11 Rendering Images in JAI

IN CHAPTER 10 WE INTRODUCED JAI IMAGE RENDERING through a couple of examples. In this chapter we'll discuss this topic in greater detail. Although the underlying device-rendering model is the same, rendering JAI images is often more complex than rendering AWT images or buffered images. The main reason is the underlying tiling mechanism, which enables the rendering of large images.

As mentioned in Chapter 10, JAI has two types of rendering modes: rendered and renderable. Each has its own rendering philosophy. We'll look at both of these layers from the JAI perspective by explaining the relevant API and then giving some examples.

Before you render a JAI image, you may need to perform an operation that enables smooth and proper rendering. This operation may involve passing some rendering hints, allocating enough memory for tiles, or even restructuring the image itself. Let's look at some of these operations before we delve into the rendering process itself.

Applying Rendering Hints

The rendering-hints concept was introduced in Java 2D (see Chapter 5). The `java.awt.RenderingHints` class represents rendering hints. In this representation, each rendering hint is a key-value pair. Rendering-related methods of the `Graphics2D` and `BufferedImageOp` classes take `RenderingHints` as an input. When there are multiple rendering hints, they are passed to these methods as the `Map` object of a key-value pair. The rendering-hints key is of type `RenderingHints.Key`, and the value is an object.

JAI doesn't use the Java 2D rendering hints for rendering a node. Instead it uses its own rendering hints, the keys for which are defined in the `JAI` class. A rendering-hint key in JAI is of type `JAI.RenderingKey`, which is an inner class of `JAI` and a subclass of `java.awt.RenderingHints.Key`.

TABLE 11.1 Rendering Hints in JAI

KEY	VALUE	COMMENTS
KEY_BORDER_EXTENDER	Objects created from `BorderExtender` and its subclasses	There are five border types: zero fill, constant fill, copy, reflection, and wrap.
KEY_TILE_CACHE	`TileCache` object	This key controls the amount of memory allotted for caching tiles.
KEY_OPERATION_BOUND	One of `OpImage.OP_COMPUTE_BOUND, OpImage.OP_IO_BOUND, or OpImage.OP_NETWORK_BOUND`	This key indicates whether an operation is computation-, I/O-, or network-bound.
KEY_OPERATION_REGISTRY	`OperationRegistry` object	With this key you can choose an operation registry other than the default.
KEY_INTERPOLATION	Objects created from the `Interpolation` class	JAI implements its own interpolation classes. There are four types of interpolation: nearest neighbor, bilinear, bicubic, and bicubic2 (see Chapters 7 and 12).
KEY_IMAGE_LAYOUT	`ImageLayout` object	This key indicates the tile layout of a JAI image.

JAI rendering hints are typically passed to an operator that creates a node in the rendering chain. The rendering hints are applied when a node is evaluated—that is, when the image is ready for rendering.

We obtain the value for a JAI rendering hint by creating an instance from an appropriate class in JAI. Table 11.1 lists the JAI rendering-hint keys and their corresponding values.

Let's look at some prerendering operations, some of which are rendering hints.

Managing Memory

As mentioned earlier, in JAI image data is available in cache as tiles. To manage memory allocation for tiles, JAI provides an interface called `TileCache.` You can create an instance of `TileCache` by calling the factor method `JAI.getTileCache().` By default, the tile cache size is set to 64MB. To change this setting, the `TileCache` interface specifies the following set method:

- **`public void setMemoryCapacity(long memoryCapacity)`**

Here's an example that sets the memory:

```
TileCache tileCache  =  JAI.getDefaultInstance().getTileCache();
tileCache.setMemoryCapacity(2048*2048L);
```

Here's the corresponding get method:

- **public long getMemoryCapacity()**

The `memoryCapacity` parameter is specified in bytes. If the memory capacity is smaller than the current capacity, the tiles in the cache are flushed to achieve the desired settings. If you set a large amount of memory for the tile cache, interactive operations are faster but the tile cache fills up very quickly. If you set a low amount of memory for the tile cache, the performance degrades. So there's a trade-off between memory and speed.

The `TileCache` interface also has a method for setting the capacity as a specific number of tiles:

- **public void setTileCapacity(int tileCapacity)**

and a corresponding get method:

- **public int getTileCapacity()**

In JAI 1.0.2, implementing these methods does nothing. In JAI 1.1, these methods are deprecated.

The other utility methods are

- **public void add(java.awt.image.RenderedImage owner,**
  int tileX,
  int tileY,
  Raster data)
- **public void remove(java.awt.image.RenderedImage owner,**
  int tileX,
  int tileY)
- **public Raster getTile(RenderedImage owner,**
  int tileX,
  int tileY)
- **public void removeTiles(RenderedImage owner)**
- **public void flush()**

In these methods the `tileX` and `tileY` parameters are the indices of the tile in the planar image. The `flush()` method discards all the tiles in the cache, and the `removeTiles()` method discards only tiles belonging to the image passed as the input parameter.

`TileCache` is also a rendering hint, and it can also be set through `JAI`'s `setTileCache()` method.

Scheduling Tiles

Like `TileCache`, `TileScheduler` is an interface. It has three methods for managing the tile-scheduling computation:

- **`public Raster scheduleTile(OpImage target, int tileX, int tileY)`**
- **`public Raster[] scheduleTiles(OpImage target, Point[] tileIndices)`**
- **`public void prefetchTiles(PlanarImage target, Point[] tileIndices)`**

Here's an example:

```
Point[] tileIndices = { new Point(0,0), new Point(1,0), new Point(2,0)};
TileCache tileCache  =  JAI.getDefaultInstance().getTileCache();
tileCache.prefetchTiles(image, tileIndices);
```

This example gives a hint to prefetch the first three tiles of an image.

Just as the `setTileCache()` method is used to set `TileCache`, you can use `setTileScheduler()` to set a `TileScheduler` object. Unlike `TileCache`, however, `TileScheduler` is not a rendering hint.

Reformatting an Image

Often an image needs to be changed before it is rendered. For example, when you load images using the JAI codec, they are not typically tiled because most formats do not support tiling. And even in formats that do support tiling (e.g., TIFF, FlashPix), images may not be properly tiled. A planar image that is not tiled is like a buffered image, and all the advantages of tiling are lost.

One solution is to tile the image in memory. The `ImageLayout` class lets you restructure the image into tiles. After loading an image in memory, sometimes you will want to change the image bounds, `SampleModel`, `ColorModel`, and so on. The `ImageLayout` class lets you do all that. The `ImageLayout` object is also used as a value of the `RenderingHints` key.

`ImageLayout` has several attributes related to the image to be generated, and each of these attributes has a bit mask (see Table 11.2).

The `ImageLayout` class has get and set methods for each of these attributes. Each bit mask can be set, unset, or checked for its state, through the following methods:

- **`public ImageLayout setValid(int mask)`**
- **`public ImageLayout unsetValid(int mask)`**
- **`public final boolean isValid(int mask)`**

TABLE 11.2 `ImageLayout` Attributes and Associated Bit Masks

ATTRIBUTE	MASK
`ColorModel`	`COLOR_MODEL_MASK`
`SampleModel`	`SAMPLE_MODEL_MASK`
`height`	`HEIGHT_MASK`
`width`	`WIDTH_MASK`
`minX`	`MIN_X_MASK`
`minY`	`MIN_Y_MASK`
`tileWidth`	`TILE_WIDTH_MASK`
`tileHeight`	`TILE_HEIGHT_MASK`
`tileGridXOffset`	`TILE_GRID_OFFSET_X_MASK`
`tileGridYOffset`	`TILE_GRID_OFFSET_Y_MASK`

- **`public int getValidMask()`**

  `ImageLayout` also has methods for setting a group of masks:

- **`public ImageLayout unsetImageBounds()`**

  This method unsets the bit masks associated with the `minX`, `minY`, `width`, and `height` attributes.

- **`public ImageLayout unsetTileLayout()`**

  This method unsets the bit masks associated with the `tileGridXOffset`, `tileGridYOffset`, `tileWidth`, and `tileHeight` attributes.

If the mask is set, the get method returns the value that was set by the `ImageLayout` object. If the mask is unset, the get method returns the attribute from the original rendered image.

Using the Format Operator

The `TileLayout` class is typically used in conjunction with the Format operator to restructure an image. The source image for the Format operator can be a rendered image or a renderable image. The tile layout is passed to the Format operator through `RenderingHints`. As mentioned earlier, the `KEY_IMAGE_LAYOUT` key in this case is defined in the `JAI` class itself. The value of this key is the `ImageLayout` object. Table 11.3 lists the parameters of the Format operator.

The example in Listing 11.1 uses the Format operator.

TABLE 11.3 Format Operator Parameters

OPERATOR NAME	PARAMETER	TYPE	DEFAULT VALUE	DESCRIPTION
Format	`transferType`	`int`	`TYPE_BYTE`	Data type of the output image. The value should be one of `TYPE_BYTE`, `TYPE_SHORT`, `TYPE_USHORT`, `TYPE_INT`, `TYPE_FLOAT`, or `TYPE_DOUBLE`. These constants are defined in the `java.awt.image.DataBuffer` class.

LISTING 11.1 Reformatting a planar image

```
public static RenderedOp reformatImage(PlanarImage img, Dimension tileDim) {
   int tileWidth = tileDim.width;
   int tileHeight = tileDim.height;
   ImageLayout tileLayout = new ImageLayout(img);
   tileLayout.setTileWidth(tileWidth);
   tileLayout.setTileHeight(tileHeight);

   HashMap map = new HashMap();
   map.put(JAI.KEY_IMAGE_LAYOUT, tileLayout);
   map.put(JAI.KEY_INTERPOLATION,
          Interpolation.getInstance(Interpolation.INTERP_BICUBIC));
   RenderingHints tileHints = new RenderingHints(map);

   ParameterBlock pb = new ParameterBlock();
   pb.addSource(img);
   return JAI.create("format", pb, tileHints);
}
```

In Listing 11.1 the `ImageLayout` object is created with a specified tile width and height. This object is then saved in `HashMap`, as is the other rendering-hint object, `Interpolation`. A `RenderingHints` object is constructed with this `HashMap` instance and then passed to `JAI.create()`. This method uses the `ParameterBlock` and `RenderingHints` parameters to execute the Format operator. The image returned by this operator has the new tile layout and other parameters set by the `ImageLayout` object.

Although the `ParameterBlock` object in this method is created for a planar image, it can be a renderable image as well.

Extending the Border

It is often necessary to add borders to images. For example, as we saw in Chapter 8, the convolution operations often cannot fill the last rows and columns of an image.

One way to fill them is to add a border. In addition, you may wish to add a decorative border to a displayed image. There are two ways you can add a border:

1. Using the subclasses of `BorderExtender`
2. Using the Border operator

Using the BorderExtender Class

Let's look at the `BorderExtender` class hierarchy first, as shown in Figure 11.1. `BorderExtender`, which is at the root of the hierarchy, is an abstract class that allows you to specify the number of pixels to be filled in all sides of the border. `BorderExtender` has five subclasses, each of which provides a different way to fill the border pixels.

The subclasses of `BorderExtender` represent the following five operations:

1. **Zero fill.** The border is filled with zeros. The constant `BorderExtender.BORDER_ZERO` represents this operation.
2. **Constant fill.** The pixels in the border are filled by a constant value. If the image has more than one band, each band can have a different constant fill value. The

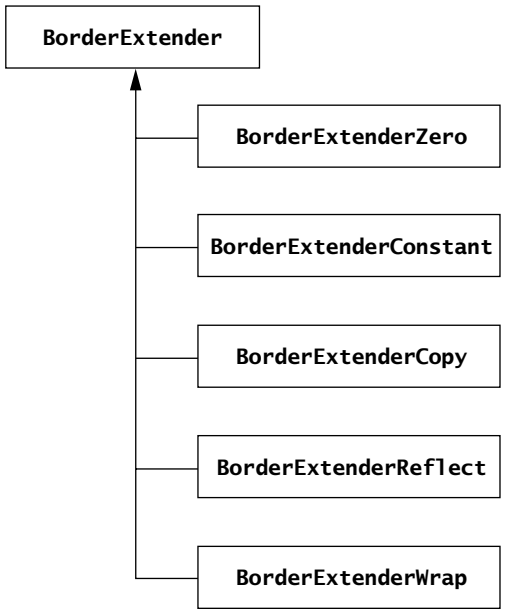

FIGURE 11.1 The `BorderExtender` class hierarchy

constant values are provided as an array through the constructor of the `BorderExtenderConstant` class.

3. **Copy.** The edge and corner pixels of the source image are copied onto the border area. The constant `BorderExtender.BORDER_COPY` represents this operation.
4. **Reflection.** The pixels on the outer edge of the source image are copied onto the border area. The constant `BorderExtender.BORDER_REFLECT` represents this operation.
5. **Wrap.** The image pixels are copied in such a way that the opposite edges of the images appear to be joined (see Figure 11.2). The `BorderExtender.BORDER_WRAP` constant represents this operation.

