Capítulo 5

Técnicas de Processamento Digital de Imagens com Java

Iális Cavalcante de Paula Júnior

Abstract

The human brain to processes and recognizes a huge data quantity from visual nature. This motivates the development of techniques and devices in order to accomplish this capacity. The knowledge about this visual system and the studies of available methodologies onto balance the image to a actual problem are important to develop automatable computational solutions. In a sense, the Digital Image Processing (DIP) permits to enable a diversity of applications in different research areas such as medicine, biology, engineering, meteorology, cartography and others one. Some of these applications are developed in programming languages with established performance and security in dealing with this type of data (digital image). In this work, it will adopt the Sun Java language in the support for DIP techniques in a simple application of regions description. It is still made a comparative among the main Application Program Interfaces (APIs) of this domain and the more adequate usage of memory.

Resumo

O cérebro humano processa e interpreta imensa quantidade de dados de natureza visual. Isso motiva o desenvolvimento de técnicas e dispositivos de modo a estender ainda mais essa capacidade. O conhecimento de como reage este sistema visual e o estudo de metodologias disponíveis para melhor adequar a imagem a um problema real são importantes para desenvolver soluções computacionais automatizáveis. Neste sentido, a área de Processamento Digital de Imagens (PDI) permite viabilizar diversas aplicações em diferentes áreas de estudo como na medicina, biologia, engenharia, meteorologia, cartografia, entre outras. Algumas dessas aplicações são desenvolvidas em linguagens de programação com reconhecido desempenho e segurança na abordagem deste tipo de dado (imagem digital). Neste trabalho, será seguida a linguagem Sun Java como base para realizar técnicas de PDI em uma aplicação simples de descrição de regiões. Também é feito ainda um comparativo entre as principais Application Program Interfaces (APIs) deste domínio e o melhor uso de memória.

5.1. Introdução

Até que ponto a interpretação humana pode ser afetada por ajustes em uma informação visual? Como adequar o processamento dos dados em cenas reais para a percepção automática através de máquinas? Esses e outros questionamentos formam a motivação para o estudo de processamento digital de imagens. E ainda, isso gera o grande interesse de diferentes áreas de aplicação no uso de suas técnicas computacionais. Como área de processamento visual de dados, alguns conceitos relacionados a Processamento Digital de Imagens (PDI) se assemelham ao que é aplicado em outras áreas de estudo como Computação Gráfica e Visão Computacional. Apesar das semelhanças, esses domínios se diferenciam justamente pelos tipos de informação utilizados em suas respectivas entradas e saídas. Como pode-se perceber pela Figura 5.1, processamento de imagens será todo processamento digital em que as entradas e saídas são imagens.

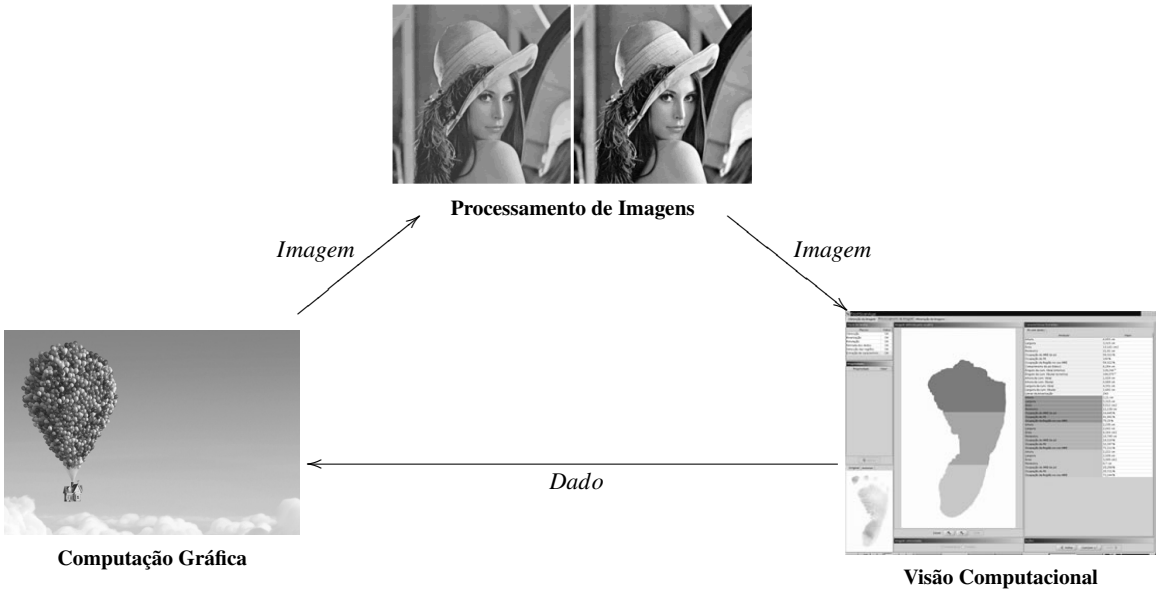


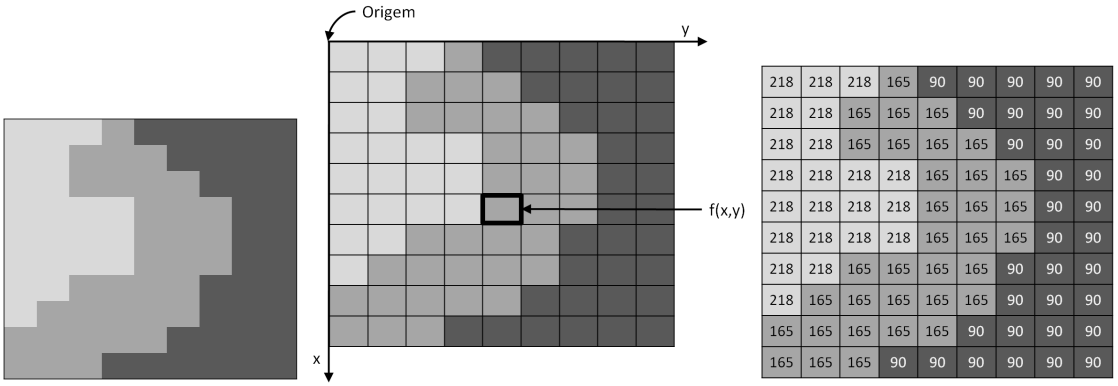
Figura 5.1. Comparativo das informações de entrada/saída entre Processamento de Imagens, Computação Gráfica (Fonte: Animação UP da Pixar Pictures - `http://www.pixar.com/featurefilms/up/`) e Visão Computacional (Fonte: Sistema FootScanAge - `http://www.imago.ufpr.br/footscan.html`).

O mesmo não acontece para os outros casos, já que um sistema de Computação Gráfica exige dados na forma vetorial ou modelos matemáticos [Gomes e Velho 1994] como entrada. Isso é bastante perceptível quando se trabalha com jogos digitais, pois no momento que o usuário aciona um comando no *joystick* é gerada uma nova imagem pelo jogo, o que resulta em uma nova cena ou movimentação/ação de um personagem. O estudo de Computação Gráfica envolve modelagem e representação de dados, visualização e projeção de curvas e superfícies, efeitos tridimensionais, animação, renderização e outras técnicas que são adequadas à geração de imagens/cenas [Azevedo et al. 2007, Velho et al. 2009].

Em Visão Computacional, dados são exibidos na saída do sistema e representam características extraídas da imagem de sua entrada. Estas características são proces-

sadas para técnicas diferenciadas, principalmente em tomadas de decisões inteligentes como sistemas especialistas, redes neurais, algoritmos genéticos, etc. Quando os dados não passam por essa etapa anterior, ainda se considera sistema de PDI (para alguns pesquisadores [Mokhtarian e Mackworth 1986, Lee et al. 1995, Paula Júnior et al. 2006, César Júnior e Costa 1996]) e envolve uma área especificada como Análise de Imagens [Costa e César Júnior 2009] - esta encontre-se em nível intermediário a Processamento de Imagens e Visão Computacional. Esta última é bastante implementada em aplicações de Robótica [Castleman 1996]. Um exemplo específico de aplicação de Visão Computacional é o sistema *FootScanAge* [Silva et al. 2006] que avalia, a partir de imagens obtidas das superfícies plantares de recém-nascidos, a idade gestacional[1] de um bebê pré-maturo (e sem histórico de pré-natal em sua gestação) para estudo de suas características neurofisiológicas. Como saída, o *FootScanAge* apresenta um *score* para essa idade gestacional que permite definir um fator de sobrevida ao paciente e seu tratamento adequado.

Com relação às técnicas de PDI, em geral são abordadas operações de realce, compressão, restauração ou transformações de informações adquiridas na forma de *pixels* (*picture elements*). Qualquer imagem digital pode ser representada por uma função bidimensional de intensidade da luz $f(x,y)$, onde x e y denotam as coordenadas espaciais [Gonzalez e Woods 2008]. Assim, para imagens em nível de cinza, o valor de f em qualquer ponto (x,y) é proporcional ao brilho (intensidade do tom de cinza) da imagem naquele ponto. A Figura 5.2 mostra a representação de uma imagem no domínio discreto e sua subdivisão em *pixels*. Assume-se uma convenção dos eixos de função para a abordagem de imagens digitais, como segue na Figura 5.2(b).



(a) Imagem original. (b) Imagem discreta. (c) Intensidade dos pixels.

Figura 5.2. Descrição dos valores de pixels em uma imagem.

