Fundação Educacional do Município de Assis
Instituto Municipal de Ensino Superior de Assis - IMESA

Allain Daleck Spera

Criação de software auxiliar para o processamento e análise de imagens aplicada à quantificação de cobertura do dossel em áreas de reflorestamento

Assis

2012

Allain Daleck Spera

Criação de *software* auxiliar para o processamento e análise de imagens aplicada à quantificação de cobertura do dossel em áreas de reflorestamento

Trabalho de Conclusão de Curso apresentado ao Instituto Municipal de Ensino de Assis – IMESA e a Fundação Educacional do Município de Assis – FEMA, como requisito do Curso de Bacharelado em Ciências da Computação para obtenção do Certificado de Conclusão.

Orientador: Felipe Alexandre Cardoso Pazinatto

Área de Concentração: Pesquisa e Desenvolvimento de software; Processamento de Imagens

Assis

2012

Allain Daleck Spera

Criação de *software* auxiliar para o processamento e análise de imagens aplicada à quantificação de cobertura do dossel em áreas de reflorestamento

Trabalho de Conclusão de Curso apresentado ao Instituto Municipal de Ensino de Assis – IMESA e a Fundação Educacional do Município de Assis – FEMA, como requisito do Curso de Bacharelado em Ciências da Computação para obtenção do Certificado de Conclusão, analisado pela seguinte banca examinadora:

Orientador: Felipe Alexandre Cardoso Pazinatto

Analisador:_____

Assis

2012

FICHA CATALOGRÁFICA

SPERA, Allain Daleck

Criação de *software* auxiliar para o processamento e análise de imagens aplicada à quantificação de cobertura do dossel em áreas de reflorestamento / Allain Daleck Spera. Fundação Educacional do Município de Assis – FEMA – Assis, 2012.

Numero de p.: 58

Orientador: Felipe Alexandre Cardoso Pazinatto.

Trabalho de Conclusão de Curso – Instituto Municipal de Ensino Superior de Assis – IMESA.

1. Computação Gráfica. 2. Processamento de Imagens

Dedicatória

Dedico este trabalho a meus pais e à minha irmã, que são meu ponto de apoio, minha base e minha única certeza de confiança para o resto da vida.

Agradecimentos

Agradeço a Deus por me dar força e vontade para passar por essa etapa da minha vida.

Ao meu orientador Prof. Felipe Alexandre Cardoso Pazinatto pela ajuda, paciência e por todo o seu potencial aplicado ao desenvolvimento do trabalho.

Ao Éliton Rodrigo da Silveira, pelo auxílio no embasamento teórico sobre as técnicas utilizadas em campo.

Ao Instituto Florestal de Assis e à Dr.ª Eliane Akiko Honda pela oportunidade e por terem me fornecido base para a pesquisa desse trabalho.

Aos professores da FEMA, pelo constante incentivo ao longo dos anos de faculdade.

A meus pais e irmã, por me darem todo o apoio, carinho e amor para que eu pudesse continuar a faculdade.

Agradeço a todos, professores e amigos, que participaram também indiretamente do desenvolvimento desse projeto.

Resumo

A cada dia a computação está mais associada à biologia, auxiliando na obtenção eficaz de dados precisos. A utilização da computação gráfica na análise de resultados de reflorestamento provê um acompanhamento real da taxa de crescimento das árvores plantadas em áreas de preservação, buscando uma equivalência com o índice de cobertura de árvores situadas em ambientes naturais.

Palavras-chave: Computação gráfica; Processamento de imagens.

Abstract

With each new day the computing is more related to biology, assisting in effective obtaining of accurate data. The utilization of computer graphics in the analyze of the result of reforesting provides a real monitoring of the increase levels of planted trees in preservation areas, looking for a similarity with canopy structure of trees situated on natural environments.

Keywords: Computer Graphics; Image Processing.

Lista de Figuras

Sumário

1 INTRODUÇÃO

Desde 1964 nota-se um crescimento acelerado da utilização do processamento de imagens para melhorar a análise e interpretação do ser humano em diversos problemas de outras áreas de estudo. Na medicina, arqueologia, astronomia, biologia e em aplicações industriais é comum encontrar o uso de softwares de processamento de imagens digitais (GONZALEZ; WOODS, 2000).

Whitmore (1989) salienta em seu trabalho que o desenvolvimento das árvores e a característica da madeira depende essencialmente da quantidade e intensidade de luz que é direcionada à arvore. Portanto, com essa preocupação com o crescimento e manutenção de áreas florestais, aliado à preocupação ambiental de desenvolvimento sustentável de empresas, a área de preservação de florestas também utiliza *softwares* de processamento de imagens, para análise de diversas variáveis, entre elas, a porcentagem de cobertura de dosséis de florestas, baseando-se na verificação de incidência de luz, verificação de doenças na agricultura, análise quantitativa de áreas devastadas, análise de regiões mais vulneráveis a pragas para posterior aplicação de insumos agrícolas, dentre outros.

1.2 OBJETIVOS

Objetiva-se desenvolver um *software* utilizando conceitos de processamento de imagens na linguagem de programação Java para auxiliar pesquisadores biólogos na análise de imagens de coberturas de dossel de áreas de reflorestamento. Para isso, propõe-se a utilização da biblioteca de processamento de imagens em Java, *Java Advanced Imaging* (JAI) para construção de rotinas que solucionem o problema citado.

1.3 JUSTIFICATIVAS

O presente trabalho demonstra a utilização de conceitos da API de processamento de imagens Java em um *software* real para uso em pesquisas de biologia. Visto que

existem muitos programas que já realizam o procedimento de análise de fotos hemisféricas de áreas florestais, o trabalho propõe o desenvolvimento de uma solução em Java que demonstre a utilização de forma simples dos métodos existentes na biblioteca JAI.

1.4 MOTIVAÇÕES

A possibilidade de desenvolver algo científico e voltado à preservação do meio ambiente, além da criação de um software que possa demonstrar a utilização de uma API gráfica para desenvolvedores, e que também possa ajudar pesquisadores e cientistas na obtenção de dados em trabalhos em laboratório.

1.5 ESTRUTURA DO TRABALHO

O trabalho será dividido em capítulos, onde no capítulo 2 serão abordados conceitos básicos sobre o processamento de imagens de forma geral, no capítulo 3 serão aplicados os conceitos vistos na linguagem de programação Java, para posterior demonstração do desenvolvimento do *software* proposto pelo trabalho no capítulo 4, apresentando os resultados obtidos no capítulo 5 e no capítulo 6 estão dispostos os anexos citados ao longo do projeto.

2 PROCESSAMENTO DE IMAGENS

O processamento de imagens tem por objetivo retirar de um arquivo toda a informação existente, para tal resultado ser analisado por uma pessoa comum. Nesse sentido, essa atividade desenvolveu técnicas para se chegar a esse resultado, como a própria análise da imagem e técnicas de melhoria da mesma, para seu estudo ser facilitado (ALBUQUERQUE, ALBUQUERQUE,2000).

O processamento de imagens é composto, de uma maneira geral, da seguinte forma:

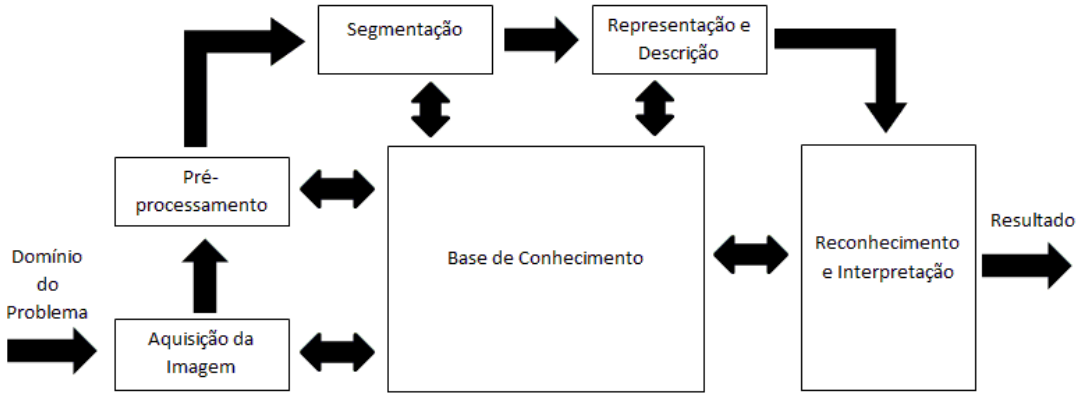

Figura 1: Etapas de um Sistema de Processamento de Imagens (In: GONZALEZ; WOODS, 2002, p. 5)

A Aquisição da Imagem é a obtenção de uma imagem digital, que pode ser através de uma câmera, um scanner, ou qualquer outro dispositivo que possa capturar sinais e digitalizá-lo em uma imagem. O Pré-processamento consiste na melhora da imagem (*image enhancement*) *para* aumentar a eficácia dos passos seguintes. Nessa etapa pode ser realizado o realce de contrastes, diminuição de ruídos, dentre outros métodos que envolvam a qualidade da imagem. A Segmentação trata de dividir a imagem em partes ou objetos constituintes. A Segmentação tem como saída os dados em forma de pixel, sendo necessária a conversão para uma forma adequada ao processamento computacional. Deve-se, portanto definir se a forma de representação adequada é a representação por fronteiras ou por regiões completas.

Além disso deve-se também, no processo de Descrição, atribuir características às classes de objetos extraídas do processo de representação. No último passo, o processo de Reconhecimento atribui um rótulo ao objeto baseado no seu descritor, sendo seguido pela Interpretação, que procura atribuir um significado a um conjunto de objetos reconhecidos (GONZALEZ; WOODS, 2000).

2.1 IMAGENS

“Uma imagem monocromática pode ser descrita matematicamente por uma função *f(x,y)* da intensidade luminosa, sendo seu valor, em qualquer ponto de coordenadas espaciais *(x,y)*, proporcional ao brilho (ou nível de cinza) da imagem naquele ponto” (MARQUES FILHO; VIEIRA NETO, 1999).

Uma imagem digital é basicamente uma matriz, onde os índices das linhas e colunas resultam em um ponto na imagem, cujo valor corresponde à intensidade do nível de cinza daquele ponto, que é chamado de *pixel*, que é abreviação de *picture elements* (elementos de figura) (GONZALEZ; WOODS, 2000).

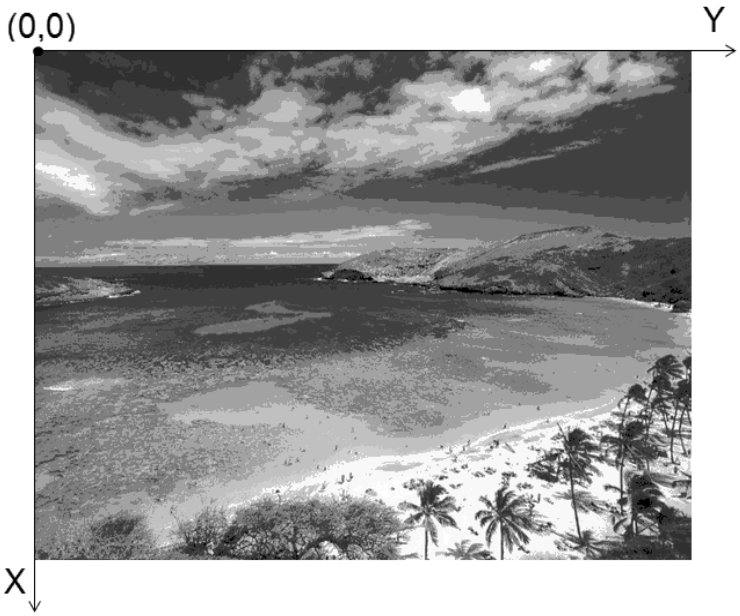

Figura 2: Imagem monocromática e representação dos pares de eixos (x,y)

De acordo com Marques Filho e Vieira Neto (1999), a função *f(x,y)* é caracterizada por dois componentes: a quantidade de luz que incide na cena representada pela função *i(x,y)* e a quantidade de luz refletida (reflectância) pelos objetos da cena, representada por *r(x,y)*. Matematicamente:

$$f(x,y) = i(x,y,).r(x,y)$$

onde:

$$0 < i(x,y) < \infty \qquad e$$

$$0 < r(x,y) < 1$$

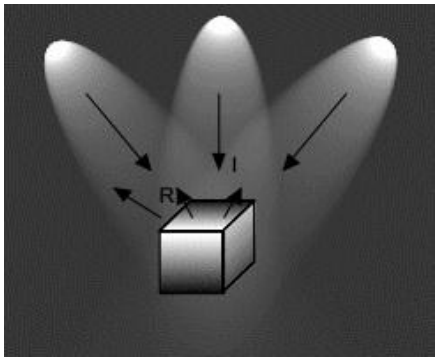

Figura 3: Iluminância (I) e Reflectância (R) de uma imagem (In: MARQUES FILHO; VIEIRA NETO, 1999, p. 20)

A intensidade monocromática obtida por *f(x,y)* é denominada nível de cinza, que estará no intervalo:

$$Lmin \leq l \leq Lmax$$

Sendo *Lmin* e *Lmax* valores positivos e finitos, e o intervalo [*Lmin*,*Lmax*] denominado escala de cinza. Comumente desloca-se o intervalo para [0,L], onde *l* = 0 é considerado preto e *l* = *L* é considerado branco, e os valores intermediários são tons de cinza (GONZALEZ; WOODS, 2000).