The `BorderExtender` class has the following two methods—one abstract and the other static:

1. **`public static BorderExtender createInstance(int extenderType)`**

   This factory method constructs an instance of a `BorderExtender` object. The `BorderExtender` class has four constants to represent the extender operations: `BORDER_ZERO`, `BORDER_COPY`, `BORDER_WRAP`, and `BORDER_REFLECT`. The `extenderType` parameter can be any of the four constants. Note that no constant

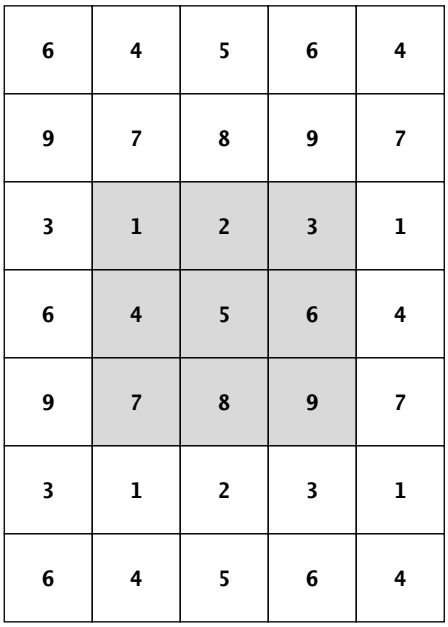

FIGURE 11.2 The border wrap operation

is defined for the constant fill operation because constant fill requires an array of constant values.

2. `public abstract void extend(java.awt.image.WritableRaster raster, PlanarImage im)`

   The subclasses of the `BorderExtender` class implement this method. As Figure 11.3 shows, the `writableRaster` parameter contains the border and the image. Depending on the operation, appropriate pixels are copied onto the border area of `writableRaster.` If the `Raster` object supplied doesn't cover the image, no pixels are copied.

   You can use a `BorderExtender` object in two ways:

1. The `BorderExtender` object can be supplied to an operation as a rendering hint. `JAI.KEY_BORDER_EXTENDER` is the key value for this rendering hint. The value can be any one of the five `BorderExtender` constants.
2. The `PlanarImage` class has the following methods that take `BorderExtender` as one of the parameters:
   - `public void copyExtendedData(WritableRaster dest, BorderExtender extender)`
   - `public Raster getExtendedData(Rectangle region, BorderExtender extender)`

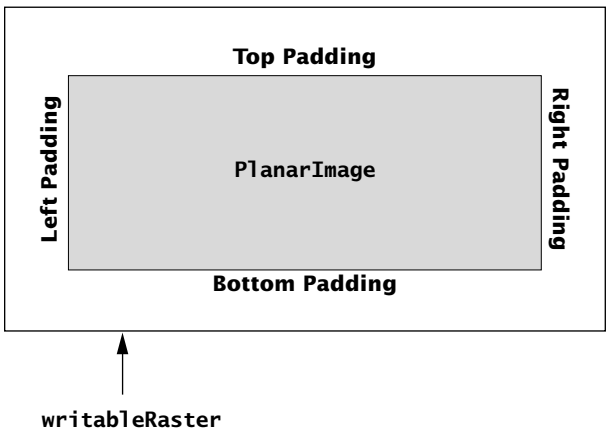

FIGURE 11.3 An image layout with borders

Using the Border Operator

You can also use the Border operator to add the border. This operator takes a rendered image as a source. Table 11.4 lists the parameters of the Border operator.

The example in Listing 11.2 uses the Border operator to add a border to images.

LISTING 11.2 Adding a border to an image

```
public static RenderedOp setConstantBorder(PlanarImage img,
                                           Dimension border, double constVal) {
   ParameterBlock pb = new ParameterBlock();
   pb.addSource(img);
   pb.add(border.width);
   pb.add(border.height);
   pb.add(border.width);
   pb.add(border.height);
   pb.add(BorderDescriptor.BORDER_CONST_FILL);
   int numbands = img.getSampleModel().getNumBands();
   double[] fillValue = new double[numbands];
   for (int i=0;i<numbands;i++) {
        fillValue[i] = constVal;
   }
   pb.add( fillValue);
   return JAI.create("border", pb);
}
```

When the border is added, the image dimensions change. The width and height of the image increase, and `minX` and `minY` coordinates become negative. Keep this in mind while writing the code for painting the image. To make the image display appear symmetrical, you can either use the Format operator to restructure the image or translate the image by an appropriate amount.

TABLE 11.4 Border Operator Parameters

OPERATOR NAME	PARAMETER	TYPE	DEFAULT VALUE	DESCRIPTION
Border	`leftPad`	`int`	0	See Figure 11.3
	`rightPad`	`int`	0	
	`topPad`	`int`	0	
	`bottomPad`	`int`	0	
	`type`	`int`	`BorderDescriptor.BORDER_ZERO_FILL`	One of the `BorderDescriptor` constants
	`constants`	`double[]`	`null`	An array of constants

A Rendering Example

Let's look at an example (see Listing 11.3) that illustrates how to use some of the rendering hints and how to add a border.

LISTING 11.3 The `ImageFormatter` class

```
import java.awt.*;
import java.util.*;
import java.awt.image.*;
import java.io.*;
import javax.swing.*;
import java.awt.geom.*;
import java.awt.image.renderable.*;
import javax.media.jai.operator.BorderDescriptor;
import javax.media.jai.*;

public class ImageFormatter extends app.JAISimpleViewer{
   protected Dimension border = new Dimension(10,10);

   public static void main(String[] args){
      ImageFormatter ip = new ImageFormatter();
      if(args.length <1) {
        System.out.println("Enter a valid image file name");
        System.exit(0);
      }
      ip.loadAndDisplay(args[0]);
   }

   public void loadAndDisplay(String filename){
      PlanarImage img = readAsPlanarImage(filename);
      TileCache tc = JAI.getDefaultInstance().getTileCache();
      System.out.println("Default Memory Capacity = "+ tc.getMemoryCapacity());
      tc.setMemoryCapacity(2024*2024);
      System.out.println("Memory Capacity = "+ tc.getMemoryCapacity());

      System.out.println("Without Border");
      System.out.println("--------");
      System.out.println("Size: "+
                         img.getWidth()+ " , "+  img.getHeight());
      System.out.println("ULHC  "+
                         img.getMinX()+ " ,"+  img.getMinY());

      RenderedOp borderImg
            = com.vistech.jai.util.JAIUtil.setConstantBorder(img, border, 70);

      System.out.println("With Border");
      System.out.println("-------");
      System.out.println("Size: "+
                         borderImg.getWidth()+ " ,"+  borderImg.getHeight());

      System.out.println("ULHC:  "+
                         borderImg.getMinX()+ " ,"+  borderImg.getMinY());

      RenderedOp opImage = reformatImage(borderImg, new Dimension(256, 256));
```

continued

```
        displayImage(opImage);
    }

    public static RenderedOp reformatImage(PlanarImage img, Dimension tileDim) {
        ImageLayout tileLayout = new ImageLayout(img);
        tileLayout.setTileWidth(tileDim.width);
        tileLayout.setTileHeight(tileDim.width);

        HashMap map = new HashMap();
        map.put(JAI.KEY_IMAGE_LAYOUT, tileLayout);
        map.put(JAI.KEY_INTERPOLATION,
                Interpolation.getInstance(Interpolation.INTERP_BICUBIC));

        RenderingHints tileHints = new RenderingHints(map);
        ParameterBlock pb = new ParameterBlock();
        pb.addSource(img);
        return JAI.create("format", pb, tileHints);
    }

    public void launchFrame(PlanarImage img, int width, int height) {
        setTitle("JAI Image Viewer");
        viewer = new BorderedPanel(img);
        viewer.setPreferredSize(new Dimension(width, height));
        Container cp = getContentPane();
        getContentPane().setLayout(new GridLayout(1,1));
        cp.add(viewer);
        pack();
        setSize(new Dimension(width, height));
        show();
        viewer.repaint();
    }

    protected class BorderedPanel extends ImagePanel {

        public BorderedPanel(PlanarImage img){ super(img);}

        public void paintComponent(Graphics gc){
            Graphics2D g = (Graphics2D)gc;
            Rectangle rect = this.getBounds();
            if((width != rect.width) || (height != rect.height)){
                double magx = rect.width/(double)width ;
                double magy = rect.height/(double)height ;
                atx.setToScale(magx, magy);
                atx.translate(border.width, border.height);
            }
            if(image != null) g.drawRenderedImage(image, atx);
        }
    }
}
```

The `ImageFormatter` class extends the `JAISimpleViewer` class that we developed in in Chapter 10 (see Listing 10.1). The `loadAndDisplay()` method first sets the tile cache and then adds a border. To add the border, `loadAndDisplay()` calls the `setConstantBorder()` method shown in Listing 11.2. Before the image is rendered, `loadAndDisplay()` calls the `reformatImage()` method to set the tile layout.

The `loadAndDisplay()` method has various print statements. When you run the application in Listing 11.3, you can see how the dimensions of the image change after the Border operator is applied. Figure 11.4 shows an image rendered with a constant border. To run the application, type "java app.ImageFormatter <image path>" on the command line.

> **Note:** The code for `ImageFormatter` is available on the book's Web page in the directory `src/chapter11/format.`

Here's an example of `ImageFormatter` output on the console:

```
Default Memory Capacity = 16777216
Memory Capacity = 4096576
Without Border
-------
Size: 208 ,222
ULHC  0 ,0
With Border
------
Size: 228 ,242
ULHC:  -10 ,-10
```

Notice that the size and the ULHC coordinates are changed after the border operation.

FIGURE 11.4 An image displayed with a border

A Closer Look at the PlanarImage Class

The `PlanarImage` class is at the heart of image rendering in JAI. It is the base class for representing an image. Figure 10.3 showed the `PlanarImage` class hierarchy. Whether it is a rendered or renderable chain, the image is normally converted to a planar image for actual rendering because the planar image represents the actual physical image.

Although we have used `PlanarImage` in many examples, we have not described it in detail. So in this section we'll look more closely at its methods.

The `PlanarImage` class has just one constructor, which takes no arguments. A `PlanarImage` object is typically constructed by an image reader or an image operator, so you may not need to use this constructor at all.

Even when an image reader returns a `RenderedImage` object, as in the case of the Image I/O API, it can be converted to a `PlanarImage` object by the `RenderedImageAdapter` class.

```
RenderedImage renderedImage = javax.imageio.ImageIO.read (new File("myImage.jpg"));
RenderedImageAdapter planarImage = new RenderedImageAdapter(renderedImage);
```

Note that `RenderedImageAdapter` is a subclass of `PlanarImage.`

As mentioned earlier, the `RenderedImage` interface, which the `PlanarImage` class implements, has set and get methods for `RenderedImage` attributes, which are listed in Tables 10.1 and 10.2. In addition, the `PlanarImage` class has methods that are needed for rendering a JAI image. The tasks accomplished by these methods include converting the image to a buffered image or a snapshot image. The `PlanarImage` class also has some static utility methods that are often used in rendering. The sections that follow review the specific methods of `PlanarImage.`

Converting to a Buffered Image

The `PlanarImage` class has two methods that can convert a JAI image to a buffered image:

- **`public BufferedImage getAsBufferedImage()`**
- **`public BufferedImage getAsBufferedImage(Rectangle rect, ColorModel colorModel)`**

The first method converts the entire image into a buffered image. If the image is huge, you may not want to use this method because `BufferedImage` keeps all the image data in memory.

The second method converts a specified rectangular area into a buffered image. The color model must be compatible with the source image. If not, this method will throw an exception: `IllegalArgumentException.` If the value of the `colorModel` parameter is `null,` the source color model is used for constructing the buffered image.

These methods are handy when you need to construct a buffered image for the purpose of rendering or saving images. For example, a bean or application that is written with the Java 2D API may use only `BufferedImage` inputs. In that case you can use JAI for loading an image, convert it to a buffered image, and then pass it to the Java 2D bean or application. Along the same lines, a codec might have been developed with only Java 2D in mind. To save a planar image using that codec, you can use these methods to convert the image to a buffered image.

Remember that the `RenderedImage` interface is an API in Java 2D and part of J2SE. So if a method in a bean or application written with the Java 2D API takes a `RenderedImage` input, the `PlanarImage` object can be passed to it. The new Image I/O API takes this approach. In this API, methods to write an image take `RenderedImage` as an input. With this API, then, there is no need to convert `PlanarImage` to `BufferedImage` before saving.

Creating a Snapshot Image

As the name suggests, the `SnapshotImage` class represents a snapshot of the image contents at a given time. The `PlanarImage` class has the following method for creating such a snapshot:

- **`public PlanarImage createSnapshot()`**

This method creates an immutable image with a copy of the source image's current contents. Multiple calls to this method don't create multiple copies of snapshot images. Instead, they create multiple references to a single `SnapshotImage` object. This means that at any given time there is one `SnapshotImage` object per `PlanarImage` instance.