Os primeiros registros de estudos em Processamento Digital de Imagens foram notificados nos anos 60, impulsionados pelos projetos de pesquisa criados pela NASA, nos Estados Unidos da América durante a Guerra Fria [Castleman 1996]. Nas décadas seguintes surgiriam mais atividades que necessitariam dos adventos proporcionados por esta área de conhecimento; entre estas estam a medicina, biologia, meteorologia, astronomia, agricultura, sensoriamento remoto, cartografia e a indústria petrolífera. Percebe-se

[1] A idade gestacional é o tempo, medido em semanas ou em dias completos, decorrido desde o início da última menstruação (ou data da última menstruação) da mulher grávida.

desta forma que o processamento de imagens é essencialmente uma área multidisciplinar, tanto na atuação quanto na derivação, pois suas técnicas baseiam-se na física, neurofisiologia, engenharia elétrica e ciência da computação. Paralelamente, com o avanço dos recursos computacionais permitiu-se alcançar um desenvolvimento mais eficiente e robusto do processamento digital de imagens.

Dentro da pesquisa na área de saúde, pode-se encontrar alguns trabalhos relevantes com PDI. Em [Passarinho et al. 2006], o tratamento fisioterápico de Reeducação Postural Global (RPG) permite uma abordagem com a captação de imagens dos pacientes durante o tratamento. Destas imagens é possível extrair a forma postural de cada paciente e a partir disso obter características desta imagem. Com essas informações, permite-se ao especialista avaliar o desempenho do tratamento em cada caso clínico. Em imagens clínicas, como exames de mamografia, imagens de ultra-som [Calíope et al. 2004] e ressonância magnética há diversas implementações de filtros digitais para possibilitarem melhor avaliação dos exames clínicos por parte dos médicos e técnicos de saúde. O estudo nesses tipos de imagens deve apresentar resultados cada vez mais precisos, como na detecção de aneurismas [Martins et al. 2008], já que uma identificação errônea de um patologia (ou ainda a não percepção desta) resulta grandes problemas tanto para pacientes quanto para a equipe médica.

A indústria petrolífera também apresenta interesse em soluções de PDI para problemas críticos como detecção de derramamento de óleo no mar [Lopez et al. 2006] e localização automática de corrosão e depredação de oleodutos e demais equipamentos marítimos e terrestres [Bento et al. 2009b, Bento et al. 2009a]. Passando para o contexto da produção agrícola, o crescimento dos agronegócios vem contribuindo para a inserção das pesquisas de processamento de imagens na agricultura. Na tentativa de distribuição de laranjas [Ramalho e Medeiros 2003] e outros produtos perecíveis, busca-se a criação de sistemas de inspeção visual automática para seleção dos alimentos, direcionando a classificação para os diferentes mercados a que o produtor esteja voltado.

Em se tratando de classificação, na biologia existem diversas abordagens para a busca de similaridade entre formas naturais [Bernier e Landry 2003]. Nesta similaridade são avaliados descritores de formas [Costa et al. 2004] para as diferentes imagens de espécies de seres e corpos trabalhadas: plantas, folhas, aves, insetos, bactérias, células, etc. Também há classificação de informação no processamento de imagens de Radar de Abertura Sintética (Synthetic Aperture Radar) [Marques et al. 2004]. Reconhecimento de regiões de cultivo, áreas de floresta e urbanas é importante para a avaliação do crescimento das cidades e do aproveitamento dos recursos naturais. Além disso, permite-se com aplicações de sensoriamento remoto a detecção de alvos [Marques et al. 2008] e acidentes marítimos nas regiões costeiras cobertas por satélites.

Este capítulo tem um caráter introdutório sobre os conceitos envolvidos em processamento de imagens e a geração de pequenas aplicações com a linguagem de programação Java neste domínio. Após apresentar uma visão geral sobre PDI e suas aplicações em diferentes áreas de pesquisa, serão tratadas as principais Application Program Interfaces (APIs) Java voltadas para sistemas de processamento de imagens na Seção 5.2. Em seguida, técnicas relevantes para aplicações de PDI são descritas na Seção 5.3. Na Seção 5.4, será apresentado ao leitor o desenvolvimento de um aplicativo simples de descrição

de regiões em imagens. Ao final, na Seção 5.5, são expostas as conclusões e considerações finais deste trabalho.

5.2. APIs Java para Imagens

Há alguns anos atrás, a comunidade de processamento de imagens era formada por um grupo pequeno de pesquisadores que tinham acesso limitado a ferramentas comerciais de PDI, algumas desnecessárias, e que eram desenvolvidas sob componentes de *softwares* proprietários. Estes eram resumidos a pequenos pacotes de *software* para carregamento de arquivos de imagem e armazenamento dos mesmos em disco e ainda apresentavam documentações incompletas e formatos de arquivos proprietários [Murray e VanRyper 1996]. Surgiram então novos formatos, inicialmente desenvolvidos para aplicações específicas, mas muitos dos quais não despontaram ou foram esquecidos [Miano 1999]. Nos anos 80 e 90 surgiram alguns novos formatos de imagens e diversos softwares para convertê-los, além do suporte dos formatos criados para modelos específico de hardware definidos pelos desenvolvedores.

Atualmente, apenas um pequeno conjunto desses formatos permaneceu na ativa: *Tagged Image File Format* (TIFF), *Graphics Interchange Format* (GIF), *Portable Network Graphics* (PNG), padrão JPEG, *Windows Bitmap* (BMP) e o *Portable Bitmap Format* (PBM) que possui os tipos PBM (*portable bitmap*) para bitmaps binários, PGM (*portable graymap*) para imagens em nível de cinza e PMN (*portable any map*) para imagens coloridas [Burger e Burge 2009]. Esses são suportados pela maioria das APIs padrões para C/C++ e Java.

Nos últimos anos, com o incremento da capacidade de compressão dos formatos de arquivos de imagem e a evolução das linguagens de programação, muitos desenvolvedores têm definido seus aplicativos utilizando a linguagem de programação Java para sistemas de PDI [Rodrigues 2001]. A linguagem possui algumas características que têm sido atrativas à comunidade de desenvolvimento [Deitel e Deitel 2005] como: é livre, portável (*"write once, run anywhere"*), segura, sintaxe similar à linguagem C, possui facilidade de internacionalização dos caracteres, vasta documentação, coletor de lixo (para desalocação automática de memória) e facilidade de criação de programação distribuída e concorrente. Há ainda o paradigma orientado a objetos [Santos 2003] que define o *modus operandi* de Java e descreve a estrutura de todas as APIs criadas a partir desta linguagem.

Esta seção apresenta uma alternativa flexível, portável e sem custos para que o desenvolvedor faça uso da linguagem Java na implementação de seus programas. Para representação e processamento de imagens, a *Java Advanced Imaging* (JAI) API (criada e mantida pela Sun Microsystems) pode ser trabalhada. Ainda que a API não faça parte de um *software* de PDI completo, as operações existentes e possibilidades de extensão aliadas ao baixo custo e implementação simples tornam esta API uma opção atrativa para desenvolvimento de algoritmos de processamento de imagens [Santos 2004].

Nesta API, a *PlanarImage* é a principal classe para representação de imagens na JAI e traz mais flexibilidade que a classe nativa *BufferedImage*. Em ambas as classes, seus *pixels* são armazenados em uma instância de *Raster* que contém uma instância de uma subclasse de *DataBuffer* (*DataBufferByte*, *DataBufferFloat*, *DataBufferInt*, *DataBufferDouble*, *DataBufferShort*, etc.) de acordo com o comportamento seguido pela instância

de uma subclasse de *SampleModel*. Uma instância de *PlanarImage* também tem uma instância de *ColorModel* associada a ela, a qual contém uma instância de *ColorSpace*, que determina como os valores de *pixels* serão deslocados aos valores de cores [Santos 2004]. Figura 5.3 mostra a composição geral da classe *PlanarImage* com as demais classes da API Java 2D [Liang e Zhang 2006].

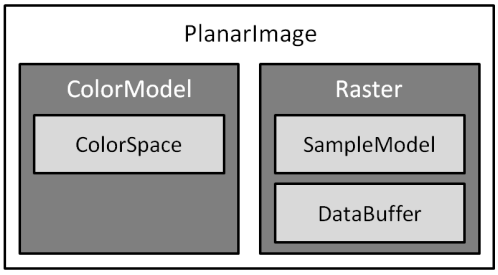
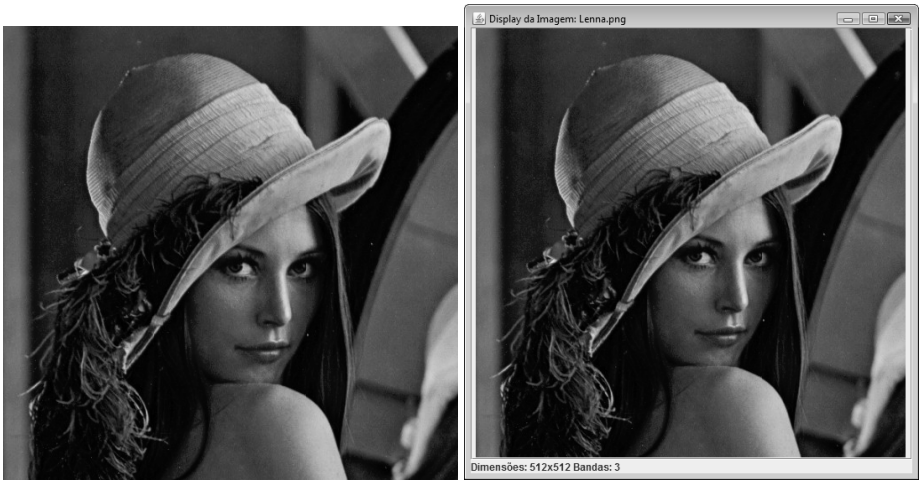


Figura 5.3. Estrutura da classe de suporte a imagem na API JAI. Adaptado de Santos (2004).