2.2 REGIÕES DE INTERESSE

Regiões de Interesse (ROI) na imagem são áreas definidas na própria imagem automaticamente ou pelo usuário onde o processamento estará concentrado (ALBUQUERQUE; ALBUQUERQUE, 2000).

Figura 4: Exemplo de seleção de Região de Interesse (ROI)

2.3 FILTROS

Os métodos de filtragem no domínio espacial referem-se ao tratamento e manipulação direta dos pixels da imagem (GONZALEZ, 2000). Funções para filtros em imagens no domínio espacial são definidas como:

$$g(x,y) = T[f(x,y)]$$

onde g(x,y) é a imagem final, f(x,y) é a imagem de entrada e T é um operador definido sobre alguma vizinhança de (x,y). A vizinhança é denotada geralmente por uma matriz quadrada envolvendo o pixel a ser processado. O centro dessa matriz quadrada é movido pixel a pixel, até que seja encontrado g para todo (x,y) (GONZALEZ, 2000).

De acordo com Gonzalez (2000), os filtros denominados “passa-baixa” têm por função atenuar ou eliminar os componentes de alta-frequência, enquanto deixam as frequências baixas intactas. O resultado desse procedimento é uma imagem borrada. Os filtros “passa-alta” por sua vez tem como objetivo atenuar bordas e detalhes finos da imagem, portanto no resultado final é apresentada uma imagem com bordas bem definidas e acentuadas.

2.3.1 Filtro espacial passa-baixa

Os filtros de suavização passa-baixa consistem basicamente na média de todos os pixels de uma área da imagem (GONZALEZ, 2000). Em um filtro 3x3, por exemplo, o resultado seria a soma dos 9 pixels envolvidos na vizinhança, e posterior divisão por 9, onde o pixel central receberia tal valor, varrendo assim a imagem em seguida até o cálculo total dos pixels.

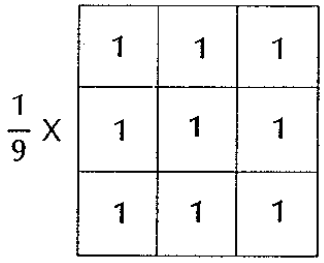

Figura 5: Exemplo de filtro passa-baixa 3x3

2.3.2 Filtro espacial passa-alta

As ideias de Gonzalez (2000) salientam que os filtros passa-alta representam o aguçamento de bordas e detalhes da imagem. O processamento desse filtro gera uma mudança nos níveis de cinza, de forma que quando há variações na intensidade, o valor resultante fica nula ou muito pequena, atenuando assim as regiões onde se alteram bruscamente a intensidade (bordas).

2.3.3 Filtro por derivadas

O trabalho de Gonzalez(2000) descreve que o filtro de diferenciação em aplicações de processamento de imagens é definido como gradiente. Para uma função f(x,y), o gradiente nas coordenadas (x,y) é demonstrado pelo vetor:

$$\nabla f = \begin{bmatrix} \frac{df}{dx} \\ \frac{df}{dy} \end{bmatrix}$$

A magnitude desse vetor

$$\nabla f = mag(\nabla f) = \left[\left(\frac{df}{dx}\right)^2 + \left(\frac{df}{dy}\right)^2 \right]^{1/2}$$

Representa a base para o processamento dos filtros, de modo que podem ser implementados vários modelos de aproximação, como por exemplo para uma vizinhança de 3x3 (Fig.6 (a)), que pode implementar o filtro de Prewitt da Fig. 6 (c) é:

$$\nabla f \approx |(z_7 + z_8 + z_9) - (z_1 + z_2 + z_3)| + |(z_3 + z_6 + z_9) - (z_1 + z_4 + z_7)|$$

(a)

z_1	z_2	z_3
z_4	z_5	z_6
z_7	z_8	z_9

(b) Roberts

1	0
0	–1

0	1
–1	0

(c) Prewitt

–1	–1	–1
0	0	0
1	1	1

–1	0	–1
–1	0	1
–1	0	1

(d) Sobel

–1	–2	–1
0	0	0
1	2	1

–1	0	–1
–2	0	2
–1	0	1

Figura 6: (a) Imagem original, (b) filtro Roberts, (c) filtro Prewitt, (d) filtro Sobel

3 PROCESSAMENTO DE IMAGENS EM JAVA

Apesar da existência de vários *softwares* de manipulação de imagens, com várias funcionalidades e propriedades distintas, existe a necessidade de se obter algoritmos específicos para tratamento de problemas que não são disponíveis nesses *softwares*, ou quando disponíveis, apresentam um nível de complexidade acima do esperado. Atendendo à esses requisitos, existem as APIs (*Application Program Interface*) para desenvolvimento de *software*, onde o programador pode utilizar seus métodos para criar suas próprias rotinas de acordo com sua necessidade, como a *Java Advanced Imaging* (JAI) específica para desenvolvimento de aplicações voltadas ao processamento de imagens (SANTOS, 2004).

3.1 JAI

A JAI é uma API que representa o suporte para o desenvolvimento de aplicações voltadas ao processamento de imagens utilizando a linguagem Java. Apesar de existirem outras APIs com esse mesmo fim, a JAI se destaca por apresentar um modelo de programação simplificado, de forma que o programador não seja obrigado a conhecer os processos internos das rotinas (SUN MICROSYSTEMS, 1999). Além disso, a JAI possui outras características que influenciam na decisão de escolhê-la como auxiliadora para desenvolvedores:

3.1.1 Multi-plataforma

A JAI segue o modelo do Java, se tornando assim uma API de imagens multi-plataforma, de modo que só é necessário uma *Java Virtual Machine* para que as aplicações desenvolvidas possam ser executadas com sucesso.

3.1.2 Arquitetura cliente-servidor

Assim como a arquitetura de rede e execução remota da plataforma Java, a JAI possui a característica de execução remota cliente-servidor, ou seja, o código em um cliente permite invocar chamadas de métodos em objetos que estão em outro computador, sem que o objeto tenha que ser movido ao cliente.

3.1.3 Orientada a objeto

A JAI é completamente orientada a objeto, unificando as ideias de imagens e operadores, de modo que um operador de objeto é instanciado apontando uma fonte de imagem e alguns parâmetros. O retorno desse operador acaba se tornando o parâmetro para outros operadores, e assim é efetuado o processamento da imagem.

3.1.4 Flexibilidade

O fato de possuir praticamente todas as funcionalidades necessárias para qualquer programa de processamento de imagens garante à JAI essa característica, além do fato de ter a flexibilidade de manipular a imagem como bem entender, para obter diferentes resultados em um mesmo método, como por exemplo, redimensionamento da imagem, dentre outros.

3.1.5 Poder e Alta Performance

JAI suporta inúmeros algoritmos complexos para o processamento, como imagens em 3 dimensões, com múltiplas bandas, redimensionamento, regiões de interesse, dentre outros, além da possibilidade de criar novos métodos, utilizando os operadores existentes e unindo funcionalidades, além do fato de possuir implementações que tirem vantagem do hardware e da capacidade dos dispositivos conectados ao computador que está executando a rotina.

3.1.6 Interoperações

Além de todas as características, a JAI também roda em conjunto com todas as APIs da plataforma Java, assim como a AWT, Java3D, e todos os outros componentes existentes.

3.2 DESENHANDO SOBRE A IMAGEM

O desenho da ROI sobre a imagem é realizado utilizando *Shapes*. A Princípio é criado um retângulo, utilizando os valores que o mouse captura quando o usuário clica para dar início e quando solta. Esses dois pontos formam o ponto inicial à esquerda superior, e o ponto direito inferior, desenhando assim a forma. Depois, uma Elipse é criada, de forma que suas coordenadas sejam proporcionais ao retângulo desenhado, ficando assim a elipse centralizada no interior dele.
A figura 5 apresenta o algoritmo com os métodos utilizados para efetuar o desenho da Região de Interesse.

```
public void selecionarROI(Point pressed, Point released){
   //testa se a imagem foi selecionada
   if(ativaSelecao = true){
       //atributo que seleciona a distancia euclidiana entre os pontos
       distancia = Math.sqrt((Math.pow((released.getX() - pressed.getX()),
       2) + Math.pow((released.getY() - pressed.getY()), 2)));
       /*Instanciando um retângulo com o ponto pressionado como ponto
       superior esquerdo*/
       Rectangle rect = new Rectangle(pressed);
       /*adicionando o ponto do mouse que foi capturado ao soltar o clique
       no canto inferior direito do retangulo*/
       rect.add(released);
       /*instanciando um objeto Ellipse2D para desenhar uma elipse
       utilizando como parametros os atributos do retângulo criado
       anteriormente*/
       Ellipse2D elip = new Ellipse2D.Float(rect.x - (rect.height / 2) +
       10, rect.y + 30, rect.height, rect.height);
       /*redimensionando o tamanho da elipse criada para o tamanho
       relativo da imagem utilizada*/
       int pressedX = (int) ((imagemOriginal.getWidth() *
       pressed.getX())/800);
       int pressedY = (int) ((imagemOriginal.getHeight() *
```

```
        pressed.getY())/600);
        int releasedX = (int)((imagemOriginal.getWidth() *
        released.getX())/800);
        int releasedY = (int)((imagemOriginal.getHeight() *
        released.getY())/600);
        /*criando os pontos redimensionados para serem passados ao novo
        retangulo e à nova elipse*/
        Point pressedCerto = new Point(pressedX, pressedY);
        Point releasedCerto = new Point(releasedX, releasedY);

        Rectangle rectNovo = new Rectangle(pressedCerto);
        rectNovo.add(releasedCerto);

        Ellipse2D elipNova = new Ellipse2D.Float(rectNovo.x -
        (rectNovo.height/2),
        rectNovo.y+((imagemOriginal.getHeight()*30)/600), rectNovo.height,
        rectNovo.height);
        raio = (releasedCerto.getY() - pressedCerto.getY())/2;
        /*passando por parâmetro a elipse para o metodo que a desenhará na
        tela*/
        adicionar(elip);
        /*passando à ROIShape a elipse redimensionada que será utilizada
        como base para o processamento da imagem*/
        roiShape = new ROIShape(elipNova);
        }

 }

public void adicionar(Shape shape) {
        //testa se a forma existe
        if (shape == null) {
                throw new IllegalArgumentException();
        }
        //testa se o vetor de formas possui mais de uma forma
        if (shapes.size() < 3) {
        //se tiver, chama o método que apaga, para deixar apenas uma
                apagar();
        }
        //adiciona em um vetor a imagem
        shapes.add(shape);
        //mostra o método de confirmar a seleção
        int retorno = JOptionPane.showConfirmDialog(null,
                "Deseja Confirmar a Seleção?");
        //se 'não', apaga a seleção que foi criada
        if (retorno == 1) apagar();
        //faz a chamada do método paint()
        repaint();

        }

public void apagar() {
        //limpa o vetor de imagens
        shapes.clear();
        //pinta na tela o vetor de imagens limpo
```

```
        repaint();
        }

@Override
public void paint(Graphics g) {
        //pinta na tela os componentes gráficos que estão criados
        super.paint(g);
        //seleciona a cor vermelha para a pintura
        g.setColor(Color.RED);
        //testa se o vetor de formas não está vazio
        if (shapes != null) {
                /*converte a forma Graphics para Graphics2D para comparação
                com a forma*/
                Graphics2D gg = (Graphics2D) g;
                /*um objeto do tipo Shape recebe cada forma que estiver no
                vetor de formas*/
                for (Shape shape : shapes) {
                        //desenha na tela a forma que foi passada ao objeto
                        gg.draw(shape);
                }
        }
```

Figura 7: Métodos utilizados para desenhar a Região de Interesse (ROI)

No caso, a *Shape* gerada ao final do método adicionar, será guardada em uma variável do tipo *RoiShape*, que pertence à JAI. Essa classe utiliza uma *Shape* que lhe é passada por parâmetro para gerar uma ROI (Região de Interesse) baseada na forma.

