Manipulating Images in Java 2D

7

AFTER YOU DISPLAY AN IMAGE, the next logical step is to manipulate it. Although image manipulation may mean different things to different people, it generally means handling an image in such a way that its geometry is changed. Operations such as panning, zooming, and rotation can be considered image manipulations.
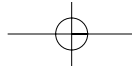
AWT imaging offers only minimal support for image manipulation. In JDK 1.1, usually images are manipulated through clever use of the `drawImage()` methods of the `Graphics` class. If you need to perform complex manipulation operations such as rotation, you must write your own transformation functions. With the introduction of the `AffineTransform` class in Java 2D, you can now implement any complex manipulation operation.

Many applications need the ability to apply image manipulations in a random order. With a map, for instance, you might want to pan it and then zoom it to look for a place of interest. Then you might want to rotate the map so that it is oriented in a direction to which you are accustomed. To inspect the map closely, you might zoom it again. To see nearby locations, you might pan it. This scenario illustrates that an application must be capable of performing manipulations in a random order in such a way that at every step operations are concatenated. Such capability would be difficult to implement without affine transformations. Because this chapter requires a thorough knowledge of affine transformations, you may want to read Chapter 4 first.

The quality of the rendered image is an important consideration in many image manipulation applications. The quality of the image often depends on the type of interpolation chosen. But quality comes with a price: The higher the quality, the more time it takes to generate the image.

This chapter will begin with an introduction to interpolation. After the basics of interpolation have been presented, we'll discuss image manipulation requirements. As we did in Chapter 6 for image rendering, we'll specify requirements for performing manipulation functions. On the basis of these specifications, we'll build a class for an image manipulation canvas. We'll then build several operator classes to operate on this canvas. Just as in Chapter 6, we'll also build an image viewer to illustrate all the

271

concepts presented here. All the operator classes are part of this image viewer, which is an extension of the image viewer of Chapter 6.

> **Note:** The source code and classes for this image viewer are available on the book's Web page in the directory `src/chapter7/manip.` To understand this chapter better, you may want to run the image viewer and perform the relevant transformations as you read.

What Is Interpolation?

As you may know already, pixels of an image occupy integer coordinates. When images are rendered or manipulated, the destination pixels may lie between the integer coordinates. So in order to create an image from these pixels, destination pixels are interpolated at the integer coordinates.

Interpolation is a process of generating a value of a pixel based on its neighbors. Neighboring pixels contribute a certain weight to the value of the pixel being interpolated. This weight is often inversely proportional to the distance at which the neighbor is located. Interpolation can be performed in one-dimensional, two-dimensional, or three-dimensional space. Image manipulation such as zooming and rotation is performed by interpolation of pixels in two-dimensional space. Volume imaging operations perform interpolation in three-dimensional space.

Java 2D supports some widely used interpolation techniques. You can choose them through the `RENDERING_HINTS` constant. The choice will depend on what is more important for your application: speed or accuracy.

Next we'll discuss different types of interpolation. Although you may not need to implement any interpolation code, knowledge of the different types of interpolation is helpful in understanding image rendering and manipulation.

Nearest-Neighbor Interpolation

In this simple scheme, the interpolating pixel is assigned the value of the nearest neighbor. This technique is fast, but it may not produce accurate images.

Linear Interpolation

In linear interpolation, immediate neighbors of the pixel to be interpolated are used to determine the value of the pixel. The distance-to-weight relationship is linear; that is, the relationship is of the form $y = ax + b$. In linear interpolation, left and right neighbors of the pixel are used to compute the pixel value (see Figure 7.1).
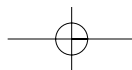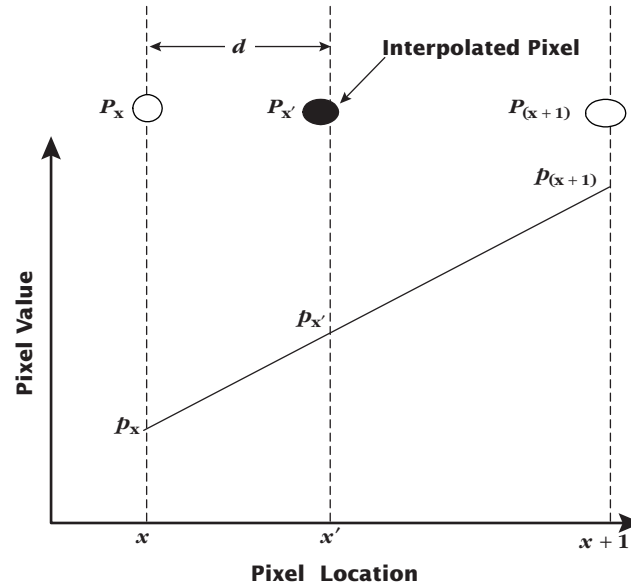
FIGURE 7.1   Linear interpolation