Uma *PlanarImage* permite somente leitura à imagem, ou seja, pode ser criada e seus valores de *pixels* podem ser lidos de várias maneiras, mas não há métodos implementados para a modificação de seus elementos [Santos]. *PlanarImage* possui uma grande flexibilidade já que pode ter a origem da imagem em uma posição diferente da coordenada (0,0), ou ainda coordenadas de *pixels* com valores negativos [Rodrigues 2001].



(a) (b)

Figura 5.4. Imagem. Fonte: Imagem adaptada de `http://www.lenna.org/`.

Para visualização, a API JAI disponibiliza um componente simples e extensível determinado pela classe *DisplayJAI*. Esta é uma subclasse de *JPanel* e pode ser trabalhado como qualquer outro componente gráfico Java. Na Tabela 5.1, é apresentado o código-fonte referente a uma classe de exibição de imagem chamada *ExibeImagem* que faz uso de uma instância de *DisplayJAI*. Essa mesma instância é associada com um objeto de *JScrollPane* para suportar (com barras de rolagem) qualquer imagem que possua dimensão

maior que a interface gráfica definida. O resultado da execução da classe *ExibeImagem* está disposto na Figura 5.4.

Tabela 5.1. Código da classe ExibeImagem.

```
01  /**
02   * Lendo imagem usando apenas APIs nativas
03   * @author ERCEMAPI 2009
04   */
05  public class ExibeImagem extends JFrame {
06    // Construtor padrão: define interface e exibe a imagem lida na mesma
07    public ExibeImagem() throws IOException {
08        BufferedImage imagem = ImageIO.read(new File("Lenna.png"));
09        String infoImagem = "Dimensões: "+imagem.getWidth()+
10            "x"+imagem.getHeight()+" Bandas: "+
11            imagem.getRaster().getNumBands();
12        ImageIcon icone = new ImageIcon(imagem);
13        JLabel labImagem = new JLabel(icone);
14        this.setTitle("Display da Imagem: "+"Lenna.png");
15        Container contentPane = this.getContentPane();
16        contentPane.setLayout(new BorderLayout());
17        contentPane.add(new JScrollPane(labImagem), BorderLayout.CENTER);
18        contentPane.add(new JLabel(infoImagem), BorderLayout.SOUTH);
19        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20        this.setSize(imagem.getWidth(),imagem.getHeight());
21        this.setVisible(true);
22    } // fim do construtor
23    // Método principal
24    public static void main(String args[]){
25        try{ // tratamento de exceção de E/S
26            ExibeImagem appExibeImg = new ExibeImagem();
27        }catch(IOException exc){
28            System.out.println("Erro de leitura! "+exc.getMessage());
29        }
30    } // fim do método principal
31  } // fim da classe
```

Outra alternativa aos desenvolvedores na área de processamento de imagens com Java é o uso da API e *software* ImageJ [Rasband 2009]. Criada por Wayne Rasband, ela apresenta em seu site oficial (`http://rsbweb.nih.gov/ij/`) a versão atual do aplicativo, pacotes, documentação, *updates*, código-fonte completo, imagens para teste e uma coleção continuamente crescente de *plugins* [Burger e Burge 2009] desenvolvidos por terceiros e que podem ser adicionados ao aplicativo. A organização da API ImageJ está disposta na Figura 5.5. Algumas classes da API são definidas a partir da API AWT de interface gráfica. Isso permite que os resultados de suas operações sejam visualizados também pela API Swing da linguagem Java, porque essa última herda as características de

AWT. Com relação aos *plugins*, os desenvolvedores podem criar seus próprios ou reaproveitar *plugins* já utilizados no aplicativo. Isso dá uma grande flexibilidade na criação de classes mas não impede a produção de códigos redundantes entre os trabalhos de vários desenvolvedores.

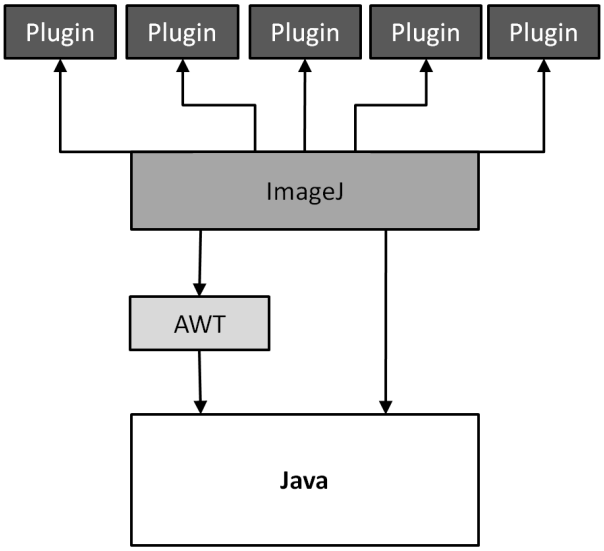


Figura 5.5. Estrutura geral da API ImageJ. Adaptado de Burger & Burge (2009).

Apesar desse problema, as classes nativas da API oferecem suporte à maioria das técnicas de PDI. Duas classes são bastante utilizadas: *ImagePlus* e *ByteProcessor*. A classe *ImagePlus* normalmente exige um objeto de *java.awt.Image* na instanciação de seus objetos e sempre carrega a imagem trabalhada para a visualização na interface gráfica (associando com a classe *ImageCanvas*). Com relação à classe *ByteProcessor*, essa é a base da execução das principais técnicas de processamento de imagens com a ImageJ. A partir de uma instância dessa classe é possível chamar diretamente um método para muitas das operações a serem vistas na próxima seção.

O aplicativo ImageJ foi desenvolvido para teste das operações implementadas pela API e pelos *plugins* mas ele tem sido utilizado também em uma abordagem didática para exibição dos resultados das técnicas de PDI. Uma implementação mais completa com esta API é descrita na Seção 5.4.

Esta seção apresentou uma visão geral sobre as API JAI e ImageJ. Essas são as principais referências para processamento de imagens usando a linguagem de programação Java. Suas definições e arquitetura foram dispostas aqui, mas o modo de uso de seus operadores será apresentado nas seções seguintes.

5.3. Técnicas de Processamento de Imagens

Esta seção tem o objetivo de apresentar técnicas relevantes de processamento de imagens que deem o embasamento teórico necessário para o desenvolvimento de um aplicativo usando a linguagem Java. Para isso, serão discutidos métodos relacionados a realce de imagens, segmentação e morfologia matemática.

Mas a princípio, é requerido um conhecimento sobre os tipo de imagens tratadas em PDI. Primeiramente, deve-se considerar as imagens binárias como um tipo de imagem digital (ver Figura 5.6(a)), em que o valor assumido por cada *pixel* é 0 (cor preta) ou 1 (cor branca). Outro tipo considerado são as imagens em nível de cinza (ver Figura 5.6(b)), onde cada *pixel* pode assumir um valor entre 0 e 255 e levando em conta que quanto menor o nível de cinza, mais próximo da cor preta, e quanto maior o nível de cinza, mais próximo da cor branca. Finalmente, as imagens coloridas (ver Figura 5.6(c)) seguem um modelo de cor para a definição dos valores de seus pixels. Nesta situação, a imagem possui mais de uma matriz de nível de cinza de acordo com o modelo assumido: modelo RGB, possui três matrizes com a influência de uma cor em cada (*Red* - Vermelha, *Green* - Verde e *Blue* - Azul); no modelo CMYK há quatro camadas de nível de cinza para cada cor definida na modelagem (*Cyan* - Ciano, *Magenta* - Magenta, *Yellow* - Amarela e *Black* - Preta); e a mesma lógica é seguida nos demais modelos de cores. As imagens coloridas não são foco deste trabalho, já que o processamento a ser proposto é direcionado a imagens em nível de cinza.

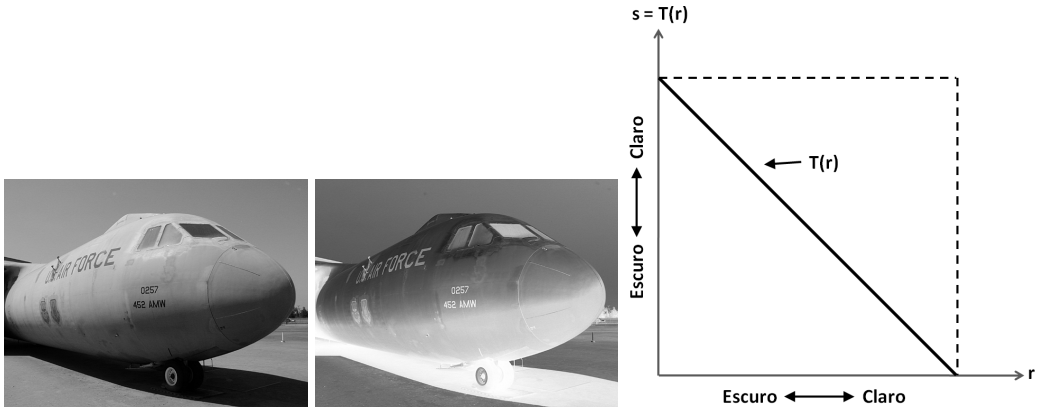
(a) Imagem binária. (b) Imagem em níveis de cinza. (c) Imagem colorida.

Figura 5.6. Tipos de Imagens. Adaptado de Meyer & Beucher (1990).

Nem sempre a imagem a ser processada encontra-se em um estado ideal para ser trabalhada. Isso quer dizer, que a imagem pode apresentar distorções indesejáveis e ter sua qualidade visual afetada. Essa problemática motiva o estudo de técnicas de realce em imagens, que tem como objetos suprimir as distorções da imagem e enfatizar características específicas da mesma. Este capítulo se resume às técnicas de realce no domínio do espaço, que seguem: negativo da imagem e filtragem (passa-baixa e passa-alta), em que todas são funções de transformação ($g(x,y) = T(f(x,y)) \leftrightarrow s = T(r)$).