No exemplo, é passada ao programa uma imagem com resolução alta, que é convertida para 800x600 para visualização. Portanto, a Região de Interesse criada é baseada nessa resolução baixa. Para resolver isso, foi criada uma rotina que converte os pontos gerados pela *Shape* para um tamanho diretamente proporcional ao tamanho real da imagem.

3.3 CONVERSÃO PARA ESCALA DE CINZA

Existem alguns métodos para conversão de imagem para escala de cinza, que serão demonstrados nesse capítulo.

A conversão para níveis de cinza é realizada, pois os algoritmos de melhoria de imagem (*image enhancement*) em imagens coloridas tratam apenas de recombinar as taxas de luminância, além de que cada componente RGB (*Red, Green, Blue*) são armazenados separadamente na memória (ALBUQUERQUE;ALBUQUERQUE, 2000).

Figura 8: Imagem hemisférica original utilizada

3.3.1 Método *GrayFilter*

Utilizando o *GrayFilter*, passa-se para o construtor da classe *GrayFilter*, que faz parte do pacote Swing, dois parâmetros:

- *true* ou *false* - para especificar se a imagem receberá brilho ou não.
- um inteiro de 0 a 100 - esse valor inteiro representa a intensidade do cinza na imagem, onde 0 é o cinza mais escuro, e 100 o mais claro.

Esse construtor então é passado a um *ImageFilter*, que serve como parâmetro para sincronizar esse filtro com a imagem, utilizando o *ImageProducer*.

```
//METODO GRAYFILTER
/*Cria um novo ImageFilter de escala de cinza com os parâmetros boolean
para saber se a imagem receberá ajuste de brilho, e um valor inteiro de
0 a 100, que corresponde à intensidade do cinza
*/
ImageFilter filtro = new GrayFilter(true,0);
/*Instancia um novo ImageProducer relacionando o filtro criado à imagem
original*/
ImageProducer producer = new FilteredImageSource(
            imgProcessamento.getSource(), filtro);
//Cria um objeto do tipo Imagem utilizando o produto criado anteriomente
Image image2 = this.createImage(producer);
```

Figura 9: Utilizando GrayFilter para conversão em Escala de Cinza

Figura 10: Resultado após a utilização do GrayFilter

3.3.2 Método *Colorspace*

A *ColorSpace* é uma classe do pacote AWT, onde é alterada na imagem o modelo de cores que ela está sendo representada. Podem ser utilizados parâmetros definido as conversões desejadas, seja para RGB, ou Escala de Cinza, ou outras famílias de cores que existem (SUN MICROSYSTEM).

```
//METODO COLORSPACE
//Instancia uma ColorSpace com o tip CS_GRAY (Escala de Cinza)
ColorSpace cs = ColorSpace.getInstance(ColorSpace.CS_GRAY);
//Cria um novo filtro conversor que utiliza o Colorspace criado
ColorConvertOp op = new ColorConvertOp(cs, null);
//Adiciona à imagem o filtro op
Image image = op.filter(imgProcessamento, null);
```

Figura 11: Utilizando ColorSpace para conversão em Escala de Cinza

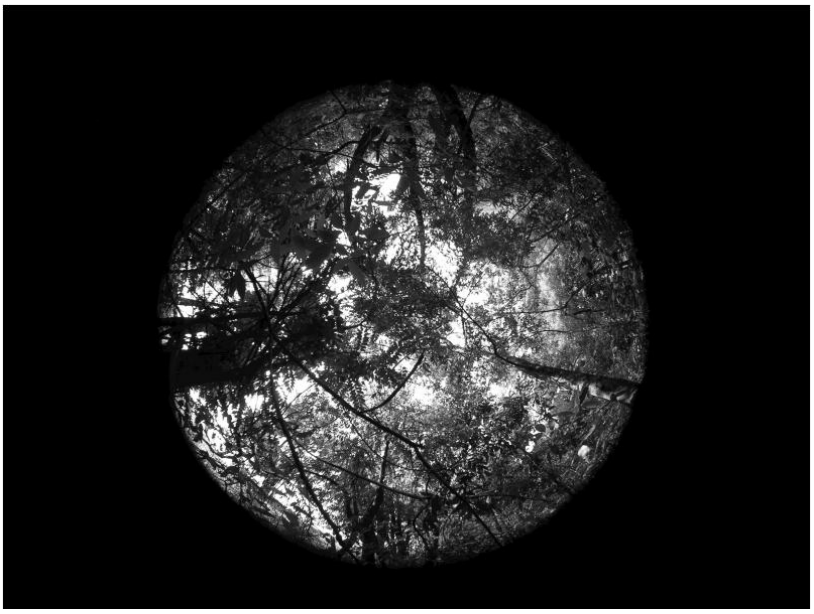

Figura 12: Resultado após a utilização do *ColorSpace*

3.3.3 Método *DrawImage*

Nesse método, é utilizado o parâmetro *TYPE_BYTE_GRAY,* que transforma a coloração dos bits da imagem que é passada no construtor do *BufferedImage*, para o tipo cinza, para ser posteriormente “desenhado” na tela pelo método *drawImage* da classe *Graphics*. Esse método funciona como se o próprio *BufferedImage* estivesse sendo criado novamente, mas com as cores da família da escala de cinza, utilizando uma imagem fonte como parâmetro para criação da nova. A imagem resultante é gerada com largura e altura referentes aos valores passados nos dois primeiros parâmetros desse construtor de *BufferedImage*.

```
//METODO DRAWIMAGE
/*Cria uma nova pseudo-imagem com as dimensões da imagem original e o
TYPE_BYTE_GRAY, que corresponde ao tipo de escala de cinza*/
BufferedImage image = new BufferedImage(imgProcessamento.getWidth(),
            imgProcessamento.getHeight(),BufferedImage.TYPE_BYTE_GRAY);
//Captura as propriedades gráficas da imagem criada
Graphics g = image.getGraphics();
/*Desenha as propriedades gráficas capturadas mesclando com a imagem
original*/
g.drawImage(imgProcessamento, 0, 0, null);
//Pausa a captura de propriedades gráficas
g.dispose();
```

Figura 13: Utilizando DrawImage para conversão em Escala de Cinza

Figura 14: Resultado após a utilização do *DrawImage*

3.3.4 Método *bandcombine*

A utilização do operador *bandcombine* da JAI é necessária para a combinação de bandas da imagem. No caso, é combinada a banda da imagem com uma matriz criada baseando-se na fórmula padrão para conversão de imagens RGB para Escala de Cinza.

```
//Cria uma matriz com os valores NTSC standard conversion formula
double[][] matriz = { {0.114D, 0.587D, 0.299D, 0.0D} };
//Cria um novo ParameterBlock para adicionar os parametros necessários
á combinação de bandas
ParameterBlock pb = new ParameterBlock();
//a imagem que será utilizada
pb.addSource(imagemOriginal);
//a matriz criada
pb.add(matriz);
/*a imagem final recebe o método 'create' com o parâmetro bandcombine
que multiplica as bandas da imagem original pelas bandas da matriz
criada*/
imagemPreProcessada =(PlanarImage) JAI.create("bandcombine", pb, null);
```

Figura 15: Utilização de bandcombine para conversão

Figura 16: Resultado após a utilização do *bandcombine*

3.4 HISTOGRAMA

Como disseram Gonzalez e Woods (2000, p. 122):

> O histograma de uma imagem digital com níveis de cinza no intervalo [0 ,L - 1] é uma função discreta $p(r_k) = n_k/n$¸ em que r_k é o k-ésimo nível de cinza, n_k é o número de pixels na imagem com esse nível de cinza, n é o número total de pixels na imagem e k = 0, 1, 2, ..., L - 1.

Os dados representados por um histograma são simplesmente a quantidade (ou porcentagem) para cada nível de cinza contido na imagem digital. Geralmente é representado por um gráfico em barras, onde é obtido quando observado, o nível de contraste da imagem, bem como se a imagem possui mais pontos claros ou escuros (MARQUES FILHO; VIEIRA NETO, 1999).

A JAI possui um operador estatístico para prover os dados da imagem, que é o *histogram*, que calcula um histograma da imagem passada, calculando também a distribuição dos pixel nos intervalos e armazena esses dados em uma pseudo-imagem de saída (SANTOS, 2004). O trecho de código demonstrado na figura 14 representa a utilização desse operador.

```
//Cria o novo ParameterBlock para utilizar na criação do histograma
ParameterBlock pb = new ParameterBlock();
//a imagem a ser utilizada como base
pb.addSource(imagemCinza);
//a região de interesse (null se quiser usar a imagem inteira)
pb.add(roiShape);
//taxa de amostragem vertical
pb.add(1);
//taxa de amostragem horizontal
pb.add(1);
//numero de intervalos para cada banda da imagem
pb.add(new int[] {256});
//limiar inferior
pb.add(new double[] {0});
//limiar superior
pb.add(new double[] {256});

//atribui a altura do gráfico na tela
dh.setHeight(160);
//atribui o multiplicador para o indice de uso da classe auxiliar
dh.setIndexMultiplier(1);
//adiciona o conteudo da classe auxiliar na janela criada
frame.getContentPane().add(dh);
//deixa visivel a janela
frame.setVisible(true);
```

Figura 17: Código-fonte da criação do Histograma

Essa rotina utiliza um *ParameterBlock* para guardar os parâmetros que serão fornecidos ao operador *histogram*. Passa-se como parâmetro uma imagem, que será a base para a criação do histograma, uma região de interesse (ou *null* se quiser a

imagem toda), as taxas de amostragem horizontal e vertical, o número de intervalos para cada banda da imagem e dois valores limiares inferior e superior.

Depois disso, é gerado em uma *RenderedImage* com o método *create* da JAI, passando como parâmetro o operador *histogram* e o *ParameterBlock* criado anteriormente. Tambem é utilizada uma classe auxiliar *DisplayHistogram*, que realiza a exibição do gráfico na tela. O código fonte dessa classe está em anexo.

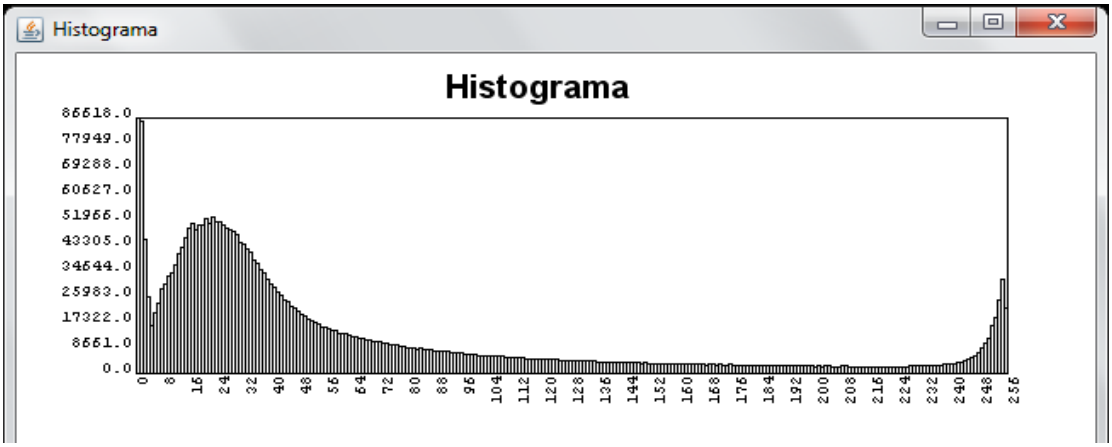

Figura 18: Histograma gerado pela JAI

3.5 THRESHOLD

A técnica de *Thresholding* ou Limiarização consiste na ideia de segmentação, onde, sendo verificado que existem dois picos principais em um histograma de uma imagem de níveis de cinza correspondentes aos níveis mais claros e mais escuros da cor, uma maneira de segmentar tal imagem é utilizar um limiar T, separando os dois picos (GONZALEZ; WOODS, 2000). A imagem obtida no caso então, é uma imagem binária, com duas regiões: uma branca (fundo) e outra preta (objetos pertencentes à foto) (ALBUQUERQUE;ALBUQUERQUE, 2001).

$$g(x,y) = \begin{cases} 1, & se\ f(x,y) > T \\ 0, & se\ f(x,y) \leq T \end{cases}$$

Onde *f(x,y)* é o nível de cinza no ponto *(x,y)*, e *g(x,y)* é a imagem limiarizada (GONZALEZ; WOODS, 2000).