Converting to a Planar Image

The following factory method in the `PlanarImage` class constructs a `PlanarImage` object from a `RenderedImage` object:

- **`public static PlanarImage wrapRenderedImage(java.awt.image.RenderedImage im)`**

This method adds various properties to the input image, such as source and sink vectors and the ability to produce snapshots to create a `PlanarImage` object. If the image is already a planar image, it is simply returned unchanged. If the input image implements the `RenderedImage` interface, this method constructs a `RenderedImageAdapter` object. If the input implements the `WritableRenderedImage` interface, the method constructs a `WritableRenderedImageAdapter` object. `RenderedImageAdapter` is a subclass of `PlanarImage`, and `WritableRenderedImageAdapter` is a subclass of `RenderedImageAdapter`.

The `wrapRenderedImage()` method is a very convenient method that converts `RenderedImage` to the JAI-specific `PlanarImage`. For instance, a `BufferedImage` object that implements the `RenderedImage` interface can be converted to a `PlanarImage` object with this method. To convert images read by the Java 2D JPEG codec and the new Image I/O API, you can use this method as shown in the following example:

```
BufferedImage renderedImage = javax.imageio.ImageIO.read(new File("myImage"));
PlanarImage planarImage = PlanarImage.wrapRenderedImage(renderedImage)
```

Converting Image Coordinates to Tile Index and Vice Versa

As we'll see later in this chapter, it is often necessary to compute the tile index when a coordinate in the image is given. For example, when an image is scrolled, the application needs to compute the ULHC of the image that is visible on the viewport. So the rendering program needs to compute tiles for the visible portion of the image. The following methods of the `PlanarImage` class do this:

- **`public int XToTileX(int x)`**
- **`public int YToTileY(int y)`**

- **`public static int XToTileX(int x, int tileGridXOffset, int tileWidth)`**
- **`public static int YToTileY(int y, int tileGridYOffset, int tileHeight)`**

The following methods do the reverse: convert a tile index to image coordinates:

- **`public static int tileXToX(int tx, int tileGridXOffset, int tileWidth)`**
- **`public static int tileYToY(int ty, int tileGridYOffset,`**
  `int tileHeight)`

- **`public int tileXToX(int tx)`**
- **`public int tileYToY(int ty)`**

Obtaining Sources and Sinks

When a `PlanarImage` object is part of a rendering chain, it is often necessary to obtain the list of sources and sinks. Here are the methods that do this:

- **`public java.util.Vector getSinks()`**
- **`public java.util.Vector getSources()`**
- **`public int getNumSources()`**

Using the RenderedOp Class

The `RenderedOp` class represents a node in the rendered layer. Even though `RenderedOp` has two public constructors, you don't normally need to construct it directly. To obtain a `RenderedOp` object, you must execute the `JAI.create()` or `JAI.createNS()` method with an operator argument.

`RenderedOp` extends the `PlanarImage` class, so it inherits all the `RenderedImage` data and properties, which are generated by application of the operator. `RenderedOp` also overrides most of the `PlanarImage` methods. In addition, it holds the name of the operation and the parameters passed to execute the operation.

Note that the `RenderedOp` class is *serializable,* which means that it implements the `Serializable` interface and the internal state of a `RenderedOp` object can be saved. Serialization is necessary for performing the image-related operations over the network using RMI (Remote Method Invocation). As we'll see in Chapter 15, only `Serializable` objects can be transported across the network. In other words, an image can be sent from a server to a client (and vice versa) only if it is serializable. As you probably know already, the AWT `Image` and `BufferedImage` objects are not serializable. Neither is `PlanarImage.`

> **Note:** The serialization feature was introduced in JDK 1.1 because it was required by many Java core APIs, such as RMI and JavaBeans. See Appendix B to learn more about serialization.

Constructing a Rendered Node

Although the `RenderedOp` class has two public constructors, you will typically use a *JAI.createXXX()* method to create a rendered node. Doing so will save you the trouble of writing and debugging several lines of code. In some instances, however, you may want to construct a `RenderedOp` object by explicitly using a `RenderedOp` constructor. There are two constructor types in the `RenderedOp` class:

1. `public RenderedOp(OperationRegistry registry,`
   `java.lang.String opName,`
   `java.awt.image.renderable.ParameterBlock pb,`
   `java.awt.RenderingHints hints)`
2. `public RenderedOp(java.lang.String opName,`
   `java.awt.image.renderable.ParameterBlock pb,`
   `java.awt.RenderingHints hints)`

To construct a node, you need four parameters:

1. `OperationRegistry`, in which an operation is registered
2. Operation name
3. `ParameterBlock`, which holds the parameters
4. `RenderingHints`

If the value of the `OperationRegistry` argument is `null`, the default value is assumed. If the value of the `RenderingHints` argument is `null`, no hints are associated with the node.

Rendering a Node

Because `RenderedOp` is a subclass of `PlanarImage`, you can directly pass the `RenderedOp` object to `Graphics2D`'s `drawRenderedImage()` to render a node. When a node is rendered directly from the `RenderedOp` object, the node is said to be *frozen*. This means that the node parameters are not allowed to change.

To avoid freezing the node, call the following method to create an instance of `PlanarImage` and then pass that instance to the `drawRenderedImage()` method:

- **`public PlanarImage createInstance()`**

It is extremely important that the node not be frozen when operations are performed dynamically in a rendering chain. A certain dynamic operation may change the sources, parameters, and rendering hints of a node. If a node is frozen, these changes cannot be made.

In addition to the case already mentioned (i.e., direct rendering with `RenderedOp`), a rendered node is frozen in the following cases:

- When the `RenderedImage` methods of `RenderedOp` are called (see the section titled Rendered Images in Chapter 10 for the `RenderedImage` interface)
- When the overridden `PlanarImage` methods, which include get and set methods for sources, are called

Even if a node is frozen, serialization can "unfreeze" it because the instance variables that hold the rendering information are `transient` and hence not saved. Upon deserialization of a `RenderedOp` object, its rendering attributes can be changed again.

The `RenderedOp` class saves the resulting image from rendering in an instance variable. You can obtain the image with the following method:

- **`public PlanarImage getRendering()`**

You can use this method for repeated rendering of the resulting image. If the node is edited, however, this method does create a new rendering. In such cases you need to call the `createInstance()` method again.

Sources and Sinks

A rendered node can have zero or more sources and sinks. Both the sources and the sinks are held in a `Vector` object. You can retrieve, add, and remove sources and sinks using the following methods:

- **`public int getNumSources()`**
- **`public java.lang.Object getNodeSource(int index)`**
- **`public void addNodeSource(Object source)`**
- **`public void setNodeSource(Object source, int index)`**
- **`public boolean removeSource(PlanarImage source)`**
- **`public void setSources(java.util.List sourceList)`**
- **`public boolean removeSource(PlanarImage source)`**
- **`public java.util.Vector getSinks()`**
- **`public void addSink(PlanarImage sink)`**
- **`public boolean removeSink(PlanarImage sink)`**

Rendered-Node Attributes

A rendered node has several attributes that are set while a node is being created. As mentioned earlier, a node can be edited. To edit a node, you typically retrieve an attribute, apply a particular rule or condition, and if applicable, assign a new value to the attribute. The `RenderedOp` class has set and get methods to do these things:

- **`public OperationRegistry getRegistry()`**
- **`public void setRegistry(OperationRegistry registry)`**

- **`public String getOperationName()`**
- **`public void setOperationName(String opName)`**

- **`public ParameterBlock getParameterBlock()`**
- **`public void setParameterBlock(ParameterBlock pb)`**

- **`public RenderingHints getRenderingHints()`**
- **`public void setRenderingHints(RenderingHints hints)`**

These attributes can be image related as well as operation related. The operation-related attributes include operation registry, operation name, operation parameters, and rendering hints. The operation parameters can be part of a `ParameterBlock` object.

Editing a Node

You can set the node attributes to edit a node, provided the node is not frozen. You can use the get and set methods in the previous section to modify a node.

To edit the parameters of an operation, you must first retrieve the `ParameterBlock`, then modify it and set it again. Alternatively, you can use the following `RenderedOp` methods to retrieve and set a desired parameter.

- Get methods:
  - `public int getNumParameters()`
  - `public java.util.Vector getParameters()`
  - `public byte getByteParameter(int index)`
  - `public char getCharParameter(int index)`
  - `public short getShortParameter(int index)`
  - `public int getIntParameter(int index)`
  - `public long getLongParameter(int index)`
  - `public float getFloatParameter(int index)`
  - `public double getDoubleParameter(int index)`
  - `public Object getObjectParameter(int index)`
- Set methods:
  - `public void setParameter(byte param, int index)`

There are similar methods for setting `short`, `int`, `long`, `float`, `double`, and `object` parameters.

Adding and Retrieving Properties

Some imaging operations may create image-related properties. The `RenderedOp` class has the following methods to help retrieve and set such properties:

- `public java.lang.String[] getPropertyNames()`
- `public String getProperty(java.lang.String name)`
- `public void setProperty(String name, Object value)`
- `public void suppressProperty(String name)`
- `public void addPropertyGenerator(PropertyGenerator pg)`

Working with Tiles

As you probably know by now, neither AWT imaging nor Java 2D can handle smooth rendering of large images. The reason is that the underlying imaging models read the entire image in memory. As mentioned in Chapter 10, the JAI pull model overcomes this problem through tiling. Although the tile concept is defined in Java 2D, JAI actually implements it. However, tiling works best when the underlying format supports tiles, and this is not the case with most image formats.

If an image is not tiled, the FileLoad operator treats the entire image as a single tile and loads it in the memory. As mentioned earlier, we use the Format operator to tile such images in memory.

Viewing and Manipulating Large Images

In Chapter 10 we developed the `JAIImageCanvas` class (see Listing 10.2) to display planar images. If we use this class for manipulating large images, the rendering is slow because `JAIImageCanvas` does not completely exploit the tiling mechanism.

In the `JAIImageCanvas` class, the `paintComponent()` method calls `Graphics2D`'s `drawRenderedImage()` method to render the image. This means that there is no control over which tiles are computed for displaying the visible part of the image.

One way to speed up rendering is to convert the visible portion of the image to a buffered image and render it using `drawImage().` In this approach, the granularity of the tile is important. If the tiles are too small compared to the viewport, there will be too many tile computations, possibly degrading performance. On the other hand, too large a tile requires a large amount of memory because `BufferedImage` stores all the image data in memory. So designing a tile layout means making a trade-off between speed and memory.

To implement a canvas for rendering large images, let's create a subclass of the `com.vistech.jai.render.JAIImageCanvas` class and extend its functionality to implement tiling. We'll call this class `RenderedImageCanvas.`

As we saw in the section titled Using the Format Operator earlier in this chapter, a planar image can be restructured to implement any tile layout through the `ImageLayout` class and the Format operator. To paint a partial image, let's override the `paintComponent()` method and implement a tile computation loop based on the coordinates of the image that covers the viewport. These tiles can then be converted into a buffered image and painted immediately.

Listing 11.4 shows the `RenderedImageCanvas` class.

> **Note:** The code for `RenderedImageCanvas` is available on the book's Web page in the directory `src/chapter11/render.`

LISTING 11.4 The `RenderedImageCanvas` class

```
package com.vistech.jai.render;
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import java.io.*;
import javax.swing.*;
import java.awt.geom.*;
```