Em alguns exames clínicos e resíduos de filme fotográfico, pode-se identificar a aplicação do processo de negativo em uma imagem (ver Figura 5.7) que é considerado um processamento ponto-a-ponto. Esta técnica se resume em submeter uma imagem à função de transformação dos níveis de cinza, como na Figura 5.7(c). Isso ocorre com imagens em nível de cinza, já que o mesmo processo em imagens binárias se torna o método de inversão dos pixels ($s = L - 1 - r$). Como resultado, no instante que a intensidade da imagem de saída diminui então a intensidade da entrada aumenta. Isso permitirá realçar regiões específicas na imagem.

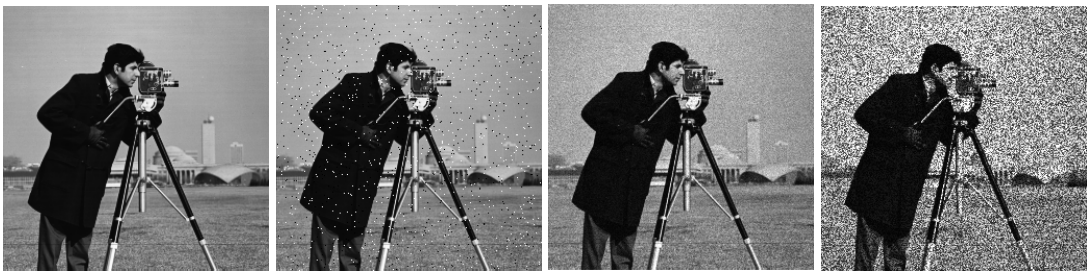
Distorções presentes em uma imagem afetam a percepção da real informação pre-



(a) Imagem original. (b) Imagem processada. (c) Função de transformação.

Figura 5.7. Processo de negativo de uma imagem. A imagem é submetida à função de transformação dos níveis de cinza. Adaptado de Burger & Burge (2009).

sente na mesma. Uma distorção crítica é a presença de ruído na imagem. Esse fator implica em nova informação na imagem, diferente do seu comportamento real. Normalmente os ruídos são definidos por uma distribuição estatística, como o modelo gaussiano, *speckle* e de poisson. Na Figura 5.8 estão identificados os efeitos de alguns tipos de ruído na mesma imagem.



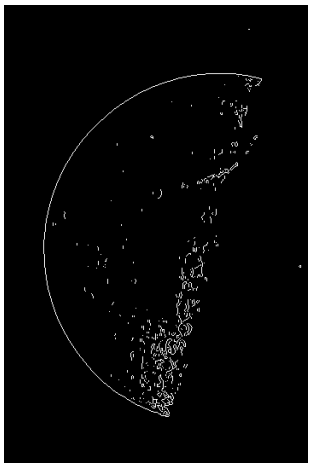
(a) Imagem original. (b) Presença de ruído "sal e pimenta". (c) Presença de ruído de Poisson. (d) Presença de ruído spackle.

Figura 5.8. Imagem sob efeito de diferentes tipos de ruído.

Para eliminação do ruído em uma imagem, faz-se necessário aplicar técnicas de realce com processamento por máscara. Isso consiste em definir uma nova imagem, de dimensão menor do que aquela a ser processada, e aplicar valores a cada um dos elementos dessa nova imagem que será chamada de máscara (ou janela). Essa janela percorre toda a imagem, e faz uma avaliação pixel a pixel de acordo com sua dimensão. A distribuição de valores nessa máscara define o tipo de filtragem a ser executada: passa-alta (preserva as informações de alta-frequência, como bordas e ruído, e elimina as demais) ou passa-baixa (preserva as informações de baixa-frequência, como regiões homogêneas em seus níveis de cinza, e elimina o restante). Na Figura 5.9 estão dispostos três exemplos de filtro passa-alta com suas respectivas máscaras. Cada máscara apresenta um efeito diferenciado no resultado da filtragem. O filtro de Roberts é um exemplo também de gradiente da

imagem, o que implica na representação do sentido de crescimento das bordas presentes nela.

(a) Imagem original.

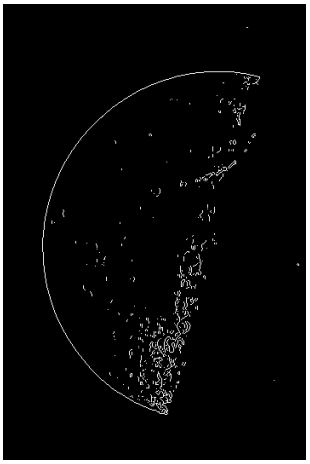


(b) Filtro de Prewitt.

-1	-1	-1
0	0	0
1	1	1

-1	0	1
-1	0	1
-1	0	1

(c) Máscaras de Prewitt.

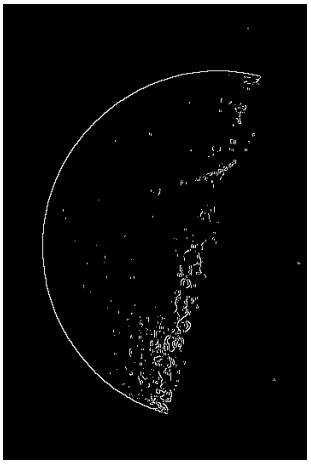


(d) Filtro de Sobel.

-1	-2	-1
0	0	0
1	2	1

-1	0	1
-2	0	2
-1	0	1

(e) Máscaras de Sobel.



(f) Filtro de Roberts.

1	0
0	-1

0	1
-1	0

(g) Máscaras de Roberts.

Figura 5.9. Detectores de bordas (gradiente da imagem).

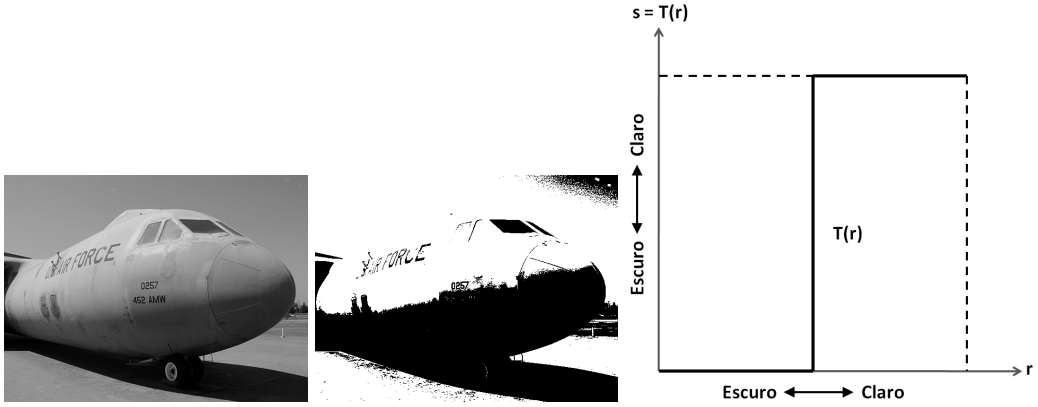
Com os filtros passa-baixa tem-se um comportamento diferenciado. A partir da Figura 5.10, pode-se notar a influência do filtro da mediana com a mudança na dimensão da janela. O filtro da mediana consiste em coletar os valores dos pixels abordados pela máscara, ordená-los e definir o valor da mediana no pixel da imagem correspondente ao centro da máscara. À medida que a janela cresce, elimina-se mais informação de alta-frequência (ruído e também as bordas da imagem). Considerando-se borda qualquer região com diferença de intensidade dos seus pixels, tem-se então uma distorção da borda proporcional à eliminação do ruído. Nesse caso, tem-se uma relação custo-benefício em que deve-se levar em conta qual a informação mais importante para a aplicação.

A importância do realce em uma imagem é percebida no momento em que o processo de segmentação é efetuado. O método de segmentação precisa que a imagem tratada

(a) (b) (c) (d)

Figura 5.10. Aplicação do filtro da mediana com diferentes máscaras. Na (a) imagem ruidosa, originada do efeito do ruído "sal e pimenta" (com maior intensidade) sob a imagem presente na Figura 5.8(a), foi aplicada a filtragem com modificação nas máscaras: (b) janela 3x3, (c) janela 5x5 e (d) janela 7x7.

esteja livre de distorções para cumprir seu objetivo que é separar as regiões presentes na imagem. A técnica mais simples é limiarização (ver Figura5.11), que consiste em definir um tom de cinza como limiar e modificar todos os pixels da imagem com base no mesmo. Isso ocorre a partir de um comparativo, em que o pixel com intensidade menor ou igual ao limiar recebe valor 0 (zero) e, em caso contrário, a intensidade do pixel receberá valor igual a 255. Assim duas regiões são separadas na imagem.



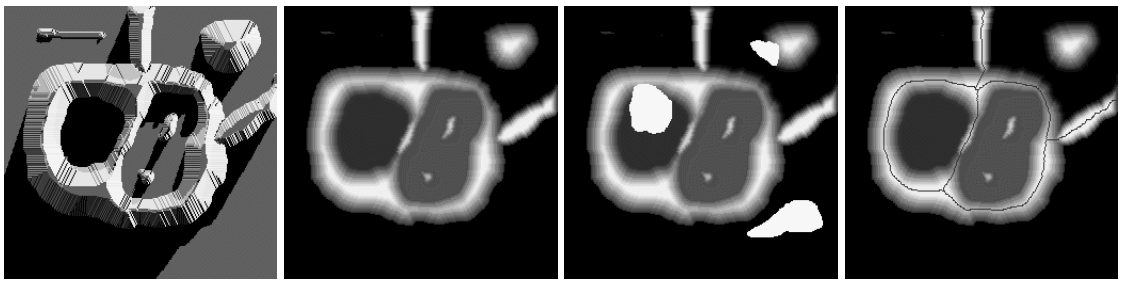
(a) Imagem original. (b) Imagem processada. (c) Função de transformação.

