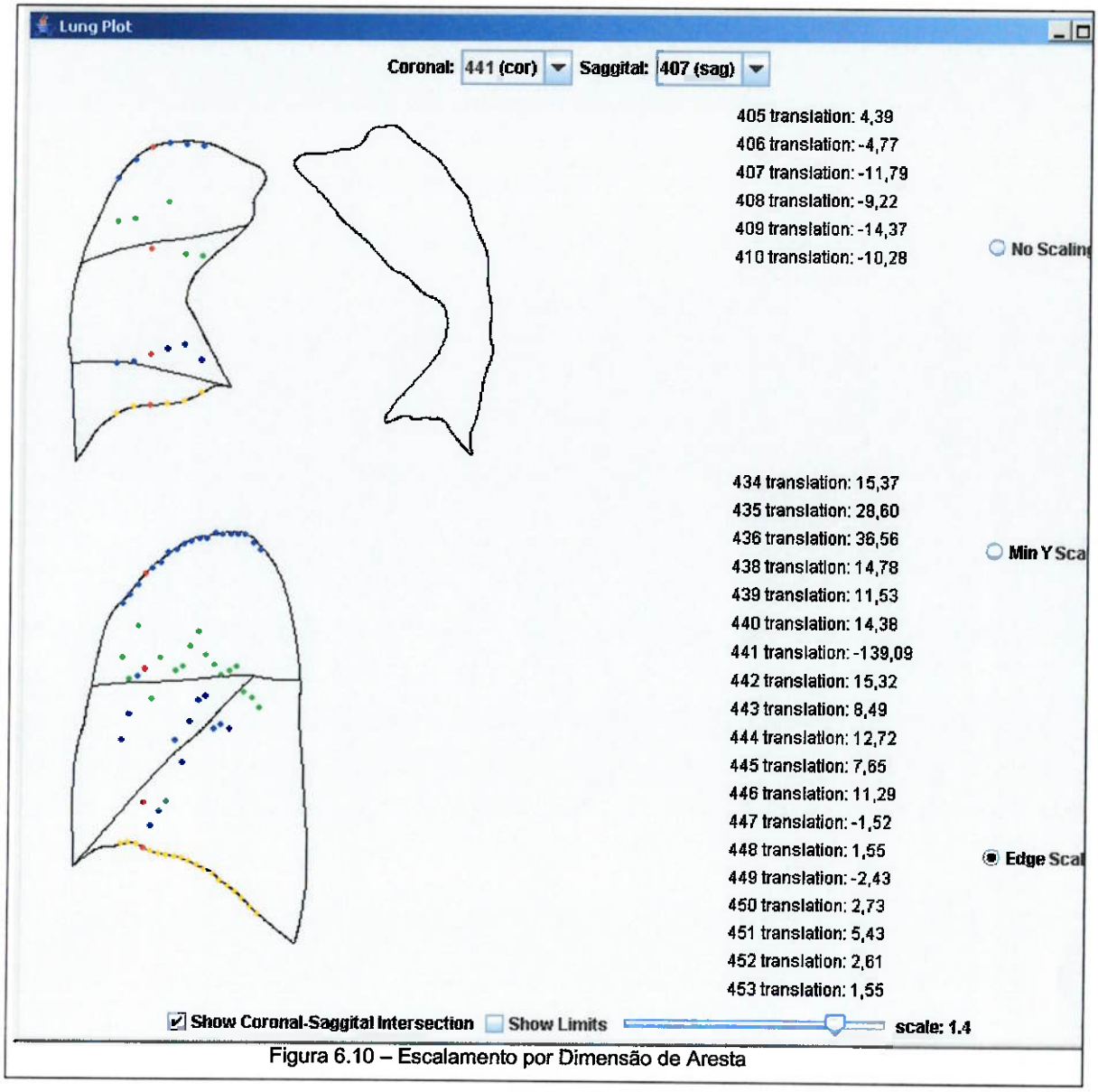

Figura 6.10 – Escalamento por Dimensão de Aresta

Conforme se pode ver facilmente na figura 6.10, mesmo para uma escala de 1.4, os pontos de intersecção inferior (amarelo) e superior (azul), estão muito próximos da curva principal. Isto se deve ao fato de que o programa encontra uma escala secundária para que ajuste as dimensões da aresta vertical da intersecção das fatias, duas a duas, e depois aplique uma translação na vertical, de forma que o ponto inferior das duas se iguale. Esta translação também é mostrada na janela do programa, para todas as fatias disponíveis.

A outra opção de escalamento também disponível é o escalamento por coordenada Y inferior, descrito na seção 4.1.2, e que pode ser ativado ao se

clicar na opção "Min Y Scaling". A figura 6.11, abaixo, ilustra o resultado obtido com esta opção.

Figura 6.11 – Escalamento por Coordenada Y Inferior

Neste modo, o aspecto mais relevante, é que apenas os pontos de intersecção inferiores (amarelo) apresentam uma grande semelhança com a fatia principal. Ao contrário do que acontecia com o escalamento por dimensão de aresta, os pontos de intersecção superiores não se aproximam tanto da curva plotada, pois neste caso, o processo somente garante que os pontos

inferiores se aproximem, não fazendo qualquer observação para a dimensão da aresta ou nenhum tipo de translação.

8. CONCLUSÃO

Pode-se dizer que os resultados obtidos nas diversas etapas do desenrolar deste projeto foram bastante satisfatórios e acabou por fornecer uma série de dados e informações importantes para a continuidade do projeto da modelagem CAD de um pulmão humano 3D animado.

Um dos principais desafios da modelagem tridimensional consistia no fato de que se partia apenas de imagens bidimensionais de várias secções diferentes do pulmão. Estas fatias eram provenientes de exames de ressonância magnética e o maior obstáculo na montagem do modelo 3D residia no fato de que cada uma destas fatias apresentava um comportamento completamente independente umas das outras. A solução encontrada para isto foi o processo de escalamento e encaixe destas fatias.

Inicialmente, existia apenas uma forma de reconstrução 3D e uma única métrica de escalamento entre as fatias, o que fornecia apenas um resultado. Adicionalmente foi implementado um novo método de reconstrução 3D e uma nova métrica de escalamento, o que permitiu ao todo 4 combinações de resultados para serem comparados e analisados.

Além disso, a exibição das divisões interlobulares tornou-se realidade no modelo 3D, o que deu-nos a chance de observar seu comportamento e constatar que, infelizmente, as linhas interlobulares das fatias do pulmão não se alinhavam perfeitamente, o que tornava impossível a criação de uma superfície interlobular coerente para o modelo tridimensional.

O programa LungPlot, desenvolvido em Java, prestou-nos grande contribuição no estudo das fatias pulmonares individualmente, algo que não era possível até então com o software em C++, o qual apenas permitia uma visualização 3D do modelo já reconstruído. Com a imagem separada das fatias, foi possível, num primeiro momento, constatar que, para determinadas escalas, a silhueta do pulmão se tornava bastante deformada, perdendo assim seu formato característico, o que de certa forma inviabilizava a utilização da mesma sob tais condições. No entanto, a maior contribuição conseguida, foi que na plotagem das intersecções entre fatias sagitais e coronais, mesmo quando

submetidas aos dois tipos de escalamento descritos. Analisando os resultados que o LungPlot nos forneceu neste aspecto, foi possível concluir que o escalamento funciona perfeitamente bem para a silhueta externa do pulmão. Porém, nas divisões interlobulares o mesmo não ocorre. O desalinhamento dos pontos de intersecção entre as curvas interlobulares das fatias sagitais e coronais evidenciou de maneira muito mais clara este problema que já havia sido percebido no modelo tridimensional. Ficou bastante claro que isso era causado pelo fato de que as métricas de escalamento utilizadas se preocupavam apenas com um perfeito alinhamento da silhueta externa do pulmão, aplicando a mesma escala para os pontos pertencentes às curvas interlobulares das fatias. Uma das possíveis soluções para este problema seria, talvez, a realização de um escalamento independente para as divisões interlobulares daquele feito para a silhueta externa, o qual garantisse também o perfeito alinhamento entre as interlobulares, e que por fim, permitisse a modelagem tridimensional destas superfícies de maneira suave o suficiente. Isso permitiria então o estudo das propriedades dos lóbulos do pulmão separadamente, tais como o cálculo do volume e área superficial, para cada instante da respiração, algo que ainda representa um desafio para a área médica.