continued

```
import java.awt.image.renderable.*;
import javax.media.jai.*;

public class RenderedImageCanvas extends JAIImageCanvas {
   protected int viewerWidth = 480, viewerHeight = 400;

   transient protected PlanarImage displayImage, origImage;
   protected int tileWidth = 256, tileHeight = 256;
   transient protected SampleModel sampleModel;
   protected ColorModel colorModel;

   protected int maxTileIndexX, maxTileIndexY;
   protected int maxTileCordX, maxTileCordY;
   protected int minTileIndexX, minTileIndexY;
   protected int minTileCordX, minTileCordY;
   protected int tileGridXOffset, tileGridYOffset;
   protected int imageWidth =0, imageHeight =0;
   protected TileCache tc;

   public RenderedImageCanvas() { }
   public RenderedImageCanvas(PlanarImage img){
      this();
      setImage(img);
   }

   public void setImage(PlanarImage img){
      origImage = img;
      panX =0; panY =0;
      atx = AffineTransform.getTranslateInstance(0.0, 0.0);
      RenderedOp op = makeTiledImage(img);

      displayImage = op.createInstance();
      sampleModel = displayImage.getSampleModel();
      colorModel = displayImage.getColorModel();
      getTileInfo(displayImage);
      fireTilePropertyChange();
      imageDrawn = false;
      repaint();
   }

   public PlanarImage getDisplayImage(){return displayImage;}

   protected RenderedOp makeTiledImage(PlanarImage img) {
      ImageLayout tileLayout = new ImageLayout(img);
      tileLayout.setTileWidth(tileWidth);
      tileLayout.setTileHeight(tileHeight);
      RenderingHints tileHints = new RenderingHints(JAI.KEY_IMAGE_LAYOUT, tileLayout);
      ParameterBlock pb = new ParameterBlock();
      pb.addSource(img);
      return JAI.create("format", pb, tileHints);
   }

   protected void getTileInfo(PlanarImage img) {
      imageWidth = img.getWidth();
      imageHeight = img.getHeight();
      tileWidth = img.getTileWidth();
      tileHeight = img.getTileHeight();
```

```
        maxTileIndexX = img.getMinTileX()+img.getNumXTiles()-1;
        maxTileIndexY = img.getMinTileY()+img.getNumYTiles()-1;
        maxTileCordX = img.getMaxX();
        maxTileCordY = img.getMaxY();
        minTileIndexX = img.getMinTileX();
        minTileIndexY = img.getMinTileY();
        minTileCordX = img.getMinX();
        minTileCordY = img.getMinY();
        tileGridXOffset = img.getTileGridXOffset();
        tileGridYOffset = img.getTileGridYOffset();
    }

    public void setTileWidth(int tw){
        tileWidth = tw;
        setImage(displayImage);
    }

    public int getTileWidth(){ return tileWidth;}
    public void setTileHeight(int th){
        tileHeight = th;
        setImage(displayImage);
    }
    public int getTileHeight(){ return tileHeight;}

    public int getMaxTileIndexX(){return maxTileIndexX;}
    public int getMaxTileIndexY(){return maxTileIndexY;}
    public int getImageWidth(){return imageWidth;}
    public int getImageHeight(){return imageHeight;}


    protected void firePropertyChange() {
        firePropertyChange("maxTileIndexX", null, new Integer(maxTileIndexX));
        firePropertyChange("maxTileIndexY", null, new Integer(maxTileIndexY));
        firePropertyChange("tileWidth", null, new Integer(tileWidth));
        firePropertyChange("tileHeight", null, new Integer(tileWidth));
        firePropertyChange("transform", null, atx);
    }

    public void paintComponent(Graphics gc){
        Graphics2D g = (Graphics2D)gc;
        Rectangle rect = this.getBounds();
        if((viewerWidth != rect.width) || (viewerHeight != rect.height)){
            viewerWidth = rect.width;
            viewerHeight = rect.height;
        }
        g.setColor(Color.black);
        g.fillRect(0, 0, viewerWidth, viewerHeight);
        if(displayImage == null) return;
        int ti =0, tj = 0;
        Rectangle bounds = new Rectangle(0, 0, rect.width, rect.height);
        bounds.translate(-panX, -panY);

        int leftIndex =  displayImage.XToTileX(bounds.x);
        if(leftIndex <  minTileIndexX) leftIndex = minTileIndexX;
        if(leftIndex > maxTileIndexX) leftIndex = maxTileIndexX;

        int rightIndex = displayImage.XToTileX(bounds.x + bounds.width - 1);
```

continued

```
        if(rightIndex <  minTileIndexX) rightIndex = minTileIndexX;
        if(rightIndex > maxTileIndexX) rightIndex = maxTileIndexX;

        int topIndex = displayImage.YToTileY(bounds.y);
        if(topIndex <  minTileIndexY) topIndex = minTileIndexY;
        if(topIndex > maxTileIndexY) topIndex = maxTileIndexY;

        int bottomIndex = displayImage.YToTileY(bounds.y + bounds.height - 1);
        if(bottomIndex <  minTileIndexY) bottomIndex = minTileIndexY;
        if(bottomIndex > maxTileIndexY) bottomIndex = maxTileIndexY;

        for(tj = topIndex; tj <= bottomIndex; tj++) {
            for(ti = leftIndex; ti <= rightIndex; ti++) {
                Raster tile = displayImage.getTile(ti, tj);
                DataBuffer dataBuffer = tile.getDataBuffer();
                WritableRaster wr = tile.createWritableRaster(sampleModel,
                                                dataBuffer, new Point(0,0));
                BufferedImage bi = new BufferedImage(colorModel,
                                                wr,
                                                colorModel.isAlphaPremultiplied(),
                                                null);
                if(bi == null) continue;
                int xInTile = displayImage.tileXToX(ti);
                int yInTile = displayImage.tileYToY(tj);
                AffineTransform tx = AffineTransform.getTranslateInstance(xInTile+panX,
                                                                    yInTile+panY);
                g.drawRenderedImage(bi, tx);
            }
        }
        imageDrawn = true;
    }
}
```

The `RenderedImageCanvas` class inherits several instance variables and methods from `JAIImageCanvas` (see Listing 10.2). `RenderedImageCanvas` adds two more instance variables—`origImage` and `displayImage`—to hold the original image and the currently displayed image. The value of the `displayImage` variable is derived by reformatting of the image with the Format operator. These variables allow users to set the desired tile dimensions. The `paintComponent()` method draws the `displayImage` object over the graphical context.

The `RenderedImageCanvas` class has various properties for describing a tile, including `tileWidth, tileHeight, maxTileIndexX,` and `maxTileIndexY.` The `maxTileIndexX` and `maxTileIndexY` properties represent the number of tile columns and the number of tile rows, respectively. These two properties are needed to compute the positions of the required tiles.

`RenderedImageCanvas` overrides the `setImage()` and `paintComponent()` methods of its superclass, `JAIImageCanvas.` The `setImage()` method uses the current tile size attributes to reformat the input image. To accomplish this reformatting, `setImage()` calls `makeTiledImage(),` which returns a reformatted planar image. The `setImage()` method assigns this image to the `displayImage` variable. Then `setImage()` calls the

`getTileInfo()` method to set the tile-related attributes. If the image is smaller than the tile, the image size becomes the current tile size.

Tile Computation

Depending on the size of the viewport, the `paintComponent()` method obtains the tiles needed to display the image within the viewport. To do so, it uses the current `panX` and `panY` variables to compute the position of the tiles within the image. The size of the viewport determines how many tiles are required to display the part of the image that covers the viewport (see Figure 11.5).

A tile in a rendered image can be obtained by the `getTile(xIndex, yIndex)` method. This means that to obtain a tile from the image, we need its position. To get all the tiles that cover the viewport, we need the starting and ending indices of the tiles in both the *x* and the *y* directions. For a given point in the image, the `PlanarImage` methods `XtoTileX()` and `YtoTileY()` compute the tile indices in the *x* and *y* directions, respectively.

To compute the tile indices, the `paintComponent()` method first starts with the upper left-hand corner of the viewport, which is at position (`panX, panY`). The tile indices corresponding to this position are `leftIndex` and `topIndex.` To compute the ending index, the `paintComponent()` method uses the lower right-hand corner

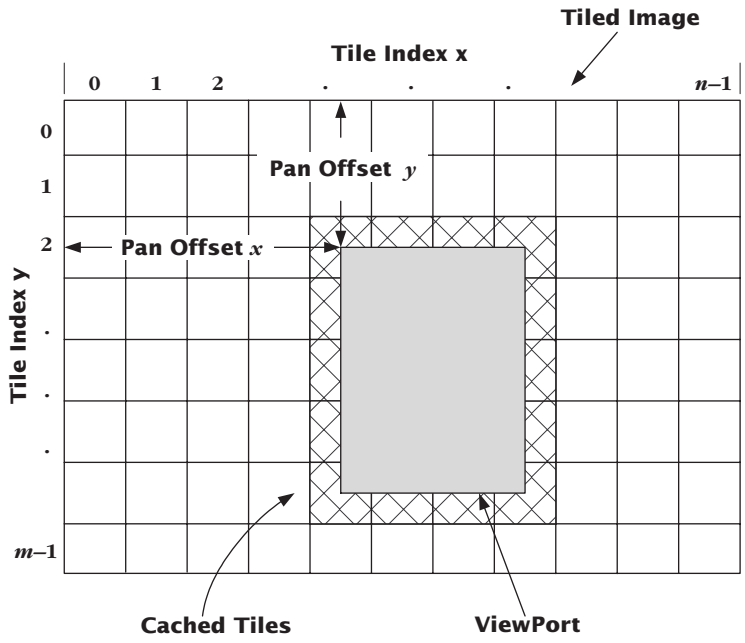

FIGURE 11.5 Viewing large images

coordinates of the viewport with respect to the image. These are given by (`panX + viewerWidth`) and (`panY + viewerHeight`), and the corresponding indices are `rightIndex` and `bottomIndex`, respectively.

Once the indices in both the *x* and the *y* directions have been computed, `paintComponent()` executes a loop that does the following:

1. Retrieves a tile from the planar image.
2. Converts that tile into a buffered image.
3. Obtains the position of the tile by calling the `tileToX()` and `tileToY()` methods. (The position of the tile in the image is needed to display this image.)
4. Constructs an `AffineTransform` object to handle the translation. (Each tile must be translated appropriately for the image to appear continuous.)
5. Draws the tile by calling the `drawImage()` method with the `BufferedImage` object produced in step 2.

A Tiled-Image Viewer

Using the `RenderedImageCanvas` class described in the previous section, let's create a viewer that is capable of displaying large images. Let's also add a panel called **Tile Grid** to show the viewport position and size with respect to the image (see Figure 11.6). We'll also add one more way to pan the image: dragging the rectangle

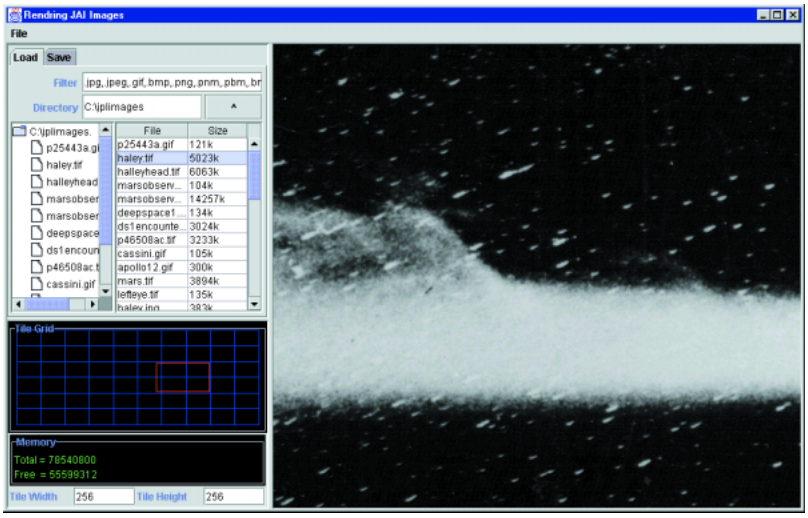

FIGURE 11.6 An image viewer for viewing large images

that represents the image to move the image within the viewport. To control the tile size, let's add two text fields for entering tile width and height.

In addition to the classes mentioned in the JAI planar-image viewer example (see Chapter 10), here are the classes used in this application:

- **`TiledImageViewer.`** This is the main class that launches the application. It is a subclass of `JAIImageViewer.`
- **`RenderedImageCanvas.`** This class displays the image. It is a subclass of `JAIImageCanvas` (see the section titled Viewing and Manipulating Large Images earlier in this chapter).
- **`RenderGrid.`** This class implements the **Tile Grid** panel. It displays the tile grid, as well as the viewport as a rectangle positioned over the tile grid. The position of this rectangle varies when you scroll the image. The position of the image changes when you drag this rectangle.

