Tutorial Java Advanced Imaging – JAI

Resumo: Este tutorial tem por objetivo mostrar como a linguagem Java e a biblioteca JAI (Java Advanced Imaging) podem ser usadas para criar aplicações de processamento e visualização de imagens. A biblioteca ou API (Application Program Interface), é licenciada sob o Java Research License (JRL) para uso não comercial e sob o e Java Distribution License (JDL) para aplicações comerciais sem necessidades de pagamento de taxas por sua utilização. Uma das principais vantagens do Java/JAI é a portabilidade multi-plataformas, a quantidade de operadores presentes na biblioteca JAI e a possibilidades de extensão da API para criação de novos métodos de processamento de imagens. O objetivo desse tutorial é apresentar os conceitos básicos da API JAI, através de uma série de exemplos de códigos que implementam simples operações de processamento e análise de imagens. Ao final o leitor terá capacidade de implementar seus próprios algoritmos usando a biblioteca.

Sumário

1. Introdução

Existe, hoje, uma grande quantidade de softwares de processamento de imagens com os mais variados métodos e utilizados pelos mais diversos tipos de usuários. Porém, frequentemente, existe a necessidade de se implementar um algoritmo específico que não estão presentes nesses softwares. Por exemplo, um pesquisador precisa implementar seu próprio método de processamento que foi desenvolvido por meio de uma pesquisa. Alguns dos softwares já disponíveis no mercado permitem que o desenvolvedor adicione plugins ou rotinas ao sistema, as vezes, utilizando a API distribuída juntamente com o software. Contudo, podem existir custos adicionais ou restrições de licença para isso.

Uma alternativa livre de custos, multi-plataformas e flexível para a implementação desses métodos de processamento de imagens seria a utilização da linguagem Java e da API JAI. A grande quantidade de operadores disponíveis na biblioteca JAI e a possibilidade de extensão da API fazem do Java/JAI uma excelente alternativa para a implementação desse tipo de sistemas.

Esse tutorial apresentará os conceitos da JAI e exemplos de código para abertura de imagens, salvamento, aplicação de operadores, visualização e manipulação de imagens.

2. Instalação

Para instalar a biblioteca JAI precisamos baixar 2 arquivos, um para o JDK e outro para o JRE. Esses instaladores são encontrados em: https://jai.dev.java.net/binary-builds.html, respectivamente, jai-1_1_3-lib-windows-i586-jdk.exe e jai-1_1_3-lib-windows-i586-jre.exe . O JDK instala a biblioteca JAI para uso juntamente com o Java JDK (para desenvolvimento), o JRE instala a biblioteca para rodar as aplicações, tanto aplicações desktop como via navegador internet.

3. Representação de uma imagem

Algoritmos de processamento de imagens usualmente requerem a manipulação dos pixels da imagem (*image data*). Nessa seção o modelo utilizado pela JAI para o armazenamento e manipulação dos pixels será apresentado, bem como as classes correspondentes.

As imagens na JAI podem ser multidimensionais (com vários valores diferentes associados a um pixel) e ter pixels representados por inteiros ou ponto flutuante (embora existam restrições para o armazenamento das imagens em disco). Os pixels podem ser representados de diferentes formas, como por exemplo um vetor. Onde diferentes modelos de cores podem ser utilizados para representar a imagem.

As classes básicas para representação de uma imagem e seus pixels na JAI são:

`PlanarImage`: classe básica para representação de uma imagem na JAI. Permite a representação de imagens no Java com maior flexibilidade do que a classe `BufferedImage` (classe básica do Java para representar imagens). Ambas `PlanarImage` e `BufferedImage` representam os valores dos pixels da seguinte forma: os pixels são armazenados numa instância da classe `Raster` a

qual contém uma instância da subclasse do tipo `DataBuffer`, que é "empacotada" de acordo com as regras definidas por uma instância da subclasse do tipo `SampleModel`. Uma instância de uma `PlanarImage` ainda tem uma `ColorModel` associada à ela, que contém uma instância de `ColorSpace`, que determina com um valor de um pixel pode ser traduzido em valores de cor. A **figura 1** mostra como essas classes são utilizadas para compor uma instância de uma `PlanarImage`.

Uma `PlanarImage` é "somente-leitura", seus pixels podem ser lidos de diversas maneiras, entretanto não existem métodos que permitam a modificação dos valores dos pixels. `PlanarImages` podem ter a origem da imagem em uma posição diferente da coordenada (0,0), ou ainda coordenadas com valores negativos.

`TiledImage`: É uma subclasse da `PlanarImage`, a qual pode ser utilizada para ler e escrever no *image data* (pixels).

`RenderedOp`: Outra subclasse de PlanarImage, a qual representa um nó em uma *rendered image chain*. Uma *rendered image chain* é um poderoso e interessante conceito da JAI que permite que o processamento de uma imagem seja especificado em uma série de passos (operadores e parâmetros) os quais podem ser aplicados a uma ou mais imagens.

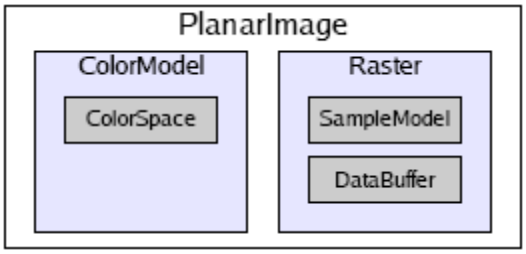


Figura 1. Estrutura de uma `PlanarImage`.

Outro conceito interessante usado na JAI são *tiled images*. *Tiles* (ladrilhos) podem ser considerados um subconjunto de imagens da imagem original distribuídos em uma grade que podem ser processados independentemente. Assim, imagens muito grandes podem ser processadas na JAI com performance razoável, desde que não seja necessário carregar toda *image data* de uma vez para a memória. Se uma imagem é *tiled*, todos os *tiles* têm a mesma altura e largura. A JAI permite diferentes origens para os pixels e para os *tiles* sobre uma imagem.

A **figura 2** mostra uma simples *tiled image,* onde a origem dos *tiles* coincidem com a origem da imagem. Cada elemento da grade tem o mesmo tamanho, podendo acontecer que o tamanho total da grade seja maior do que a imagem original (isso geralmente acontece), como demonstrado na figura 2. O conteúdo do *tile* que não está sobre a imagem é tratado como não definido. Mais informações *tiled images* podem ser encotrado no Java Advanced Imaging Programmer's Guide em http://jai.dev.java.net.

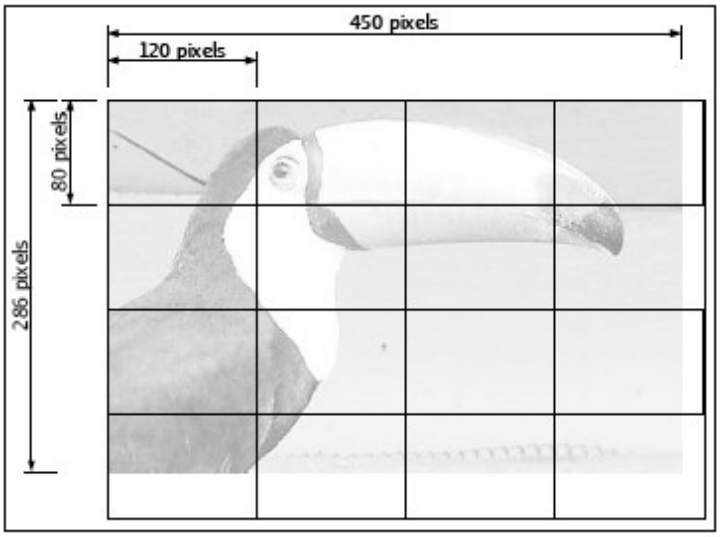


Figura 2. Uma *tiled image.*

Sabendo quais classes são utilizadas para representação da imagem, o processo de manipulação se torna mais simples.

Dois exemplos para criação de imagens serão apresentados, o primeiro será a criação de uma imagem em escala de cinza, e o segundo uma imagem RGB. Ambos exemplos usarão o tipo *byte* para representar os pixels e deverão seguir os passos:

1. Criar o *image data* num vetor em memória. Este vetor deve ser unidimensional, embora, por simplicidade um vetor multidimensional possa ser criado e "achatado" posteriormente.
2. Criar uma instância da subclasse `DataBuffer`, usando um dos seus construtores e o vetor que representa o *image data*.
3. Criar uma instância de `SampleModel` com o mesmo tipo do `DataBuffer` e com as dimensões desejadas. O método *factory* da classe `RasterFactory` pode ser utilizado para isso.
4. Criar uma instância de `ColorModel` compatível com o *sample model* que está sendo usado. O método estático `PlanarImage.createColorModel` pode ser usado, com o *sample model* como argumento.
5. Criar uma instância de `Raster` usando o *sample model* e o vetor *image data*. Pode-se utilizar o método `RasterFactory.createWritableRaster.`
6. Criar uma imagem para escrita (instância de `TiledImage`) usando o *sample model*, *color model* e as dimensões desejadas.
7. Associar a instância do `Raster` com a imagem através do método `setData` da classe `TiledImage`.
8. Salvar em disco ou mostrar na tela a instânia de `TiledImage`.