Este projeto possibilitará ainda, uma continuação no estudo do pulmão humano e suas propriedades. Tendo validado os métodos de escalamento para a reconstrução do pulmão em 3D com o LungPlot. O próximo passo neste momento seria tentar reconstruir o pulmão utilizando novas técnicas como, por exemplo, utilizando fatias horizontais.

9. REFERÊNCIAS BIBLIOGRÁFICAS

[1] TSUZUKI, M.S.G.; TAKASE, F.K.; GOTOH, T.; KAGEI, S.; ASAKURA, A.; INOUE, T.; IWASAWA, T. - 4D Lung Modelling from Sequential MRI Base don Respiratory Motion Analysis, submetido a Computer Aided Design – Janeiro/2006.

[2] TSUZUKI, M.S.G. – Introdução ao CAD/CAM, Apostila do Curso de PMR2520.

[3] SOBOTTA, J. – Atlas de Anatomia Humana, Vol. 2, Ed. GUANABARA KOOGAN, 21ª edição - 1993.

[4] STROUSTRUP, B. – The C++ Programming Language – Ed. Addison-Wesley Professional, 3rd Edition – Junho/1997

[5] Informações gerais sobre a biblioteca OpenGL em http://www.opengl.org, acessado em Abril/2006.

[6] SCHREINER, D.; WOO, M.; NEIDER, T.; DAVIS, T. – OpenGL® Programming Guide: The Official Guide to Learning OpenGL®, Ed. Addison-Wesley Professional, 5ª edição – Agosto/2005

[7] SCHREINER, D. – OpenGL® Reference Manual: The Official Reference Document to OpenGL, Ed. Addison-Wesley Professional, 4ª edição – Março/2004

[8] HORNAK, Joseph P. – The Basics of MRI – 1996

ANEXO A - LISTAGEM COMPLETA DO CÓDIGO

A seguir, a listagem completa do código comentado do programa desenvolvido em Java, devidamente dividido em classes.

CLASSE Point

Classe responsável por armazenar informações relativas a um ponto no plano (coordenadas x e y e seus respectivos Deltas).

```
/*************************************************************************/
/*                                                                       */
/* AUTHORS: Cesar Henrique Keiti Kuroiwa                                 */
/*                  Alexandre Marques Lima                               */
/*                                                                       */
/*************************************************************************/

package LungPlot;

import java.lang.Math;

// Class to store Point data: X and Y coordinates and delta
public class Point {

	// Point Coordinates
	private double x, y;

	// Point deltas
	private double deltaX, deltaY;

	public Point() {
		this.x = 0;
		this.y = 0;
		this.deltaX = 0;
		this.deltaY = 0;
	}

	public Point(double x, double y, double deltaX, double deltaY) {
		this.x = x;
		this.y = y;
		this.deltaX = deltaX;
		this.deltaY = deltaY;
	}

	public Point(Point p) {
		this.x = p.getX();
		this.y = p.getY();
		this.deltaX = p.getDeltaX();
		this.deltaY = p.getDeltaY();
	}

	public double getDeltaX() {
		return deltaX;
	}
	public void setDeltaX(double deltaX) {
		this.deltaX = deltaX;
	}
	public double getDeltaY() {
		return deltaY;
```

```
	}
	public void setDeltaY(double deltaY) {
		this.deltaY = deltaY;
	}

	public double getX() {
		return x;
	}

	public void setX(double x) {
		this.x = x;
	}

	public double getY() {
		return y;
	}

	public void setY(double y) {
		this.y = y;
	}

	// Adds two points as if two vectors
	public Point add(Point p) {
		double x = this.x + p.getX();
		double y = this.y + p.getY();
		double deltaX = this.deltaX + p.getDeltaX();
		double deltaY = this.deltaY + p.getDeltaY();
		return new Point(x, y, deltaX, deltaY);
	}

	// Multiplies a point by a scalar value
	public Point times(double s) {
		return new Point(s*x, s*y, s*deltaX, s*deltaY);
	}

	// Does scalar product between two points (vectors)
	public double scalar(Point p) {
		double result = getX()*p.getX() + getY()*p.getY();
		return result;
	}

	// Calculates the length of a vector
	public double length() {
		return Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2));
	}

	// Normalizes the vector
	public void normalize() {
		double len = length();
		x = x/len;
		y = y/len;
		deltaX = deltaX/length();
		deltaY = deltaY/length();
	}
}
```

CLASSE Polygon

Classe que contém dados de um Polígono, tais como seus pontos.

```
/*****************************************************************************/
/*                                                                           */
/* AUTHORS: Cesar Henrique Keiti Kuroiwa                                     */
/*                    Alexandre Marques Lima                                 */
/*                                                                           */
/*****************************************************************************/

package LungPlot;

import java.util.Vector;

// Class that represents a Polygon as group of points
public class Polygon {
        // Polygon points
        private Vector<Point> points;

        // cyclic flag
        private boolean cyclic;

        public Polygon() {
                points = new Vector<Point>();
                cyclic = true;
        }

        public Vector<Point> getPoints() {
                return points;
        }

        // Adds a point to the polygon
        public void pushPoint(Point p) {
                points.addElement(p);
        }

        // Returns the number of points in the polygon
        public int getSize() {
                return points.size();
        }

        public void clear() {
                points.clear();
        }

        public boolean isCyclic() {
                return cyclic;
        }

        public void setCyclic(boolean cyclic) {
                this.cyclic = cyclic;
        }

        public double getMinY(double x, double scale) {
                double minY = 1000;
                Point prev;
                Point next;
                for (int i = 0; i < points.size() - 1; i++) {
                        prev = points.get(i);
                        double prevX = prev.getX() + scale*prev.getDeltaX();
                        next = points.get(i + 1);
                        double nextX = next.getX() + scale*next.getDeltaX();
                        if (x >= prevX && x <= nextX ||
                                x <= prevX && x >= nextX) {
                                double prevY = prev.getY() + scale*prev.getDeltaY();
                                double nextY = next.getY() + scale*next.getDeltaY();
```

```
                        double tmpY = (prevY + nextY)/2;
                        minY = tmpY < minY ? tmpY : minY;
                    }
                }
                return minY;
        }

        public double getMaxY(double x, double scale) {
                double maxY = -1000;
                Point prev;
                Point next;
                for (int i = 0; i < points.size() - 1; i++) {
                        prev = points.get(i);
                        double prevX = prev.getX() + scale*prev.getDeltaX();
                        next = points.get(i + 1);
                        double nextX = next.getX() + scale*next.getDeltaX();
                        if (x >= prevX && x <= nextX ||
                                x <= prevX && x >= nextX) {
                                double prevY = prev.getY() + scale*prev.getDeltaY();
                                double nextY = next.getY() + scale*next.getDeltaY();
                                double tmpY = (prevY + nextY)/2;
                                maxY = tmpY > maxY ? tmpY : maxY;
                        }
                }
                return maxY;
        }

        public double getEdgeSize(double x, double scale) {
                return getMaxY(x, scale) - getMinY(x, scale);
        }
}
```