Now let's look at the code. We have already discussed the `RenderedImageCanvas` code, so let's look at the `RenderGrid` class. Even though this class uses no JAI API, we list it here because it draws heavily from the Java 2D graphical and rendering APIs. Listing 11.5 shows the code for `RenderGrid.`

LISTING 11.5 The `RenderGrid` class

```
package com.vistech.jai.render;
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import java.io.*;
import javax.swing.*;
import java.awt.geom.*;

public class RenderGrid extends JPanel {
   protected int maxTileIndexX, maxTileIndexY;
   protected int tileWidth, tileHeight;
   protected Point vpPos = new Point(0,0);
   protected Rectangle currentShape;
   protected int panX =0, panY =0;
   protected int width, height;
   protected boolean dragOn = false;
   private Point scrollAnchor  = new Point(0,0);
   protected AffineTransform atx = new AffineTransform();
   protected Dimension vpSize;

   public RenderGrid() {
      enableEvents(AWTEvent.MOUSE_MOTION_EVENT_MASK |
                           AWTEvent.MOUSE_EVENT_MASK);
   }

   public Dimension getViewportDimension(){
      return new Dimension(width, height);
   }
```

continued

```
public void setTileIndices(int maxXIndex, int maxYIndex){
   this.maxTileIndexX = maxXIndex;
   this.maxTileIndexY = maxYIndex;
   repaint();
}

public void setTileDimension(int width, int height){
   tileWidth = width; tileHeight = height;
   repaint();
}

public void setTileWidth(int tw){
   tileWidth = tw; repaint();
}
public int getTileWidth(){ return tileWidth;}

public void setTileHeight(int th){
   tileHeight = th; repaint();
}
public int getTileHeight(){ return tileHeight;}
public void setMaxTileIndexX(int tix){
   maxTileIndexX = tix; repaint();
}

public int getMaxTileIndexX(){return maxTileIndexX;}

public void setMaxTileIndexY(int tiy){
   maxTileIndexY = tiy; repaint();
}
public int getMaxTileIndexY(){return maxTileIndexY;}
public void setTransform(AffineTransform at){atx = at;}
public void setViewportSize(Dimension size){vpSize = size;}

public void  processMouseEvent(MouseEvent e){
   switch(e.getID()){
       case MouseEvent.MOUSE_PRESSED:
          setCursor(new Cursor(Cursor.MOVE_CURSOR));
          int x = e.getX();
          int y = e.getY();
          if(currentShape.contains(x,y)) {  dragOn = true;}
          break;
       case MouseEvent.MOUSE_CLICKED:
          break;
       case MouseEvent.MOUSE_RELEASED:
          setCursor(Cursor.getDefaultCursor());
          dragOn = false;
          break;
   }
}

public void processMouseMotionEvent(MouseEvent e){
    switch(e.getID()){
       case MouseEvent.MOUSE_DRAGGED:
          if(dragOn) scroll(e.getX(), e.getY());
          break;
   }
}
```

```
public void stopScroll(){ setCursor(Cursor.getDefaultCursor());}

public void startScroll(int x, int y){
   if(dragOn) return;
   scrollAnchor.x = x- panX;
   scrollAnchor.y = y- panY;
   repaint();
}

public void scroll(int x, int y){
   panX = x-scrollAnchor.x;
   panY = y-scrollAnchor.y;
   setPan(new Point(panX, panY));
   repaint();
}

public void setPan(Point pos){
   firePropertyChange("viewportPosition",this.vpPos, pos);
   int x = (int)( pos.x * (maxTileIndexX*tileWidth)/(width-20));
   int y = (int)( pos.y * (maxTileIndexY*tileHeight)/(height-20));
   this.vpPos = new Point(x,y);
   repaint();
}

public void setViewportPosition(Point vpPos){
   this.vpPos = vpPos; repaint();
}

public void paintComponent(Graphics gc) {
   Graphics2D g = (Graphics2D)gc;
   Rectangle bounds = this.getBounds();
   g.setColor(Color.black);
   width = bounds.width-20;
   height = bounds.height-20;
   double gridWidth =  (width)/(double)maxTileIndexX;
   double gridHeight = (height)/(double)maxTileIndexY;
   g.fillRect(0,0,bounds.width, bounds.height);
   if ((maxTileIndexX == 0)|| (maxTileIndexY == 0))  return;
   g.setColor(Color.blue);
   double vertStartX = 10.0; double vertStartY = 10.0;
   double vertEndX = 10.0;
   double vertEndY = (double)maxTileIndexY*gridHeight+vertStartY;
   // Horizontal lines
   for(int i= 0; i<maxTileIndexX; i++) {
       g.drawLine((int)vertStartX, (int)vertStartY, (int)vertEndX, (int)vertEndY);
       vertStartX += gridWidth;
       vertEndX   += gridWidth;
   }
   g.drawLine((int)vertStartX, (int)vertStartY,
              (int)vertEndX, (int)vertEndY);
   double horizStartX = 10.0; double  horizStartY = 10.0;
   double horizEndX = maxTileIndexX*gridWidth+horizStartX;
   double horizEndY = 10.0;
   // Horizontal lines
   for(int i=0; i<maxTileIndexY; i++) {
       g.drawLine((int)horizStartX, (int)horizStartY,
                  (int)horizEndX, (int)horizEndY);
```

continued

```
            horizEndY += gridHeight;
            horizStartY   += gridHeight;
        }
        g.drawLine((int)horizStartX, (int)horizStartY,
                   (int)horizEndX, (int)horizEndY);
        g.setColor(Color.red);
        vertStartX = 10.0; vertStartY = 10.0;
        int vpWid, vpHt;
        if(vpSize == null){ vpWid = (int)gridWidth; vpHt = (int)gridHeight;}
        else {
           vpWid = vpSize.width; vpHt = vpSize.height;
        }

        if(dragOn) {
           currentShape = new Rectangle(-(int)vertStartX+panX,
                                        -(int)vertStartY+panY,vpWid, vpHt);
         }else {
           int x = (int)(vertStartX +
                   vpPos.x * ((width)/(double)(maxTileIndexX*tileWidth)));
           int y = (int)(vertStartY +
                   vpPos.y * ((height)/(double)(maxTileIndexY*tileHeight)));
           currentShape = new Rectangle(x,y,vpWid,vpHt);
        }

        g.draw(currentShape);
     }
}
```

The `RenderGrid` class handles `mouse` and `mouseMotion` events internally. When the mouse is pressed, the `processMouseEvent()` method checks whether the position of the cursor is within the rectangle. If so, it sets the `dragOn` variable to `true.` When the mouse is dragged, the `processMouseMotion()` method moves the rectangle. This method also calls the `setPan()` method, which fires the `propertyChange` event for the `viewportPosition` property.

The use of `propertyChange` events eliminates the need to call `RenderedImage Canvas` directly. An interposing object can receive this event and invoke appropriate methods in `RenderedImageCanvas.` In this case, the interposing object is the `Tiled ImageViewer` object, which upon receipt of this event, calls the `setViewport Position()` method of `RenderedImageCanvas.`

Likewise, when the mouse is dragged over the image, `RenderedImageCanvas` fires the `propertyChange` event for its `viewportPosition` property. Again this event is captured by the `TiledImageViewer` object, which in turn invokes `RenderGrid`'s `setViewportPosition()` method. Thus, by using `propertyChange` events, you can avoid circular dependency between `RenderGrid` and `RenderedImageCanvas.` Moreover, both `RenderedImageCanvas` and `RenderGrid` can be used as beans and can be visually connected by binding properties (see Appendix B).

The `paintComponent()` method draws first the grid and then the rectangle representing the viewport. If the value of `dragOn` is `true,` which means the user is moving the rectangle, the `panX` and `panY` variables are used for moving the rectangle. If it is

false, vpPos is used for moving the rectangle. The vpPos variable is set by the setViewportPosition() method, which is called by TiledImageViewer whenever it receives a propertyChange event for the viewportPosition property. The TiledImageViewer object also sets the tile dimensions, on the basis of which RenderGrid draws the grid. Listing 11.6 shows the code for TiledImageViewer.

LISTING 11.6 The TiledImageViewer class: Part 1 (continued in Listing 11.7)

```
public class TiledImageViewer extends JAIImageViewer{
   protected RenderedImageCanvas viewer;
   protected RenderGrid renderGrid;
   protected JTextField twidth, theight;
   protected JTextArea memoryMessageBar;

   public void createUI() {
      Dimension dim = Toolkit.getDefaultToolkit().getScreenSize();
      int width = (int)(dim.width *3/4.0);
      int height = (int)(dim.height*3/4.0);

      setTitle("Rendering JAI Images");
      viewer = new RenderedImageCanvas();
      Dimension d = getViewerSize(width/(double)height);
      viewer.setPreferredSize(new Dimension(d.width, d.height));

      createImageLoaderAndSaver();
      loader.addPlanarImageLoadedListener(
          new PlanarImageLoadedListener() {
             public void imageLoaded(PlanarImageLoadedEvent e) {
                PlanarImage image = e.getImage();
                if(image == null) return;
                setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
                SwingUtilities.invokeLater(new ImagePaint(image));
             }
          }
      );

      twidth = new JTextField(5);
      theight = new JTextField(5);
      renderGrid = new RenderGrid();
      renderGrid.setBorder(BorderFactory.createTiledBorder("Tile Grid"));
      theight.setText(Integer.toString(viewer.getTileHeight()));
      twidth.setText(Integer.toString(viewer.getTileWidth()));
           //...  Gridbag layout code not shown.

   }

   protected void addPropertyChangeEventAdapters(){
        viewer.addPropertyChangeListener(
          new PropertyChangeListener() {
             public void propertyChange(PropertyChangeEvent e) {
                if(e.getPropertyName().equals("viewportPosition")){
                   twidth.setText(Integer.toString(viewer.getTileWidth()));
                   theight.setText(Integer.toString(viewer.getTileHeight()));
                   if(renderGrid != null) updateRenderGrid();
                   Point p =(Point)e.getNewValue();
```

continued

```
                renderGrid.setViewportPosition(p);
                }
                if(e.getPropertyName().equals("maxTileIndexX")){
                   renderGrid.setMaxTileIndexX(((Integer)e.getNewValue()).
                   intValue());
                }
                if(e.getPropertyName().equals("maxTileIndexY")){
                   renderGrid.setMaxTileIndexY(((Integer)e.getNewValue()).
                   intValue());
                }
                if(e.getPropertyName().equals("tileWidth")){
                   renderGrid.setTileWidth(((Integer)e.getNewValue()).intValue());
                }
               if(e.getPropertyName().equals("tileHeight")){
                   renderGrid.setTileHeight(((Integer)e.getNewValue()).intValue());
               }
               if(e.getPropertyName().equals("transform")){
                  int wid = renderGrid.getViewportDimension().width;
                  int ht = renderGrid.getViewportDimension().height;
                  double scaleX = wid/(double)viewer.getSize().width;
                  double scaleY = ht/(double)viewer.getSize().height;
                  AffineTransform atx =  (AffineTransform)e.getNewValue();
                  renderGrid.setTransform(atx);
               }
            }
         }
      );

      renderGrid.addPropertyChangeListener(
          new PropertyChangeListener() {
             public void propertyChange(PropertyChangeEvent e) {
                if(e.getPropertyName().equals("viewportPosition")){
                   double scaleX, scaleY;
                   int wid = renderGrid.getViewportDimension().width;
                   int ht = renderGrid.getViewportDimension().height;
                   scaleX = viewer.getImageWidth()/wid;
                   scaleY = viewer.getImageHeight()/ht;
                   if(e.getNewValue() instanceof Point){
                      Point p =(Point)e.getNewValue();
                      viewer.pan(-p.x*scaleX, -p.y*scaleY);
                   }
                }

            }
         }
      );
   }

   protected void updateRenderGrid() {
      if(viewer == null) return;
      int tw1 = viewer.getTileWidth();
      int th1 = viewer.getTileHeight();
      int xInd1 = viewer.getMaxTileIndexX();
      int yInd1 = viewer.getMaxTileIndexY();
      renderGrid.setTileDimension(tw1,th1);
      renderGrid.setTileIndices(xInd1, yInd1);
      int wid = renderGrid.getViewportDimension().width;
```

```
        int ht = renderGrid.getViewportDimension().height;
        double scaleX = viewer.getSize().width/(double)viewer.getImageWidth();
        double scaleY = viewer.getSize().height/(double)viewer.getImageHeight();
        double scaledVpX = wid*scaleX;
        double scaledVpY = ht*scaleY;
        renderGrid.setViewportSize(new Dimension((int)scaledVpX,(int)scaledVpY));
        renderGrid.repaint();
    }
}
```

`TiledImageViewer` is the main application class that launches the application. To run the application, type "java app.TiledImageViewer" or "java app.TiledImageViewer <image path>" on the command line.

The `TiledImageViewer` class extends `JAIImageViewer` and overrides the `createUI()` method. As Figure 11.6 shows, the left-hand panel of the window that represents this application has the **Tile Grid** pane and two text fields, for setting the tile width and height.

`TiledImageViewer` acts as an event adapter between `RenderGrid` and `RenderedImageCanvas` objects. When the GUI is created, `TiledImageViewer` registers to receive the `propertyChange` events from both `RenderGrid` and `RenderedImageCanvas` objects. The call to the `addPropertyChangeEventAdapters()` method in Listing 11.6 shows that this registration is done through anonymous inner classes.

When a user drags the mouse over the image, `RenderImageCanvas` fires a `propertyChange` event for the `viewportPosition` property. When `TiledImageViewer` receives this event, it calls the `setViewportPosition()` method in the `RenderGrid` object. Likewise, when a user moves the rectangle in the Tile Grid pane, the `TiledImageViewer` object receives the `propertyChange` event for the `viewportPosition` property. In this case `TiledImageViewer` calls the `pan()` method in `RenderedImageCanvas.` Figure 11.7 shows how `TiledImageViewer` facilitates communication between `RenderGrid` and `RenderedImageCanvas` via `propertyChange` events.

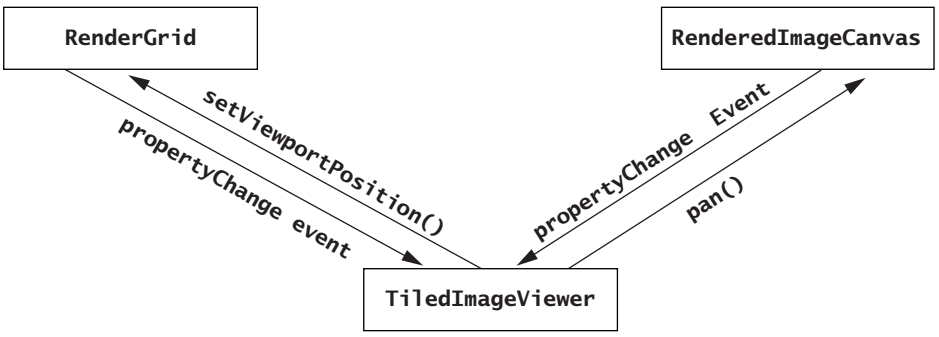

FIGURE 11.7 The `propertyChange` event flow

`TiledImageViewer` acts as an event adapter for other properties as well, including `tileWidth`, `tileHeight`, `maxTileIndexX`, and `maxTileIndexY`.