O código para resolução do exercício é mostrado a seguir:

Código 1. Criando uma imagem em escala de cinza.

```
public class CreateGrayImage{

      public static void main(String[] args){
        int width = 500; int height = 500; // dimensões da imagem
        byte[] imageData = new byte[width*height]; // vetor de pixels
        int count = 0; // contador auxiliar
        for(int w=0;w<width;w++) // preenche o vetor com um padrao dégradé
            for(int h=0;h<height;h++)
                    imageData[count++] = (byte)(w/2);
        // cria um DataBuffer a partir do vetor de pixels
        DataBufferByte dbuffer = new DataBufferByte(imageData,width*height);
        // cria o SampleModel com o tipo byte
        SampleModel sampleModel = RasterFactory.createBandedSampleModel(
            DataBuffer.TYPE_BYTE,width,height,1);
        // cria o ColorModel a partir do sampleModel
        ColorModel colorModel = PlanarImage.createColorModel(sampleModel);
        // cria o Raster
        Raster raster = RasterFactory.createWritableRaster(sampleModel,
            dbuffer,new Point(0,0));
        // cria uma TiledImage usando o SampleModel e o ColorModel
        TiledImage tiledImage = new TiledImage(0,0,width,height,0,0,
            sampleModel,colorModel);
        // seta o Raster que contém o dataBuffer na tiledImage
        tiledImage.setData(raster);
        // salva a imagem num arquivo
        JAI.create("filestore",tiledImage,"gray.tif","TIFF");
        // mostrar imagem na tela
        DisplayJAI display = new DisplayJAI(tiledImage);
        JFrame frame = new JFrame();
        frame.add(display);
        frame.setVisible(true);
        frame.pack();
    }
}
```

O código para criação da imagem RGB é similar:

Código 2. Criando uma imagem RGB.

```
public class CreateRGB{

      public static void main(String[] args){
        int width = 500; int height = 500; // dimensões da imagem
        byte[] imageData = new byte[width*height*3]; // vetor de pixels
        int count = 0; // contador auxiliar
        for(int w=0;w<width;w++) // preenche o vetor com um padrao qualquer
            for(int h=0;h<height;h++){
                    imageData[count+0] = (count % 2 == 0) ? (byte)255: (byte) 0;
                    imageData[count+1] = 0;
                    imageData[count+2] = (count % 2 == 0) ? (byte) 0: (byte)255;
                    count += 3;
            }
        // cria um DataBuffer a partir do vetor de pixels
        DataBufferByte dbuffer = new DataBufferByte(imageData,width*height*3);
        // cria o SampleModel com o tipo byte
        SampleModel sampleModel = RasterFactory.createPixelInterleavedSampleModel(
```

```
            DataBuffer.TYPE_BYTE,width,height,3);
        // cria o ColorModel a partir do sampleModel
        ColorModel colorModel = PlanarImage.createColorModel(sampleModel);
        // cria o Raster
        Raster raster = RasterFactory.createWritableRaster(sampleModel,
            dbuffer,new Point(0,0));
        // cria uma TiledImage usando o SampleModel e o ColorModel
        TiledImage tiledImage = new TiledImage(0,0,
            width,height,0,0,sampleModel,colorModel);
        // seta o Raster que contém o dataBuffer na tiledImage
        tiledImage.setData(raster);
        // salva a imagem num arquivo
        JAI.create("filestore",tiledImage,"color.tif","TIFF");
        // mostrar imagem na tela
        DisplayJAI display = new DisplayJAI(tiledImage);
        JFrame frame = new JFrame();
        frame.add(display);
        frame.setVisible(true);
        frame.pack();
    }
}
```

Para descobrir informações sobre uma imagem já existente, existem alguns métodos das classes PlanarImage, SampleModel and ColorModel. Alguns desses métodos são apresentados no exemplo abaixo.

Código 3. Lendo informações de uma imagem existente.

```
public class ImageInfo{

    public static void main(String[] args){
        // abrir uma imagem
        PlanarImage pi = JAI.create("fileload",
            "C:/Users/Animati/Desktop/testes/lena.png");
        // mostrar o tamanho da imagem
        File image = new File("C:/Users/Animati/Desktop/testes/lena.png");
        System.out.println("Tamanho do arquivo: "+image.length()+" bytes.");
        // mostrar as dimensões e coordenadas da imagem
        System.out.print("Dimensões: ");
        System.out.print(pi.getWidth()+"x"+pi.getHeight()+" pixels");
        System.out.println(" (from "+pi.getMinX()+","+pi.getMinY()+" to " +
            (pi.getMaxX()-1)+","+(pi.getMaxY()-1)+")");
        if((pi.getNumXTiles() != 1)||(pi.getNumYTiles() != 1)){//é uma tiled image?
            // numero dos tiles, dimensões e coordenadas
            System.out.print("Tiles: ");
            System.out.print(pi.getTileWidth()+"x"+pi.getTileHeight()+" pixels"+ "
                  ("+pi.getNumXTiles()+"x"+pi.getNumYTiles()+" tiles)");
            System.out.print(" (from "+pi.getMinTileX()+","+pi.getMinTileY()+" to
                 "+pi.getMaxTileX()+","+pi.getMaxTileY()+")");
            System.out.println(" offset: "+pi.getTileGridXOffset()+","+
            pi.getTileGridXOffset());
        }
        // mostrar informação sobre o SampleModel da imagem
        SampleModel sm = pi.getSampleModel();
        System.out.println("Número de bandas: "+sm.getNumBands());
        System.out.print("Tipo do pixel: ");
        switch(sm.getDataType()){
```

```
            case DataBuffer.TYPE_BYTE: System.out.println("byte"); break;
            case DataBuffer.TYPE_SHORT: System.out.println("short"); break;
            case DataBuffer.TYPE_USHORT: System.out.println("ushort"); break;
            case DataBuffer.TYPE_INT: System.out.println("int"); break;
            case DataBuffer.TYPE_FLOAT: System.out.println("float"); break;
            case DataBuffer.TYPE_DOUBLE: System.out.println("double"); break;
            case DataBuffer.TYPE_UNDEFINED:System.out.println("undefined"); break;
        }
        // mostrar informação sobre o ColorModel
        ColorModel cm = pi.getColorModel();
        if (cm != null){
            System.out.println("Número de componentes de cor: "+
            cm.getNumComponents());
            System.out.println("Bits por pixel: "+cm.getPixelSize());
            System.out.print("Transparência: ");
            switch(cm.getTransparency()){
                case Transparency.OPAQUE: System.out.println("opaque"); break;
                case Transparency.BITMASK: System.out.println("bitmask"); break;
                case Transparency.TRANSLUCENT:
                System.out.println("translucent"); break;
            }
        }else System.out.println("Não possui colorModel.");
    }
}
```

4. Operadores da JAI

A JAI contém um série de operadores que podem ser aplicados sobre imagens. Os operadores seguem o conceito de uma *rendered imaging chain*, onde os passos para o processamento da imagem são definidos, porém executados somente quando necessário (deferred execution).

As operações são especificadas da seguinte maneira: uma instância da classe `ParameterBlock` é criada, basicamente é um vetor com os parâmetros que serão utilizados na operação. Essa instância será passada ao método estático `create` da JAI para efetuar a operação. O método `create` pega como argumentos o nome da operação e a instância do `ParemeterBlock`, e retorna uma instância de `RenderedOp`, que pode ser manipulada como uma `PlanarImage`. Para adicionar os parâmetros ao `ParameterBlock` utilizamos os métodos `addSource` e `add`. O método `addSource` é utilizado para adição de imagens ao `ParameterBlock`, e o método `add` para demais parâmetros necessários ao processamento, inteiros, ponto flutuante, etc. Alguns operadores não necessitam de um `ParameterBlock`.

Um exemplo de um operador JAI é o "`filestore`", utilizado nos exemplos de código 1 e 2 para armazenar uma instância de `PlanarImage` (ou uma de suas subclasses) em disco. A chamada para o método `JAI.create` usa como argumentos o nome do operador, a instância da PlanarImage, o nome do arquivo que será salvo e uma string contendo o formato desejado ("TIFF", "JPEG", "PNG", etc.).

Outro exemplo de operador, que também não utiliza uma instância de `ParameterBlock` foi mostrado no código 3: uma chamada para `JAI.create("fileload", caminhoImagem);` irá carregar e retornar a imagem apontada pela string `caminhoImagem`. Alguns exemplos de operadores serão apresentados a seguir. A lista completa de operadores da JAI API podem ser encontrados na sua documentação em http://jai.dev.java.net.

O operador "`invert`" requer somente uma PlanarImage como argumento e pode ser executado como mostrado no código 4.