CLASSE Slice

Classe responsável por armazenar diversas informações relacionadas a uma fatia do pulmão, tais como suas curvas, dicom e dados de escalamento.

```
/*****************************************************************************/
/*                                                                           */
/* AUTHORS: Cesar Henrique Keiti Kuroiwa                                     */
/*                    Alexandre Marques Lima                                 */
/*                                                                           */
/*****************************************************************************/

package LungPlot;

import java.util.Vector;

// Slice class
public class Slice {
	// X and Y offset for the viewport
	private static final int X_OFFSET = 30;
	private static final int Y_OFFSET = 30;

	// Y saggital viewport offset
	public static final int SAGGITAL_Y_OFFSET = 280;

	// Number of Points in the Bezier Interpolation
	public static final int BEZIERPOINTS = 50;

	public final double SCALE_STEP = 0.025;
	public final double TOLERANCE = 1.5;

	// Image scaling factor
	private final double FACTOR = 1.0;

	// Polygons
	private Vector<Polygon> polys;

	// Bezier Polygons
	private Vector<Polygon> bezierPolys;

	// Scaling Data for all the other slices
	private Vector<ScalingData> scales;

	// DICOM information
	private Dicom dicom;

	// Slice window and viewport
	private Window window;
	private ViewPort viewPort;

	// Slice number
	private int number;

	// Slice type: "cor" or "sag"
	private String type;

	public Slice() {
		polys = new Vector<Polygon>();
		bezierPolys = new Vector<Polygon>();
		scales = new Vector<ScalingData>();
		dicom = new Dicom();
	}

	public String toString() {
		return Integer.toString(number) + " (" + type + ")";
	}
```

```
public void pushPoly(Polygon poly) {
        polys.add(poly);
}

public Vector<Polygon> getPolys() {
        return polys;
}

public void pushBezierPoly(Polygon bezierPoly) {
        bezierPolys.add(bezierPoly);
}

public Vector<Polygon> getBezierPolys() {
        return bezierPolys;
}

public void pushScalingData(ScalingData scale) {
        scales.add(scale);
}

public Vector<ScalingData> getScales() {
        return scales;
}

public Dicom getDicom() {
        return dicom;
}

public void setDicom(Dicom dicom) {
        this.dicom = dicom;
}

public Window getWindow() {
        return window;
}

public void setWindow(Window window) {
        this.window = window;
}

public ViewPort getViewPort() {
        return viewPort;
}

public void setViewPort(ViewPort viewPort) {
        this.viewPort = viewPort;
}

public int getNumber() {
        return number;
}

public void setNumber(int number) {
        this.number = number;
}

public String getType() {
        return type;
}

public void setType(String type) {
        this.type = type;
}

public void calculateViewPort() {
        viewPort = new ViewPort();
        int offsetX = X_OFFSET;
        int offsetY = Y_OFFSET;
        if (type == "sag") {
                offsetY += SAGGITAL_Y_OFFSET;
        }
        viewPort.setTop(offsetY);
        viewPort.setBottom(offsetY + window.getHeight()*FACTOR);
        viewPort.setLeft(offsetX);
        viewPort.setRight(offsetX + window.getWidth()*FACTOR);
}
```

```
        // Obtain the Polygons with Bezier Interpolation
        public void calculateBezierPolys() {
                for (int i = 0; i < polys.size(); i++) {
                        Polygon poly = polys.get(i);
                        int nPoints = poly.getSize();
                        Vector<Point> points = poly.getPoints();
                        boolean cyclic = poly.isCyclic();
                        Polygon bezierPoly = new Polygon();
                        bezierPoly.setCyclic(poly.isCyclic());
                        for (int j = 0; j < nPoints; j++) {
                                if (j == nPoints - 1 && cyclic == false) {
                                        continue;
                                }
                                int start_j = j;

                                int prev_j;
                                if (start_j - 1 >= 0) {
                                        prev_j =  start_j- 1;
                                } else if (cyclic == true) {
                                        prev_j = nPoints - 1;
                                } else {
                                        prev_j = start_j;
                                }

                                int end_j;
                                if (start_j + 1 < nPoints) {
                                        end_j = start_j + 1;
                                } else {
                                        end_j = 0;
                                }

                                int next_j;
                                if (end_j + 1 < nPoints) {
                                        next_j = end_j + 1;
                                } else if (cyclic == true) {
                                        next_j = 0;
                                } else {
                                        next_j = end_j;
                                }

                                // Get the Points and Scale them
                                Point prev = new Point(points.get(prev_j));
                                Point start = new Point(points.get(start_j));
                                Point end = new Point(points.get(end_j));
                                Point next = new Point(points.get(next_j));

                                // Obtain the Bezier Control Points
                                Point controlPoints[] = Bezier.getControlPoints(prev, start,
end, next);

                                // Bezier Interpolation
                                for (int k = 0; k <= BEZIERPOINTS; k++) {
                                        double t = (double) k/BEZIERPOINTS;
                                        Point bezierPoint = Bezier.getBezierPoint(controlPoints,
t);
                                        double bezierX = bezierPoint.getX();
                                        double bezierY = bezierPoint.getY();
                                        double bezierDeltaX = bezierPoint.getDeltaX();
                                        double bezierDeltaY = bezierPoint.getDeltaY();
                                        bezierPoly.pushPoint(new Point(bezierX, bezierY,
bezierDeltaX, bezierDeltaY));
                                } // Bezier Points
                        } // Curve Points
                        bezierPolys.add(bezierPoly);
                } // Polygons
        }

        public void calculateScalingData(Vector<Slice> slices) {
                double constCoord =
                        type.equals("cor") ? dicom.getPositionY() : dicom.getPositionX();

                for (int i = 0; i < slices.size(); i++) {
                        Slice slice = slices.get(i);
                        String sliceType = slice.getType();
                        Dicom sliceDicom = slice.getDicom();
                        double sliceConstCoord = sliceType.equals("cor") ?
```

```
                        sliceDicom.getPositionY() : sliceDicom.getPositionX();
                Vector<Polygon> sliceBezierPolys = slice.getBezierPolys();

                Polygon bezierPoly = bezierPolys.get(0);
                Polygon sliceBezierPoly = sliceBezierPolys.get(0);

                // Calculate the slice scale and y translation
                ScalingData scalingData = new ScalingData();
                double minYScale = LungPlot.MIN_SCALE - 1.0;
                double maxYScale = minYScale + 3.5;
                double minEdgeScale = LungPlot.MIN_SCALE - 1.0;
                double maxEdgeScale = minEdgeScale + 3.5;
                for (double scale = LungPlot.MIN_SCALE; scale <= LungPlot.MAX_SCALE;
                        scale += 0.05) {
                        scalingData.pushScale(scale);

                        double yScale = scale;
                        double minY = bezierPoly.getMinY(sliceConstCoord, scale);
                        for (double s = minYScale; s <= maxYScale; s += SCALE_STEP) {
                                double sliceMinY = sliceBezierPoly.getMinY(constCoord,
s);
                                if (Math.abs(sliceMinY - minY) < TOLERANCE) {
                                        yScale = s;
                                        minYScale = yScale;
                                        maxYScale = minYScale + 3.5;
                                        break;
                                }
                        }
                        scalingData.pushYScale(yScale);

                        double yTranslation = 0;
                        double edgeScale = scale;
                        double edge = bezierPoly.getEdgeSize(sliceConstCoord, scale);
                        for (double s = minEdgeScale; s <= maxEdgeScale; s +=
SCALE_STEP) {
                                double sliceEdge =
sliceBezierPoly.getEdgeSize(constCoord, s);
                                if (Math.abs(edge - sliceEdge) < TOLERANCE) {
                                        edgeScale = s;
                                        minEdgeScale = edgeScale;
                                        maxEdgeScale = minEdgeScale + 3.5;
                                        yTranslation = bezierPoly.getMinY(sliceConstCoord,
scale) -
                                                sliceBezierPoly.getMinY(constCoord,
edgeScale);
                                        break;
                                }
                        }
                        scalingData.pushEdgeScale(edgeScale);
                        scalingData.pushYTranslation(yTranslation);
                }
                scales.add(scalingData);
        }
    }
}
```