Setting the Tile Parameters

You can change the tile layout by entering the tile width and height in the text fields located at the bottom of the left-hand pane. This pane also has a memory status bar that shows the total available and free memory. The memory status is updated whenever the mouse is clicked on the image. Note that the memory readings may not always be accurate. Listing 11.7 shows the code that handles events from the text fields and the code that updates the memory status bar.

LISTING 11.7 The `TiledImageViewer` class: Part 2 (continued from Listing 11.6)

```
public JPanel createTileSetPanel(){
   memoryMessageBar = createMemoryMessageBar();
   memoryMessageBar.setBorder(BorderFactory.createTiledBorder("Memory"));
   //..Gridbag layout code

   return pan;
}

protected void addTileParamsEventAdapters(){
   twidth.addActionListener(
      new ActionListener(){
         public void actionPerformed(ActionEvent e) {
            try {
               String str = ((JTextField) e.getSource()).getText();
               int wid = Integer.parseInt(str);
               viewer.setTileWidth(wid);
               str = theight.getText();
               int ht = Integer.parseInt(str);
               viewer.setTileHeight(wid);
               if(renderGrid != null) updateRenderGrid();
            } catch (Exception e2){}
          }
       } );
   theight.addActionListener(
      new ActionListener(){
         public void actionPerformed(ActionEvent e) {
            try {
               String str = ((JTextField) e.getSource()).getText();
               int ht = Integer.parseInt(str);
               viewer.setTileHeight(ht);
               str = twidth.getText();
               int wid = Integer.parseInt(str);
               viewer.setTileWidth(wid);
               if(renderGrid != null) updateRenderGrid();
            } catch (Exception e1){}
          }
       });

   viewer.addMouseListener(
```

```
        new MouseAdapter() {
          public void mouseReleased(MouseEvent e){
             updateMemoryMessageBar(memoryMessageBar);
          }
        });
    }

    protected JTextArea createMemoryMessageBar(){
       JTextArea messageBar = new JTextArea();
       messageBar.setBackground(Color.black);
       messageBar.setForeground(Color.green);
       updateMemoryMessageBar(messageBar);
       return messageBar;
    }

    protected void updateMemoryMessageBar(JTextArea mbar) {
       Runtime rt = Runtime.getRuntime();
       long totalMemory = rt.totalMemory();
       rt.gc();
       long freemem = rt.freeMemory();
       mbar.setText("Total = "+ totalMemory+  "\n" +
                    "Free  = " + freemem);
    }
```

The `updateMemoryMessageBar()` method calls some utility methods in the `java.lang.Runtime` class to obtain the amount of total available memory and free memory. It also calls the `System.gc()` method to run the garbage collector. As we learned in Chapter 10, however, calling this method does not guarantee that the garbage collector will run. The `updateMemoryMessageBar()` method posts the amount of free memory to the memory message bar.

Painting the Image in a Thread

The `TiledImageViewer` class also overrides the `ImagePaint` class, which is shown in Listing 11.8. Running the image-painting routine on a thread is essential for painting large images.

LISTING 11.8 The `ImagePaint` class

```
class ImagePaint implements Runnable {
      PlanarImage image;
      boolean firstTime = true;
      public ImagePaint(PlanarImage image){this.image = image;}
      public void run() {
         if(firstTime) {
            try {
             firstTime = false;
             setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
             viewer.setImage(image);
             int wid = image.getWidth();
             int height = image.getHeight();
             saver.setDisplayImage(viewer.getDisplayImage());
```

continued

```
                viewer.repaint();
                SwingUtilities.invokeLater(this);
               } catch(Exception e){SwingUtilities.invokeLater(this);}
            }
            else {
               if(!viewer.isImageDrawn()) SwingUtilities.invokeLater(this);
               else{
                  setCursor(Cursor.getDefaultCursor());
                  updateRenderGrid();
                  updateMemoryMessageBar(memoryMessageBar);
              }
            }
         }
      }
```

The `ImagePaint` thread is created by an anonymous class that retrieves the image from the `PlanarImageLoaded` event object (see the `createUI()` method in Listing 11.6). In the first pass, the `run()` method calls `RenderedImageCanvas`'s `setImage()` method, which calls the `repaint()` method to start the image painting.

The `SwingUtilities.invokeLater()` method runs the `ImagePaint` thread repeatedly until `isImageDrawn()` returns `true.` When that happens, the `run()` method calls `updateRenderGrid(),` which gets the latest tile settings from the viewer and assigns them to `renderGrid.` It also updates the memory message bar and changes the hourglass cursor to the default cursor.

Writing to Pixels

In our examples so far, we have used only planar images. As stated earlier, `PlanarImage` is read-only. To write pixels, you need to use the `TiledImage` class, which extends `PlanarImage` and implements `WritableRenderedImage.` As we saw in Chapter 10, the `WritableRenderedImage` interface handles the writing of pixel data to tiles.

To write pixels, you need to obtain a `WritableRaster` object for the desired rectangular region. You cannot write to this instance of `WritableRaster` immediately because unlike `BufferedImage, TiledImage` does not have all its data resident in the memory. So you need to get all the tiles that cover the rectangular region. Recall from Chapter 10 that the `WritableRenderedImage` interface specifies methods for checking out, writing, and releasing tiles. The `TiledImage` class implements those methods.

Constructing a Tiled Image

First let's see how to construct a `TiledImage` object. The `TiledImage` class has three public constructors:

1. **`public TiledImage(int minX,`**

```
                    int minY,
                    int width,
                    int height,
                    int tileGridXOffset,
                    int tileGridYOffset,
                    java.awt.image.SampleModel sampleModel,
                    java.awt.image.ColorModel colorModel)
2. public TiledImage(Point origin, SampleModel sampleModel,
                    int tileWidth,
                    int tileHeight)
3. public TiledImage(SampleModel sampleModel,
                    int tileWidth,
                    int tileHeight)
```

These constructors take attributes of the tiled image as input parameters. For a detailed explanation of the parameters, see Tables 10.1 and 10.2. When parameters are not provided, default values are assumed.

Notice that the `TiledImage` constructors don't have an argument for data. These methods construct a skeleton image. You can set the data later, either by calling the `setData()` method or by drawing over it using its graphical context. In this respect, a tiled image is just like a buffered image.

One way to create a tiled image is to start from scratch—that is, create an instance of `SampleModel` and of `ColorModel` and pass them to the `TiledImage` constructor along with other parameters. You can also use the factory methods that will be described shortly to create a tiled image.

More often, however, it is necessary to construct a tiled image from a planar image. To do so, use the existing `PlanarImage` object to get the attributes and then copy the `PlanarImage` data into the `TiledImage` object.

Here's an example of code that creates a tiled image from a planar image:

```
public static TiledImage createDisplayImage(PlanarImage image){
   SampleModel sampleModel = image.getSampleModel();
   ColorModel colorModel = image.getColorModel();

   TiledImage ti = new TiledImage(image.getMinX(), image.getMinY(),
                                  image.getWidth(), image.getHeight(),
                                  image.getTileGridXOffset(),
                                  image.getTileGridYOffset(),
                                  sampleModel, colorModel);
   ti.setData(image.copyData());
   return ti;
}
```

In addition to its constructors, the `TiledImage` class has two factory methods for creating an instance of `TiledImage`:

```
1. public static TiledImage createBanded(int minX,
```

continued

```
                                                int minY,
                                                int width,
                                                int height,
                                                int dataType,
                                                int tileWidth,
                                                int tileHeight,
                                                int[] bankIndices,
                                                int[] bandOffsets)
2. public static TiledImage createInterleaved(int minX,
                                                int minY,
                                                int width,
                                                int height,
                                                int numBands,
                                                int dataType,
                                                int tileWidth,
                                                int tileHeight,
                                                int[] bandOffsets)
```

The first static method creates an instance of `TiledImage` with the band-interleaved sample model, and the second method creates the pixel-interleaved sample model. See Chapter 8 for a detailed explanation of these sample models.

The `dataType` argument indicates the data type. The Java 2D API supports only `int, byte, short,` and unsigned `short` types. The JAI APIs support `float` and `double` types in addition to the basic types. The `DataBuffer` class, however, can take any constant representing any type.

As stated earlier, the constructors don't have image data as one of the input parameters. To set the data, the `TiledImage` class provides different flavors of *setXXX()* methods. Using these methods, you can set an individual pixel or pixels covering a rectangular region.

Setting the Pixel Value

The `TiledImage` class does not have a `setPixel()` method per se. However, it has three flavors of `setSample(),` which set the value of a sample. Recall from Chapter 8 that a sample is a component of a pixel; that is, a pixel in an RGB image has three samples—one each for red, green, and blue. Unlike a Java 2D image, a JAI image sample can assume `float` or `double` values. Accordingly, the `setSample()` method has the following three flavors for setting `int, float,` and `double` values:

```
1. public void setSample(int x, int y, int b, int s)
2. public void setSample(int x, int y, int b, float s)
3. public void setSample(int x, int y, int b, double s)
```

The `x` and `y` parameters represent the coordinates of a pixel, `b` represents the band index, and `s` represents the sample value. Corresponding to the `setSample()` methods, three methods of the `TiledImage` class are available for retrieving an individual sample value:

1. **`public int getSample(int x, int y, int b)`**
2. **`public float getSampleFloat(int x, int y, int b)`**
3. **`public double getSampleDouble(int x, int y, int b)`**

There is another way to set a pixel value: by using the `WritableRenderedImage` methods. We explained these methods in Chapter 8. To write a pixel, first get the index of the tile in which the pixel is located, then check out that tile for writing. Here's an example:

```
public static void writePixel(int x, int y,  short[] pixelValue TiledImage image) {
   Int xIndex = image.XtoTileX(x);
   int yIndex = image.YtoTileY(y);
   WritableRaster  tileRaster  =  image.getWritableTile (xIndex, yIndex);
   If (tileRaster != null) tileRaster.setPixel(x,y, pixelValue);
   ReleaseWritableTile(xIndex, yIndex);
}
```

Setting a Rectangular Region

There are two ways to write to a rectangular region of an image:

1. **Writing tile by tile.** You can use the `WritableRenderedImage` interface methods to write to a rectangular region.

2. **Writing to a rectangular region directly.**

Here are the methods:

- **`public void set(java.awt.image.RenderedImage im)`**
- **`public void set(java.awt.image.RenderedImage im, ROI roi)`**

Both of these methods copy the specified `RenderedImage` data onto a `TiledImage` object. The sample model of the source image should be compatible with that of the target image. The position and size of the overlapping area are determined from the source image attributes. If the source image area does not overlap the target image, these methods have no effect.

In the case of the latter method, which has an `ROI` parameter, the ROI overlaps the target image. The overlapping area and its position are computed from the corresponding ROI attributes.

The following two methods use the `Raster` parameter to pass the image data:

1. **`public void setData(java.awt.image.Raster r)`**
2. **`public void setData(java.awt.image.Raster r,ROI roi)`**

The `Raster` data provided by the input parameter is written over the target image. The overlapping area and position are computed from the coordinates of the input raster. These methods use the `WritableRenderedImage` methods to write to tiles. When the data is being written, the affected tiles are checked out and locked.

Creating a Subimage

Just as in `BufferedImage`, the `TiledImage` methods create a subimage from a tiled image:

- `public TiledImage getSubImage(int x, int y, int w, int h)`
- `public TiledImage getSubImage(int[] bandSelect)`
- `public TiledImage getSubImage(int x, int y, int w, int h, int[] bandSelect)`

The subimage follows the same coordinate system as the source image. The `x` and `y` parameters represent the upper left-hand corner coordinates of the subimage. The `w` and `h` parameters represent the width and height of the subimage, respectively. The `bandSelect` parameter is an array of indices representing the band indices in the source images. Using this parameter, you can create a subimage containing only the selected bands. The `subimage()` methods are useful in creating cropped images.

Obtaining an Off-Screen Graphical Context

Again like `BufferedImage`, the `TiledImage` class has methods for drawing graphical objects over it. The method signatures are the same as for the `BufferedImage` methods:

- `public java.awt.Graphics getGraphics()`
- `public java.awt.Graphics2D createGraphics()`

The first method returns the AWT graphical context, and the second returns the Java 2D graphical context.

Creating an Aggregate Image

Often it is necessary to create a single image from a set of images. You can use the `TiledImage` class to do this. As mentioned earlier, you can populate a `TiledImage` object in two different ways:

1. Drawing over its graphical context
2. Writing to tiles

Let's explore the first approach. This approach may not be very accurate, but it serves the purpose in most cases. The idea here is to first construct a skeleton tiled image as big as the target image, and then draw images from the set one by one, each time translating and scaling the image appropriately. Listing 11.9 shows the relevant code.