Código 4. Abrindo e invertendo uma imagem.

```
// carrega a imagem indicada pelo endereço caminhoImagemRead the image
PlanarImage input = JAI.create("fileload", caminhoImagem);
// inverte a imagem
PlanarImage output = JAI.create("invert", input);
```

O operador "`scale`" altera a escala da imagem. Também é possível transladar a imagem. Para usar esse operador, é necessário a criação de um `ParameterBlock` onde são adicionados a imagem original, dois valores em ponto-flutuante que correspondem à escala em X e Y, e outros dois valores em ponto-flutuante para translação da imagem em X e Y. Quando a imagem é reescalada pode-se aplicar uma interpolação de pixels com a finalidade de "suavizar" o resultado. Para isso adicionamos ao `ParameterBlock` uma instância de uma subclasse de `javax.media.jai.Interpolation`. O código 5 mostra um exemplo de utilização desse operador.

Código 5. Modificando a escala de uma imagem.

```
float escala = 2.0f;
ParameterBlock pb = new ParameterBlock();
pb.addSource(imagem);
pb.add(escala);
pb.add(escala);
pb.add(0.0f);
pb.add(0.0f);
pb.add(new InterpolationNearest());
PlanarImage imageReescalada = JAI.create("scale", pb);
```

O operador "`rotate`" rotaciona uma imagem usando um ângulo em radianos. De forma semelhante ao operador "`scale`", podemos utilizar o método de interpolação. Para usar o operador, devemos criar um ParameterBlock, adicionar a imagem e, nessa ordem, as duas coordenadas para o centro da rotação, o ângulo de rotação e a instância de uma subclasse de `Interpolation`. O código 6 mostra um exemplo de uma rotação de 45 graus de um imagem em torno do seu centro.

Código 6. Rotacionando uma imagem.

```
float angulo = (float)Math.toRadians(45);
float centroX = imagem.getWidth()/2f;
float centroY = imagem.getHeight()/2f;
ParameterBlock pb = new ParameterBlock();
pb.addSource(imagem);
pb.add(centroX);
pb.add(centroY);
pb.add(angulo);
pb.add(new InterpolationBilinear());
PlanarImage imagemRotacionada = JAI.create("rotate", pb);
```

Convoluções podem ser utilizadas facilmente através da JAI. O operador "`convolve`" pode ser aplicado em uma imagem com um *kernel* ou filtro, que é criado como uma instância da classe

`KernelJAI`. Essa instância é criada com um vetor que representa os valores do *kernel*, e então a instância de KernelJAI pode ser usada através de um `ParameterBlock` ou diretamente no método `JAI.create`. O código 7 mostra como criar um *kernel* de *smoothing* de tamanho 15 x 15 a aplicá-lo a uma imagem.

Código 7. Aplicando a operação de convolução com um filtro passa-baixa.

```
int kernelTamanho = 15;
float[] kernelMatriz = new float[kernelTamanho*kernelTamanho];
for(int k=0;k<kernelMatriz .length;k++)
      kernelMatriz[k] = 1.0f/(kernelTamanho*kernelTamanho);
KernelJAI kernel = new KernelJAI(kernelTamanho,kernelTamanho,kernelMatriz);
PlanarImage imagemProcessada = JAI.create("convolve", imagem, kernel);
```

O código 8 mostra outro exemplo de convolução, aplicando o método Sobel horizontal para detecção de bordas.

Código 8. Aplicando a operação de convolução para detecção de bordas.

```
float[] kernelMatriz = { -1, -2, -1,
                          0,  0,  0,
                          1,  2,  1 };
KernelJAI kernel = new KernelJAI(3,3,kernelMatriz);
PlanarImage imagemProcessada = JAI.create("convolve", imagem, kernel);
```

É possível utilizar operadores para manipular as bandas ou canais de uma imagem. Por exemplo, podemos fazer as seleções de algumas bandas de uma imagem multi-bandas para compor uma nova imagem. O operador "`bandselect`" usa como parâmetros de entrada uma imagem e um vetor de inteiros com os índices das bandas na ordem em que devem ser apresentados. No código 9 temos um exemplo de inversão nos canais deu uma imagem RGB, criando uma nova imagem na ordem reversa (BGR). Também podemos selecionar um determinado canal de uma imagem multi-bandas para criar uma imagem em escala de cinza a a partir do canal desejado. Para esse operador não é necessário utilizar um `ParameterBlock`.

Código 9. Selecionando as bandas de uma imagem.

```
//inverte os canais de uma imagem
PlanarImage imgResultado = JAI.create("bandselect", imagem, new int[] {2,1,0});

//seleciona  a banda 2 de uma imagem
PlanarImage imgResultado = JAI.create("bandselect", imagem, new int[] {2});
```

Outro manipulador de bandas é o operador "bandmerge", que utiliza uma série de bandas de imagens para criar uma imagem multi-bandas. Este método pode ser utilizado para criar uma imagem RGB a partir de três imagens que representem os canais, vermelho, verde e azul, por exemplo. Assumindo que tivéssemos três imagens em escala de cinza já carregadas em memória poderíamos combiná-las para criar uma imagem RGB como mostra o código 10.

Código 10. Combinando bandas para criar uma imagem multi-bandas.

```
//cria o parameterBlock e adiciona as imagens que correspondem a cada banda
ParameterBlock pb = new ParameterBlock();
pb.addSource(imagemCanalVermelho);
pb.addSource(imagemCanalVerde);
pb.addSource(imagemCanalAzul);
PlanarImage imagemResultado = JAI.create("bandmerge", pb);
```

Operadores como “`add`”, “`subtract`”, “`multiply`” e “`divide`”, são utilizados para aplicar operações aritméticas básicas em duas imagens, retornando uma terceira como resultado. Um simples exemplo é mostrado abaixo, para somar duas imagens.

Código 11. Operações aritméticas entre imagens.

```
ParameterBlock pb = new ParameterBlock();
pb.addSource(imagem1);
pb.addSource(imagem2);
imagemResultado = JAI.create("add", pb);
```

A lista completa de operadores da JAI com suas respectivas descrições o modos de utilização podem ser encontrados no Java Advanced Imaging Programmer's Guide em http://jai.dev.java.net.

5. Acessando os pixels de uma imagem

Frequentemente é necessário acessar os pixels de uma imagem, para aplicar um determinada operação. Por exemplo, algoritmos de classificação, necessitam acessar todos os pixels de uma imagem para sua avaliação e classificação.

Uma forma simples para acessar os pixels de uma imagem é a utilização de *iterators*. Estes permitem o acesso aos pixels de uma imagem respeitando uma ordem definida. Por exemplo, uma instância de `RectIter` lê a imagem coluna por coluna, de cima para baixo, atualizando as coordenadas dos pixels de forma automática conforme for sendo feita a leitura dos pixels (somente-leitura). Outro iterador, o `RamdomIter` permite o acesso direto um pixel usando sua coordenada dentro da imagem, x e y.

Ambos *iterators* são interfaces, e podem ser instanciados pelas classes `RectIterFactory` e `RandomIterFactory`, respectivamente. Esses métodos requerem dois argumentos: uma instância de `PlanarImage` (que é a imagem que será processada) e uma instância de `Rectangle`, que determina uma sub-região retangular da imagem que será considerada para o processamento. Se `null` for utilizado ao invés de uma instância de `Rectangle`, toda a imagem será considerada para o processamento.

O código 12 mostra como exibir no console todos valores dos pixels de uma imagem. O exemplo pega as dimensões da imagem, cria um vetor que se ajusta para ser preenchido com um pixel, e percorre toda a imagem imprimindo os valores dos pixels. Note que o método `nextPixel` precisa ser chamado para incrementar as coordenadas para o próximo pixel.

Código 12. Acessando os pixels de uma imagem com `RecIter`.

```
int largura = imagem.getWidth();
int altura = imagem.getHeight();
SampleModel sm = imagem.getSampleModel();
int nbandas = sm.getNumBands();
int[] pixel = new int[nbandas];
RectIter iterator = RectIterFactory.create(imagem, null);
for(int a=0;a<altura;a++)
    for(int l=0;l<largura;l++)
    {
        iterator.getPixel(pixel);//pixel onde os valores serão armazenados é
            passado como parâmetro
        System.out.print("em ("+l+","+a+"): ");
        for(int banda=0;banda<nbandas;banda++)
            System.out.print(pixel[banda]+" ");
        System.out.println();
        iterator.nextPixel();//pula para o próximo pixel
    }
```

O código 13 é similar ao 12, exceto pela instância de `RandomIter` que é usada. Então, quando o método `getPixel` é invocado devemos passar como argumento as coordenadas x e y do pixel desejado.

Código 13. Acessando os pixels de uma imagem com `RandomIter`.

```
int largura = imagem.getWidth();
int altura = imagem.getHeight();
SampleModel sm = imagem.getSampleModel();
int nbandas = sm.getNumBands();
int[] pixel = new int[nbandas];
RandomIter iterator = RandomIterFactory.create(imagem, null);
for(int a=0;a<altura;a++)
    for(int l=0;l<largura;l++)
    {
        iterator.getPixel(l, a, pixel);//coordenas e pixel onde os valores serão
            armazenados é passado como parâmetro
        System.out.print("em ("+l+","+a+"): ");
        for(int banda=0;banda<nbandas;banda++)
            System.out.print(pixel[banda]+" ");
        System.out.println();
    }
```

Embora o acesso dos pixels com iteradores seja simples, ele causa um overhead na performance da aplicação, pelo número de chamadas que são efetuadas pelo método internamente. Um forma mais eficiente de acessar os pixels de uma imagem é através de um `Raster` da imagem.