CLASSE LungPlot

Classe principal do programa, responsável pela interface gráfica.

```
/*************************************************************************/
/*                                                                       */
/* AUTHORS: Cesar Henrique Keiti Kuroiwa                                 */
/*                                                                       */
/*                      Alexandre Marques Lima                           */
/*                                                                       */
/*                                                                       */
/*************************************************************************/

package LungPlot;

import java.io.File;
import java.io.FileInputStream;
import java.io.BufferedReader;
import java.io.FilenameFilter;
import java.io.InputStreamReader;

import java.util.StringTokenizer;
import java.util.Arrays;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Vector;

import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.awt.GridBagLayout;
import java.awt.FlowLayout;
import java.awt.Dimension;

import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JSlider;
import javax.swing.JLabel;
import javax.swing.JCheckBox;
import javax.swing.JComboBox;
import javax.swing.JRadioButton;
import javax.swing.ButtonGroup;

// Main Class
public class LungPlot extends JFrame {
        private static final long serialVersionUID = 1L;

        public static final double MAX_SCALE = 2.0;
        public static final double MIN_SCALE = -1.0;

        private JPanel contentPane = null;
        private JPanel topPanel = null;
        private JPanel bottomPanel = null;
        private JPanel rightPanel = null;
        private PolyPanel drawPanel = null;

        // Scaling mode radio buttons
        private ButtonGroup scalingMode = null;
        private JRadioButton noScaling = null;
        private JRadioButton bottomScaling = null;
        private JRadioButton edgeScaling = null;

        // Scale slider
        private JSlider slider = null;

        private JLabel corSliceLabel = null;
        private JLabel sagSliceLabel = null;
```

```
// Status label
private JLabel statusLabel = null;

private JCheckBox showLimits = null;
private JCheckBox showCorSagIntersection = null;

// Coronal and Saggital slices combo box
private JComboBox corSliceBox = null;
private JComboBox sagSliceBox = null;

// Coronal and Saggital slices
private Vector<Slice> corSlices = null;
private Vector<Slice> sagSlices = null;

public LungPlot() {
        super();
        corSlices = new Vector<Slice>();
        sagSlices = new Vector<Slice>();
        initialize();
}

private void initialize() {
        loadSlices();

        this.setSize(900, 1000);
        this.setExtendedState(this.getExtendedState()|JFrame.MAXIMIZED_BOTH);
        this.setContentPane(getJContentPane());
        this.setTitle("Lung Plot");
        this.setVisible(true);

        getStatusLabel().setText("Calculating Scaling Data. Please wait...");
        // After all slices have been loaded, calculate the scalingData
        for (int i = 0; i < corSlices.size(); i++) {
                corSlices.get(i).calculateScalingData(sagSlices);
        }
        for (int i = 0; i < sagSlices.size(); i++) {
                sagSlices.get(i).calculateScalingData(corSlices);
        }
        getStatusLabel().setText("");
}

private JPanel getJContentPane() {
        if (contentPane == null) {
                contentPane = new JPanel();
                contentPane.setLayout(new BorderLayout());

                // Mounts the Graphical Interface
                topPanel = new JPanel();
                topPanel.setLayout(new FlowLayout());
                topPanel.add(getCorSliceLabel());
                topPanel.add(getCorSliceBox());
                topPanel.add(getSagSliceLabel());
                topPanel.add(getSagSliceBox());

                drawPanel = new PolyPanel();
                drawPanel.setCorSlices(corSlices);
                drawPanel.setSagSlices(sagSlices);

                bottomPanel = new JPanel();
                bottomPanel.setLayout(new GridBagLayout());
                bottomPanel.add(getShowCorSagIntersection());
                bottomPanel.add(getShowLimits());
                bottomPanel.add(getSlider());
                bottomPanel.add(getStatusLabel());

                rightPanel = new JPanel();
                rightPanel.setLayout(new GridLayout(3, 1));
                rightPanel.add(getNoScaling());
                rightPanel.add(getBottomScaling());
                rightPanel.add(getEdgeScaling());

                scalingMode = new ButtonGroup();
                scalingMode.add(getNoScaling());
                scalingMode.add(getBottomScaling());
                scalingMode.add(getEdgeScaling());
```

```
                contentPane.add(topPanel, BorderLayout.NORTH);
                contentPane.add(drawPanel, BorderLayout.CENTER);
                contentPane.add(bottomPanel, BorderLayout.SOUTH);
                contentPane.add(rightPanel, BorderLayout.EAST);


        }
        return contentPane;
}

private JRadioButton getNoScaling() {
        if (noScaling == null) {
                noScaling = new JRadioButton();
                noScaling.setText("No Scaling");
                noScaling.addItemListener(new ScalingModeHandler());
                noScaling.setSelected(true);
        }
        return noScaling;
}

private JRadioButton getBottomScaling() {
        if (bottomScaling == null) {
                bottomScaling = new JRadioButton();
                bottomScaling.setText("Min Y Scaling");
                bottomScaling.addItemListener(new ScalingModeHandler());
        }
        return bottomScaling;
}

private JRadioButton getEdgeScaling() {
        if (edgeScaling == null) {
                edgeScaling = new JRadioButton();
                edgeScaling.setText("Edge Scaling");
                edgeScaling.addItemListener(new ScalingModeHandler());
        }
        return edgeScaling;
}

// Scaling Mode Radio Buttons Handler Class
private class ScalingModeHandler implements ItemListener {
        public void itemStateChanged(ItemEvent event) {
                if (event.getSource() == noScaling) {
                        drawPanel.setScalingMode(0);
                } else if (event.getSource() == bottomScaling) {
                        drawPanel.setScalingMode(1);
                } else if (event.getSource() == edgeScaling) {
                        drawPanel.setScalingMode(2);
                }
                drawPanel.repaint();
        }
}

private JLabel getCorSliceLabel() {
        if (corSliceLabel == null) {
                corSliceLabel = new JLabel();
                corSliceLabel.setText("Coronal:");
        }
        return corSliceLabel;
}

private JLabel getSagSliceLabel() {
        if (sagSliceLabel == null) {
                sagSliceLabel = new JLabel();
                sagSliceLabel.setText("Saggital:");
        }
        return sagSliceLabel;
}

private JComboBox getCorSliceBox() {
        if (corSliceBox == null) {
                corSliceBox = new JComboBox();
                corSliceBox.addItem("");
                corSliceBox.addItemListener(new java.awt.event.ItemListener() {
                        public void itemStateChanged(java.awt.event.ItemEvent e) {
                                int corIndex = corSliceBox.getSelectedIndex();
                                drawPanel.reset();
                                drawPanel.setCorIndex(corIndex - 1);
```

```
                                        if (corIndex == 0) {
                                                drawPanel.setPlotCoronal(false);
                                                repaint();
                                                return;
                                        }
                                        drawPanel.setPlotCoronal(true);
                                        slider.setEnabled(true);
                                        showLimits.setEnabled(true);
                                        showCorSagIntersection.setEnabled(true);
                                        statusLabel.setText("scale: " + drawPanel.getScale());
                                        repaint();
                                }
                        });

                        /* Add the Coronal Slices to the ComboBox */
                        for (int i = 0; i < corSlices.size(); i++) {
                                corSliceBox.addItem(corSlices.get(i));
                        }
                }
                return corSliceBox;
        }

        private JComboBox getSagSliceBox() {
                if (sagSliceBox == null) {
                        sagSliceBox = new JComboBox();
                        sagSliceBox.addItem("");
                        sagSliceBox.addItemListener(new java.awt.event.ItemListener() {
                                public void itemStateChanged(java.awt.event.ItemEvent e) {
                                        int sagIndex = sagSliceBox.getSelectedIndex();
                                        drawPanel.reset();
                                        drawPanel.setSagIndex(sagIndex - 1);
                                        if (sagIndex == 0) {
                                                drawPanel.setPlotSaggital(false);

                                                repaint();
                                                return;
                                        }
                                        drawPanel.setPlotSaggital(true);
                                        slider.setEnabled(true);
                                        showLimits.setEnabled(true);
                                        showCorSagIntersection.setEnabled(true);
                                        statusLabel.setText("scale: " + drawPanel.getScale());
                                        repaint();
                                }
                        });
                        /* Add the Sagittal Slices to the ComboBox */
                        for (int i = 0; i < sagSlices.size(); i++) {
                                sagSliceBox.addItem(sagSlices.get(i));
                        }
                }
                return sagSliceBox;