Figura 5.11. Processo de Limiarização. Imagem é submetida à função de transformação dos níveis de cinza. Adaptado de Burger & Burge (2009) e Gonzalez & Woods (2008).

Uma técnica de segmentação bastante conhecida, mas que apresenta maior complexidade, é o método *watershed*[2] [Vincent e Soille 1991, Meyer e Beucher 1990]. Aqui são necessárias duas informações: a imagem de gradiente e a imagem de marcador(es), que são definidas com base na imagem processada. Essa imagem de marcador(es) deve apresentar em que região(ões) da imagem o processo terá início e o gradiente informa

[2]Animação didática sobre funcionamento da técnica está disponível em `http://cmm.ensmp.fr/~ beucher/wtshed.html`.

onde o processo é encerrado. De um modo geral, a técnica de *watershed* traz uma representação de uma "represa" que inicia uma "inundação" a partir dos marcadores de tal forma que encha a imagem de "água" até alcançar a região identificada pelo gradiente. A execução do método pode ser acompanhado na Figura 5.12.



(a) Imagem original. (b) Imagem em níveis de cinza. (c) Imagem de marcadores. (d) Imagem de gradiente.

Figura 5.12. Aplicação da técnica *watershed*. Adaptado de Meyer & Beucher (1990).

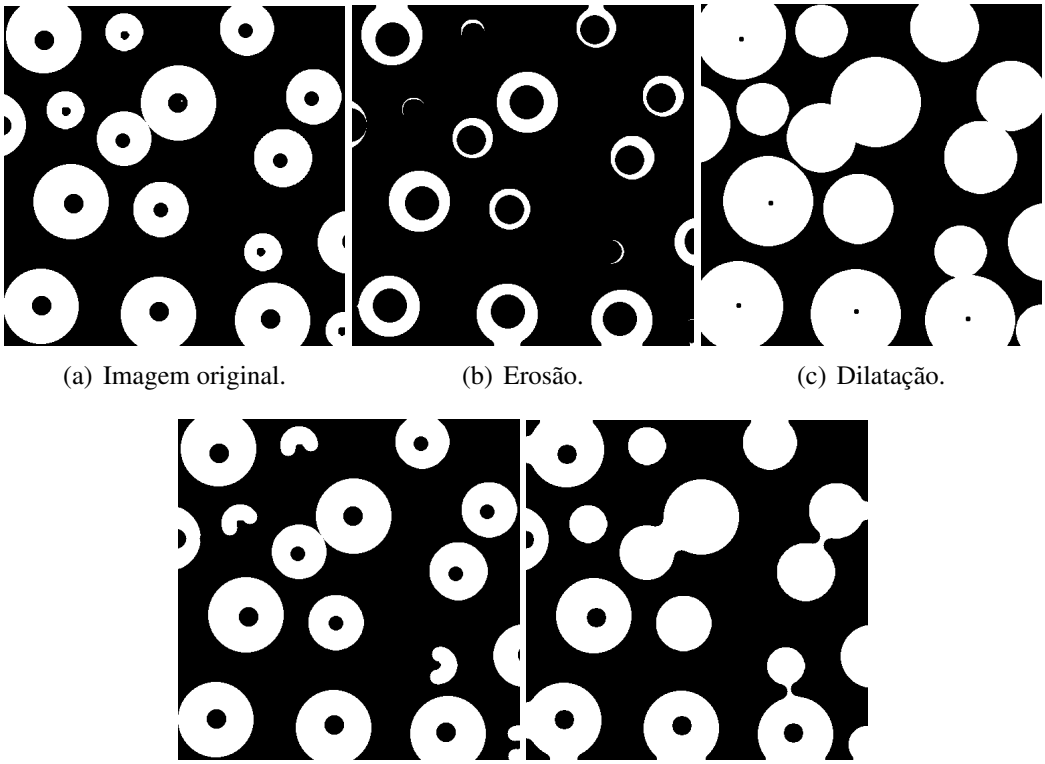
Dentro de PDI há o estudo de Morfologia Matemática (MM), a qual tem como base os conceitos de teoria dos conjuntos [Soille 2004]. O método *watershed* está inserido nessa área. O primeiro conceito a ser relacionado é o de Elemento Estruturante (EE), o qual indica o comportamento do processo a ser seguido. A forma assumida pelo EE (disco, quadrado, linha, losango, etc.) influencia diretamente no resultado do método.

Outros dois conceitos importantes são o de erosão e dilatação. A partir da forma do EE, pode erodir ou dilatar a borda de uma região na imagem processada. A combinação desses dois últimos tópicos faz surgir outros operadores morfológicos, que são a abertura (erosão seguida de dilatação com o mesmo EE) e o fechamento (erosão seguida de dilatação, também com o mesmo EE). Os efeitos destes quatro operadores com um elemento estruturante de disco (com raio igual a 11 pixels) são apresentados na Figura 5.13.

Os assuntos envolvidos nessa seção destacam importantes métodos presentes no domínio de processamento de imagens. Na próxima seção, um pequeno aplicativo será desenvolvido com base nos tópicos discutidos até aqui. Assim será possível entender como integrar os conceitos de PDI apresentados com as definições da linguagem de programação Java.

5.4. Arquitetura de um Aplicativo de PDI

Para entendimento do relacionamento das técnicas de processamento de imagens, sugere-se aqui a proposta de um aplicativo simples de PDI. Esta aplicação trata da extração de um objeto presente em uma imagem ruidosa e seu funcionamento é descrito na Figura 5.14. Esta imagem apresenta a interferência do ruído "sal e pimenta" [Gonzalez e Woods 2008] na distribuição dos seus *pixels*. Esse tipo de ruído é eliminado, com maior eficiência, pelo filtro da mediana [Marques Filho e Vieira Neto 1999]. O filtro será então aplicado e terá como resultado uma imagem mais adequada para a segmentação. Nesta etapa, a imagem será processada com o objetivo de identificar o objeto (estrela) presente na



(a) Imagem original. (b) Erosão. (c) Dilatação.

(d) Abertura. (e) Fechamento.

Figura 5.13. Operações de morfologia matemática em imagens binárias.

mesma (ver Figura 5.15). O objeto de interesse estará presente na imagem resultante após a segmentação. Em caso de falha na etapa anterior, permite-se um pós-processamento no qual se resume a adequar a região segmentada em uma formatação próxima da saída do aplicativo. Nesta seção, a classe a ser criada seguirá a API ImageJ para definir a aplicação. Com esta API, todas as operações sobre as imagens passam pelos métodos da classe *ByteProcessor* que funciona como base para a execução das tarefas mais simples de PDI sem necessidade de mudança de parametrização.

A imagem a ser tratada nesta aplicação (Figura 5.15(a)) se mostra bastante desafiadora. A estrela (objeto de interesse) a ser destacada possui o tom de cinza mais claro em relação aos demais elementos da imagem (com exceção de alguns *pixels* do ruído). Com essa informação, deve-se eliminar o ruído a fim de manter a homogeneidade nos tons de cinza por todas as regiões e, por consequência, também remover a linha de cor preta que percorre toda a imagem (inclusive sobre o objeto). Efetua-se então a filtragem na imagem com o método *medianFilter* (ver Tabela 5.2) definido na classe *ByteProcessor*. Este método aplica o filtro da mediana em toda a imagem com uma janela 3x3 de tal maneira que mantenha as informações de borda das regiões mesmo que ainda permaneçam artefatos do ruído e da linha preta na imagem. Isso é adotado por ser possível a remoção dessa informação nas próximas etapas do aplicativo.

Ao executar o código sugerido na Tabela 5.2, obtém-se o resultado da filtragem apresentado na Figura 5.15(b). Pode-se perceber até aqui, que o ruído "sal e pimenta" foi

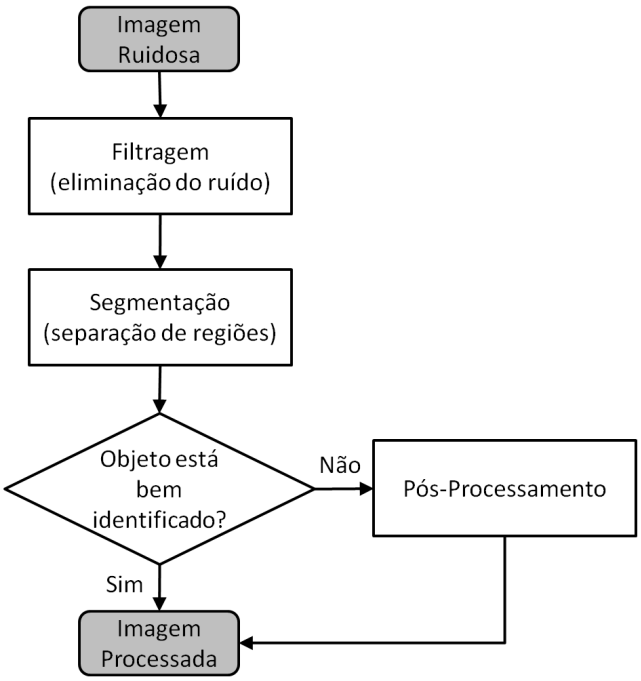
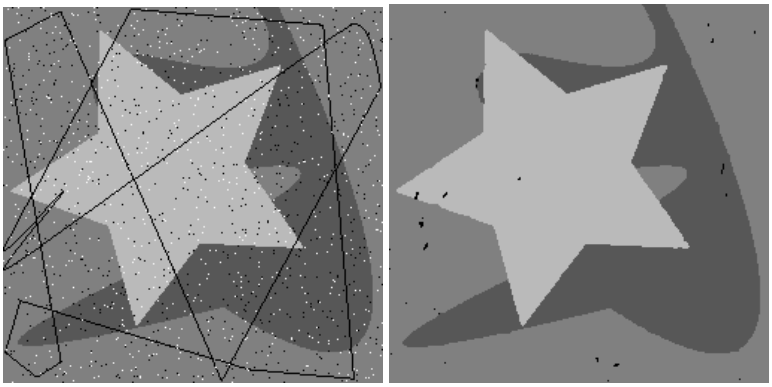


Figura 5.14. Fluxograma do aplicativo.