Como visto na seção 3, os pixels de uma imagem são armazenados em um Raster, que encapsula um `DataBuffer` e um `SampleModel`. O desenvolvedor não necessita saber como pixel está armazenado dentro do Raster, o seu método getPixel e suas variações pega os pixels como um vetor e o método `getSample` pega o valor de uma banda em determinado pixel. Extraindo o `Raster` de uma imagem e um sub-região deste, haverão menos chamas a métodos internos e conseqüentemente menor overhead, melhorando a performance da aplicação. Por outro lado mais memória pode ser necessária se o processamento da imagem for feito em pedaços e dependendo do tamanho das sub-regiões.

O código 14 mostra como acessar os pixels de uma imagem utilizando um `Raster`. O código é

similar ao 12 e 13, exceto por uma instância de `Raster` que é criada pelo método `getData` da classe `PlanarImage`, e então o método `getPixels` do *raster* é chamado para pegar todos os pixels da imagem e armazená-los em um vetor com as dimensões da imagem. O método `getPixels` tem como parâmetros as coordenadas do primeiro pixel superior à esquerda, a altura, a largura da imagem e o vetor onde os valores dos pixels serão armazenados. No exemplo a imagem inteira foi utilizada. O vetor onde as os valores são armazenados deve ser unidimensional e o usuário necessita calcular o deslocamento nesse vetor para acessar os pixels desejados.

Código 14. Acessando os pixels de uma imagem através de um `Raster`.

```
int largura = imagem.getWidth();
int altura = imagem.getHeight();
SampleModel sm = imagem.getSampleModel();
int nbandas = sm.getNumBands();
Raster raster = imagem.getData();
int[] pixels = new int[nbandas*largura*altura];
raster.getPixels(0,0,largura,altura,pixels);
int deslocamento; // deslocamento no vetor
for(int a=0;a<altura;a++)
    for(int l=0;l<largura;l++){
        deslocamento = a*largura*nbandas+l*nbandas; // calcula o deslocamento do
            //pixel atual no vetor
        System.out.println(deslocamento);
        System.out.print("em ("+l+","+a+"): ");
        for(int banda=0;banda<nbandas;banda++)
            System.out.print(pixels[deslocamento+banda]+" ");
        System.out.println();
    }
```

PlanarImages e Rasters são somente-leitura, mas podemos criar uma aplicação que processa os pixels de uma imagem e os armazena. A partir de uma instância de um `Raster` podemos criar um instância de um `WritableRaster` com a mesma estrutura (mas sem os valores dos pixels) através do método `Raster.createCompatibleWritableRaster`. Os valores dos pixes podem ser obtidos como mostrado no código 14. Depois de processar os pixels pelo vetor de valores, esse vetor pode ser armazenado novamente na instância de `WritableRaster` pelo método `setPixels`, cujos argumentos são os mesmo utilizados em `Raster.getPixels`.

Um `Raster` ou `WritableRaster` não pode ser inserido novamente dentro de uma `PlanarImage`, mas uma `TiledImage` pode ser criada a partir destes. No construtor da `TiledImage` utilizamos como parâmetros a `PlanarImage` original, e tamanho desejado do *tiles*. A `TiledImage` terá as mesmas dimensões e propriedades da imagem original, e o método `setData` com um `Raster` ou `WritableRaster` como parâmetro é usado para armazenar os valores dos pixels na nova imagem. Esta `TiledImage` pode ser processada novamente, visualizada ou armazenada em disco.

O código 15 mostra todo o processo, onde todos os pixels com valores igual a zero são modificados para 255. A entrada do método é um `PlanarImage` e sua saída uma `TiledImage` com os valores originais modificados.

Código 15. Acessando os pixels de uma imagem para leitura e escrita.

```
int largura = imagem.getWidth();
```

```
int altura = imagem.getHeight();
SampleModel sm = imagem.getSampleModel();
int nbandas = sm.getNumBands();
Raster rasterLeitura = imagem.getData();
WritableRaster rasterEscrita = rasterLeitura.createCompatibleWritableRaster();
int[] pixels = new int[nbandas*largura*altura];
rasterLeitura.getPixels(0,0,largura,altura,pixels);
int deslocamento;
for(int a=0;a<altura;a++)
    for(int l=0;l<largura;l++){
        deslocamento = a*largura*nbandas+l*nbandas;
        for(int banda=0;banda<nbandas;banda++)
            if (pixels[deslocamento+banda] == 100) pixels[deslocamento+banda] =
255;
    }
rasterEscrita.setPixels(0,0,largura,altura,pixels);
TiledImage ti = new TiledImage(imagem,1,1);
ti.setData(rasterEscrita);
JAI.create("filestore",ti,"imagem.tif","TIFF");
```

Também é possível usar *iterators* de escrita. Por exemplo, uma instância de `WrritableRandomIter` pode ser criada através do método `RandomIterFactory.createWritable` e passando como parâmetros uma `TiledImage` e uma instância de `Rectangle` para definir a área de processamento, ou null para toda a imagem. Um *iterator* de escrita pode ser utilizado de forma similar ao de somente-leitura. O *iterator* de escrita armazena os pixels diretamente na imagem resultado, por meio dos métodos `setPixel` ou `setSample`.

6. Visualização de imagens

Visualização é uma etapa importante em uma aplicação de processamento de imagens. Embora seja possível ler uma imagem, processá-la, armazenar os resultados em disco e utilizar outra aplicação para visualizar os resultados, existem alguns tipos de imagens, as quais não são visualizáveis através de aplicações genéricas – imagens em ponto-flutuante ou com mais de quatro bandas, por exemplo. Assim, pode ser mais interessante fazer o processamento e visualização em uma mesma aplicação Java.

A JAI API dispõe de um simples, mas eficiente, componente para exibir imagens, implementado pela classe `DisplayJAI`. Esse componente é uma herança de `JPanel` e pode ser usado em outros componentes gráficos do Java. Este componente também pode ser estendido para diferente propósitos.

Um exemplo é mostrado no código 16. Essa aplicação Java mostra uma imagem na tela. Um instância de `DisplayJAI` é criada, usando como argumento um instância de `PlanarImage` (a imagem no `DisplayJAI` pode ser modificado posteriormente através do método `set`). A instância do `DisplayJAI` é associada com um `JScrollPane`, assim imagens maiores que a tela podem ser visualizada através de rolagens.

Código 16. Visualizando imagens através do componente `DisplayJAI`.

```
//cria a planarImage
PlanarImage imagem = JAI.create("fileload", "C:/Documents and
Settings/Jean/Desktop/testes/lena.png");
//cria o displayJai
DisplayJAI display = new DisplayJAI(imagem);
//cria o frame do swing onde a imagem será exibida
JFrame frame = new JFrame("Imagem X");
//cria o scroll para efetuar as rolagens caso a imagem seja muito grande
JScrollPane scroll = new JScrollPane();
//adiciona o display dentro do scroll
scroll.setViewportView(display);
//adiciona o scroll dentro do frama
frame.add(scroll);
//define a operação padrão para o botão fechar do frame
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setSize(400,400); //define o tamanho do frame
frame.setVisible(true); //manda o frame ser exibido
```

6.1 Visualização utilizando uma imagem intermediária

Com componente `DisplayJAI` é possível visualizar imagens cujos pixels não são valores em bytes, porém o resultado é imprevisível, pois não existe uma conversão interna para adequar os pixels ao formato correto de exibição. Nesta seção criaremos uma extensão de um `DisplayJAI`. Este exemplos tem dois pontos interessantes : ele usa uma imagem intermediária para exibição, que será criada a partir da imagem original; e permite uma básica interação com o usuário, que poderá ver o valor original do pixel da imagem pelo cursor do mouse.

O componente será adaptado para exibir imagens de um modelo digital de terrenos (DEM – digital elevation model), que são, imagens com um canal de cor com pixels representados em ponto-flutuante, onde os pixels representam a elevação do terreno em relação ao nível do mar. Para criar a imagem intermediária que irá representar visualmente a DEM, precisamos criar e normalizar os novos valores dos pixels baseados nos valores em ponto-flutuante da imagem original. Os pixels da imagem intermediária devem estar na faixa [0,255], normalizados de acordo com o mínimo e o máximo (*min* e *max*) valores dos pixels da imagem DEM – em outras palavras, todos os pixels da imagem intermediária serão calculados pelo valor do pixel correspondente da uma imagem DEM, multiplicados por 255/(*max* – *min*) e adicionados com *min.* Os pixels da imagem intermediária serão reformatados para o tipo byte.