3.5.1 ThresholdDescriptor

O *ThresholdDescriptor* é um operador que gera em uma pseudo-imagem, uma imagem limiarizada de acordo com os valores passados por parâmetro a seu método *create*, que são o valor mínimo do limiar, o máximo.

```
/*Uma pseudo-imagem recebe como o método create do ThresholdDescriptor e
como parâmetro a imagem a ser processada, o limiar inferior, o limiar
superio e as constantes que serão mapeadas*/
RenderedImage imagemLimiar =
      ThresholdDescriptor.create(imagemPreProcessada,
                                 new double[] {0,0,0},
                                 new double[] {40,40,40},
                                 new double[] {0,0,0}, null);
/*No caso, será realizada duas vezes o processamento, uma de 0 a 40
(acima) e outra de 41 a 255*/
RenderedImage imagemLimiar1 =
      ThresholdDescriptor.create(imagemLimiar,
                                 new double[] {41,41,41},
                                 new double[] {255,255,255},
                                 new double[] {255,255,255}, null);
```

Figura 19: Utilização de ThresholdDescriptor

No caso, está sendo passado um valor fixo para o método, porém é possível utilizar valores dinâmicos, sendo estes definidos pelo usuário.

Figura 20: Resultado gerado pelo *ThresholdDescriptor* com limiar padrão

O limiar utilizado no exemplo foi um valor aleatório, que gerou uma imagem binarizada, porém não detectando corretamente os objetos (pontos pretos) e fundo (pontos brancos) da imagem original. Portanto, definindo uma barra junto com a imagem do histograma, é possível selecionar o ponto que divide mais corretamente a imagem em fundo e objetos, tornando mais eficaz o processamento e a análise da imagem.

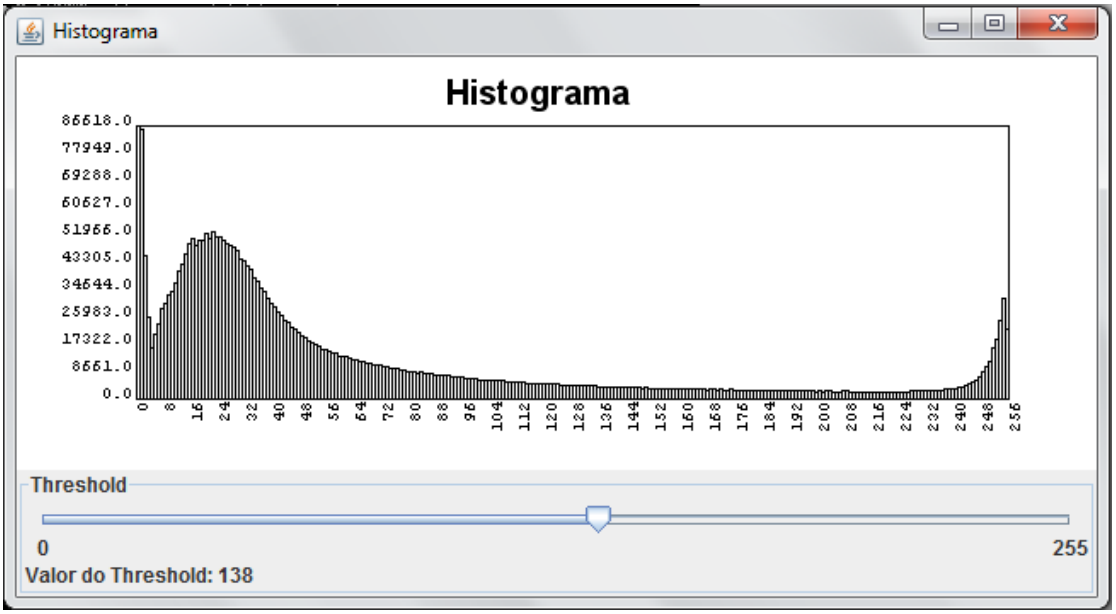

Figura 21: Seleção de limiar utilizando o histograma

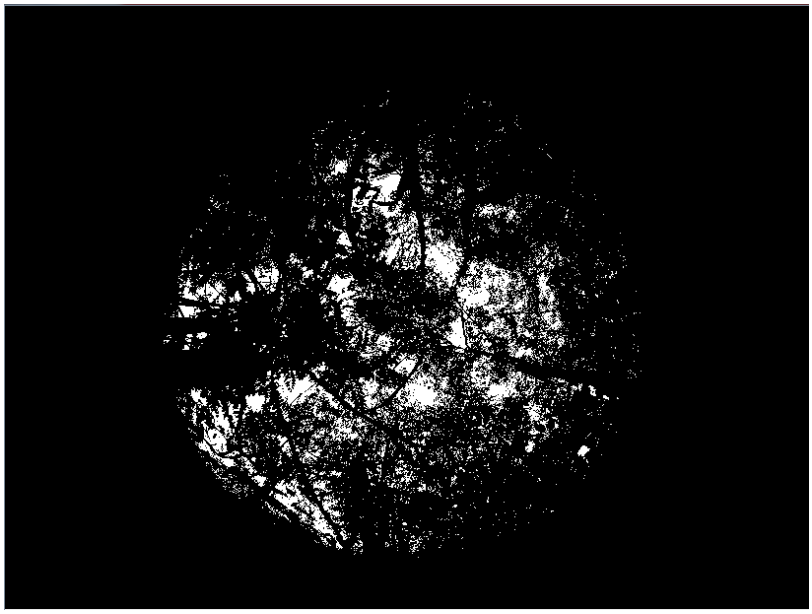

Figura 22: Resultado gerado pela seleção de limiar

3.5.2 Operador *threshold*

O operador da JAI *threshold* realiza a mesma operação que o *ThresholdDescriptor*, porém encapsulada em um método da API. Os parâmetros utilizados pelo *ThresholdDescriptor* são passados através do *ParameterBlock*.

```
//valor limiar inferior
double low[] = { 0d };
//valor limiar superior
double high[] = { 170d };
//ParameterBlock que será utilizado no operador threshold
ParameterBlock pb = new ParameterBlock();
//imagem a ser processada
pb.addSource(imagemPreProcessada);
//valor limiar inferior
pb.add(low);
//valor limiar superior
pb.add(high);
//constantes
pb.add(low);
/*pseudo-imagem recebendo o operador threshold com os
parametros criados*/
imagemProcessada = JAI.create("threshold", pb);
```

Figura 23: Utilização do operador threshold

O operador *threshold*, assim como o *ThresholdDescriptor* pode utilizar ao invés de um limiar fixo, um limiar dinâmico indicado pelo usuário.