Let $P_{x'}$ be the pixel that lies between P_x and P_{x+1}, the respective pixel values of which are p_x and p_{x+1}. Let d be the distance between $P_{x'}$ and the left neighbor, P_x. The value of the pixel $P_{x'}$ is given by

$$P_{x'} = p_x + [(p_{x+1} - p_x) \times d]$$
$$= p_{x'}(1 - d) + (p_{x+1} \times d)$$

There are two types of linear interpolation: bilinear and trilinear.

Bilinear Interpolation

Bilinear interpolation is the method used for two-dimensional operations—for instance, magnifying an image. The interpolation is performed in a 2×2 neighborhood (see Figure 7.2).

Linear interpolation is performed in one direction, and the result is applied to the linear interpolation in the other direction. For example, if $P_{(x', y')}$ is the pixel at d_x and d_y from the upper left-hand neighbor, its value is computed by

$$p_u = [p_{(x,y)} - (1 \times d_x)] + (p_{(x+1,y)} \times d_x) \qquad \textbf{(7.1)}$$
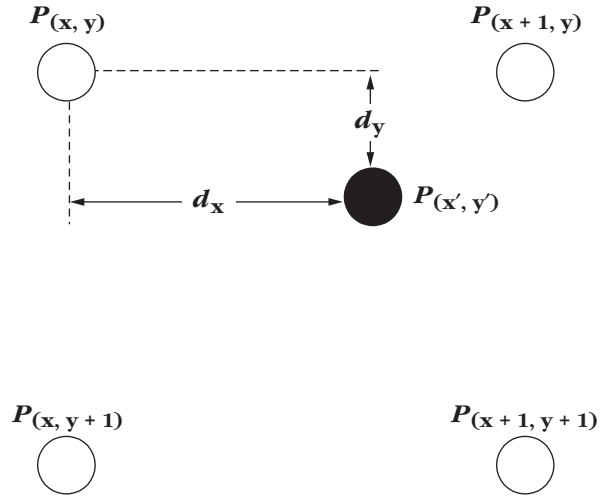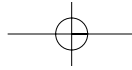
FIGURE 7.2 Bilinear interpolation

which represents the contribution to the upper row, and by

$$p_{l} = [p_{(x,y+1)} \times (1 - d_{x})] + (p_{(x+1,y+1)} - d_{x}) \tag{7.2}$$

which represents the contribution to the lower row.

From equations (7.1) and (7.2), we get $P_{(x',y')} = [p_{u} \times (1 - d_{y})] + (p_{l} \times d_{y})$.
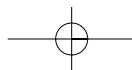
Trilinear Interpolation

Trilinear interpolation is computed in a 3×3 neighborhood (see Figure 7.3). To compute trilinear interpolation, bilinear interpolation is first performed in the *xyz* plane. Then the linear interpolation is applied to the resulting value in the *z* direction.

Cubic Interpolation

Cubic interpolation is performed in a neighborhood of four pixels (see Figure 7.4). The cubic equation is of the form $P_{x} = ax^3 + bx^2 + cx + d$.