Para criar a imagem intermediária com esses regras, três operadores da JAI serão utilizados. O primeiro é o operador "`extrema`", que usa uma imagem como parâmetro. Depois de aplicar o operador, é possível, pelo método `getProperty` do objeto `RenderedOp`, utilizando "`maximun`" ou "`minimum`" como argumento, pegar um vetor double com os valores correspondentes aos máximos e mínimos valores dos pixels de cada canal da imagem. Nesse exemplo considera-se que a imagem DEM tem somente uma banda.

O segundo operador utilizado é o "`rescale`", que usa como parâmetros (por meio de uma `ParameterBlock`) uma imagem de entrada, um vetor de valores double para multiplicação dos pixels da imagem de entrada, e outro vetor de valores double para adição aos pixels da imagem de entrada. Se a dimensão do vetor for a mesma do número de bandas que a imagem, a multiplicação e adição serão efetuadas levando em consideração o primeiro elemento do vetor com a primeira banda da imagem, o segundo elemento com a segunda banda e assim por diante. De outra forma, somente o primeiro valor do vetor será utilizado para todas as bandas. Os pixel da imagem resultante são

calculados como: *saida* = *entrada* * *m* + *a,* onde *m* e *a* são os vetores de multiplicação e adição respectivamente.

O terceiro operador é o "`format`", que recebe como parâmetros a imagem de entrada e uma das constantes `TYPE_BYTE`, `TYPE_SHORT`, `TYPE_USHORT`, `TYPE_INT`, `TYPE_FLOAT` ou `TYPE_DOUBLE`, que são definidas pela classe `DataBuffer`. Os pixels da imagem resultante serão formatados para o tipo definido no `ParameterBlock` pela constante do `DataBuffer`.

O código 17 mostra como criar o novo componente `Display-DEM`. Esse componente cria uma imagem intermediária no seu construtor, usando a imagem original e então procede com os passos de normalização e formatação, e também cria uma `RandomIter` para obter os valores da imagem original. Esse exemplo também mostra como extrair os valores dos pixels sob a posição do mouse.

Código 17. Código para o componente `DisplayDEM`.

```
public class DisplayDEM extends DisplayJAI implements MouseMotionListener{
    protected StringBuffer pixelInfo; //informação do pixel (formatado como
            //stringBuffer)
    protected double[] dpixel; // informação dos pixels como vetor de doubles
    protected RandomIter readIterator; // um RandomIter para pegar os valores dos
            //pixels
    protected PlanarImage surrogateImage; // a imagem intermediária
    protected int width,height; // dimensões da imagem
    protected double minValue,maxValue; // intervalo dos valores da imagem

    /**
    * O construtor da classe, que cria a estrutura da imagem intermediária
    */
    public DisplayDEM(RenderedImage image)
    {
        readIterator = RandomIterFactory.create(image, null);
        // pega informações da imagem
        width = image.getWidth();
        height = image.getHeight();
        dpixel = new double[image.getSampleModel().getNumBands()];
        // precisamos saber o pixel de valor máximo da imagem. Utilizamos o
        //operador extrema
        ParameterBlock pbMaxMin = new ParameterBlock();
        pbMaxMin.addSource(image);
        RenderedOp extrema = JAI.create("extrema", pbMaxMin);//note que utilizamos
                                    //um RenderedImage ao invés de uma PlanarImage
        double[] allMins = (double[])extrema.getProperty("minimum");
        double[] allMaxs = (double[])extrema.getProperty("maximum");
        minValue = allMins[0]; // assume que imagem tenho somente uma banda.
        maxValue = allMaxs[0];
        // aplica o operador rescale na imagem
        double[] multiplyByThis = new double[1];
        multiplyByThis[0] = 255./(maxValue-minValue);
        double[] addThis = new double[1];
        addThis[0] = minValue;
        ParameterBlock pbRescale = new ParameterBlock();
        pbRescale.add(multiplyByThis);
        pbRescale.add(addThis);
        pbRescale.addSource(image);
        surrogateImage = (PlanarImage)JAI.create("rescale", pbRescale);
        // convertemos a imagem para o tipo byte para visualizacao
        ParameterBlock pbConvert = new ParameterBlock();
        pbConvert.addSource(surrogateImage);
```

```
        pbConvert.add(DataBuffer.TYPE_BYTE);
        surrogateImage = JAI.create("format", pbConvert);
        set(surrogateImage);
        // criamos uma instancia de StringBuffer para colocar as informacoes dos
        //pixels
        pixelInfo = new StringBuffer(50);
        addMouseMotionListener(this); // registrar os listener
     }

    // este método está aqui somente para implementar a interface
    //MouseMotionListener
    public void mouseDragged(MouseEvent e) { }

    // este método é chamado quando o mouse e movimentado sobre a imagem
    public void mouseMoved(MouseEvent me){
        pixelInfo.setLength(0); // limpa o StringBuffer
        int x = me.getX(); // pega as coordenadas do mouse
        int y = me.getY();
        if ((x >= width) || (y >= height)) // considera somente os pixels
                                           // dentro da imagem
        {
        pixelInfo.append("No data!");
        return;
        }

        pixelInfo.append("(DEM data) "+x+","+y+": ");
        readIterator.getPixel(x,y,dpixel); // le o pixel da imagem original
        pixelInfo.append(dpixel[0]); // adiciona o valor do pixel na string
    }

    // permite que outras classes acessem as informações do pixels
    public String getPixelInfo(){
        return pixelInfo.toString();
    }
}
```

O componente `DisplayDEM` pode ser utilizado com qualquer aplicação Java com interface com usuário. No código 18 temos o exemplo de uma aplicação simples que gera uma imagem em escala de cinza com os pixels com valores aleatórios em ponto flutuante. Essa aplicação usa o `DisplayDEM` para exibir a imagem, e também mostra os valores originais dos pixels quando movimentamos o mouse sobre a imagem.

Código 18. Aplicação que utiliza o componente `DisplayDEM`.

```
public class DisplayDEMApp extends JFrame implements MouseMotionListener{

    private DisplayDEM dd; // um instancia do componente DisplayDEM
    private JLabel label; // Label para mostrar a infomacao sobre a imagem

    public DisplayDEMApp(PlanarImage image){
        setTitle("Move the mouse over the image !");
        getContentPane().setLayout(new BorderLayout());
        dd = new DisplayDEM(image); // cria o componente
        getContentPane().add(new JScrollPane(dd),BorderLayout.CENTER);
        label = new JLabel("---"); // cria o Label
        getContentPane().add(label,BorderLayout.SOUTH);
        dd.addMouseMotionListener(this); // registra os eventos do mouse.
```

```
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(400,200);
        setVisible(true);
    }

    // este método está aqui somente para implementar a interface
      //MouseMotionListener
    public void mouseDragged(MouseEvent e) { }

    // este método será executado quando o mouse é movimentado sobre a aplicação
    public void mouseMoved(MouseEvent e){
        label.setText(dd.getPixelInfo()); // atualiza o label com a info do
                                          // DisplayDEM
    }

    //main para rodar a aplicação
    public static void main(String[] args){
        new DisplayDEMApp(createGrayImage());
    }

    //cria uma imagem de uma banda com os pixels com valores aleatórios em float
    static PlanarImage createGrayImage(){
        int width = 250; int height = 250; // dimensões da imagem
        float[] imageData = new float[width*height]; // vetor de pixels
        int count = 0; // contador auxiliar
        for(int w=0;w<width;w++) // preenche o vetor com um padrao dégradé
            for(int h=0;h<height;h++){
                    imageData[count] = (float)(Math.random()*100);
                    count++;
            }
        // cria um DataBuffer a partir do vetor de pixels
        DataBufferFloat dbuffer = new DataBufferFloat(imageData,width*height);
        // cria o SampleModel com o tipo byte
        SampleModel sampleModel = RasterFactory.createBandedSampleModel(
            DataBuffer.TYPE_FLOAT,width,height,1);
        // cria o ColorModel a partir do sampleModel
        ColorModel colorModel = PlanarImage.createColorModel(sampleModel);
        // cria o Raster
        Raster raster = RasterFactory.createWritableRaster(sampleModel,
            dbuffer,new Point(0,0));
        // cria uma TiledImage usando o SampleModel e o ColorModel
        TiledImage tiledImage = new TiledImage(0,0,width,height,0,0,
            sampleModel,colorModel);
        // seta o Raster que contém o dataBuffer na tiledImage
        tiledImage.setData(raster);
        // retorna a imagem criada
        return tiledImage;
    }
}
```

6.2 Visualização de imagens com anotações

Outra tarefa freqüente em aplicações de processamento de imagens é mostrar uma imagem com algum tipo de anotação sobre ela – marcadores sobre a imagem, texto, delimitadores de regiões de interesse, etc. Nessa seção descreveremos a criação de anotações sobre imagens usando uma extensão

da classe `DisplayJAI`.

Para apresentar um exemplo mais completo, uma classe abstrata que encapsula um anotação é criada. A classe `DrawableAnnotation` é mostrada no código 19, onde são declarados o método abstrato `paint` e um objeto `Color`, com os métodos `set` e `get` para esse objeto. As classes que estendem a `DrawableAnnotation` devem implementar o método `paint`, que irá desenhar a anotação desejada usando uma instância de `Graphics2D`.