        }

        private JLabel getStatusLabel() {
                if (statusLabel == null) {
                        statusLabel = new JLabel();
                        statusLabel.setText("");
                }
                return statusLabel;
        }

        private JSlider getSlider() {
                if (slider == null) {
                        slider = new JSlider();
                        slider.setEnabled(false);
                        slider.setSize(new Dimension(300, 20));
                        slider.setMaximum((int) (MAX_SCALE*20));
                        slider.setMinimum((int) (MIN_SCALE*20));
                        slider.setValue(0);
                        slider.addChangeListener(new javax.swing.event.ChangeListener() {
                                public void stateChanged(javax.swing.event.ChangeEvent e) {
                                        // Every time the value of slider changes, redraw the
whole slice
                                        double value = (double)slider.getValue()/20.0;
                                        drawPanel.setScale(value);
                                        statusLabel.setText("scale: " + drawPanel.getScale());
                                        drawPanel.repaint();
```

```
                }
            });
        }
        return slider;
    }

    private JCheckBox getShowLimits() {
        if (showLimits == null) {
            showLimits = new JCheckBox();
            showLimits.setText("Show Limits");
            showLimits.setEnabled(false);
            showLimits.addItemListener(new java.awt.event.ItemListener() {
                public void itemStateChanged(java.awt.event.ItemEvent e) {
                    drawPanel.setLimits(showLimits.isSelected());
                    drawPanel.repaint();
                }
            });
        }
        return showLimits;
    }

    private JCheckBox getShowCorSagIntersection() {
        if (showCorSagIntersection == null) {
            showCorSagIntersection = new JCheckBox();
            showCorSagIntersection.setText("Show Coronal-Saggital Intersection");
            showCorSagIntersection.setEnabled(false);
            showCorSagIntersection.addItemListener(new java.awt.event.ItemListener()
{
                public void itemStateChanged(java.awt.event.ItemEvent e) {

    drawPanel.setShowCorSagIntersection(showCorSagIntersection.isSelected());
                    drawPanel.repaint();
                }
            });
        }
        return showCorSagIntersection;
    }

    public void loadSlices() {
        File dir = new File(".");

        /* Filter out names that don't start with "slicePoints" */
        FilenameFilter filter = new FilenameFilter() {
            public boolean accept(File dir, String name) {
                return name.startsWith("slicePoints");
            }
        };

        String files[] = dir.list(filter);
        if (files == null) {
            getStatusLabel().setText("No Files Found!");
            return;
        }

        /* Sort the files by name */
        ArrayList<String> fileList = new ArrayList<String>(Arrays.asList(files));
        Collections.sort(fileList);

        for (int i = 0; i < fileList.size(); i++) {
            String fileName = fileList.get(i);
            int beginIndex = fileName.indexOf("_") + 1;
            String sliceNumber = fileName.substring(beginIndex, beginIndex + 3);
            Slice slice = readSlice(fileName);
            slice.setNumber(Integer.parseInt(sliceNumber));
            if (fileName.contains("cor")) {
                corSlices.add(slice);
                slice.setType("cor");
            } else {
                sagSlices.add(slice);
                slice.setType("sag");
            }
        }
    }

    public Slice readSlice(String filename) {
        double minX = 300, maxX = -300;
        double minY = 300, maxY = -300;
```

```
                Slice slice = new Slice();
                try {
                        //      Reads input data from text files
                        FileInputStream fis =
                                new FileInputStream(filename);
                        InputStreamReader isr = new InputStreamReader(fis);
                        BufferedReader reader = new BufferedReader(isr);

                        // Read in the number of Areas
                        String line = readLine(reader);
                        int num_areas = Integer.parseInt(line);
                        for (int i = 0; i < num_areas; i++) {
                                // Read in the number of Curves
                                line = readLine(reader);
                                int num_curves = Integer.parseInt(line);
                                int curve_counter = 0;
                                for (int j = 0; j < num_curves; j++) {
                                        Polygon poly = new Polygon();
                                        if (curve_counter == 0) {
                                                /* Only the first curve is cyclic */
                                                poly.setCyclic(true);
                                        } else {
                                                poly.setCyclic(false);
                                        }

                                        // Read in the number of Points
                                        line = readLine(reader);
                                        int num_lines = Integer.parseInt(line);

                                        for (int k  = 0; k < num_lines; k++) {
                                                // Each line contains the coordinates for one
point

                                                line = readLine(reader);
                                                StringTokenizer st = new StringTokenizer(line);
                                                double x = Double.parseDouble(st.nextToken());
                                                double y = Double.parseDouble(st.nextToken());
                                                double deltaX =
Double.parseDouble(st.nextToken());

                                                double deltaY =
Double.parseDouble(st.nextToken());

                                                Point p = new Point(x, y, deltaX, deltaY);
                                                poly.pushPoint(p);

                                                /* Calculate the maximum and minimum X and Y */
                                                if (x < minX) {
                                                        minX = x;
                                                }
                                                if (x > maxX) {
                                                        maxX = x;
                                                }
                                                if (y < minY) {
                                                        minY = y;
                                                }
                                                if (y > maxY) {
                                                        maxY = y;
                                                }
                                        } // Points
                                        slice.pushPoly(poly);

                                        curve_counter++;
                                } // Curves
                        } // Areas

                        Window window = new Window();
                        window.setLeft(minX);
                        window.setRight(maxX);
                        window.setTop(maxY);
                        window.setBottom(minY);
                        slice.setWindow(window);

                        // Read in Dicom data
                        Dicom dicom = new Dicom();
                        // xx, xy, xz
                        line = readLine(reader);
                        StringTokenizer st = new StringTokenizer(line);
                        double xx = Double.parseDouble(st.nextToken());
                        double xy = Double.parseDouble(st.nextToken());
```

```
                        double xz = Double.parseDouble(st.nextToken());
                        dicom.setXCos(xx, xy, xz);

                        // yx, yy, yz
                        line = readLine(reader);
                        st = new StringTokenizer(line);
                        double yx = Double.parseDouble(st.nextToken());
                        double yy = Double.parseDouble(st.nextToken());
                        double yz = Double.parseDouble(st.nextToken());
                        dicom.setYCos(yx, yy, yz);

                        // sx, sy, sz
                        line = readLine(reader);
                        st = new StringTokenizer(line);
                        double sx = Double.parseDouble(st.nextToken());
                        double sy = Double.parseDouble(st.nextToken());
                        double sz = Double.parseDouble(st.nextToken());
                        dicom.setPosition(sx, sy, sz);

                        // psx, psy
                        line = readLine(reader);
                        st = new StringTokenizer(line);
                        double psx = Double.parseDouble(st.nextToken());
                        double psy = Double.parseDouble(st.nextToken());
                        dicom.setSpacing(psx, psy);
                        slice.setDicom(dicom);
                        slice.calculateBezierPolys();
                        return slice;
                } // Try
                catch(Exception ex) {
                        statusLabel.setText("Error opening file " + filename);
                        return null;
                }
        }

        // Read a line from the file
        public String readLine(BufferedReader reader) throws Exception {
                String line = "";
                do {
                        line = reader.readLine();
                } while (line == "" || line.startsWith("#") == true);
                return line;
        }

        public static void main(String[] args) {
                LungPlot lp = new LungPlot();
                lp.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        }
}
```