4 DESENVOLVIMENTO DO SOFTWARE

4.1 Diagrama de Classes

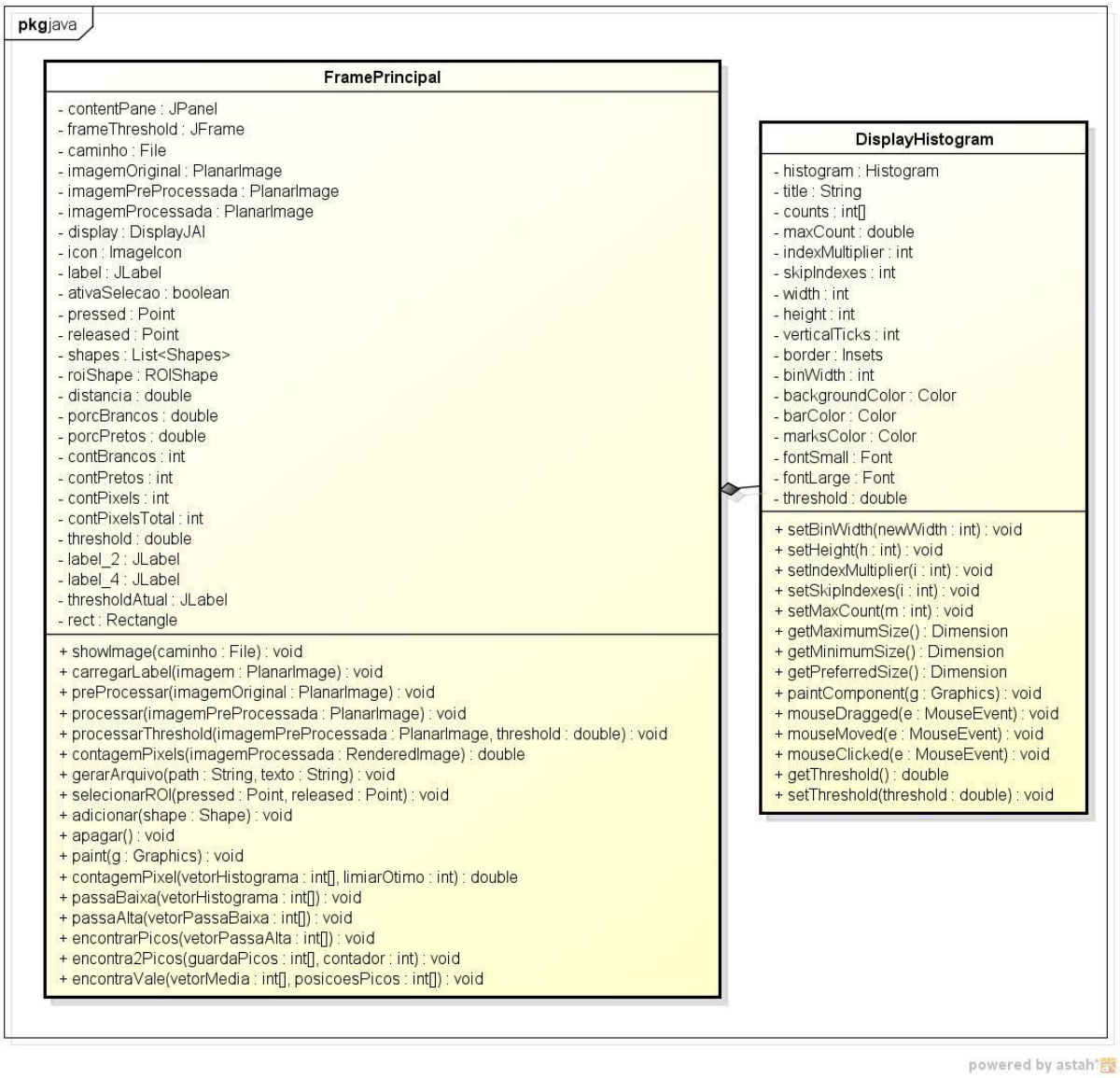

Figura 24: Diagrama de Classes do Projeto

4.2 Diagrama de Caso de Uso

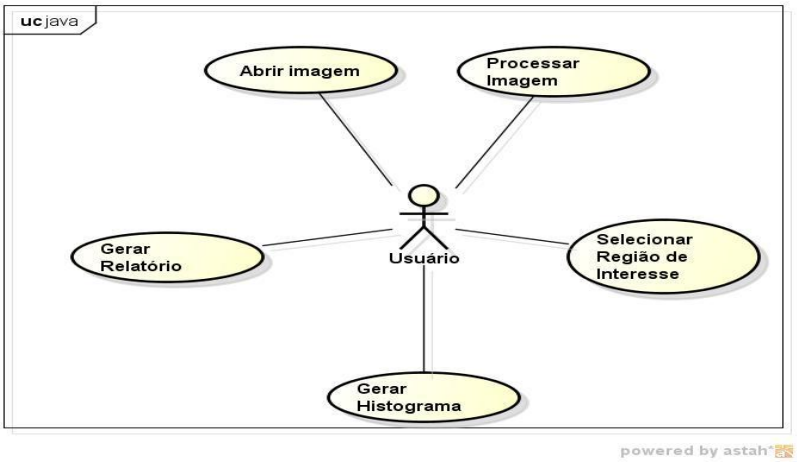

Figura 25: Diagrama de Casos de Uso

4.3 Diagrama de Estados

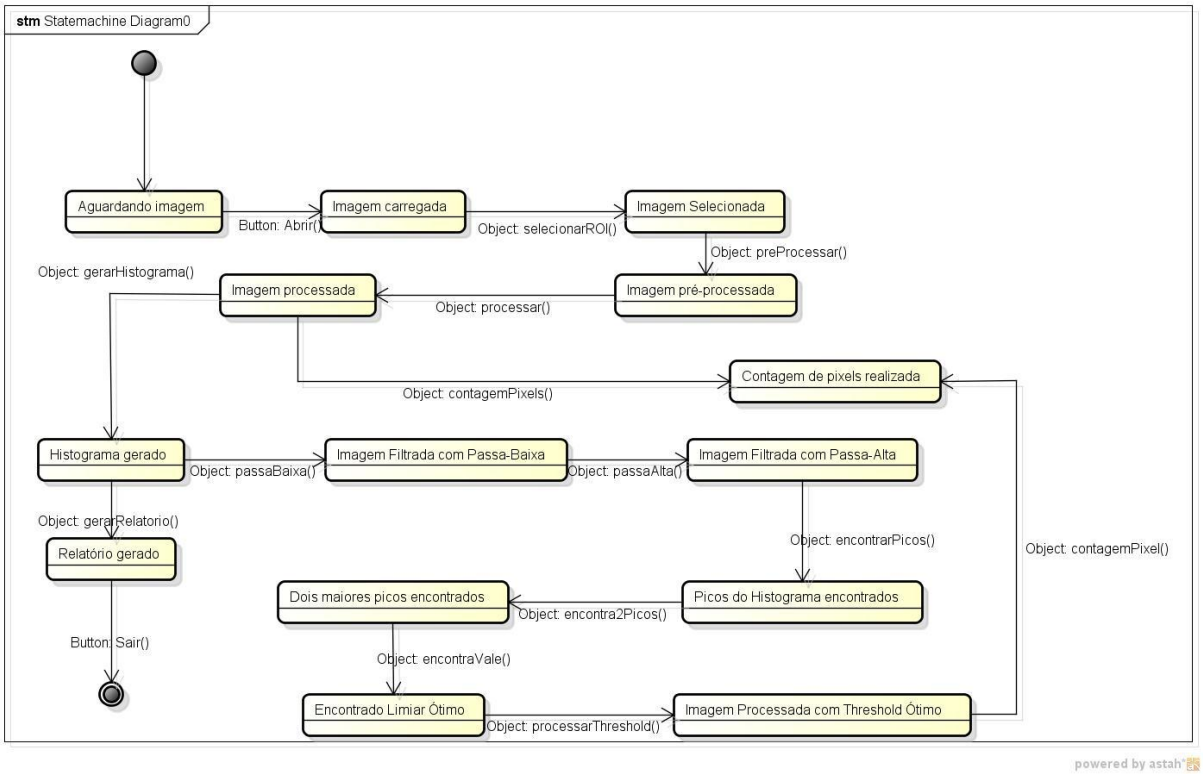

Figura 26: Diagrama de Estados

5 RESULTADOS E MÉTODOS

O produto final demonstra uma aplicação de processamento de imagens digitais, voltada à análise de fotografias hemisféricas capturadas de regiões florestais do Instituto Florestal de Assis. O processamento dessas imagens propõe uma análise da incidência de luz na região de interesse da foto, porcentagem que por sua vez auxiliará pesquisadores biólogos no acompanhamento e manutenção do crescimento das árvores da reserva.

A primeira tela do programa é a Fig. 27, que traz dois menus ‘Menu’ e ‘Ferramentas’

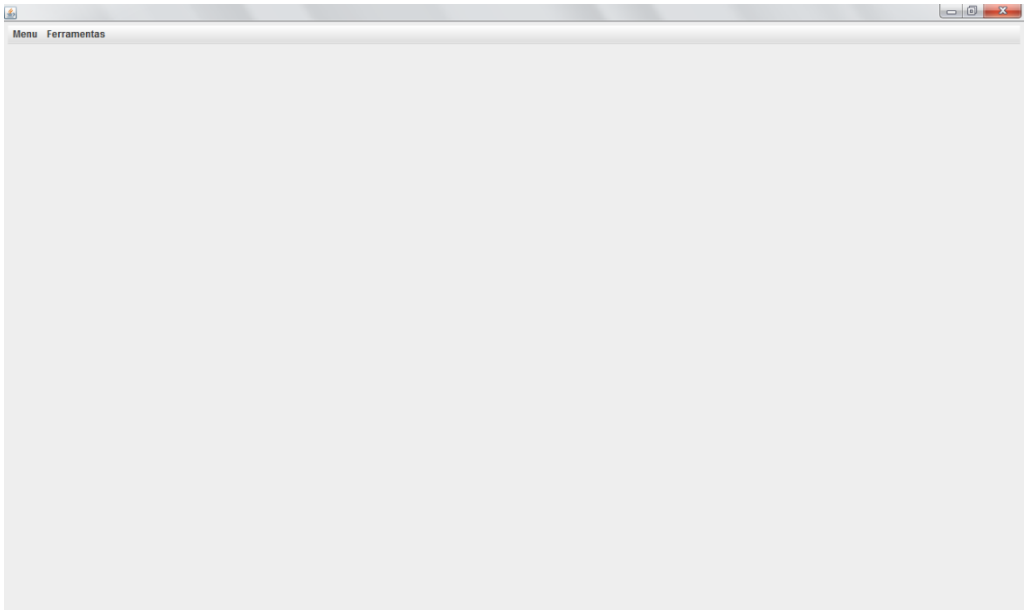

Figura 27: Tela Principal

No ‘Menu’ duas opções são apresentadas: ‘Abrir’, botão que carrega uma imagem do computador para ser processada (Fig. 30), e o botão de ‘Sair’.

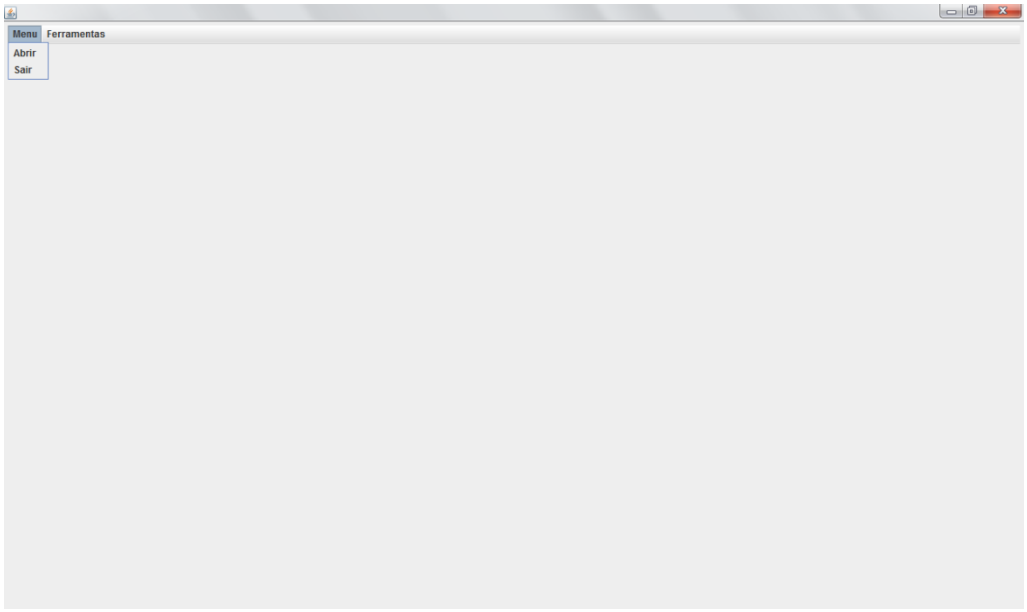

Figura 28: Demonstração do Menu Principal

No menu ‘Ferramentas‘ as funcionalidades do software são demonstradas. É possível ‘Selecionar ROI’ (a região de interesse que o processamento será concentrado), ‘Processar Imagem’ (efetua o pré-processamento), ‘Gerar Histograma’ (gera o gráfico Histograma de níveis de cinza da imagem, e realiza o processamento com o limiar encontrado no histograma).

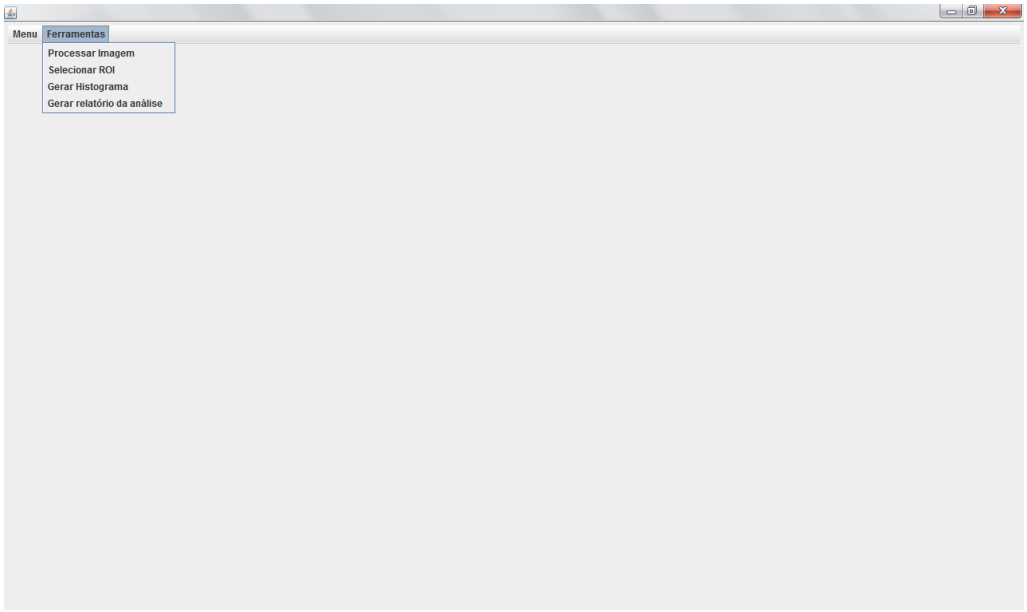

Figura 29: Demonstração do Menu Ferramentas

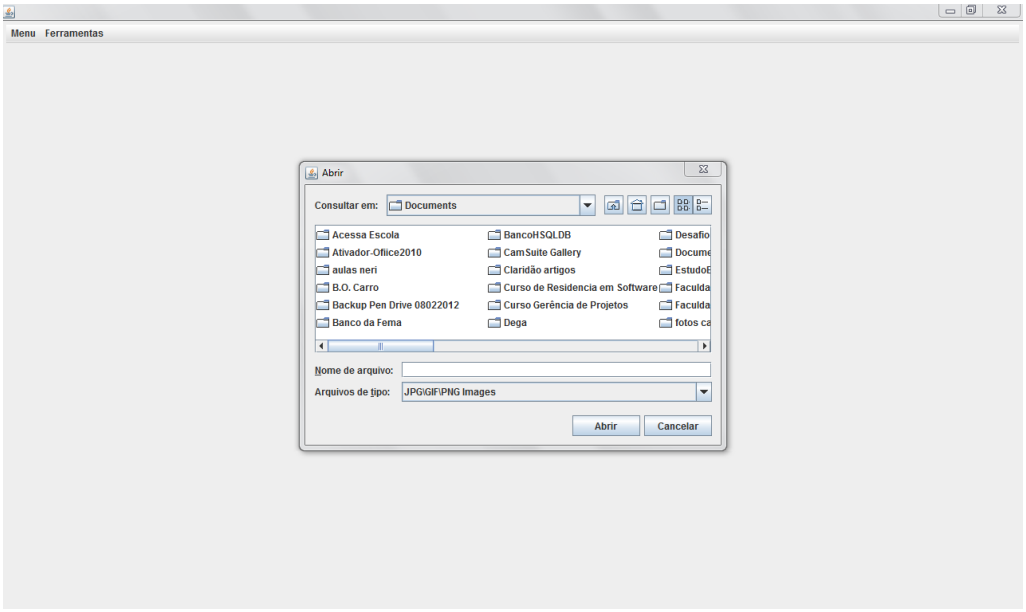

Figura 30: Tela de Seleção de imagens

O carregamento da imagem e amostragem na tela (Fig. 32) é realizado pelos métodos demonstrados na Fig. 31.

```
public void showImage(File caminho){
            //carregando a imagem a ser processada
            imagemOriginal = JAI.create("fileload", caminho.getPath());
            //chamada do método para mostrar na tela a imagem
            carregarLabel(imagemOriginal);
      }

private void carregarLabel(PlanarImage imagem){
            //carrega em um Icon a imagem passada por parâmetro
            icon = new ImageIcon(imagem.getAsBufferedImage());
            //seta a imagem redimensionando para ser mostrada na tela
      com 800x600 pixels
            icon.setImage(icon.getImage().getScaledInstance(
              icon.getIconWidth() > 800 ? 800 : icon.getIconWidth(),
              icon.getIconHeight() > 600 ? 600 : icon.getIconHeight(),
            100));
            //adiciona à label da janela o ícone
            label.setIcon(icon);

      }
```