Código 19. Classe abstrata que encapsula a anotação.

```
public abstract class DrawableAnnotation{
	private Color color;

	public abstract void paint(Graphics2D g2d);

	public void setColor(Color color){
		this.color = color;
	}

	public Color getColor(){
		return color;
	}
}
```

A implementação da classe `DrawableAnnotation` é mostrada no código 20. A implementação permite o desenho de uma anotação em forma de diamante, usando como parâmetros em seu construtor o ponto central para a anotação, a altura e largura da anotação em pixels, e a bitola da linha de desenho.

Código 20. Classe que encapsula a anotação em forma de diamante.

```
public class DiamondAnnotation extends DrawableAnnotation{
	private Point2D center; // ponto central da anotação
	private double width; // largura da anotação
	private double height; // altura da anotação
	private BasicStroke stroke; // bitola da linha de desenho

	// construtor da classe
	public DiamondAnnotation(Point2D c,double w,double h,float pw){
		center = c;
		width = w;
		height = h;
		stroke = new BasicStroke(pw);
	}

	// implementa do método paint, que faz o desenho na tela
	public void paint(Graphics2D g2d){
		int x = (int)center.getX();
		int y = (int)center.getY();
		int xmin = (int)(x-width/2);
		int xmax = (int)(x+width/2);
		int ymin = (int)(y-height/2);
		int ymax = (int)(y+height/2);
		g2d.setStroke(stroke);
		g2d.setColor(getColor());
```

```
            g2d.drawLine(x,ymin,xmin,y);
            g2d.drawLine(xmin,y,x,ymax);
            g2d.drawLine(x,ymax,xmax,y);
            g2d.drawLine(xmax,y,x,ymin);
      }
}
```

A classe main nessa seção é a classe que herda de `DisplayJAI` e pode mostrar a imagem e mostrar as anotações (instâncias das classes que herdam de `DrawableAnnotation`) sobre ela. As anotações são armazenadas em uma lista, e a classe possui métodos para adicionar anotações à lista.

Esta classe sobrescreve (override) o método paint da DisplayJAI, então após a imagem ser desenhada na tela (através da chamada `super.paint`) todas as instâncias das anotações na lista tem seus métodos `paint` executados, usando o mesmo contexto de desenho para desenhar a imagem.

O código 21 mostra a extensão da classe `DisplayJAI`, a `DisplayJAIWithAnnotations`.

Código 21. Estendendo a classe `DisplayJAI` para mostrar imagem com anotações.

```
public class DisplayJAIWithAnnotations extends DisplayJAI{
      protected ArrayList annotations; // lista das anotaçoes que serao desenhadas

      // construtor da classe
      public DisplayJAIWithAnnotations(RenderedImage image){
            super(image); // chama o construtor da DisplayJAI
            annotations = new ArrayList(); // lista de anotações
      }

      // este método desenha a imagem e suas anotações
      public void paint(Graphics g){
            super.paint(g);
            Graphics2D g2d = (Graphics2D)g;
            for (int a=0;a<annotations.size();a++) // para cada anotação...
            {
                  DrawableAnnotation element =
(DrawableAnnotation)annotations.get(a);
                  element.paint(g2d);
            }
      }

      // adiciona uma anotação na lista de anotações
      public void addAnnotation(DrawableAnnotation a)
      {
            annotations.add(a);
      }
}
```

Por fim, uma aplicação que utiliza o `DisplayJAIWithAnnotations` é mostrada no código 22. A aplicação cria três instâncias de `DiamondAnnotation` e as adiciona à instância do `DisplayJAIWithAnnotations`, que será desenhado dentro de um `JFrame`.

Código 22. Aplicação que usa o componente `DisplayJAIWithAnnotations`.

```
public class DisplayJAIWithAnnotationsApp{
```

```
	public static void main(String[] args){
		PlanarImage image = JAI.create("fileload","datasets/bloodimg02.jpg");
		DisplayJAIWithAnnotations display = new
			DisplayJAIWithAnnotations(image);
		// cria as anotações em forma de diamante
		DiamondAnnotation d1 = new DiamondAnnotation(new
			Point2D.Double(229,55),20,20,2);
		d1.setColor(Color.BLACK);
		DiamondAnnotation d2 = new DiamondAnnotation(new
			Point2D.Double(249,84),20,20,3);
		d2.setColor(Color.BLACK);
		DiamondAnnotation d3 = new DiamondAnnotation(new
			Point2D.Double(303,33),35,35,5);
		d3.setColor(Color.GRAY);
		// adiciona as anotações à instância de DisplayJAIWithAnnotations.
		display.addAnnotation(d1);
		display.addAnnotation(d2);
		display.addAnnotation(d3);
		// cria um novo frama e adiciona o DisplayJAIWithAnnotations
		JFrame frame = new JFrame();
		frame.setTitle("Anotações sobre a imagem");
		frame.getContentPane().add(new JScrollPane(display));
		frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		frame.setSize(500,200); // define o tamanho do frame
		frame.setVisible(true);
	}
}
```

A figura 3 mostra um screenshot da aplicação.

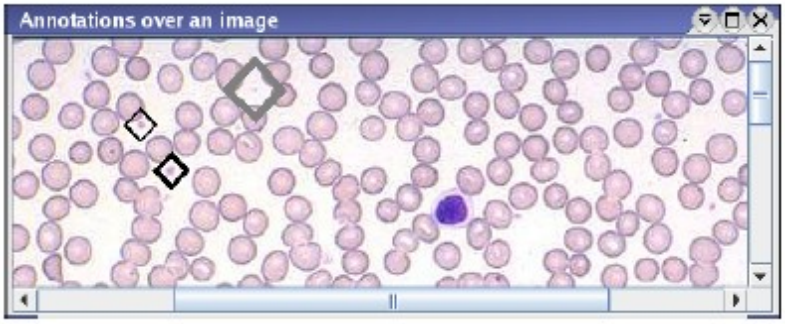


***Figura 3**. Imagem com anotações*

7. Estendendo a JAI

Embora a API JAI suporte um grande número de operação sobre imagens, ela foi projetada para encorajar programadores a escreverem extensões da biblioteca ao invés de manipularem as imagens diretamente. A JAI teoricamente permite que qualquer algoritmo de processamento de imagens possa

ser adicionado a sua API e ser usado como se fosse parte da biblioteca (JAI Guide, 1999). O mecanismo para adicionar funcionalidades a API possui múltiplos níveis de encapsulamento e complexidade. Isso permite aos programadores adicionar operações simples à biblioteca bem como operações complexas e com vários níveis de processamento.

7.1. Escrevendo novos métodos

Para a criação de novos operadores é necessário estender três classes com operações básicas da biblioteca. Após, deve-se utilizar o mecanismo de registro da JAI para permitir com que os novos métodos possam ser invocados de forma nativa (como pertencente à biblioteca) pelo programador. As classes necessárias para a criação de novos métodos são:

a) Uma classe estendendo a classe OpImage ou alguma de suas subclasses.

Essa classe é responsável pelo processamento da imagem, ou seja, as operações pixel a pixel; As subclasses de OpImage que já estão implementadas na JAI são:

9. AreaOpImage: classe abstrata para operadores que necessitam computar uma região ao redor de um determinado pixel;
10. PointOpImage: classe abstrata para operadores que necessitam processar sobre o conjunto de pixels de uma imagem e armazenar os valores correspondentes em uma imagem resultado;
11. SourcelessOpImage: classe abstrata para processamento de métodos que não contêm imagens;
12. StatisticsOpImag: classe abstrata que retorna o processamento sobre uma determinada região da imagem na forma de dados estatísticos.
13. WarpOpImag: classe abstrata para operações geométricas sobre as imagens.

Após estender a classe OpImage ou uma de suas subclasses, existem duas funções que devem ser implementadas (sobrescritas), getTile e computeRect. A função getTile é responsável por abrir uma região da imagem para leitura e retornar os valores calculados. A função computeRect processa o método no nível dos pixels e é chamada pela getTile. Dentro da função computeRect é necessária a utilização dos iteradores da JAI que definem a forma como os pixels serão acessados para leitura ou escrita. Por exemplo, a classe RasterAcessor.

Obs.: Caso a classe getTile não seja sobrescrita (nem sempre é necessário), a chamada ao método computeRect é feita pela implemetação da getTile que foi herdada.

b) Uma classe estendendo a classe OperationDescriptorImpl

Classe que descreve a operação, lista de parâmetros e demais detalhes do método. Os métodos que são chamados através da JAI pelo comando JAI.create precisam estar registrados na biblioteca em registryFile, incluído em jai.core.jar. Para isso os métodos devem estender a classe OperationDescriptorImpl, onde serão providas as descrições textuais do método, e definições para o formato das imagens e parâmetros.

c) Uma classe estendendo a classe CRIFImpl

Classe que permite chamadas ao operador através de cadeias de operação (chain ou graphs operations).

7.2. Exemplo

A operação `SubtractBackground` que remove o padrão de iluminação irregular de uma imagem com defeito.