> **Note:** The code for aggregating images is available on the book's Web page in the directory `src/jaiutil/JAIUtil.java`.

LISTING 11.9 A method that aggregates images

```
public static TiledImage aggregateImages(PlanarImage[] imageset,
                                         Dimension celldim,
                                         int cols, int rows) {
   int imWid = cols*celldim.width;
   int imHt =  rows*celldim.height;
   PlanarImage img = imageset[0];
   TiledImage ti = new TiledImage(0,0,(int)imWid,(int)(imHt), 0,0,
                                  img.getSampleModel(), img.getColorModel());

   Graphics2D tg2d = ti.createGraphics();
   int x = 0; int y=0;
   for(int i=0; i< rows; i++) {
       for(int j=0;j<cols; j++) {
           if(i*cols+j >= imageset.length) break;
           img = imageset[i*cols+j];
           if(img != null){
              double magx = celldim.width/(double)img.getWidth() ;
              double magy = celldim.height/(double)img.getHeight() ;
              AffineTransform atx = new AffineTransform();
              atx.setToTranslation((double)x,(double)y);
              atx.scale(magx, magy);
              tg2d.drawRenderedImage(img,atx);
              x += celldim.width;
           }
       }
       x =0;
       y += celldim.height;
    }
   return ti;
}
```

The `aggregateImages()` method takes an array of planar images and converts it into a single planar image. It also has parameters to determine the size and layout of the target image. The `cellDim` parameter represents the size of an image to be drawn, which is assumed to be the same for all images. The `rows` and `cols` parameters specify how the input images are to be laid out—that is, in a rows-by-columns grid. It is expected that `rows` × `cols` = `imageset.length`.

This method assumes that all the images in `imageset` have the same sample model and color model. So it gets these parameters from the first image and then creates a skeleton tiled image. It then gets a `Graphics2D` context from this `TiledImage` object.

The next step is to draw the images from the `imageset` array one by one. There are two `for` loops, which create a single image by drawing the input images in a grid. Each time, the image is translated and then scaled so that the images fit closely. If you need a gap between the images, add the border to each image before passing to this method (see the section titled Using the Border Operator earlier in this chapter).

A JAI Image Browser

The Java browser application in Chapter 6 displays multiple images in a single frame and saves them as a single image. Let's modify that application to browse and save JAI images. We'll use the code in Listing 11.9 to aggregate multiple images into a single image.

There are two main classes: `JAIImageBrowser` and `ViewerPanel.` The `JAIImageBrowser` class code is similar to the `ImageBrowser` class code (see the section titled Image Browser: An Example of Aggregating Multiple Images in Chapter 6). `JAIImageBrowser` embeds the modified `MultiImageLoader` bean that can read image formats supported by JAI. When users select a set of images, `JAIImageBrowser` fires a `planarImageLoaded` event to the registered listeners. `JAIImageBrowser` receives this event, extracts the image set, and passes it on to `ViewerPanel.` Listing 11.10 shows the save methods of the modified `ViewerPanel` class.

> **Note:** The source code for the JAI image browser application is available on the book's Web page in the directory `src/chapter11/imagebrowser.`

LISTING 11.10 The `ViewerPanel` class

```
protected void save(){
   try{
     PlanarImage img = (PlanarImage) JAIUtil.aggregateImages(imageset,
                                     new Dimension(cellWidth, cellHeight),
                                     cols,rows);
     ImageSaverJAI.saveAsJPEG(img, "aggr.jpg");
   } catch(Exception e){}
}
```

For a detailed explanation of the `ViewerPanel` class, see the section titled Image Browser: An Example of Aggregating Multiple Images in Chapter 6. Figure 11.8 shows

a screen shot of a frame with multiple images launched by the `JAIImageBrowser` application. Each image in this frame is drawn on an `ImagePanel` object (see Listing 11.11).

LISTING 11.11 Displaying an image with a border

```
class ImagePanel extends JComponent {
   PlanarImage image;
   AffineTransform atx = new AffineTransform();
   int width, height;
   Dimension border = new Dimension(2,2);
   public ImagePanel(){ }
   public ImagePanel(PlanarImage img){
      RenderedOp op = JAIUtil.setConstantBorder(img,border,255);
      image = op.createInstance();
      width = image.getWidth();
      height = image.getHeight();
   }
   public void setImage(PlanarImage img){image = img;}

   public void paintComponent(Graphics gc){
      Graphics2D g = (Graphics2D)gc;
      Rectangle rect = this.getBounds();
      if((width != rect.width) || (height != rect.height)){
          double magx = rect.width/(double)width ;
```

continued

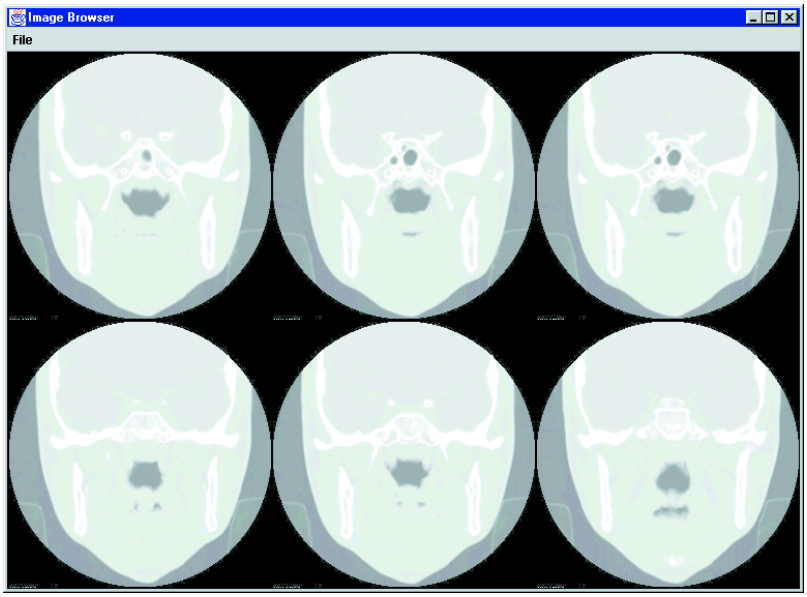

FIGURE 11.8 The image browser

```
            double magy = rect.height/(double)height ;
            atx.setToTranslation(border.width, border.height);
            atx.scale(magx, magy);
        }
        if(image != null) g.drawRenderedImage(image, atx);
    }
}
```

The `ImagePanel` class is a lightweight component for displaying images. The `JAIImageBrowser` object creates an `ImagePanel` object for each image.

When images are displayed side by side, a border is necessary to separate them. The `ImagePanel` constructor calls the method in Listing 11.11 to add the border to the input image.

The `paintComponent()` method translates the image so that the image border is visible on the left-hand side. It scales the image to fit the viewport and then draws it onto the graphical context.

The Renderable Layer

We already described the `RenderableImage` interface in Chapter 10. Many JAI standard operators take `RenderableImage` as an input source. The `RenderableOp` class is the equivalent of `RenderedOp` for a renderable layer. In other words, `RenderableOp` represents a node in the renderable chain.

A rendering chain has a source and a sink. The first step in building a renderable chain is to create a renderable source—that is, a renderable image.

Even though the renderable image has no size and lacks many other physical attributes, a real image is needed as a source. To construct a renderable chain, a renderable image is derived from this image.

An image in Java can be loaded as an `Image, BufferedImage,` or `PlanarImage` object. There is no equivalent of `PlanarImage` for the renderable chain. Instead, JAI provides an operator named Renderable, which converts `RenderedImage` to `RenderableImage.` Table 11.5 shows the parameters for this operator.

Obviously the Renderable operator does not take the renderable image as a source. This operator creates a pyramid of low-resolution images. But normally you would want to create a single image. Listing 11.12 shows an example that uses the Renderable operator.

LISTING 11.12 Converting to a renderable node

```
public static RenderableOp convertToRenderable(PlanarImage image) {
   ParameterBlock pb = new ParameterBlock();
   pb.addSource(image);
   pb.add(null).add(null).add(null).add(null).add(null);
   return  JAI.createRenderable("renderable", pb);
}
```

TABLE 11.5 Renderable Operator Parameters

OPERATOR NAME	PARAMETER	TYPE	DEFAULT VALUE	DESCRIPTION
Renderable	downSampler	RenderedOp	null	Rendered chain that supplies low-resolution images
	maxLowResDim	float	64	
	minX	float	0.0f	
	minY	float	0.0f	
	height	int	1.0f	

To apply standard operators to a renderable image, use the `JAI.createRenderable()` method, which returns a `RenderableOp` node.

Creating a Renderable Node

In the renderable chain, the `RenderableOp` class represents a renderable node. The `RenderableOp` class implements the `RenderableImage` interface. Recall from Chapter 10 that the `RenderableImage` interface has three different methods for creating a rendering: `createDefaultRendering()`, `createScaledRendering()`, and `createRendering()`. Each one returns a `RenderedImage` object.

To evaluate a renderable node, call one of the three rendering methods. While these methods are being executed, the size is assigned to the image. The `RenderedImage` object returned by these methods can be rendered through `Graphics2D`'s `drawRenderedImage()` method. This means that the same components we developed for displaying planar images can be used for rendering a renderable node.

However, the `Graphics2D` class has a method that directly renders a renderable image:

- `public abstract void drawRenderableImage(RenderableImage img, AffineTransform xform)`

Let's use this method to display a renderable image. As stated earlier, a renderable image has no size. This means that it has no width or height properties. However, a renderable image does have an aspect ratio. Because the renderable image has no size, the image viewer will need to provide the input. The options are

- **To fit the viewport.** Regardless of the size of a viewport, the size of the displayed image is increased or decreased to fit the viewport.
- **Scaled.** The size of the image fits the viewport, but the aspect ratio is maintained.
- **Original size.** The size matches the size of the source image from which the renderable image is generated.

Let's first build a canvas to display the renderable image. Listing 11.13 shows the class (`RenderableImageCanvas`) that can do this. It is similar to `JAIImageCanvas`, except for the `paintComponent()` method.

> **Note:** The source code for `RenderableImageCanvas` is available on the book's Web page in the directory `src/chapter11/simplerenderable`.

LISTING 11.13 The `RenderableImageCanvas` class

```
package com.vistech.jai.render;
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import javax.swing.*;
import java.awt.geom.*;
import java.awt.image.renderable.*;
import javax.media.jai.*;
import com.vistech.imageviewer.*;
public class RenderableImageCanvas extends JComponent implements ScrollController {
   public final static int TO_FIT =0;
   public final static int SCALED = 1;
   public final static int MAX_SIZE = 4;

   protected RenderableImage sourceImage;
   protected AffineTransform atx = new AffineTransform();
   protected boolean imageDrawn = false;
   protected int panX =0, panY =0;
   private Point scrollAnchor  = new Point(0,0);
   protected boolean scrollOn = true;
   protected int viewerWidth = 480, viewerHeight = 400;
   protected Point panOffset = new Point(0,0);
   protected int displayMode = SCALED;
   protected  int interpolationMode = Interpolation.INTERP_BILINEAR;
   protected float sourceImageHeight = 1.0f;
   protected float sourceImageWidth = 1.0f;
   protected int maxHeight =1024, maxWidth = 1024;

   public RenderableImageCanvas() {}

   public RenderableImageCanvas(RenderableImage img){
      atx = new AffineTransform();
      setImage(img);
   }

   public boolean isImageDrawn(){ return imageDrawn;}

   public void setDisplayMode(int dispMode) {
      displayMode = dispMode;
      createScalingTransform();
   }
```

```
public int getDisplayMode(){ return displayMode;}

public void setOrigImageSize(Dimension size){
   maxWidth = size.width;
   maxHeight = size.height;
}

public void setImage(RenderableImage rImg){
   sourceImage =  rImg;
   panX =0; panY =0;
   sourceImageHeight = sourceImage.getHeight();
   sourceImageWidth = sourceImage.getWidth();
   createScalingTransform();
   imageDrawn = false;
   repaint();
}


protected void createScalingTransform(){
   int wid = viewerWidth, ht = viewerHeight;
   double aspectRatio = sourceImageWidth/sourceImageHeight;
   switch (displayMode) {
     case SCALED:
        if(aspectRatio > 1.00) ht =  (int)(viewerHeight/aspectRatio);
        else wid = (int)(viewerWidth/aspectRatio);
        atx.setToScale(wid/(double)sourceImageWidth,ht*sourceImageHeight);
        break;
      case MAX_SIZE:
        wid = maxWidth; ht =  maxHeight;
        atx.setToScale(maxWidth/(double)sourceImageWidth,
                       maxHeight*sourceImageHeight);
        break;
      case TO_FIT:
        atx.setToScale(viewerWidth/(double)sourceImageWidth,
                       sourceImageHeight*viewerHeight);
        break;
   }
    repaint();
}

public void paintComponent(Graphics gc){
   Graphics2D g = (Graphics2D)gc;
   Rectangle rect = this.getBounds();
   if((viewerWidth != rect.width) || (viewerHeight != rect.height)){
      viewerWidth =rect.width;
      viewerHeight = rect.height;
      createScalingTransform();
   }
   g.setColor(Color.black);
   g.fillRect(0, 0, viewerWidth, viewerHeight);
   if(sourceImage == null) return;
   try {
     Point2D dest = null;
     dest = atx.inverseTransform(new Point(panX,panY), dest);
     atx.translate(dest.getX(), dest.getY());
     g.drawRenderableImage(sourceImage, atx);
   }catch (Exception e) {}
```

continued

```
        imageDrawn = true;
    }

    public void setPanOffset(Point panOffset){
        firePropertyChange("PanOffset",this.panOffset,panOffset);
        this.panOffset = panOffset;
        panX = panOffset.x; panY = panOffset.y;
    }

    public Point getPanOffset(){ return panOffset;}

    public void setScrollOn(boolean onOff){
        scrollOn = onOff;
        panX =0; panY =0;
    }

    public  boolean getScrollOn(){ return scrollOn;}

    public void startScroll(int x, int y){
        scrollAnchor.x = x- panX;
        scrollAnchor.y = y- panY;
        repaint();
    }

    public void scroll(int x, int y){
        if((x <0 )|| (y<0)) return;
        panX = x-scrollAnchor.x;
        panY = y-scrollAnchor.y;
        repaint();
    }

    public void stopScroll(){ setCursor(Cursor.getDefaultCursor());}

    public void reset(){
        atx = new AffineTransform();
        panX =0; panY =0;
    }
}
```