Figura 31: Métodos utilizados para carregar a imagem principal na tela

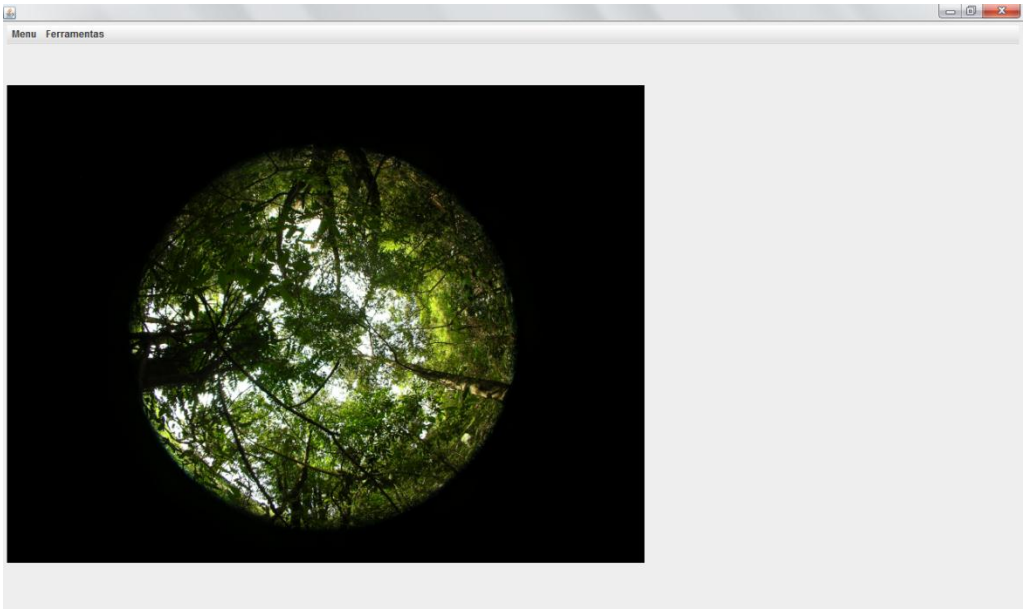

Figura 32: Tela após seleção de imagem

A seleção da Região de Interesse é efetuada utilizando o mouse, onde o clique inicial é no topo do círculo, e, arrastando até a base do mesmo, a região em vermelho surge destacando o foco principal do processamento.

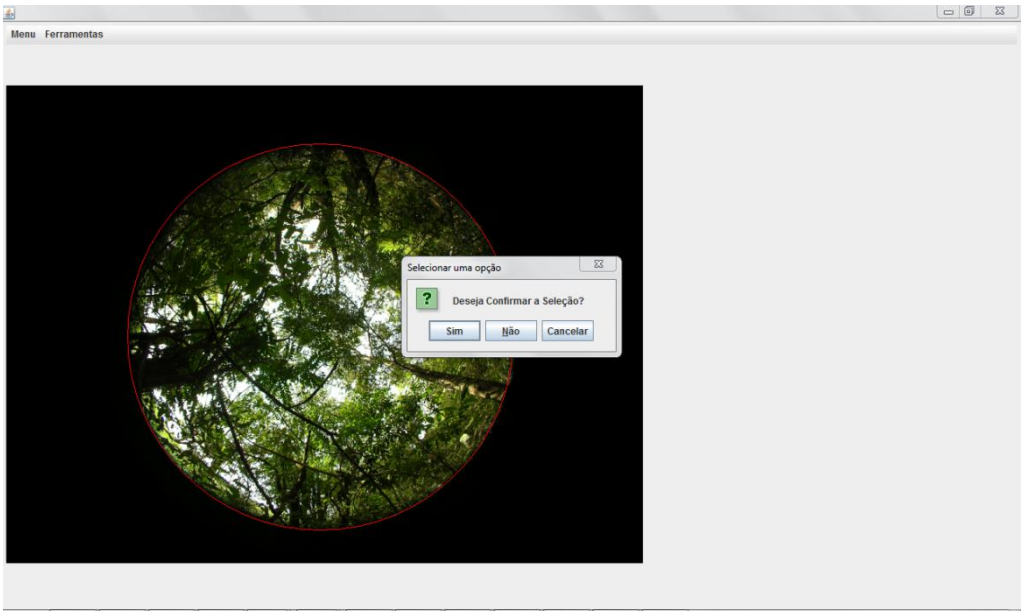

Figura 33: Confirmação da seleção de Região de Interesse

O processamento da imagem resulta em duas telas, a tela com a imagem convertida para Escala de Cinza, e a tela com a imagem em escala de cinza já limiarizada. O detalhe fica por conta das informações contidas na imagem limiarizada, pois é efetuada a contagem dos pixels (Fig. 34) que estão dentro da região de interesse, e efetuada uma porcentagem para quantificação da informação.

```
public double contagemPixel(int [] vetorHistograma, int limiarOtimo){
      //loop que varrerá o vetorHistograma inteiro
      for(int i=0; i< 256; i++){
      /*se a posição for maior que o limiar ótimo que separa
      o fundo da imagem com objeto, é branco, senão é preto*/
      if(i>limiarOtimo){
            contBrancos += vetorHistograma[i];
      } else contPretos += vetorHistograma[i];
      //soma dos pixels da ROI
      contPixels+=vetorHistograma[i];
      }
      //calculo da porcentagem de brancos
      return porcBrancos = (contBrancos*100)/contPixels;

      }
```

Figura 34: Método utilizado para efetuar a contagem de pixels

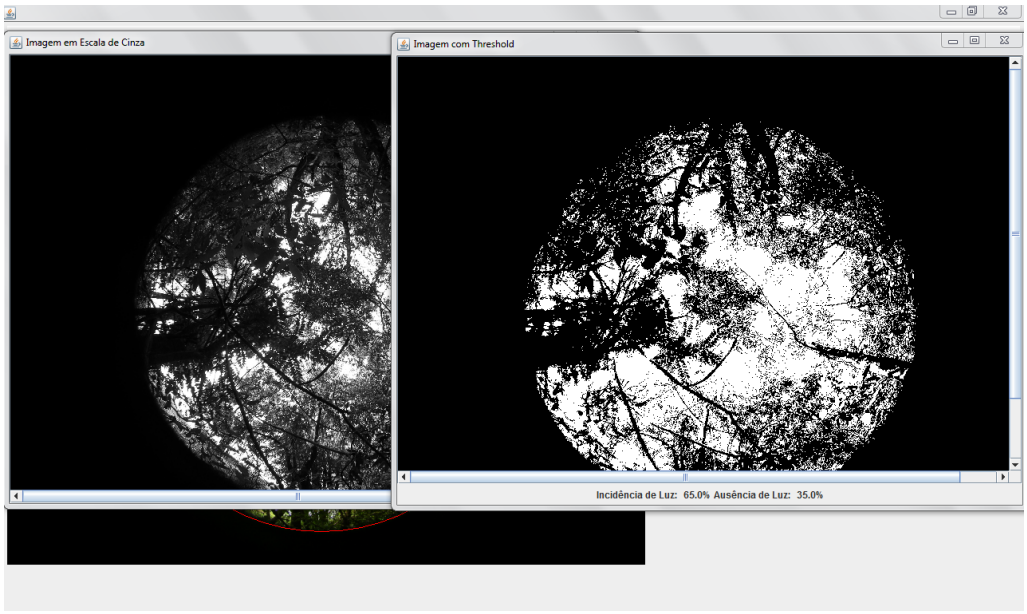

Figura 35: Tela após processamento da imagem

O histograma é gerado ao clicar no botão ‘Gerar Histograma’, onde além de sua criação, é encontrado um limiar ótimo baseando-se em suas informações. Esse limiar ótimo utiliza informações obtidas ao efetuar um filtro passa-baixa, um passa-alta. A utilização de um filtro passa-baixa no vetor do histograma suaviza suas bordas, de modo que a utilização de um filtro passa-alta atenua as variações de intensidade, ou seja, encontra efetivamente os picos e vales do histograma. Essas informações auxiliam na busca pelo limiar ótimo, que no caso é caracterizado como o vale (menor valor, com um superior antes e um superior depois) principal do histograma.

Os métodos utilizados para o processamento dos filtros são demonstrados nas Figuras 36, 37, 38, 39 e 40.

```
private void passaBaixa(int[] vetorHistograma){
      //vetor que receberá os valores após a filtragem
      vetorMedia = new int[256];
      //loop que varrerá o vetor recebido com os valores do histograma
      for(int i=0;i<(vetorHistograma.length-4);i++){
            /*cada posição do vetor receberá a soma de 5 posições
            somadas e suas respectivas médias*/
            vetorMedia[i+2] = (vetorHistograma[i]+vetorHistograma[i+1]+
                              vetorHistograma[i+2]+vetorHistograma[i+2]+
                              vetorHistograma[i+4])/5;
            }
      /*como o calculo da media possui menos valores que o
      original, os valores das 3 primeiras e 3 ultimas casas são
      repetidos*/
      vetorMedia[0] = vetorMedia[2];
      vetorMedia[1] = vetorMedia[2];
      vetorMedia[254] = vetorMedia[253];
      vetorMedia[255] = vetorMedia[253];
      //chamada do filtro passa-alta com o vetor gerado de média
      passaAlta(vetorMedia);

      }
```

Figura 36: Método utilizado para o Filtro Passa-Baixa

```
private void passaAlta(int[] vetorPassaBaixa){
      /*vetor que será comparado ao vetorPassaBaixa,
      contendo os valores do filtro de Prewitt*/
      int [] vetorPrewitt = {-1,0,0,0,0,0,0,0,0,0,0,0,0,0,1};
      //vetor que receberá os valores após a filtragem
      vetorPassaAlta = new int[256];
      //loop que varrerá o vetor com as intensidades do vetorPassaBaixa
      for(int i=0;i<(vetorPassaBaixa.length-14);i++){
      /*cada posição do vetor receberá a soma de 15 posições com cada
      uma multiplicada pelo vetor de Prewitt*/
      vetorPassaAlta[i+7] = ((vetorPassaBaixa[i]*vetorPrewitt[0])+
                        (vetorPassaBaixa[i+1]*vetorPrewitt[1])+
                        (vetorPassaBaixa[i+2]*vetorPrewitt[2])+
                        (vetorPassaBaixa[i+3]*vetorPrewitt[3])+
                        (vetorPassaBaixa[i+4]*vetorPrewitt[4])+
                        (vetorPassaBaixa[i+5]*vetorPrewitt[5])+
                        (vetorPassaBaixa[i+6]*vetorPrewitt[6])+
                        (vetorPassaBaixa[i+7]*vetorPrewitt[7])+
                        (vetorPassaBaixa[i+8]*vetorPrewitt[8])+
                        (vetorPassaBaixa[i+9]*vetorPrewitt[9])+
                        (vetorPassaBaixa[i+10]*vetorPrewitt[10])+
                        (vetorPassaBaixa[i+11]*vetorPrewitt[11])+
                        (vetorPassaBaixa[i+12]*vetorPrewitt[12])+
                        (vetorPassaBaixa[i+13]*vetorPrewitt[13])+
                        (vetorPassaBaixa[i+14]*vetorPrewitt[14]));
            }


       /*como o calculo da filtragem possui menos valores que o
       recebido, os valores das 7 primeiras e73 ultimas casas são
       repetidos*/
       vetorPassaAlta[0] = vetorPassaAlta[7];
       vetorPassaAlta[1] = vetorPassaAlta[7];
       vetorPassaAlta[2] = vetorPassaAlta[7];
       vetorPassaAlta[3] = vetorPassaAlta[7];
       vetorPassaAlta[4] = vetorPassaAlta[7];
       vetorPassaAlta[5] = vetorPassaAlta[7];
       vetorPassaAlta[6] = vetorPassaAlta[7];
       vetorPassaAlta[249] = vetorPassaAlta[248];
       vetorPassaAlta[250] = vetorPassaAlta[248];
       vetorPassaAlta[251] = vetorPassaAlta[248];
       vetorPassaAlta[252] = vetorPassaAlta[248];
       vetorPassaAlta[253] = vetorPassaAlta[248];
       vetorPassaAlta[254] = vetorPassaAlta[248];
       vetorPassaAlta[255] = vetorPassaAlta[248];
       //chamada da busca pelos picos do vetorPassaAlta
       encontrarPicos(vetorPassaAlta);
       }
```