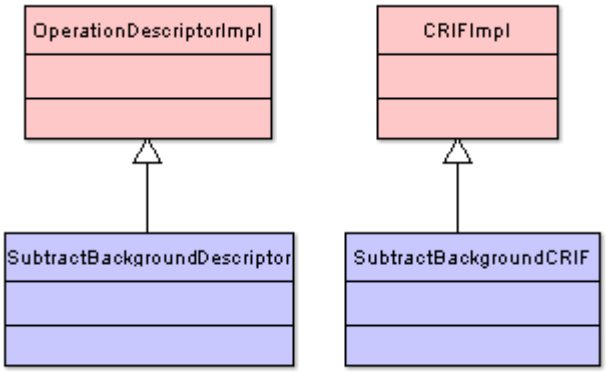
São 3 classes para a implementação do método:

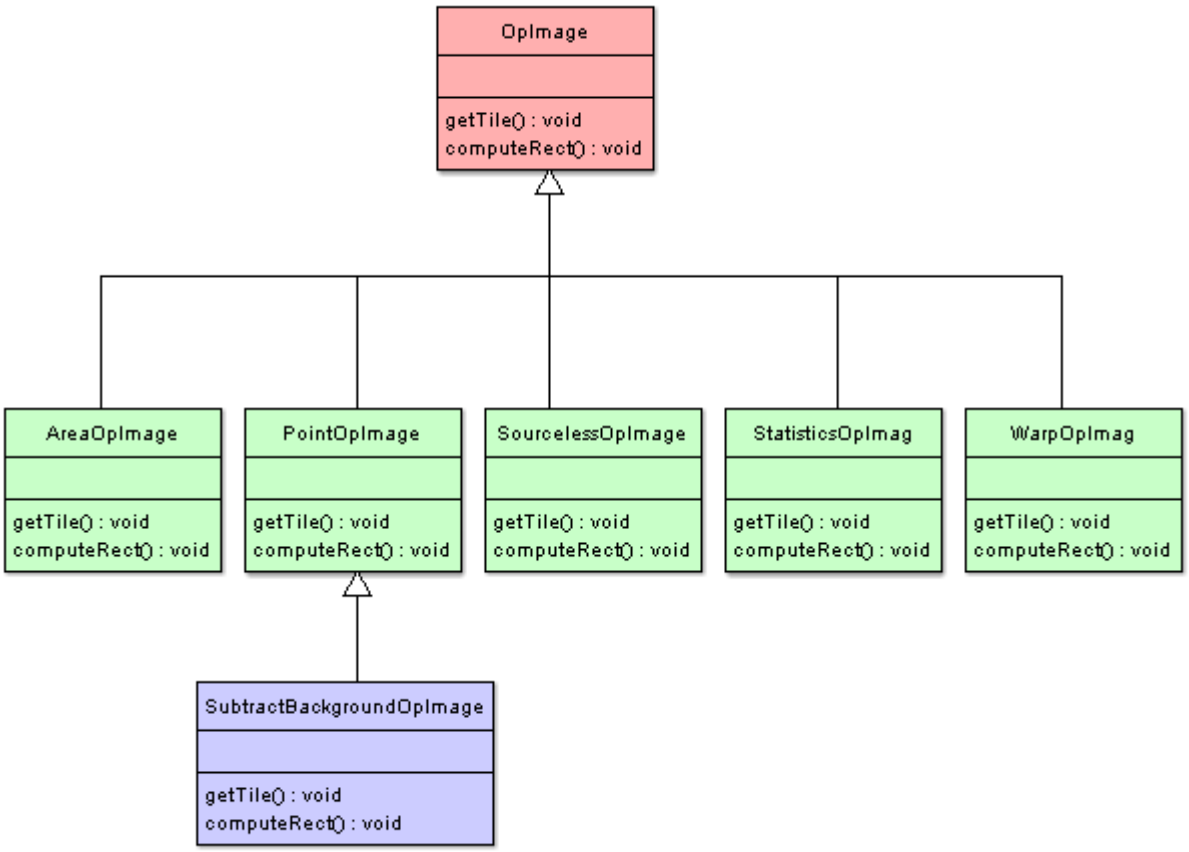
`SubtractBackgroundOpImage.java` - estende a classe `PointOpImage` que é subclasse de `OpImage`. Por questões de padronização o nome da classe fica: "`nome do método`" + `OpImage`.

`SubtractBackgroundDescriptor.java` - estende a classe `OperationDescriptorImpl`. O nome da classe fica: "`nome do método`" + `Descriptor`

`SubtractBackgroundCRIF.java` - estende a classe `CRIFImpl`. O nome da classe fica: "`nome do método`" + `CRIF`.

A figura abaixo mostra o diagrama UML das extensões.





7.3. Código Fonte

a) Classe SubtractBackgroundCRIF

```
/**
 * A <code>CRIF</code> supporting the "SubtractBackground" operation in the
 * rendered and renderable image layers.
 *
 * Um <code>CRIF</code> suportanto a operação "SubtractBackground" para
 * rendered e renderable image layers.
 */
public class SubtractBackgroundCRIF extends CRIFImpl {

    /** Constructor. */
    public SubtractBackgroundCRIF() {
        super("subBackground");
    }

    /**
     * Creates a new instance of <code>AddOpImage</code> in the rendered
     * layer. This method satisfies the implementation of RIF.
     *
       * @param paramBlock    The two source images to be added and the offset
parameter.
```

```
     * @param renderHints  Optionally contains destination image layout.
     */
    public RenderedImage create(ParameterBlock paramBlock,
                                RenderingHints renderHints) {
        // Get ImageLayout from renderHints if any.
        ImageLayout layout = RIFUtil.getImageLayoutHint(renderHints);


        return new SubtractBackgroundOpImage(paramBlock.getRenderedSource(0),
                                paramBlock.getRenderedSource(1),
                                paramBlock.getIntParameter(0),
                                renderHints,
                                layout);
    }

    /**
     * Creates a new instance for renderable layer - NOT IMPLEMENTED
     *
     * public RenderedImage createRenderable(ParameterBlock paramBlock ...
     */
}
```

b) Classe SubtractBackgroundDescriptor

```
/**
 * <b>DESCRIPTION</b>
 * <br><br>
 * SubtractBackground
 * <br><br>
 * Name for JAI invocation: "subbackground".
 *
 * <p> The SubtractBackground operation takes a source image with vignetting
 * effect, the background pattern and a value of offset to adjust the pixel color
 * level. For every pair of pixels, source image and the background source image
 * subtracts each pixel in the correspondent band and adds the offset.<br>
 * The two source images need to have same number of bands. The result image
 * size will be equal to the smallest source image. If the result of the operation
 * underflows/overflows the minimum/maximum value supported by the destination
 * data type, then it will be clamped to the minimum/maximum value respectively.
 *
 * <p> The destination pixel values are defined by the pseudocode:
 * <pre>
 * dst[x][y][dstBand] = clamp((srcs[0][x][y][src0Band] - srcs[1][x][y][src1Band]) +
offset);
 * </pre>
 *
 * <b>SOURCES</b> <br>
 *   source0 - source image with vignetting effect <br>
 *   source1 - background representation of the source image
 * <br><br>
 *
 * <b>PARAMETERS</b> <br>
 *   parameter0 - offset
 * <br><br>
 *
 * <b>RESULTS</b> <br>
 *   result0 - a RenderedImage
 * <br><br>
```

```
 *
 * <b>EXAMPLE</b> <br>
 * ParameterBlock pb = new ParameterBlock(); <br>
 * pb.addSource(source0); <br>
 * pb.addSource(source1); <br>
 * pb.add(parameter0); <br>
 * SubtractBackgroundDescriptor.register(); <br>
 * PlanarImage result0 = JAI.create("subbackground",pb);
 */
public class SubtractBackgroundDescriptor extends OperationDescriptorImpl {

    // The resource strings that provide the general documentation
    private static final String[][] resources = {
        {"GlobalName",  "subBackground"},
        {"LocalName",   "subBackground"},
        {"Vendor",      "br.com.animati.anima"},
           {"Description", "Subtract the background from a image with vignetting
effect"},
        {"DocURL",      "http://www.animati.com.br/"},
        {"Version",     "1.0"}
    };

    //supported modes
    private static final String[] supportedModes = {"rendered"};

    //source names
    private static final String[] sourceNames = {"source0","source1"};

     //source classes. Matriz [i,j], where i correspond to supportedModes and j to
sourceNames
         private  static  final  Class[][]  sourceClasses  ={  {PlanarImage.class,
PlanarImage.class } };

    //parameter names
    private static final String[] paramNames = {"offset"};

    //parameters classes
    private static final Class[] paramClasses = {java.lang.Integer.class};

    //parameter defaults
    private static final Object[] paramDefaults = null;//{new Integer(128)};

    //define the valid parameters values
    private static final Range[] validParamValues =
    {
        new Range(Integer.class, // 1st parameter
            new Integer(Integer.MIN_VALUE), new Integer(Integer.MAX_VALUE))
    };

    //variable to verify if the method is already registered
    private static boolean registered = false;



    /** Constructor. */
    public SubtractBackgroundDescriptor() {
        //super(resources, sourceClasses);
        super(resources, supportedModes, sourceNames, sourceClasses, paramNames,
                paramClasses, paramDefaults, validParamValues);
```

```
    }


    /**
     * <p>Creates a <code>ParameterBlockJAI</code> from all
     * supplied arguments except <code>hints</code> and invokes
     * {@link JAI#create(String,ParameterBlock,RenderingHints)}.
     */
    public static RenderedOp create(RenderedImage source0,
                                    RenderedImage source1,
                                    Integer offset,
                                    RenderingHints hints)  {
        ParameterBlockJAI pb =
            new ParameterBlockJAI("subBackground",
                                  RenderedRegistryMode.MODE_NAME);

        pb.setSource("source0", source0);
        pb.setSource("source1", source1);
        pb.setParameter("parameter0", offset);

        return JAI.create("subBackground", pb, hints);
    }

    //register the new operation to be accessed by JAI.create
    public static void register()
    {
        if(registered==false){
            SubtractBackgroundCRIF rif = new SubtractBackgroundCRIF();
                                                    OperationRegistry    opreg    =
JAI.getDefaultInstance().getOperationRegistry();
            opreg.clearPropertyState("rendered");
            opreg.registerDescriptor(new SubtractBackgroundDescriptor());
            RIFRegistry.register(opreg, "subBackground", "Anima", rif);
            registered = true;
        }
    }
}
```