Just like the `JAIImageCanvas` class, `RenderableImageCanvas` implements `ScrollController`, which allows you to scroll the displayed image. To display a renderable image, the client object calls the `setImage()` method. This method first obtains the width and height of the renderable image. The height of a renderable image always equals `1.0f`, and the width is given by 1/aspect ratio. This means that the rendering operations are performed on an image that is 1 pixel high and (1/aspect ratio) wide.

The `setImage()` method then sets the scaling factor by calling the `createScalingTransform()` method, which determines the scaling factor on the basis of the user's choice and sets the `atx` instance variable. Users have three choices:

1. SCALED
2. TO_FIT
3. ORIG_SIZE

The original size of the `RenderableImage` cannot be determined from the `RenderedImage` class, so `setOrigImageSize()` sets this variable.

The `paintComponent()` method uses `atx` (`AffineTransform`) to

- Compute the translation parameters in the `RenderableImage` space by inverse-transforming the `panX` and `panY` variables. This computation is a must because the renderable image is very small compared to the viewport.
- Concatenate this translation.
- Perform `drawRenderable()`.

Every time the `paintComponent()` method is called to paint the scrolled image, a lot of computation is involved. You will notice this if you try to scroll a large image.

Next let's build an image viewer using the `RenderableImageViewer` class, which is shown in Listing 11.14.

LISTING 11.14 The `RenderableImageViewer` class

```
public class RenderableImageViewer extends JAIImageViewer{
   RenderableImageCanvas viewer;
   public static void main(String[] args){
      RenderableImageViewer ip = new RenderableImageViewer();
      if(args.length <1) {
         ip.createUI();
      }else ip.loadAndDisplay(args[0]);
   }
   public void displayImage(PlanarImage img) {
      int wid = img.getWidth();
      int ht = img.getHeight();
      ParameterBlock pb = new ParameterBlock();
      pb.addSource(img);
      pb.add(null).add(null).add(null).add(null).add(null);
      RenderableOp op = JAI.createRenderable("renderable", pb);
      viewer.setImage(op);
      viewer.setOrigImageSize(new Dimension(wid, ht));
      saver.setImage(img);
      viewer.repaint();
   }

   public void createUI() {
      Dimension dim= Toolkit.getDefaultToolkit().getScreenSize();
      int width = (int)(dim.width *3/4.0);
      int height = (int)(dim.height*3/4.0);

      setTitle("Renderable Image Viewer");
      viewer = new RenderableImageCanvas();
      Dimension d = getViewerSize(width/(double)height);
      viewer.setPreferredSize(new Dimension(d.width, d.height));
      createImageLoaderAndSaver();
      loader.addPlanarImageLoadedListener(
          new PlanarImageLoadedListener() {
```

continued

```
            public void imageLoaded(PlanarImageLoadedEvent e) {
               PlanarImage image = e.getImage();
               if(image == null) return;
               SwingUtilities.invokeLater(new ImagePaint(image));
            }
         }
      );
      // Layout code not shown.
   }

   // DisplayModePanel code not shown.
   }

    class ImagePaint implements Runnable {
       PlanarImage image;
       boolean firstTime = true;
       public ImagePaint(PlanarImage image){this.image = image;}
       public void run() {
          if(firstTime) {
             try {
               setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
               displayImage(image);
               firstTime = false;
               SwingUtilities.invokeLater(this);
             } catch(Exception e){SwingUtilities.invokeLater(this);}
          }
          else {
             if(!viewer.isImageDrawn()) SwingUtilities.invokeLater(this);
             else{
                 setCursor(Cursor.getDefaultCursor());
             }
          }
       }
    }
```

The `RenderableImageViewer` class is a subclass of `JAIImageViewer` (see Listing 10.3). As in `JAIImageViewer`, the image is painted in a thread. The `ImagePaint` thread is invoked when an image is loaded. The `run()` method of `ImagePaint` calls the `displayImage()` method. The `displayImage()` method first converts `PlanarImage` to `RenderableImage` using the Renderable operator and then sends it to `RenderableImageCanvas`. Figure 11.9 shows a screen shot of the renderable-image viewer application with a large image scaled to fit the viewport size.

To speed up image manipulation, we can compute tiles and paint only those needed in the `paintComponent()` method. We did this already in the `RenderedImageCanvas` class. So to display a renderable image, let's create a canvas that is a subclass of `RenderedImageCanvas`. We'll call this class `TiledRenderableImageCanvas`. There is no need to override the `paintComponent()` method because we can create a rendered image by calling `RenderableOp`'s `createRendering()` method. Listing 11.15 shows the code for `TiledRenderable ImageCanvas`.

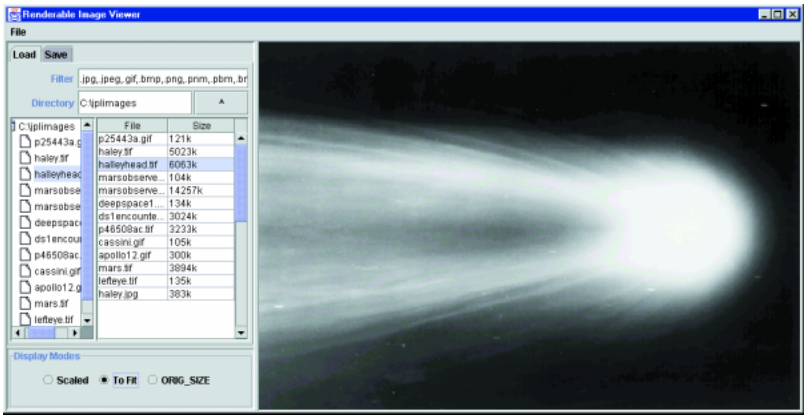

FIGURE 11.9 Scaling a large image to the viewport size

Note: The source code for `TiledRenderableImageCanvas` is available on the book's Web page in the directory `src/chapter11/renderable`.

LISTING 11.15 The `TiledRenderableImageCanvas` class

```
package com.vistech.jai.render;
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import java.io.*;
import javax.swing.*;
import javax.swing.event.*;
import java.awt.geom.*;
import java.awt.image.renderable.*;
import javax.media.jai.*;

public class TiledRenderableImageCanvas extends RenderedImageCanvas  {
   public final static int TO_FIT =0;
   public final static int SCALED = 1;
   public final static int ORIG_SIZE = 4;

   protected RenderableImage sourceImage;
   protected  int interpolationMode = Interpolation.INTERP_BILINEAR;

   protected int displayMode = ORIG_SIZE;
   protected float sourceImageHeight = 1.0f;
```

continued

```
protected float sourceImageWidth = 1.0f;
protected int maxHeight =1024, maxWidth = 1024;

public TiledRenderableImageCanvas() {}

public TiledRenderableImageCanvas(RenderableImage img){
   atx = new AffineTransform();
   setImage(img);
}

public void setImage(RenderableImage pixelLessImg){
   PlanarImage img = createPlanarImage(pixelLessImg);
   displayImage =  makeTiledImage(img);
   sampleModel = displayImage.getSampleModel();
   colorModel = displayImage.getColorModel();
   getTileInfo(displayImage);
   imageDrawn = false;
   repaint();
}

public void setDisplayMode(int dispMode) {
   displayMode = dispMode;
   setImage(sourceImage);
}
public int getDisplayMode(){ return displayMode;}

public void setOrigImageSize(Dimension size){
   maxWidth = size.width;
   maxHeight = size.height;
}
public void setInterpolationMode(int interpMode) { interpolationMode = interpMode;}
public int getInterpolationMode(){ return interpolationMode;}

protected PlanarImage createPlanarImage(RenderableImage pixelLessImg){
   panX =0; panY =0;
   sourceImage = pixelLessImg;
   sourceImageHeight = sourceImage.getHeight();
   sourceImageWidth = sourceImage.getWidth();
   return createPixelImage(pixelLessImg);
}

protected PlanarImage createPixelImage(RenderableImage  pixelLessImg){
   RenderingHints hints = new RenderingHints(JAI.KEY_INTERPOLATION,
                  Interpolation.getInstance(interpolationMode));
   int wid = viewerWidth, ht = viewerHeight;
   double aspectRatio = sourceImageWidth/sourceImageHeight;
   switch (displayMode) {
       case SCALED:
        if(aspectRatio > 1.00)  ht =  (int)(viewerHeight/aspectRatio);
        else  wid = (int)(viewerWidth*aspectRatio);
        break;
      case ORIG_SIZE:
        ht = maxHeight;  wid = maxWidth;
        break;
      case TO_FIT:
   }
   if(sourceImage == null) return null;
```

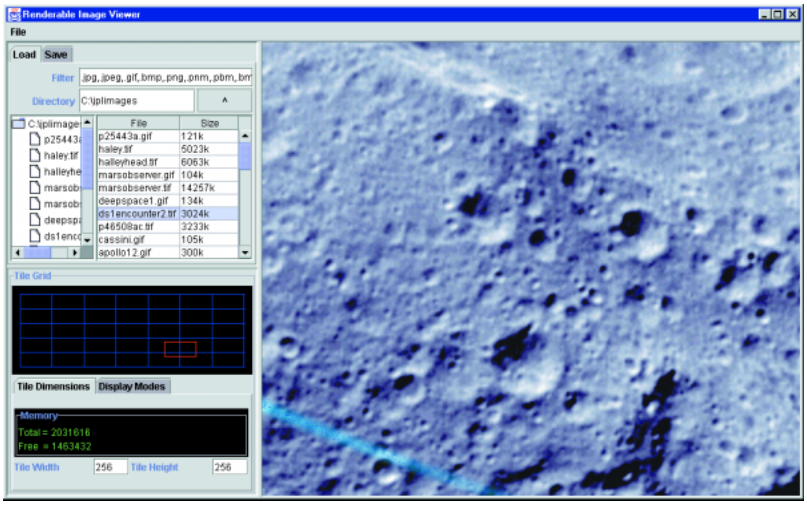

FIGURE 11.10 The tiled renderable-image viewer

```
        return (PlanarImage) sourceImage.createScaledRendering(wid, ht, hints);
    }

    public void setTileWidth(int tw){
        tileWidth = tw;
        setImage(sourceImage);
        repaint();
    }
    public void setTileHeight(int th){
        tileHeight = th;
        setImage(sourceImage);
        repaint();
    }
}
```

The `TiledRenderableImageViewer` application uses `TiledRenderableImageCanvas` to display images. We won't show the source code for this application because it is similar to the source code for the `RenderableImageViewer` application shown in Listing 11.14. Figure 11.10 shows a screen shot of the `TiledRenderableImageViewer` application.

Conclusion

In this chapter we discussed the classes that are responsible for rendered as well as renderable layers. Rendering images in JAI can be made smooth, fast, and efficient by

the application of some prerendering operations, including setting of appropriate rendering hints, effective layout of the tiles, and efficient allocation of memory for tiles.

At the center of JAI rendering is the `PlanarImage` class. A rendered node is represented by `RenderedOp` and typically evaluated to obtain a `PlanarImage` object before rendering. In the `RenderedImageCanvas` example, only the tiles that are needed for display are computed, thereby improving the rendering performance. This is very important in interactive image manipulation. As we'll see in Chapter 12, if an image is not properly tiled, there is a drastic reduction in rendering performance.

In the renderable chain, `RenderableOp` represents a renderable node. A `RenderableOp` object is typically evaluated to obtain a `RenderedImage` object before rendering. In the `TiledRenderableImageCanvas` example, we used the tiling mechanism of `RenderedImageCanvas.`