CLASSE PolyPanel

Classe de grande importância, responsável por toda tarefa de plotagem das imagens das fatias.

```
/*************************************************************************/
/*                                                                       */
/* AUTHORS: Cesar Henrique Keiti Kuroiwa                                 */
/*                                                                       */
/*                    Alexandre Marques Lima                             */
/*                                                                       */
/*                                                                       */
/*************************************************************************/

package LungPlot;

import java.util.Vector;

import java.text.NumberFormat;

import java.awt.Color;
import java.awt.Graphics;
import java.awt.GridBagLayout;

import javax.swing.JPanel;

// Class that plots the Polygons onto the screen
public class PolyPanel extends JPanel {
        private static final long serialVersionUID = 1L;

        // Boolean Flags
        private boolean plotCoronal;
        private boolean plotSaggital;

        // Show Limits Flag
        private boolean limits;

        // Show Coronal-Saggital Intersection Flag
        private boolean showCorSagIntersection;

        // Show Source points flag
        private boolean showSourcePoints;

        // Current Scale
        private double scale;

        // Scaling Mode: 0 - no scaling, 1 - bottom scaling, 2 - edge scaling
        private int scalingMode;

        // Coronal and Saggital Slices
        private Vector<Slice> corSlices;
        private Vector<Slice> sagSlices;

        // Current Coronal and Saggital Slices
        private int corIndex;
        private int sagIndex;

        // Coronal and Saggital pixel Spacing
        private double corSpacing;
        private double sagSpacing;

        public PolyPanel() {
                super();
                initialize();
        }

        private void initialize() {
                reset();
```

```
        scalingMode = 0;

        plotCoronal = false;
        plotSaggital = false;
        showCorSagIntersection = false;
        showSourcePoints = true;

        corSpacing = 1;
        sagSpacing = 1;
        corIndex = -1;
        sagIndex = -1;
        this.setLayout(new GridBagLayout());
    }

    public void reset() {
        limits = false;
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        normalizeSpacing();

        g.setColor(Color.black);
        // Plots Coronal Slice
        if (plotCoronal == true) {
            plotCoronal(g);
        }

        // Plot Saggital Slice
        if (plotSaggital == true) {
            plotSaggital(g);
        }

        /* Check the need to show the inner and outer limits */
        if (limits == false) {
            return;
        }

        double tmpScale = scale;
        boolean tmpCorSagIntersection = showCorSagIntersection;
        showCorSagIntersection = false;
        showSourcePoints = false;

        /* Maximum Scale: 1 */
        g.setColor(Color.CYAN);
        scale = 2;
        if (plotCoronal == true) {
            plotCoronal(g);
        }
        if (plotSaggital == true) {
            plotSaggital(g);
        }

        /* Minimum Scale: 0 */
        g.setColor(Color.PINK);
        scale = -1;
        if (plotCoronal == true) {
            plotCoronal(g);
        }
        if (plotSaggital == true) {
            plotSaggital(g);
        }

        scale = tmpScale;
        this.setShowCorSagIntersection(tmpCorSagIntersection);
    }

    public void setPlotCoronal(boolean plot) {
        this.plotCoronal = plot;
    }

    public void setPlotSaggital(boolean plot) {
        this.plotSaggital = plot;
    }

    public void setLimits(boolean limits) {
```

```
                this.limits = limits;
        }

        public double getScale() {
                return scale;
        }

        public void setScale(double scale) {
                this.scale = scale;
        }

        // Normalize Spacings, so that corSpacing = 1.0 and sagSpacing = sagSpacing/corSpacing;
        public void normalizeSpacing() {
                if (plotCoronal == true && plotSaggital == true) {
                        Slice sagSlice = sagSlices.get(sagIndex);
                        Slice corSlice = corSlices.get(corIndex);
                        sagSpacing =
sagSlice.getDicom().getSpacingX()/corSlice.getDicom().getSpacingX();
                } else {
                        sagSpacing = 1.0;
                }
                corSpacing = 1.0;
        }

        // Plot Coronal Slice
        public void plotCoronal(Graphics g) {
                Slice corSlice = corSlices.get(corIndex);
                corSlice.calculateViewPort();
                ViewPort corViewPort = corSlice.getViewPort();
                Window corWindow = corSlice.getWindow();

                // Window - ViewPort relation
                double sx = corViewPort.getWidth()/corWindow.getWidth();
                double sy = corViewPort.getHeight()/corWindow.getHeight();
                double dx = corViewPort.getLeft() - sx*corWindow.getLeft();
                double dy = corViewPort.getTop() - sy*corWindow.getTop();

                // Loop through all the polygons in the slice
                Vector<Polygon> corBezierPolys = corSlice.getBezierPolys();
                for (int i = 0; i < corBezierPolys.size(); i++) {
                        Polygon bezierPoly = corBezierPolys.get(i);
                        int nPoints = bezierPoly.getSize();
                        Vector<Point> points = bezierPoly.getPoints();

                        // Loop through all the bezier points in the polygon
                        int[] X = new int[nPoints];
                        int[] Y = new int[nPoints];
                        for (int j = 0; j < nPoints; j++) {
                                Point bezierPoint = points.get(j);

                                // Get the scaled coordinates
                                double bezierX = bezierPoint.getX() +
scale*bezierPoint.getDeltaX();
                                double bezierY = bezierPoint.getY() +
scale*bezierPoint.getDeltaY();
                                bezierX *= corSpacing;
                                bezierY *= corSpacing;

                                X[j] = (int) (sx*bezierX + dx);
                                Y[j] = (int) (sy*bezierY + dy);

                                // Draw small squares at the points
                                if (j % Slice.BEZIERPOINTS == 0 && showSourcePoints == true) {
                                        g.fillRect(X[j] - 2, Y[j] - 2, 4, 4);
                                }
                        } // Bezier Points
                        g.drawPolyline(X, Y, nPoints);
                } //Polygons

                // Show Coronal-Saggital Intersections
                if (showCorSagIntersection == false) {
                        return;
                }
                double corConstCoord = corSlice.getDicom().getPositionY();

                // Go through all the saggital slices
                NumberFormat nf = NumberFormat.getInstance();
```

```
                nf.setMaximumFractionDigits(2);
                nf.setMinimumFractionDigits(2);
                for (int j = 0; j < sagSlices.size(); j++) {
                        Slice sagSlice = sagSlices.get(j);
                        double sagConstCoord = sagSlice.getDicom().getPositionX();

                        // Get the saggital polygons
                        Vector<Polygon> sagBezierPolys = sagSlice.getBezierPolys();

                        // Calculate the secondary scale and y translation
                        ScalingData scalingData = corSlice.getScales().get(j);
                        Vector<Double> scales = scalingData.getScales();
                        double scale2 = scale;
                        double yTranslation = 0.0;
                        if (scalingMode == 1) {
                                Vector<Double> yScales = scalingData.getYScales();
                                for (int k = 0; k < scales.size(); k++) {
                                        if (Math.abs(scales.get(k) - scale) < 0.05) {
                                                scale2 = yScales.get(k);
                                                break;
                                        }
                                }
                        } else if (scalingMode == 2) {
                                Vector<Double> edgeScales = scalingData.getEdgeScales();
                                Vector<Double> yTranslations = scalingData.getYTranslations();
                                for (int k = 0; k < scales.size(); k++) {
                                        if (Math.abs(scales.get(k) - scale) < 0.05) {
                                                scale2 = edgeScales.get(k);
                                                yTranslation = yTranslations.get(k);
                                                break;
                                        }
                                }
                                g.drawString(corSlice.getNumber() + " translation: " +
                                                nf.format(yTranslation), 600, 20 + 20*j);
                        }
                        g.drawString(corSlice.getNumber() + " Scale: " + nf.format(scale2),
                                                500, 20 + 20*j);
                        if (j == sagIndex) {
                                g.setColor(Color.RED);
                        }

                        sagConstCoord *= corSpacing;
                        for (int k = 0; k < sagBezierPolys.size(); k++) {
                                Polygon bezierPoly = sagBezierPolys.get(k);
                                double maxY = bezierPoly.getMaxY(corConstCoord, scale2) +
yTranslation;
                                if (maxY == -1000) {
                                        continue;
                                }
                                maxY *= corSpacing;
                                int x = (int) (sx*sagConstCoord + dx);
                                int y = (int) (sy*maxY + dy);

                                if (g.getColor() != Color.RED) {
                                        if (k % 2 == 0) {
                                                g.setColor(Color.BLUE);
                                        } else {
                                                g.setColor(Color.GREEN);
                                        }
                                }
                                g.fillOval(x - 2, y - 2, 4, 4);

                                // For cyclic polygons, do the same for minY
                                if (bezierPoly.isCyclic() == true) {
                                        double minY = bezierPoly.getMinY(corConstCoord, scale2) +
yTranslation;
                                        minY *= corSpacing;
                                        y = (int) (sy*minY + dy);

                                        if (g.getColor() != Color.RED) {
                                                g.setColor(Color.ORANGE);
                                        }
                                        g.fillOval(x - 2, y - 2, 4, 4);
                                }
                        }
                        g.setColor(Color.BLACK);
                }
```

```
        }

        public void plotSaggital(Graphics g) {
                Slice sagSlice = sagSlices.get(sagIndex);
                sagSlice.calculateViewPort();
                ViewPort sagViewPort = sagSlice.getViewPort();
                Window sagWindow = sagSlice.getWindow();

                // Window - ViewPort relation
                double sx = sagViewPort.getWidth()/sagWindow.getWidth();
                double sy = sagViewPort.getHeight()/sagWindow.getHeight();
                double dx = sagViewPort.getLeft() - sx*sagWindow.getLeft();
                double dy = sagViewPort.getTop() - sy*sagWindow.getTop();

                // Loop through all the polygons in the slice
                Vector<Polygon> sagBezierPolys = sagSlice.getBezierPolys();
                for (int i = 0; i < sagBezierPolys.size(); i++) {
                        Polygon sagBezierPoly = sagBezierPolys.get(i);
                        int nPoints = sagBezierPoly.getSize();
                        Vector<Point> points = sagBezierPoly.getPoints();

                        // Loop through all the bezier points in the polygon
                        int[] X = new int[nPoints];
                        int[] Y = new int[nPoints];
                        for (int j = 0; j < nPoints; j++) {
                                Point bezierPoint = points.get(j);

                                // Get the scaled coordinates
                                double bezierX = bezierPoint.getX() +
scale*bezierPoint.getDeltaX();
                                double bezierY = bezierPoint.getY() +
scale*bezierPoint.getDeltaY();
                                bezierX *= sagSpacing;
                                bezierY *= sagSpacing;

                                X[j] = (int) (sx*bezierX + dx);
                                Y[j] = (int) (sy*bezierY + dy);

                                // Draw small squares at the points
                                if (j % Slice.BEZIERPOINTS == 0 && showSourcePoints == true) {
                                        g.fillRect(X[j] - 2, Y[j] - 2, 4, 4);
                                }
                        } // Bezier Points
                        g.drawPolyline(X, Y, nPoints);
                } //Polygons

                // Show Coronal-Saggital Intersections
                if (showCorSagIntersection == false) {
                        return;
                }
                double sagConstCoord = sagSlice.getDicom().getPositionX();

                // Go through all the coronal slices
                NumberFormat nf = NumberFormat.getInstance();
                nf.setMaximumFractionDigits(2);
                nf.setMinimumFractionDigits(2);
                for (int j = 0; j < corSlices.size(); j++) {
                        Slice corSlice = corSlices.get(j);
                        double corConstCoord = corSlice.getDicom().getPositionY();

                        // Get the coronal polygons
                        Vector<Polygon> corBezierPolys = corSlice.getBezierPolys();

                        ScalingData scalingData = sagSlice.getScales().get(j);
                        Vector<Double> scales = scalingData.getScales();
                        double scale2 = scale;
                        double yTranslation = 0.0;
                        if (scalingMode == 1) {
                                Vector<Double> yScales = scalingData.getYScales();
                                for (int k = 0; k < scales.size(); k++) {
                                        if (Math.abs(scales.get(k) - scale) < 0.05) {
                                                scale2 = yScales.get(k);
                                                break;
                                        }
                                }
                        } else if (scalingMode == 2) {
                                Vector<Double> edgeScales = scalingData.getEdgeScales();
```

```
                                Vector<Double> yTranslations = scalingData.getYTranslations();
                                for (int k = 0; k < scales.size(); k++) {
                                        if (Math.abs(scales.get(k) - scale) < 0.05) {
                                                scale2 = edgeScales.get(k);
                                                yTranslation = yTranslations.get(k);
                                                break;
                                        }
                                }
                                g.drawString(corSlice.getNumber() + " translation: " +
                                                nf.format(yTranslation), 600,
Slice.SAGGITAL_Y_OFFSET + 20*j);
                        }
                        g.drawString(corSlice.getNumber() + " Scale: " + nf.format(scale2),
                                                500, Slice.SAGGITAL_Y_OFFSET + 20*j);

                        if (j == corIndex) {
                                g.setColor(Color.RED);
                        }

                        corConstCoord *= sagSpacing;
                        for (int k = 0; k < corBezierPolys.size(); k++) {
                                Polygon bezierPoly = corBezierPolys.get(k);
                                double maxY = bezierPoly.getMaxY(sagConstCoord, scale2) +
yTranslation;
                                if (maxY == -1000) {
                                        continue;
                                }
                                maxY *= sagSpacing;

                                int x = (int) (sx*corConstCoord + dx);
                                int y = (int) (sy*maxY + dy);
                                if (g.getColor() != Color.RED) {
                                        if (k % 2 == 0) {
                                                g.setColor(Color.BLUE);
                                        } else {
                                                g.setColor(Color.GREEN);
                                        }
                                }
                                g.fillOval(x - 2, y - 2, 4, 4);

                                // For cyclic polygons, do the same for minY
                                if (bezierPoly.isCyclic() == true) {
                                        double minY = bezierPoly.getMinY(sagConstCoord, scale2) +
yTranslation;
                                        minY *= sagSpacing;
                                        y = (int) (sy*minY + dy);
                                        if (g.getColor() != Color.RED) {
                                                g.setColor(Color.ORANGE);
                                        }
                                        g.fillOval(x - 2, y - 2, 4, 4);
                                }
                        }
                        g.setColor(Color.BLACK);
                }
        }

        public int getScalingMode() {
                return scalingMode;
        }

        public void setScalingMode(int scalingMode) {
                this.scalingMode = scalingMode;
        }

        public int getCorIndex() {
                return corIndex;
        }

        public void setCorIndex(int corIndex) {
                this.corIndex = corIndex;
        }

        public Vector<Slice> getCorSlices() {
                return corSlices;
        }

        public void setCorSlices(Vector<Slice> corSlices) {
```

```
            this.corSlices = corSlices;
        }

        public int getSagIndex() {
            return sagIndex;
        }

        public void setSagIndex(int sagIndex) {
            this.sagIndex = sagIndex;
        }

        public Vector<Slice> getSagSlices() {
            return sagSlices;
        }

        public void setSagSlices(Vector<Slice> sagSlices) {
            this.sagSlices = sagSlices;
        }

        public void setShowCorSagIntersection(boolean showCorSagIntersection) {
            this.showCorSagIntersection = showCorSagIntersection;
            this.showSourcePoints = !this.showCorSagIntersection;
        }
}
```