Tabela 5.2. Aplicação do filtro da mediana na definição do sistema.

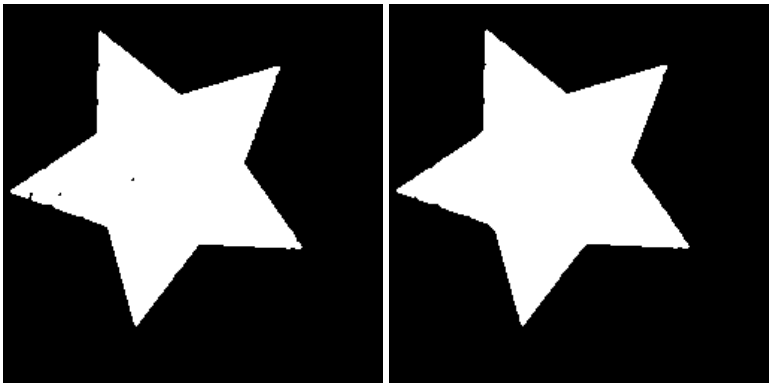
```
// objeto "imagem"" da classe java.awt.Image instanciado
// com a imagem original
ByteProcessor processador = new ByteProcessor(imagem);
processador.medianFilter();
// resultado do filtro é passado ao objeto "imagem"
imagem = processador.createImage();
```

eliminado, bem como a linha preta que passava pelo objeto de interesse. Mesmo assim, o efeito da filtragem em cima da linha preta fez permanecer alguns *pixels* de tonalidade de cor preta de tal forma que distorce a homogeneidade das regiões da imagem. Apesar desses resíduos, a imagem ganha um novo aspecto visual (ver Figura 5.15(b)) e o resultado da filtragem facilitará a separação da região da estrela. Isso porque o próximo passo a ser seguido é a segmentação da imagem. Nesse processo, será utilizada a técnica de limiarização para separar a região de interesse das demais. No exemplo trabalhado, a imagem possui uma diferença bem perceptível entre os tons de cinza de cada região. Desse modo, basta identificar o valor do tom de cinza predominante no objeto e usá-lo como limiar para a segmentação. Realiza-se a limiarização com o método *threshold* (ver Tabela 5.3) que também é definido na classe *ByteProcessor*. Este método tem como argumento um valor inteiro que indica o limiar da segmentação. A partir dessa informação, o processo de segmentação converte qualquer *pixel*, com valor maior ou igual a esse limiar, ao tom de cinza igual a 0 (zero) - indicando a cor preta - e em caso contrário, o *pixel* assume o valor de 255 (cor branca).



(a) Imagem de Entrada (adaptada de Gonzalez & Woods (2008)).

(b) Imagem Filtrada.



(c) Imagem Segmentada.

(d) Imagem de Saída.

Figura 5.15. Etapas do aplicativo proposto: processa imagem ruidosa para extração da região do objeto de interesse.

Tabela 5.3. Aplicação da segmentação na definição do sistema.

```
// objeto "imagem" da classe java.awt.Image instanciado
// com a imagem filtrada
ByteProcessor processador = new ByteProcessor(imagem);
// "valorLimiar" é inteiro e que indica o limiar da segmentação
processador.threshold(valorLimiar);
// resultado da limiarização é passado ao objeto "imagem"
imagem = processador.createImage();
```

O resultado da segmentação pode ser observado na Figura 5.15(c). Percebe-se na imagem segmentada a separação das regiões: objeto de interesse e o fundo da imagem. Mas nota-se também a presença de "buracos" na forma obtida, como pode ser notado na Figura 5.15(c). Isso se deve aos processos assumidos até aqui. Após a filtragem, permaneceram ainda *pixels* que se diferenciavam da região em que estavam presentes; e na segmentação, esses *pixels* destoantes foram considerados elementos do fundo mesmo estando na região do objeto. Como a segmentação não foi totalmente eficiente, deve-se

ajustar o resultado para a obtenção de uma região com todos os seus *pixels* assumindo o mesmo valor. Por conta destes "buracos", aplica-se um pós-processamento para obter a homogeneidade dos *pixels* na região do objeto.

A etapa de pós-processamento será realizada por uma operação de fechamento na imagem, cuja execução se dá por uma operação de dilatação seguida de uma erosão na mesma imagem e com o mesmo elemento estruturante [Soille 2004]. A API ImageJ não traz a operação de fechamento implementada em suas classes nativas, mas a partir da classe *ByteProcessor* adota-se as chamadas dos métodos *dilate* e *erode* correspondendo (ver Tabela 5.4), respectivamente, às operações de dilatação e erosão [Rasband 2009]. As duas operações fazem uso de uma janela de forma quadrada com as dimensões 3x3.

Tabela 5.4. Aplicação da operação de fechamento na definição do sistema.

```
// objeto "imagem" da classe java.awt.Image instanciado
// com a imagem filtrada
ByteProcessor processador = new ByteProcessor(imagem);
// etapa de dilatação - 1º passo do fechamento
processador.dilate(1,0);
// resultado da dilatação é passado ao objeto "imagem"
imagem = processador.createImage();
processador = new ByteProcessor(imagem);
// etapa de erosão - 2º passo do fechamento
processador.erode(1,0);
// resultado da erosão é passado ao objeto "imagem"
imagem = processador.createImage();
```

Ao fim de todo esse processamento, alcança-se o resultado final com o objeto de interesse separado do fundo de acordo com os *pixels* que compõem duas regiões correspondentes (ver Figura 5.15(d)). Com o pós-processamento foi possível alcançar uma segmentação completa e com as classes e métodos já apresentados pode-se preparar uma implementação melhorada para esse aplicativo utilizando interface gráfica. Permite-se criar neste trabalho uma classe de nome SegRdI (acrônimo de Segmentador de Região de Interesse) que suportará todas as técnicas descritas nesta seção, baseadas na API ImageJ para a linguagem Java.

Na Tabela 5.5, define-se a estrutura geral da classe no aplicativo a ser implementado. Na linha 05, define-se a classe SegRdI como subclasse de *JFrame*. Isso implica que será utilizada interface gráfica na aplicação [Deitel e Deitel 2005], e esta mesma será baseada na API Swing (que herda suas características da API AWT para desenho de componentes gráficos, assim como a API ImageJ). Outros componentes também serão integrados a essa interface a partir da API Swing: *JButton*, *JFileChooser*, *JPanel*, *JOptionPane* e *JScrollPane*. Mas para a geração desta interface gráfica, é implementado o método *iniciaComponentes* que tem a criação de alguns botões de ação. Para estes botões, são descritos métodos para aplicar uma funcionalidade para cada um na interface da aplicação (métodos *abrirActionPerformed*, *processarActionPerformed*, *limparActionPerformed* e *sairActionPerformed*). No construtor desta classe, há apenas a chamada ao método de inicialização

Tabela 5.5. Código da classe SegRdI (visão geral).

```
01  /**
02   * Lendo imagem ruidosa e extraindo o objeto
03   * @author ERCEMAPI 2009
04   */
05  public class SegRdI extends JFrame {
06    // atributos
07    JPanel painelPrinc, painelBotoes;
08    JButton btnAbrir, btnProc, btnLimpar, btnSair;
09    File fileName;
10    ImagePlus imagemIJ;
11    // inicialização dos componentes gráficos
12    public void iniciaComponentes() { ... }
13    // método da ação ao clicar o botão Abrir
14    private void abrirActionPerformed(ActionEvent evt) { ... }
15    // método da ação ao clicar o botão Processar
16    private void processarActionPerformed(ActionEvent evt) { ... }
17    // método da ação ao clicar o botão Limpar
18    private void limparActionPerformed(ActionEvent evt) { ... }
19    // método da ação ao clicar o botão Sair
20    private void sairActionPerformed(ActionEvent evt) { System.exit(0); }
21    // construtor padrão
22    public SegRdI() { this.iniciaComponentes(); }
23    // método principal
24    public static void main(String[] args) {
25        SegRdI aplicacao = new SegRdI();
26    } // fim do método principal
27  } // fim da classe
```

dos componentes gráficos da interface. Por consequência, no método principal há apenas a instanciação da classe. Isso porque buscou-se evitar o acúmulo de variáveis estáticas no método principal [Arnold et al. 2000] para auxiliar o desempenho do coletor de lixo da máquina virtual Java. O uso de variáveis estáticas deve ser evitado porque ocupam muito recurso de memória e são as últimas instâncias a serem eliminadas pelo *garbage collector* [Jones e Lins 1996].

O método *iniciaComponentes*, descrito na Tabela 5.6, traz uma implementação simples da interface gráfica. Nele são definidas as propriedades mais relevantes dos botões e a disposição dos demais componentes gráficos na tela. Para cada botão é associada um método correspondente que também está presente nesta seção.