Just as bilinear interpolation is performed in a 2×2 neighborhood, bicubic interpolation is performed in a 4×4 neighborhood.
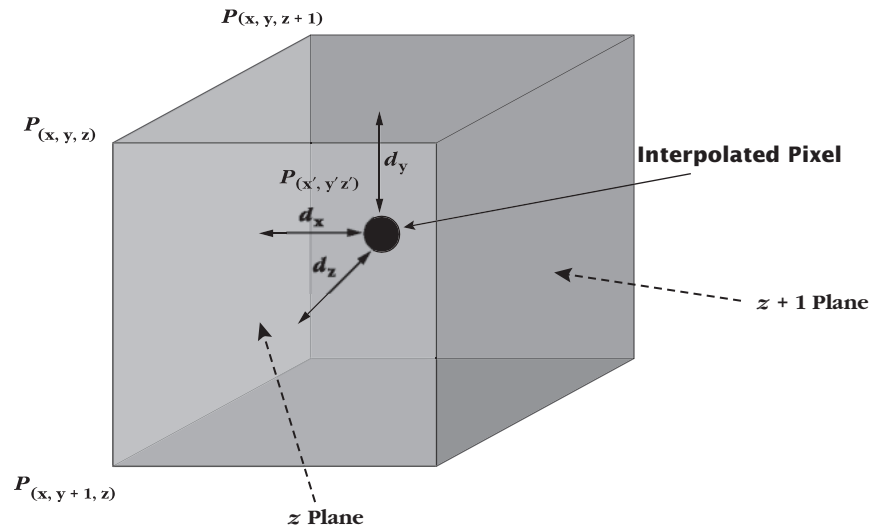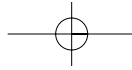
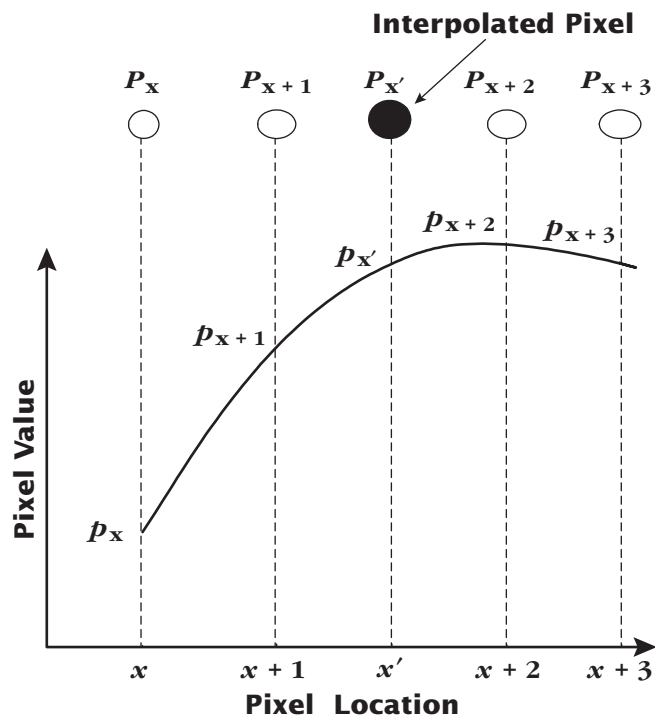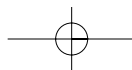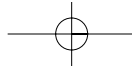**FIGURE 7.3**   Trilinear interpolation

FIGURE 7.4   Cubic interpolation

Applying Affine Transformation to Images

In Chapter 4 you may have noticed that the `AffineTransform` class does not explicitly provide a way to apply the affine transformation to an image. However, we can do this by using the `Graphics2D` class or the `AffineTransformOp` class, as described in the sections that follow.

Using the Graphics2D Class

The `Graphics2D` class has a flavor of the `drawImage()` method that takes an `AffineTransform` object as an input parameter. This method applies the transformation to the image before rendering. Listing 7.1 gives an example.

LISTING 7.1   The `applyTransform()` method

```
protected BufferedImage applyTransform(BufferedImage bi,
                                       AffineTransform atx,
                                       int interpolationType){
    Dimension d = getSize();
    BufferedImage displayImage =
              new BufferedImage(d.width, d.height, interpolationType);
    Graphics2D dispGc = displayImage.createGraphics();
    dispGc.drawImage(bi, atx, this);
    return displayImage;
}
```

The `drawImage()` method used in Listing 7.1 applies the transformation parameter `atx` to the `BufferedImage` object before it draws onto the `dispGc` graphical context that belongs to `displayImage.` So in effect, one buffered image is transformed and drawn onto another buffered image. If `displayImage` is drawn on a context obtained through `paint()` or `paintComponent(),` you can see the transformed image.

As mentioned in Chapter 5, the `Graphics2D` object maintains a `transform` attribute. This attribute is applied after the transformation from the input parameter is applied.

Using the AffineTransformOp Class

Here's a quick introduction to the `AffineTransformOp` class. The class has two constructors:

◆ **public AffineTransformOp(AffineTransform xform, int interpolationType)**
◆ **public AffineTransformOp(AffineTransform xform, RenderingHints hints)**

The `interpolationType` parameter in the first constructor should be one of `TYPE_BILINEAR` and `TYPE_NEAREST_NEIGHBOR`, which are defined in the `Affine`
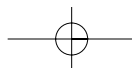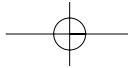
`TransformOp` class itself. If you use the second constructor, you may have more choices of interpolation techniques, depending on your platform.