CLASSE ScalingData

Classe responsável por armazenar todas as informações de escalamento entre as fatias, para os métodos de escalamento por dimensão de aresta e coordenada y inferior.

```
package LungPlot;

import java.util.Vector;

public class ScalingData {
        private Vector<Double> scales;
        private Vector<Double> edgeScales;
        private Vector<Double> yTranslations;
        private Vector<Double> yScales;

        public ScalingData() {
                scales = new Vector<Double>();
                edgeScales = new Vector<Double>();
                yTranslations = new Vector<Double>();
                yScales = new Vector<Double>();
        }

        public Vector<Double> getEdgeScales() {
                return edgeScales;
        }

        public void pushEdgeScale(double edgeScale) {
                edgeScales.add(edgeScale);
        }

        public Vector<Double> getScales() {
                return scales;
        }

        public void pushScale(double scale) {
                scales.add(scale);
        }

        public Vector<Double> getYScales() {
                return yScales;
        }

        public void pushYScale(double scale) {
                yScales.add(scale);
        }

        public Vector<Double> getYTranslations() {
                return yTranslations;
        }

        public void pushYTranslation(double translation) {
                yTranslations.add(translation);
        }
}
```

CLASSE Dicom

Classe que contém dados relativos ao Dicom, tais como fatores de rotação e translação das imagens entre 2D e 3D.

```
/*****************************************************************************/
/*                                                                           */
/* AUTHORS: Cesar Henrique Keiti Kuroiwa                                     */
/*                                                                           */
/*                  Alexandre Marques Lima                                   */
/*                                                                           */
/*                                                                           */
/*****************************************************************************/

package LungPlot;

// Dicom data class
public class Dicom {

        // Pixel spacing
        private double psx, psy;

        // 3D image initial positions
        private double sx, sy, sz;

        // 3D image direction cosines
        private double xx, xy, xz;
        private double yx, yy, yz;

        public void setSpacing(double psx, double psy) {
                this.psx = psx;
                this.psy = psy;
        }

        public double getSpacingX() {
                return psx;
        }

        public double getSpacingY() {
                return psy;
        }

        public void setPosition(double sx, double sy, double sz) {
                this.sx = sx;
                this.sy = sy;
                this.sz = sz;
        }

        public double getPositionX() {
                return sx;
        }

        public double getPositionY() {
                return sy;
        }

        public double getPositionZ() {
                return sz;
        }

        public void setXCos(double xx, double xy, double xz) {
                this.xx = xx;
                this.xy = xy;
                this.xz = xz;
        }

        public double getXX() {
                return xx;
        }

        public double getXY() {
```

```
            return xy;
        }

        public double getXZ() {
            return xz;
        }

        public void setYCos(double yx, double yy, double yz) {
            this.yx = yx;
            this.yy = yy;
            this.yz = yz;
        }

        public double getYX() {
            return yx;
        }

        public double getYY() {
            return yy;
        }

        public double getYZ() {
            return yz;
        }
}
```