O botão relacionado à ação de limpar a tela (objeto *btnLimpar*) tem sua implementação apresentada na Tabela 5.7. Nela descreve-se o método *limparActionPerformed* que remove a imagem destacada no aplicativo. Nota-se aqui o uso da classe *ImagePlus* que carrega a imagem a ser exibida na interface gráfica e instancia o objeto suportado pela

Tabela 5.6. Código da classe SegRdI – método *iniciaComponentes* que define a interface gráfica do aplicativo.

```
01  public void iniciaComponentes(){
02    this.setLayout(new BorderLayout());
03    painelPrinc = new JPanel(); painelPrinc.setLayout(new BorderLayout());
04    this.add(painelPrinc, BorderLayout.CENTER);
05    painelBotoes = new JPanel(); painelBotoes.setLayout(new FlowLayout());
06    this.add(painelBotoes, BorderLayout.SOUTH);
07    // adicionando botões
08    btnAbrir = new JButton(); painelBotoes.add(btnAbrir);
09    btnAbrir.setText(" Abrir ... ");
10    btnProc = new JButton(); painelBotoes.add(btnProc);
11    btnProc.setText("Processar");
12    btnLimpar = new JButton(); painelBotoes.add(btnLimpar);
13    btnLimpar.setText(" Limpar ");
14    btnSair = new JButton(); painelBotoes.add(btnSair);
15    btnSair.setText(" Sair ");
16    // configurar ações dos botões
17    btnAbrir.addActionListener(new ActionListener() {
18       public void actionPerformed(ActionEvent evt) {
19          abrirActionPerformed(evt); } });
20    btnProc.addActionListener(new ActionListener() {
21       public void actionPerformed(ActionEvent evt) {
22          processarActionPerformed(evt); } });
23    btnLimpar.addActionListener(new ActionListener() {
24       public void actionPerformed(ActionEvent evt) {
25          limparActionPerformed(evt); } });
26    btnSair.addActionListener(new ActionListener() {
27       public void actionPerformed(ActionEvent evt) {
28          sairActionPerformed(evt); } });
29    this.setVisible(true);
30    this.setSize(450,350);
31    this.setDefaultCloseOperation(
32          javax.swing.WindowConstants.EXIT_ON_CLOSE);
33  } // fim do método initComponents
```

classe *ImageCanvas*. Esta última classe habilita a integração de uma imagem trabalhada pelas classes da API ImageJ aos componentes oferecidos pela API Swing. Portanto, para "limpar" a tela do sistema basta instanciar um novo objeto de *ImagePlus* e associá-lo à interface. Já que o objeto *ImagePlus* é instanciado com seu construtor padrão, então nenhuma imagem é apresentada na tela.

O método *abrirActionPerformed* tem sua implementação detalhada na Tabela 5.8 e detalha a implementação do botão com ação de abrir uma nova imagem na interface. Permite-se com esse método o surgimento de uma caixa de diálogo (linhas 03 a 05) para

Tabela 5.7. Código da classe SegRdI – método *limparActionPerformed* que remove a imagem destacada no aplicativo.

```
01  private void limparActionPerformed(ActionEvent evt) {
02      ImagePlus imp = new ImagePlus();
03      ImageCanvas ic = new ImageCanvas(imp);
04      painelPrinc.removeAll();
05      painelPrinc.add(ic,BorderLayout.CENTER);
06  } // fim do método limparActionPerformed
```

Tabela 5.8. Código da classe SegRdI – método *abrirActionPerformed* definido para busca e visualização da imagem a ser trabalhada.

```
01  private void abrirActionPerformed(ActionEvent evt) {
02      // exibe caixa de diálogo para abrir arquivo de imagem
03      JFileChooser dialogo = new JFileChooser();
04      dialogo.setFileSelectionMode(JFileChooser.FILES_ONLY);
05      int result = dialogo.showOpenDialog(this);
06      if (result == JFileChooser.CANCEL_OPTION) return;
07      // recupera arquivo selecionado
08      fileName = dialogo.getSelectedFile();
09      // exibe erro se inválido
10      if (fileName == null || fileName.getName().equals("")) {
11          JOptionPane.showMessageDialog(this, "Nome de Arquivo Inválido",
12              "Nome de Arquivo Inválido", JOptionPane.ERROR_MESSAGE);
13          return;
14      imagemIJ = new ImagePlus(fileName.toString());
15      JScrollPane sp = new JScrollPane(
16          JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,
17          JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);
18      ImageCanvas ic = new ImageCanvas(imagemIJ); sp.add(ic);
19      sp.setSize(imagemIJ.getWidth(), imagemIJ.getHeight());
20      painelPrinc.add(sp,BorderLayout.CENTER);
21  } // fim do método abrirActionPerformed
```

seleção do arquivo da imagem a ser trabalhada. Se o arquivo for válido, a imagem escolhida é apresentada na tela do aplicativo.

Por fim, na Tabela 5.9 apresenta-se a descrição do método *processarActionPerformed* com a ação de processar a segmentação da imagem. Como já descrito nesta seção, nas linhas 02 a 07 define-se a chamada ao filtro da mediana na imagem. Na linha 08 há o carregamento da imagem filtrada para um objeto de *ImagePlus*, o que permite a visualização (quando necessário) do resultado da filtragem. Em seguida, uma busca pelo *pixel* de maior intensidade na imagem é feita nas linhas 10 a 15. Nesse trecho de código, uma variável inteira armazena o valor assumido por cada *pixel* consultado na imagem. Com o

Tabela 5.9. Código da classe SegRdI – método *processarActionPerformed* que realiza todo o processamento em torno das informações da imagem trabalhada.

```
01  private void processarActionPerformed(ActionEvent evt) {
02      Image image = imagemIJ.getImage();
03      ByteProcessor byteProc = new ByteProcessor(image);
04      byteProc.medianFilter();
05      image = byteProc.createImage();
06      ImagePlus imFilt = new ImagePlus("filtragem",image);
07      // descobrindo maior valor de nível de cinza
08      int max = -1;
09      for(int lin = 0; lin < imFilt.getHeight(); lin++)
10          for(int col = 0; col < imFilt.getWidth(); col++){
11              int[] pixels = imFilt.getPixel(col, lin);
12              if(pixels[0]>max) max = pixels[0]; }
13      image = imFilt.getImage(); byteProc = new ByteProcessor(image);
14      // aplicando a segmentação através de limiarização
15      byteProc.threshold(max-1);
16      image = byteProc.createImage();
17      ImagePlus imSeg = new ImagePlus("segmentacao",image);
18      image = imSeg.getImage(); byteProc = new ByteProcessor(image);
19      // inicialmente aplica-se a dilatação
20      byteProc.dilate(1,0);
21      image = byteProc.createImage();
22      ImagePlus imDil = new ImagePlus("dilatacao",image);
23      image = imDil.getImage(); byteProc = new ByteProcessor(image);
24      // posteriormente aplica-se a erosão
25      byteProc.erode(1,0);
26      image = byteProc.createImage();
27      ImagePlus imErosao = new ImagePlus("erosao",image);
28      JScrollPane sp = new JScrollPane(
29          JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,
30          JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);
31      ImageCanvas ic = new ImageCanvas(imErosao); sp.add(ic);
32      sp.setSize(imErosao.getWidth(), imErosao.getHeight());
33      painelPrinc.removeAll();
35      painelPrinc.add(sp,BorderLayout.CENTER);
36  } // fim do método processarActionPerformed
```

método *getPixel* (linha 13) da classe *ImagePlus* é possível recuperar um array com quatro elementos distintos: o primeiro é o valor de nível de cinza do *pixel* e os demais são o valores nas camadas RGB. Neste caso, apenas a informação do primeiro elemento é relevante.

Como mencionado anteriormente, os trechos de código expostos nas linhas 16 a 20 são aplicados para a execução da limiarização da imagem. Na linha 18 da Tabela 5.9,

percebe-se que o limiar assumido possui o valor do *pixel* de maior nível de cinza decrementado por uma unidade. Isso se deve ao fato que o método converte para 0 qualquer *pixel* com intensidade menor ou igual ao argumento passado no método *threshold*. Como na região de interesse todos os *pixels* possuem o valor máximo de tom de cinza na imagem, então decrementa-se esse limiar para que o objeto possa ser destacado. Após este passo, realiza-se o pós-processamento nas linhas 22 a 33 com a operação de fechamento. Ao final dessas operações, é repassada a imagem com o resultado final para componentes gráficos apresentarem a imagem pós-processada na tela do aplicativo.

Nota-se no resultado final que as bordas do objeto de interesse na imagem de saída não se assemelham com as bordas deste mesmo objeto na imagem de entrada. Pode-se observar, mesmo com a presença do ruído, que as bordas da imagem de entrada trazem um aspecto visual de linhas mais retilíneas que na imagem final. É possível enumerar alguns motivos para essa ocorrência. Primeiramente, a linha preta que passa por toda a imagem também atravessa a região do objeto e isso faz perder a informação de continuidade da sua borda. Em segundo lugar, o comportamento do ruído "sal e pimenta" - que converte um *pixel* aleatório em um dos valores extremos de nível de cinza - pode afetar o resultado da filtragem quando um *pixel* de borda possui seu valor modificado. E por último, na operação de fechamento a borda terá uma degradação em sua topologia se seu comportamento local (nos valores dos *pixels* tratados pelo elemento estruturante) foi afetado pelos motivos anteriores.

Apesar dessas dificuldades, o resultado final ainda se mantém bastante eficiente com relação à identificação da região de interesse. Pode-se afirmar que outras imagens, que sigam o mesmo padrão de distribuição de tons de cinza e deste mesmo ruído, obterão resultados semelhantes na segmentação dos objetos.