Figura 37: Método utilizado para o filtro Passa-Alta

```
private void encontrarPicos(int[] vetorPassaAlta){
      //vetor que guardará os picos do vetorPassaAlta
      int [][] guardaPicos = new int [100][4];
      int k=0;
      int i;
      //loop que varrerá todo o vetor buscando os picos
      for(k=0,i=1;i<vetorPassaAlta.length-1;i++){
            /*se a posição tiver um valor inferior antes
            e um valor inferior depois, ele é um pico*/
            if(vetorPassaAlta[i-1]<vetorPassaAlta[i] &&
               vetorPassaAlta[i]>vetorPassaAlta[i+1] && i>5){
                  /*a posicao 0 guarda o valor anterior,
                  a 1 guarda o valor do pico,
                  a 2 guarda o valor posterior e
                  a 3 guarda a posição*/
                  guardaPicos[k][0]= vetorPassaAlta[i-1];
                  guardaPicos[k][1]= vetorPassaAlta[i];
                  guardaPicos[k][2]= vetorPassaAlta[i+1];
                  guardaPicos[k][3]= i;
                  k++;
                  }
            }
            //chamada da busca pelos 2 maiores picos
            encontra2Picos(guardaPicos,k);

      }
```

Figura 38: Método utilizado para encontrar os Picos do histograma

```
private void encontra2Picos(int [][] guardaPicos, int contador){
	//vetor que guarda os 2 maiores picos
	int[] maioresPicos = new int[2];

	int picoMaior =0;
	int posicaoMaior =0;
	int posicaoApagar =0;
	//vetor que guardará as posições dos 2 maiores picos
	int[] posicoesPicos = new int[2];
	//loop que varrerá o vetor de picos, e encontrará os 2 maiores
	for(int i=0;i<2;i++){
		for(int j=0;j<contador;j++){
			/*se o pico for o maior, é guardada no picoMaior o
			seu valor, e em posicaoMaior a posição do pico, além
			de guardar a posição desse pico maior para ele não
			entrar na próxima comparação*/
			if(picoMaior<guardaPicos[j][1]){
				picoMaior=guardaPicos[j][1];
				posicaoMaior=guardaPicos[j][3];
				posicaoApagar=j;
				}
			}
		maioresPicos[i]=picoMaior;
		posicoesPicos[i]=posicaoMaior;
		picoMaior=0;
		posicaoMaior=0;
		//apaga o maior pico para não entrar na próxima comparação
		guardaPicos[posicaoApagar][1]=0;
		}
	//chamada da busca pelo limiar ótimo (vale)
	encontraVale(vetorMedia, posicoesPicos);

	}
```

Figura 39: Método que encontra os dois maiores picos

```
private void encontraVale(int [] vetorMedia, int[] posicoesPicos){
      //vetor que guardará os picos do vetor
      int [][]guardaVale = new int[100][4];
      int k=0;
      int i;
      /*loop que varrerá  as posições dos picos e
      encontrará a correspondência no vetor de média
      O teste no loop é para caso os valores não estejam em ordem
      crescente*/
      for(k=0,
          i= (posicoesPicos[0]>posicoesPicos[1])?posicoesPicos[1]:
          posicoesPicos[0];
          i < ((posicoesPicos[0]>posicoesPicos[1])?posicoesPicos[0]:
          posicoesPicos[1]);
          i++){
            /*se a posição anterior e posterior forem
            maiores que a atual, é um vale*/
            if(vetorMedia[i-1]>vetorMedia[i] &&
               vetorMedia[i]<vetorMedia[i+1]){
                        /*o vetor receberá o valor correspondente
                        do vale encontrado, no vetorMedia*/
                        guardaVale[k][0]= vetorMedia[i-1];
                        guardaVale[k][1]= vetorMedia[i];
                        guardaVale[k][2]= vetorMedia[i+1];
                        guardaVale[k][3]= i;
                        k++;
                  }
            }
      //variável que guardará o menor valor
      int menorValor = 1000000;
      //variável que guardará a posição do menor valor
      int posicaoMenor = 0;
      for(i=0; i<100; i++){
            //se o valor for o menor e não for preto
            if(guardaVale[i][1]<menorValor && guardaVale[i][1]!= 0){
                  menorValor=guardaVale[i][1];
                  posicaoMenor=guardaVale[i][3];
                  }
            }
      /*chamada para o método de processar a imagem em escala de cinza,
      utilizando a posiçao encontrada na filtragem como limiar ótimo*/
      processarThreshold(imagemPreProcessada, posicaoMenor);

      }
```

Figura 40: Método utilizado para encontrar o Limiar Ótimo

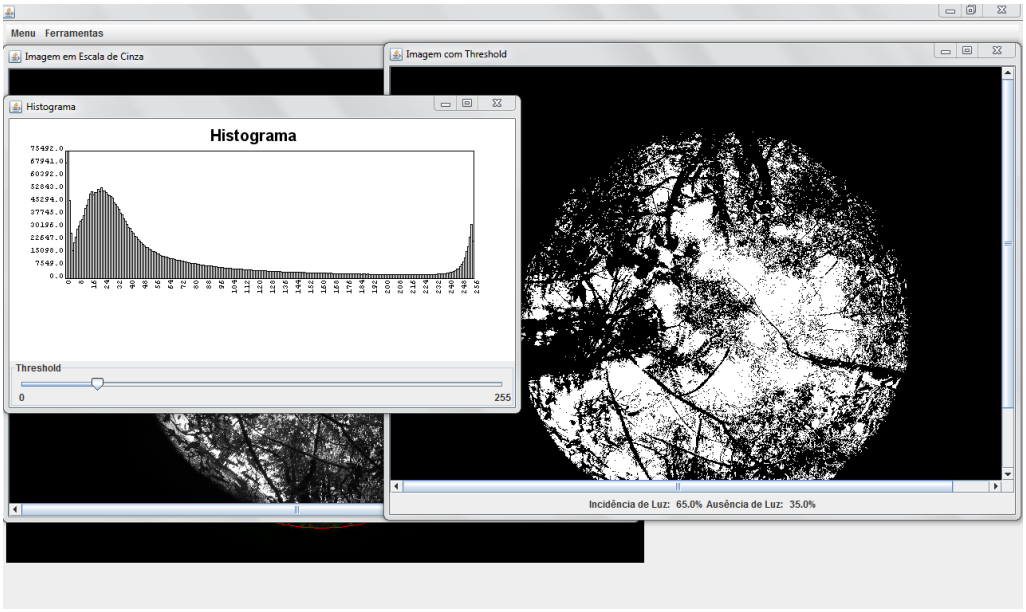

Figura 41: Tela após geração do Histograma

O relatório é gerado após a interação do usuário, depois de realizar todas as operações com a imagem, através do clique no botão ‘Gerar Relatório’ do menu ‘Ferramentas’. Nesse relatório, é discriminado o caminho da pasta que a imagem se encontra, seu nome, tamanho da imagem em pixels na relação altura x largura, quantidade de pixels da Região de Interesse, a incidência de luz (quantidade de pixels da cor branca) em porcentagem e quantidade de pixels e a cobertura da área (quantidade de pixels da cor preta) em porcentagem e quantidade de pixels.

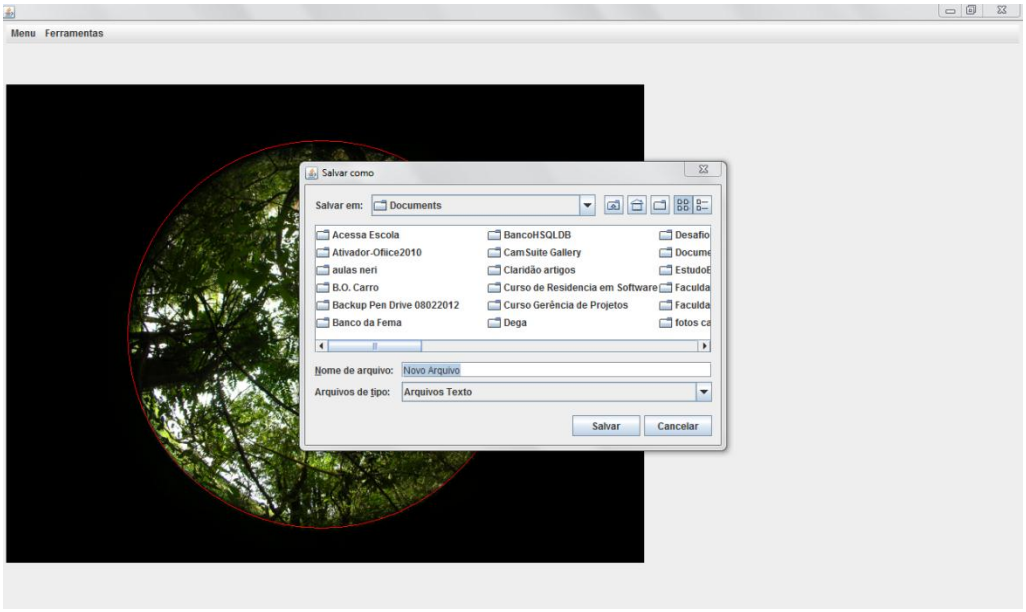

Figura 42: Tela após clique em “Gerar relatório da análise”

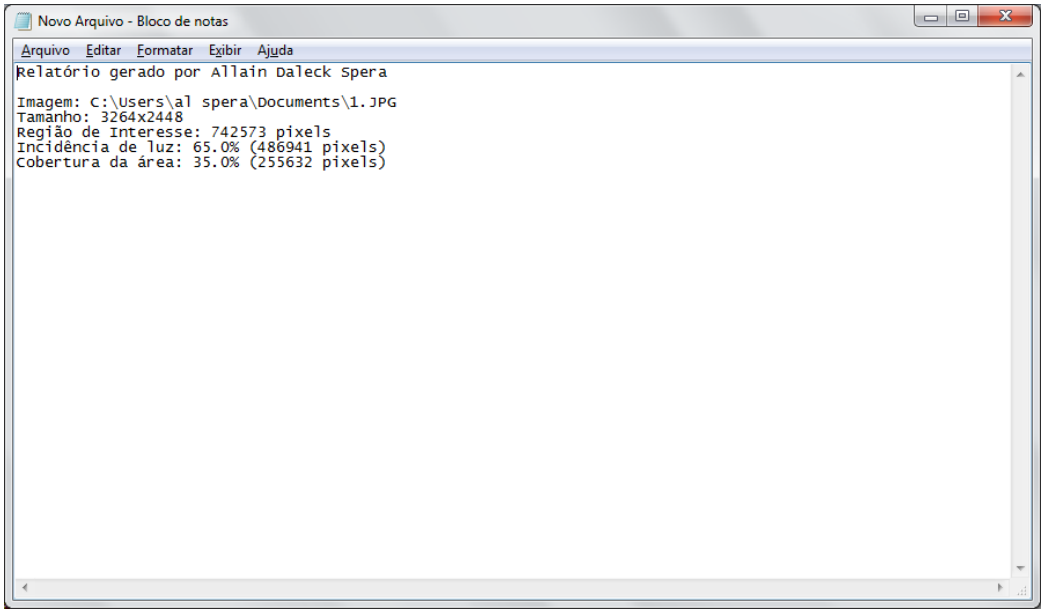

Figura 43: Relatório gerado ao final da análise

6 CONCLUSÃO

O desenvolvimento de um *software* que auxilie a análise e obtenção de dados utilizados em pesquisas voltadas para a preservação ambiental apresentou resultados satisfatórios. É clara a necessidade de uma solução simples e eficiente, que foi solucionada graças à apresentação do trabalho.