CLASSE Bezier

Classe auxiliar que contém métodos responsáveis por obter a interpolação de curvas por Bezier.

```
/*****************************************************************************/
/*                                                                           */
/* AUTHORS: Cesar Henrique Keiti Kuroiwa                                     */
/*                  Alexandre Marques Lima                                   */
/*                                                                           */
/*****************************************************************************/

package LungPlot;

/* Class Responsible For Interpolating a Bezier Curve */
public class Bezier {

        // Calculates a Point in the Bezier Curver, given 4 control points and parameter 't'
        public static Point getBezierPoint(Point[] controlPoints, double t) {
                Point p1 = controlPoints[0];
                Point p2 = controlPoints[1];
                Point p3 = controlPoints[2];
                Point p4 = controlPoints[3];
                double x, y;
                double deltaX, deltaY;

                if (p1 != p2 && p3 != p4) {
                        // All 4 points are different (4 control points)
                        x = p4.getX()*t*t*t + 3*p3.getX()*t*t*(1-t) + 3*p2.getX()*t*(1-t)*(1-t) +
                                p1.getX()*(1-t)*(1-t)*(1-t);
                        y = p4.getY()*t*t*t + 3*p3.getY()*t*t*(1-t) + 3*p2.getY()*t*(1-t)*(1-t) +
                                p1.getY()*(1-t)*(1-t)*(1-t);
                        deltaX = p4.getDeltaX()*t*t*t + 3*p3.getDeltaX()*t*t*(1-t) +
                                3*p2.getDeltaX()*t*(1-t)*(1-t) + p1.getDeltaX()*(1-t)*(1-t)*(1-t);
                        deltaY = p4.getDeltaY()*t*t*t + 3*p3.getDeltaY()*t*t*(1-t) +
                                3*p2.getDeltaY()*t*(1-t)*(1-t) + p1.getDeltaY()*(1-t)*(1-t)*(1-t);
                } else if (p1 == p2) {
                        // First 2 points are the same (3 control points)
                        x = p4.getX()*t*t + 2*p3.getX()*t*(1-t) + p2.getX()*(1-t)*(1-t);
                        y = p4.getY()*t*t + 2*p3.getY()*t*(1-t) + p2.getY()*(1-t)*(1-t);
                        deltaX = p4.getDeltaX()*t*t + 2*p3.getDeltaX()*t*(1-t) +
                                p2.getDeltaX()*(1-t)*(1-t);
                        deltaY = p4.getDeltaY()*t*t + 2*p3.getDeltaY()*t*(1-t) +
                                p2.getDeltaY()*(1-t)*(1-t);
                } else if (p3 == p4){
                        // Last 2 points are the same (3 control points)
                        x = p3.getX()*t*t + 2*p2.getX()*t*(1-t) + p1.getX()*(1-t)*(1-t);
                        y = p3.getY()*t*t + 2*p2.getY()*t*(1-t) + p1.getY()*(1-t)*(1-t);
                        deltaX = p3.getDeltaX()*t*t + 2*p2.getDeltaX()*t*(1-t) +
                                p1.getDeltaX()*(1-t)*(1-t);
                        deltaY = p3.getDeltaY()*t*t + 2*p2.getDeltaY()*t*(1-t) +
                                p1.getDeltaY()*(1-t)*(1-t);
                } else { /* p1 == p2 && p3 == p4 */
                        // Only 2 control points
                        x = p3.getX()*t + p2.getX()*(1-t);
                        y = p3.getY()*t + p2.getY()*(1-t);
                        deltaX = p3.getDeltaX()*t + p2.getDeltaX()*(1-t);
                        deltaY = p3.getDeltaY()*t + p2.getDeltaY()*(1-t);
                }

                Point bezierPoint = new Point(x, y, deltaX, deltaY);
                return bezierPoint;
        }

        // Calculates 4 Bezier control points, given 4 points
        public static Point[] getControlPoints(Point prev, Point start, Point end, Point next) {
```

```
                Point[] points = new Point[4];

                Point prev_start = start.add(prev.times(-1));
                Point start_end = end.add(start.times(-1));
                double len = start_end.length();
                Point end_next = next.add(end.times(-1));
                prev_start.normalize();
                start_end.normalize();
                end_next.normalize();

                Point prev_end = prev_start.add(start_end);
                Point start_next = start_end.add(end_next);
                prev_end.normalize();
                start_next.normalize();

                double cos1 = Math.abs(prev_end.scalar(start_end));
                double cos2 = Math.abs(start_next.scalar(start_end));

                points[0] = new Point(start);
                points[3] = new Point(end);

                if (cos1 > Math.sqrt(2)/2 && prev_start.length() > 0) {
                        points[1] = new Point(start.getX() + prev_end.getX()*len/3,
                                                                start.getY() +
prev_end.getY()*len/3,
                                                                start.getDeltaX() +
prev_end.getDeltaX()/3,
                                                                start.getDeltaY() +
prev_end.getDeltaY()/3);
                } else {
                        points[1] = points[0];
                }

                if (cos2 > Math.sqrt(2)/2 && end_next.length() > 0) {
                        points[2] = new Point(end.getX() - start_next.getX()*len/3,
                                                                end.getY() -
start_next.getY()*len/3,
                                                                end.getDeltaX() -
start_next.getDeltaX()/3,
                                                                end.getDeltaY() -
start_next.getDeltaY()/3);
                } else {
                        points[2] = points[3];
                }
                return points;
        }
}
```

CLASSES Window e ViewPort

Classes responsáveis por armazenar dados relacionados à transformação entre coordenadas de Window e ViewPort.

```
/*************************************************************************/
/*                                                                       */
/* AUTHORS: Cesar Henrique Keiti Kuroiwa                                 */
/*                                                                       */
/*                    Alexandre Marques Lima                             */
/*                                                                       */
/*                                                                       */
/*************************************************************************/

package LungPlot;

// Window class
public class Window {
        private double left;
        private double right;
        private double top;
        private double bottom;

        public double getBottom() {
                return bottom;
        }
        public void setBottom(double bottom) {
                this.bottom = bottom;
        }
        public double getLeft() {
                return left;
        }
        public void setLeft(double left) {
                this.left = left;
        }
        public double getRight() {
                return right;
        }
        public void setRight(double right) {
                this.right = right;
        }
        public double getTop() {
                return top;
        }
        public void setTop(double top) {
                this.top = top;
        }

        public double getWidth() {
                return right - left;
        }

        public double getHeight() {
                return top - bottom;
        }
}
```

```
/*************************************************************************/
/*                                                                       */
/* AUTHORS: Cesar Henrique Keiti Kuroiwa                                 */
/*                                                                       */
/*                     Alexandre Marques Lima                            */
/*                                                                       */
/*                                                                       */
/*************************************************************************/

package LungPlot;

// Viewport class
public class ViewPort {
        private double left;
        private double right;
        private double top;
        private double bottom;

        public double getBottom() {
                return bottom;
        }
        public void setBottom(double bottom) {
                this.bottom = bottom;
        }
        public double getLeft() {
                return left;
        }
        public void setLeft(double left) {
                this.left = left;
        }
        public double getRight() {
                return right;
        }
        public void setRight(double right) {
                this.right = right;
        }
        public double getTop() {
                return top;
        }
        public void setTop(double top) {
                this.top = top;
        }

        public double getWidth() {
                return right - left;
        }

        public double getHeight() {
                return top - bottom;
        }
}
```