Com este pequeno aplicativo foi possível relacionar diferentes técnicas de PDI para alcançar um único resultado para uma determinada imagem. Buscou-se, com esse exemplo de aplicação, apresentar algumas funcionalidades resumidas da API ImageJ de forma integrada com APIs voltadas à geração de interface gráfica na linguagem de programação Java. Verificou-se ainda como o resultado de uma etapa deste aplicativo interferiu no resultado alcançado por uma etapa conseguinte. Esta mesma aplicação poderia ser desenvolvida usando a API JAI, ou alguma outra API direcionada a imagens, mas mesmo assim não teria uma implementação mais breve do que essa apresentada. Pode-se comprovar isso estudando a documentação destas APIs [Rodrigues 2001, Rasband 2009, Microsystems] e será observado que a API ImageJ possui descrição mais curtas em suas chamadas de métodos e implementações mais simples para as técnicas básicas de processamento de imagens.

5.5. Conclusões e Considerações Finais

Neste capítulo foram apresentadas definições de processamento de imagens e a implementação de um aplicativo direcionado para a descrição de regiões em uma imagem de nível de cinza. Para obter maior embasamento neste desenvolvimento, apresentou-se aqui aplicações de PDI em diferentes áreas de pesquisa, técnicas de realce, segmentação e morfologia matemática e as APIs Java dedicadas a processamento de imagens. Em relação ao realce observou-se o funcionamento do filtro da mediana e negativo de uma imagem, em

tempo que com segmentação, as ideias de limiarização e *watershed* foram apresentadas e, como consequência, a visão geral de morfologia matemática com exposição dos conceitos de erosão, dilatação, abertura e fechamento. Todas essas noções passadas serviram para a definição do aplicativo Java proposto na seção anterior.

Com relação às APIs, pode-se notar que a JAI trabalha melhor com entrada e saída das imagens na aplicação e a ImageJ traz definições mais simplificadas e implementação rápida das técnicas mais básicas de PDI. As duas APIs são flexíveis e bem extensíveis, o que facilita a continuidade de suas distribuições e a popularidade de cada uma na comunidade de desenvolvimento. Elas ainda fazem uso de classes em comum, o que possibilita a criação de uma aplicativo que combine estas APIs. Mas deve-se ficar atento com a redundância de implementações com mesmos resultados em classes diferentes. Outras implementações com o uso dos recursos citados nesse capítulo podem ser encontrado em `http://engcomp.sobral.ufc.br/professores/ialis/ercemapi2009`.

Referências

[Arnold et al. 2000] Arnold, K., J., G., e Holmes, D. (2000). *The* Java Programming Language. Prentice-Hall, 3rd edition.

[Azevedo et al. 2007] Azevedo, E., Conci, A., e Leta, F. (2007). *Computação Gráfica: Processamento de Imagens Digitais*, volume 2. Editora Campus.

[Bento et al. 2009a] Bento, M., Medeiros, F., Paula Júnior, I., e Ramalho, G. (2009a). Image processing techniques applied for corrosion damage analysis. In *Proceedings of the XXII Brazilian Symposium on Computer Graphics and Image Processing*, Rio de Janeiro. RJ.

[Bento et al. 2009b] Bento, M., Medeiros, F., Ramalho, G., e Medeiros, L. (2009b). Image processing techniques to monitor atmospheric corrosion. In *Anais da X Conferência sobre Tecnologia de Equipamentos (COTEQ)*, Salvador, BA.

[Bernier e Landry 2003] Bernier, T. e Landry, J. (2003). A new method for representing and matching shapes of natural objects. *Pattern Recognition*, 36(8):1711–1723.

[Burger e Burge 2009] Burger, W. e Burge, M. J. (2009). *Principles of Digital Image Processing: Fundamental Techniques*. Springer.

[Calíope et al. 2004] Calíope, P., Medeiros, F., Marques, R., e Costa, R. (2004). A comparison of filters for ultrasound images. *Lecture Notes in Computer Science*, 3124:1034–1040.

[Castleman 1996] Castleman, K. R. (1996). *Digital Image Processing*. Prentice Hall, Upper Saddle River.

[Costa e César Júnior 2009] Costa, L. e César Júnior, R. (2009). *Shape* Analysis and Classification: Theory and Practice. CRC Press, 2nd edition.

[Costa et al. 2004] Costa, L., Dos Reis, S., Arantes, R., Alves, A., e Mutinari, G. (2004). Biological shape analysis by digital curvature. *Pattern Recognition*, 37(3):515–524.

[César Júnior e Costa 1996] César Júnior, R. e Costa, L. (1996). Towards effective planar shape representation with multiscale digital curvature analysis based on signal processing techniques. *Pattern Recognition*, 28(9):1559–1569.

[Deitel e Deitel 2005] Deitel, H. e Deitel, P. (2005). *Java:* Como Programar. Prentice-Hall, 6th edition.

[Gomes e Velho 1994] Gomes, J. e Velho, L. (1994). *Computação* Gráfica: Imagem. IMPA.

[Gonzalez e Woods 2008] Gonzalez, R. e Woods, R. (2008). *Digital Image Processing*. Prentice Hall, Nova York, EUA, 3rd edition.

[Jones e Lins 1996] Jones, R. e Lins, R. (1996). *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons.

[Lee et al. 1995] Lee, J.-S., Sun, Y.-N., e Chen, C.-H. (1995). Multiscale corner detection by using wavelet transform. *IEEE Transactions on Image Processing*, 4(1):100–104.

[Liang e Zhang 2006] Liang, Y. e Zhang, H. (2006). *Computer Graphics Using Java 2d And 3d*. Prentice-Hall.

[Lopez et al. 2006] Lopez, D., Ramalho, G., Medeiros, F., Costa, R., e Araújo, R. (2006). Combining features to improve oil spill classification in sar images. *Lecture Notes in Computer Science*, 4109:928–936.

[Marques et al. 2004] Marques, R., Carvalho, E., Costa, R., e Medeiros, F. (2004). Filtering effects on sar images segmentation. *Lecture Notes in Computer Science*, 3124:1041–1046.

[Marques et al. 2008] Marques, R., Medeiros, F., e Ushizima, D. (2008). Target detection in sar images based on a level set approach. *IEEE Transactions on Systems, Man and Cybernetics - Part C, Applications and Reviews*, 39(2):214–222.

[Marques Filho e Vieira Neto 1999] Marques Filho, O. e Vieira Neto, H. (1999). *Processamento Digital de Imagens*. Brasport.

[Martins et al. 2008] Martins, C., Veras, R., Ramalho, G., Medeiros, F., e Ushizima, D. (2008). Automatic microaneurysm detection and characterization through digital color fundus images. In *Anais do Simpósio Brasileiro de Redes Neurais (SBRN)*, Salvador, BA.

[Meyer e Beucher 1990] Meyer, F. e Beucher, S. (1990). Morphological segmentation. *Journal of Visual Communication and Image Representation*, 1(1):21–46.

[Miano 1999] Miano, J. (1999). *Compressed Image File Formats*. ACM Press - Addison-Wesley, Reading, MA.

[Microsystems] Microsystems, S. Java Advanced Imaging (JAI) API. http://java.sun.com/javase/technologies/desktop/media/jai/.

[Mokhtarian e Mackworth 1986] Mokhtarian, F. e Mackworth, A. (1986). Scale-based description and recognition of planar curves and two-dimensional shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(1):34–43.

[Murray e VanRyper 1996] Murray, L. e VanRyper, W. (1996). *Encyclopedia of Graphics File Formats*. O'Reilly, 2nd edition. Sebastopol, CA.

[Passarinho et al. 2006] Passarinho, C., Cintra, L., Medeiros, F., Oliveira, I., e Paula Júnior, I. (2006). Análise de similaridade e correspondência de formas aplicada à reeducação postural global. In *Anais do XX Congresso Brasileiro de Engenharia Biomédica*, pages 117–120, São Pedro, SP.

[Paula Júnior et al. 2006] Paula Júnior, I., Medeiros, F., Mendonça, G., Passarinho, C., e Oliveira, I. (2006). Correlating multiple redundant scales for corner detection. In *Proceedings of the 2006 IEEE International Telecommunication Symposium (ITS)*, pages 650–655.

[Ramalho e Medeiros 2003] Ramalho, G. e Medeiros, F. (2003). Um sistema de inspeção visual automático aplicado à classificação e seleção de laranjas. In *Anais do IV Congresso Brasileiro da Sociedade Brasileira de Informática Aplicada à Agropecuária e Agroindústria (SBIAGRO)*, volume 2, pages 197–200, Porto Seguro, BA.

[Rasband 2009] Rasband, W. S. (1997-2009). ImageJ: Image Processing and Analysis in Java. http://rsbweb.nih.gov/ij/. U. S. National Institute of Health.

[Rodrigues 2001] Rodrigues, L. H. (2001). *Building* Imaging Applications with Java Technology: Using AWT Imaging, Java 2D, and Java Advanced Imaging (JAI). Addison-Wesley Professional.

[Santos] Santos, R. Java Image Processing Cookbook (online). Disponível em www.lac.inpe.br/˜ rafael.santos/JIPCookbook/.

[Santos 2003] Santos, R. (2003). *Introdução à Programação Orientada a Objetos Usando* Java. Campus.

[Santos 2004] Santos, R. (2004). Java Advanced Imaging API: A Tutorial. *Revista de Informática Teórica e Aplicada*, 11(1):93–123.

[Silva et al. 2006] Silva, L., Bellon, O., Lemes, R., Meira, J., e Cat, M. (2006). An image processing tool to support gestational age determination. In *Proceedings of the 19th IEEE International Symposium on Computer-Based Medical System*, pages 867–874.

[Soille 2004] Soille, P. (2004). *Morphological Image Analysis: Principles and Applications*. Springer-Verlag, Nova York.

[Velho et al. 2009] Velho, L., Frery, A., e Gomes, J. (2009). *Image Processing for Computer Graphics and Vision*. Springer, 2nd edition.

[Vincent e Soille 1991] Vincent, L. e Soille, P. (1991). Watersheds in digital spaces: An efficient algorithm based on immersion simulations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(6):583–598.