O programa resolve o problema apresentado, permitindo a análise de imagens em alta qualidade e sua utilização em variados computadores graças à portabilidade da tecnologia Java, calculando com precisão as informações obtidas da imagem e apresentando um relatório geral da consulta.

Visto que as ferramentas utilizadas pelo Instituto Florestal de Assis necessitavam de um ajuste prévio da resolução da imagem, bem como sua extensão, o produto final deste trabalho terá grande utilidade, pois a facilidade de se obter o relatório inclui apenas capturar a foto com a câmera e adicioná-la ao computador.

O projeto serve como base de estudos para a utilização do processamento de imagens aliado à tecnologia Java, assim como apresenta um aprofundamento sobre as ferramentas disponibilizadas pela API *Java Advanced Imaging* e suas aplicações em um trabalho real.

Como trabalho futuro é proposto salvar os dados em *XML*, bem como procurar uma forma de automaticamente encontrar o região circular de cobertura foliar. Pode-se ainda comparar o valor do Limiar Ótimo encontrado com base no filtro Prewitt e a posterior varredura do vetor média com os métodos tradicionais de encontrar o Limiar, visando validar o método proposto.

7 ANEXOS

7.1 Anexo A

```
/*
 * Created on Jun 10, 2005
 * @author Rafael Santos (rafael.santos@lac.inpe.br)
 *
 * Part of the Java Advanced Imaging Stuff site
 * (http://www.lac.inpe.br/~rafael.santos/Java/JAI)
 *
 * STATUS: Complete, but could be improved, for example, with:
 *   - Plotting more than one band of the histogram.
 *   - Considering the minimum number of pixels in a bin.
 *   - Customization as a JavaBean.
 *
 * Redistribution and usage conditions must be done under the
 * Creative Commons license:
 * English: http://creativecommons.org/licenses/by-nc-sa/2.0/br/deed.en
 * Portuguese: http://creativecommons.org/licenses/by-nc-
sa/2.0/br/deed.pt
 * More information on design and applications are on the projects'
page
 * (http://www.lac.inpe.br/~rafael.santos/Java/JAI).
 *
 *Alterado por: Allain Daleck Spera
 *Data da alteração: 30 de Julho de 2012
 *Atributos adicionados: private double threhsold = 0.0D
 *Métodos adicionados: public double getThreshold()
 *                     public void setThreshold(double threshold)
 *                     public int[] getCounts()
 *                     public void setCounts(int[] counts)
 *email: allaindspera@hotmail.com
 */

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Insets;
import java.awt.RenderingHints;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionListener;
import javax.media.jai.Histogram;
import javax.swing.JComponent;

/**
 * This class displays a histogram (instance of Histogram) as a
```

```
component.
 * Only the first histogram band ins considered for plotting.
 * The component has a tooltip which displays the bin index and bin
count for the
 * bin under the mouse cursor.
 */
public class DisplayHistogram extends JComponent implements
MouseMotionListener {

    // The histogram and its title.
    private Histogram histogram;
    private String title;
    // Some data and hints for the histogram plot.
    private int[] counts;
    private double maxCount;
    private int indexMultiplier = 1;
    private int skipIndexes = 8;
    // The components' dimensions.
    private int width,  height = 250;
    // Some constants for this component.
    private int verticalTicks = 10;
    private Insets border = new Insets(40, 70, 40, 30);
    private int binWidth = 3;
    private Color backgroundColor = Color.WHITE;
    private Color barColor = Color.BLACK;
    private Color marksColor = Color.BLACK;
    private Font fontSmall = new Font("monospaced", 0, 10);
    private Font fontLarge = new Font("Arial", Font.BOLD, 20);

    private double threshold = 0.0D;

      /**
     * The constructor for this class, which will set its fields' values
and get some information
     * about the histogram.
     * @param histogram the histogram to be plotted.
     * @param title the title of the plot.
     */
    public DisplayHistogram(Histogram histogram, String title) {

        this.histogram = histogram;
        this.title = title;
        // Calculate the components dimensions.
        width = histogram.getNumBins(0) * binWidth;
        // Get the histogram data.
        counts = histogram.getBins(0);
        // Get the max and min counts.
        maxCount = Integer.MIN_VALUE;
        for (int c = 0; c < counts.length; c++) {
            maxCount = Math.max(maxCount, counts[c]);
        }
        addMouseMotionListener(this);
    }

    /**
     * Override the default bin width (for plotting)
     */
    public void setBinWidth(int newWidth) {
```

```
        binWidth = newWidth;
        width = histogram.getNumBins(0) * binWidth;
    }

    /**
     * Override the default height for the plot.
     * @param h the new height.
     */
    public void setHeight(int h) {
        height = h;
    }

    /**
     * Override the index multiplying factor (for bins with width != 1)
     */
    public void setIndexMultiplier(int i) {
        indexMultiplier = i;
    }

    /**
     * Override the index skipping factor (determines how many labels
will be
     * printed on the index axis).
     */
    public void setSkipIndexes(int i) {
        skipIndexes = i;
    }

    /**
     * Set the maximum value (used to scale the histogram y-axis). The
default value
     * is defined in the constructor and can be overriden with this
method.
     */
    public void setMaxCount(int m) {
        maxCount = m;
    }

    /**
     * This method informs the maximum size of this component, which will
be the same as the preferred size.
     */
    public Dimension getMaximumSize() {
        return getPreferredSize();
    }

    /**
     * This method informs the minimum size of this component, which will
be the same as the preferred size.
     */
    public Dimension getMinimumSize() {
        return getPreferredSize();
    }

    /**
     * This method informs the preferred size of this component, which
will be constant.
     */
```

```
    public Dimension getPreferredSize() {
        return new Dimension(width + border.left + border.right, height +
border.top + border.bottom);
    }

    /**
     * This method will paint the component.
     */
    protected void paintComponent(Graphics g) {
        Graphics2D g2d = (Graphics2D) g;
        // Draw the background.
        g2d.setColor(backgroundColor);
        g2d.fillRect(0, 0, getSize().width, getSize().height);
        // Draw some marks.
        g2d.setColor(marksColor);
        g2d.drawRect(border.left, border.top, width, height);
        // Draw the histogram bars.
        g2d.setColor(barColor);
        for (int bin = 0; bin < histogram.getNumBins(0); bin++) {
            int x = border.left + bin * binWidth;
            double barStarts = border.top + height * (maxCount -
counts[bin]) / (1. * maxCount);
            double barEnds = Math.ceil(height * counts[bin] / (1. *
maxCount));
            g2d.drawRect(x, (int) barStarts, binWidth, (int) barEnds);
        }
        // Draw the values on the horizontal axis. We will plot only
1/8th of them.
        g2d.setColor(marksColor);
        g2d.setFont(fontSmall);
        FontMetrics metrics = g2d.getFontMetrics();
        int halfFontHeight = metrics.getHeight() / 2;
        for (int bin = 0; bin <= histogram.getNumBins(0); bin++) {
            if (bin % skipIndexes == 0) {
                String label = "" + (indexMultiplier * bin);
                int textHeight = metrics.stringWidth(label); // remember
it will be rotated!
                int x = border.left + bin * binWidth + binWidth / 2;
                g2d.translate(border.left + bin * binWidth +
halfFontHeight, border.top + height + textHeight + 2);
                g2d.rotate(-Math.PI / 2);
                g2d.drawString(label, 0, 0);
                g2d.rotate(Math.PI / 2);
                g2d.translate(-(border.left + bin * binWidth +
halfFontHeight), -(border.top + height + textHeight + 2));
            }
        }
        // Draw the values on the vertical axis. Let's draw only some of
them.
        double step = (int) (maxCount / verticalTicks);
        for (int l = 0; l <= verticalTicks; l++) // last will be done
separately
        {
            String label;
            if (l == verticalTicks) {
                label = "" + maxCount;
            } else {
                label = "" + (l * step);
```

```
            }
            int textWidth = metrics.stringWidth(label);
            g2d.drawString(label, border.left - 2 - textWidth, border.top
+ height - l * (height / verticalTicks));
        }
        // Draw the title.
        g2d.setFont(fontLarge);
        g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
        metrics = g2d.getFontMetrics();
        int textWidth = metrics.stringWidth(title);
        g2d.drawString(title, (border.left + width + border.right -
textWidth) / 2, 28);
    }

    /**
     * This method does not do anything, it is here to keep the
MouseMotionListener interface happy.
     */
    public void mouseDragged(MouseEvent e) {
    }

    /**
     * This method will be called when the mouse is moved over the
component. It will
     * set the tooltip text on the component to show the histogram data.
     */
    public void mouseMoved(MouseEvent e) {
        int x = e.getX();
        int y = e.getY();
        // Don't show anything out of the plot region.
        if ((x > border.left) && (x < border.left + width) && (y >
border.top) && (y < border.top + height)) {
            // Convert the X to an index on the histogram.
            x = (x - border.left) / binWidth;
            y = counts[x];
            setToolTipText((indexMultiplier * x) + ": " + y);
        } else {
            setToolTipText(null);
        }
    }


    public void mouseClicked(MouseEvent e){
      int x = e.getX();
      int y = e.getY();
      if ((x > border.left) && (x < border.left + width) && (y >
border.top) && (y < border.top + height)) {
            // Convert the X to an index on the histogram.
            x = (x - border.left) / binWidth;
            y = counts[x];
            setThreshold(y);

        } else {
            setToolTipText(null);
        }
    }

    public double getThreshold() {
```

```
            return threshold;
      }


    public void setThreshold(double threshold) {
            this.threshold = threshold;

      }

    public int[] getCounts() {
            return counts;
      }

    public void setCounts(int[] counts) {
            this.counts = counts;

      }

} // end class
```

REFERÊNCIAS

ALBUQUERQUE, Márcio Portes de; ALBUQUERQUE, Marcelo Portes de. **Processamento de Imagens: Métodos e Análise**. 12p. Centro Brasileiro de Pesquisas Físicas – CBPF/MCT, Rio de Janeiro – RJ.

DEITEL, H. M.; DEITEL, P. J.. **Java: como programar.** 6. ed. São Paulo: Pearson Prentice Hall, 2005.

GONZALEZ, R. C.; WOODS, R. E.. **Processamento de Imagens Digitais.** São Paulo: Editora Edgard Blücher Ltda., 2000.

LI, Xin. ***To convert RGB to gray***. Disponível em: < http://www.math.ucf.edu/~xli/To%20display%20image%20ETC.pdf >. Acesso em: 20 jul. 2012.

MARQUES FILHO, O.; VIEIRA NETO, H. **Processamento digital de imagens**. Rio de Janeiro: Brasport Livros e Multimídia, 1999.

ORACLE. ***Java Advanced Imaging: Package Summary***. Disponível em: < http://docs.oracle.com/cd/E17802_01/products/products/java-media/jai/forDevelopers/jai-apidocs/javax/media/jai/package-summary.html >. Acesso em: 15 jul. 2012.

SANTOS, Rafael. *Java Advanced Imaging API: A Tutorial*. Divisão de Sensoriamento Remoto – Instituto de Ensinos Avançados – Centro Técnico Aeroespacial. **Revista de Informática Teórica e Aplicada**, v.11, n.1, 2004.

SUN MICROSYSTEMS, Inc. ***Java Advanced Imaging API – Tutorial***. Disponível em: < http://java.sun.com/developer/onlineTraining/javaai/jai/ >. Acesso em: 15 jul. 2012.

SUN MICROSYSTEMS, Inc. ***Programming in Java Advanced Imaging*****.** Disponível em: < http://java.sun.com/products/java-media/jai/forDevelopers/jai1_0_1guide-unc/JAITOC.fm.html >. Acesso em: 03 jul. 2012.

WHITMORE, T. C. *Canopy gaps and the two major groups of forest trees*. ***Ecology***, v. 70, n. 3, p.536-538, 1989.