c) Classe SubtractBackgroundCRIF

```
/**
 * <b>Description</b>
 *
 * An <code>OpImage</code> implementing the "SubtractBackground" operation as
 * described in <code>SubtractBackgroundDescriptor</code>.
 *
 * Uma <code>OpImage</code> implementando a operacao "SubtractBackground", que
 * está descrita em <code>SubtractBackgroundDescriptor</code>.
 */
final class SubtractBackgroundOpImage extends PointOpImage {

    //the offset //deslocamento de intensidade de cor
    private int offset = 0;

    /**
     * Constructs an <code>SubtractBackground</code>.
     *
     * @param source1  The first source image.
```

```
     * @param source2  The second source image.
     * @param offset   The offset to adjust the collor level.
     * @param layout   The destination image layout.
     */
    public SubtractBackgroundOpImage(RenderedImage source1,
                      RenderedImage source2,
                      int offset,
                      Map config,
                  ImageLayout layout) {
        super(source1, source2, layout, config, true);

        //Verify if the sources have the same number of bands, if not abort.
        //verifica se as imagens fonte possui o mesmo numero de bandas, se nao
aborta.
        if(source1.getSampleModel().getNumBands()!
=source2.getSampleModel().getNumBands()){
            String className = this.getClass().getName();
            throw new RuntimeException(className + "sources with different number
of bands");
        }

        //sets offset value //seta o valor do offset
        this.offset = offset;
    }

    /**
     * Subtracts the background within a specified rectangle.
     *
     * @param sources   Cobbled sources, guaranteed to provide all the
     *                  source data necessary for computing the rectangle.
     * @param dest      The tile containing the rectangle to be computed.
     * @param destRect  The rectangle within the tile to be computed.
     */
    protected void computeRect(Raster[] sources,
                               WritableRaster dest,
                               Rectangle destRect) {

        // Retrieve format tags.
        RasterFormatTag[] formatTags = getFormatTags();

        RasterAccessor s1 = new RasterAccessor(sources[0], destRect,
                                               formatTags[0],
                                               getSourceImage(0).getColorModel());
        RasterAccessor s2 = new RasterAccessor(sources[1], destRect,
                                               formatTags[1],
                                               getSourceImage(1).getColorModel());
        RasterAccessor d = new RasterAccessor(dest, destRect,
                                              formatTags[2], getColorModel());

        //if the dataType is supported
        if(d.getDataType() == DataBuffer.TYPE_BYTE){
            computeRectByte(s1, s2, d);
        }else {
            //else the source has the DataBuffer type not foreseen
            //se nao o tipo de dataBuffer nao foi previsto
            String className = this.getClass().getName();
            throw new RuntimeException(className + " - Source DataBuffer type not
suported");
        }
```

```
    }

    //perform the pixel computation
    private void computeRectByte(RasterAccessor src1,
                                 RasterAccessor src2,
                                 RasterAccessor dst) {

        byte[][] s1Data = src1.getByteDataArrays();
        byte[][] s2Data = src2.getByteDataArrays();
        byte[][] dData = dst.getByteDataArrays();

        //get values of destiny image (used for acess to the data)
        //pega os tamanhos e valores correspondentes a imagem de destino para
acessar seus dados
        int dwidth = dst.getWidth(); //width //largura da imagem
        int dheight = dst.getHeight(); //height //altura da imagem
        int bands = dst.getNumBands(); //numero de bandas da imagem
        int lineStride = dst.getScanlineStride(); //line length (width x bands)
//comprimento de uma linha(largura x bandas)
        int pixelStride = dst.getPixelStride(); //pixel lenght (=number of
bands) //tamanho do pixel (=num de bandas)
        int[] bandOffsets = dst.getBandOffsets(); //the band bandDataOffsets
//deslocamento entre as bandas

        for (int b = 0; b < bands; b++) {
            //get the pointers for the bands data //ponteiros para para as bandas
da imagem
            byte[] s1 = s1Data[b];
            byte[] s2 = s2Data[b];
            byte[] d = dData[b];

            int lineOffset = bandOffsets[b]; //band offset in the line
//deslocamento do pixel do canal b

            for (int h = 0; h < dheight; h++) {
                int pixelOffset = lineOffset; // pixel offset in the line // inicio
de deslocamento do pixel na linha
                lineOffset += lineStride; // next line //proxima linha

                for (int w = 0; w < dwidth; w++) {
                    // execute the operation  //executa a operacao
                    int diff = ((s1[pixelOffset]&0xFF) - (s2[pixelOffset]&0xFF)) +
offset;
                    d[pixelOffset] = clampByte(diff); //store the result

                    pixelOffset += pixelStride; // next pixel in the line //proximo
pixel na linha
                }
            }
        }
    }

    //clamp byte //aparar os bytes para que fiquem no intervalo entre 0 e 255
    private byte clampByte(int v) {
        if ( v > 255 ) {
             return (byte)255;
        } else if ( v < 0 ) {
             return (byte)0;
        } else {
```

```
                return (byte)v;
            }
        }
}
```

7.4. Exemplo de utilização

Criamos uma aplicação para subtrair o padrão de iluminação de uma imagem com problema de iluminação. O operador utilizado é o "`subBackground`" que foi criado conforme as classes acima. Os parâmetros que foram definidos para essa operação são: a imagem original, a imagem do padrão de iluminação e deslocamento intensidade de cor. Para maiores informações sobre o método, buscar em http://www-app.inf.ufsm.br/bdtg/tg.php?id=279.

```
public class Main {
    /**
     * Aplicação que subtrai o background de uma imagem com problema
     * de iluminação
     *
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        PlanarImage imagem = JAI.create("fileload",
"C:\\caminhoDaImagem\\imagem.bpm");
        PlanarImage padraoIluminacao = JAI.create("fileload",
"C:\\caminhoDaImagem\\padraoIluminacao.bpm");

        //registra a aperação para poder ser invocada por meio do JAI.crete.
        SubtractBackgroundDescriptor.register();

        //cria o parameterBlock e adiciona os argumentos
        ParameterBlock pb = new ParameterBlock();
        pb.addSource(imagem);
        pb.addSource(padraoIluminacao);
        pb.add(50);
        //então invocamos o operador subBackground através do JAI.create
        PlanarImage dst = JAI.create("subbackground", pb);

        //exibimos o resultado na tela
        DisplayJAI display = new DisplayJAI(dst);
        JFrame frame = new JFrame();
        frame.add(display);
        frame.setVisible(true);
        frame.pack();

    }
}
```

8. Créditos

Este tutorial tem como base o Java Advanced Imaging API: A Tutorial, de Rafael Santos, que foi apresentado no SIBGRAPI 04 (International Symposium on Computer Graphics, Image Processing and Vision, 2004) . Traduzimos o tutorial e modificamos alguns exemplos. O tutorial original pode ser encontrado em https://jaistuff.dev.java.net/docs/jaitutorial.pdf.

9. Referências

Java Advanced Imaging Programmer's Guide - Guia do programador para a biblioteca JAI http://java.sun.com/products/java-media/jai/forDevelopers/jai1_0_1guide-unc/

Java Advanced Imaging Tutorial – Tutorial online com exemplos de utilização da JAI. http://java.sun.com/developer/onlineTraining/javaai/

JAI Stuff - Material desenvolvido por Rafael Santos do Impe, com exemplos de utilização da biblioteca, métodos de processamento de imagens, etc. (em inglês). https://jaistuff.dev.java.net/

Java Advanced Imaging API**: A Tutorial** - Tutorial desenvolvido por Rafael Santos, apresentado no SIBGRAPI04. https://jaistuff.dev.java.net/docs/jaitutorial.pdf