The `AffineTransformOp` class implements two interfaces: `BufferedImageOp` and `RasterOp.` It applies the affine transformation to the source image and creates a transformed destination image. While transforming the image, `AffineTransformOp` applies the interpolation technique provided through its constructor. It has two filter methods:

- **`public final BufferedImage filter(BufferedImage src,`**
  `BufferedImage dst)`
- **`public final WritableRaster filter(Raster src, WritableRaster dst)`**

In this chapter only the first filter method is of interest. We'll explore the second one in Chapter 8.

Listing 7.2 shows a sample implementation of the `applyTransform()` method using the `AffineTransformOp` class.

LISTING 7.2 The `applyTransform()` method using `AffineTransformOp`

```
protected BufferedImage applyTransform(BufferedImage bi,
                                       AffineTransform atx,
                                       int interpolationType){
    Dimension d = getSize();
    BufferedImage displayImage = new BufferedImage(d.width,
                                                   d.height,
                                                   interpolationType);
    Graphics2D dispGc = displayImage.createGraphics();
    AffineTransformOp atop = new AffineTransformOp(atx, interpolationType);
    return atop.filter(bi, displayImage);
}
```
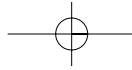
In Listing 7.2 an `AffineTransformOp` object is created with `AffineTransform` and `interpolationType` parameters. The `filter()` method in the `AffineTransformOp` class is invoked next. This method takes two arguments: source and destination `BufferedImage` object. It applies the transformation to the source buffered image and puts the resulting image in the destination buffered image.

With both `Graphics2D` and `AffineTransformOp,` the transformation is applied to a `BufferedImage` object. The resulting `BufferedImage` object is then rendered to a `Graphics2D` context in the paint methods.

Although we didn't mention it explicitly in Chapter 6, there was some image manipulation in the `ImageCanvas` class, which was achieved through the `drawImage()` methods. For instance, setting up display modes and flip modes involved some image manipulation. These operations can now be performed with the `AffineTransform` class.
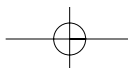
Image Manipulation Requirements

In Chapter 6 we enumerated various display requirements and designed an interface based on these requirements. Let's continue the discussion and extend it to incorporate the image manipulation requirements. These requirements will be the basis for designing interfaces to specify image manipulation methods.

Although image manipulation may mean different things to different people, certain common operations are performed by many imaging applications, including pan, zoom, rotate, and flip. In this chapter we'll consider each of these in turn. First, however, we need a canvas that can display manipulated images. To develop such a canvas, let's consider the following two approaches:

1. **Using the `ImageCanvas` class.** In this case image manipulation must be performed in separate class(es), and the resulting image is displayed on `ImageCanvas.` This approach is simple and works well for noninteractive operations. In interactive operations, however, the current transformation and other information must be retained. So a separate class is needed to manage such things.

2. **Creating a new class by extending the `ImageCanvas` class and implementing all the manipulation functions in it.** This is not a good solution because not all manipulation functions are required in an application. Even if they are needed, they may not be needed at a given time.

What is desirable is a flexible design that helps build only the desired manipulation functions at runtime. To achieve this, we'll adopt an approach that is a hybrid of the two approaches just presented. In this hybrid approach we'll build a controller class for each manipulation function. This controller class will operate on the image that is to be manipulated in the canvas. With this approach, an application needs to construct only required controller objects at any time.

Another important point to consider is the GUI required for generating manipulation parameters. The GUI and the controller classes must be clearly separate. The reason for this requirement is obvious: An application must be able to choose its own GUI for generating manipulation parameters.

The design pattern here is that the manipulation GUI feeds the data to the controller object, which in turn operates on the image canvas to manipulate the image. Often the manipulation GUI may need to capture some events from the image canvas. For example, `mouse` events are often needed to get the cursor position.

We'll design our interfaces from a user's point of view. First, let's design the `ImageManipulator` interface, which will be implemented by a component class to support image manipulation functionality. Such functionality includes pan, zoom, rotate, shear, and flip. For each of the manipulation types, we can define a property, the value of which can be set and retrieved by the clients of `ImageManipulator` objects.
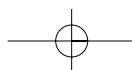
In addition, we'll allow the client objects to choose an interpolation mechanism through a property called `interpolationType.` Currently, the `AffineTransformOp` class supports only two types of interpolation techniques: nearest-neighbor and bilinear. The `interpolationType` property can assume either of two values: `Affine TransformOp.TYPE_BILINEAR` and `AffineTransformOp.TYPE_NEAREST_NEIGHBOR.`

Because images are manipulated through the affine transformation, we need to specify the current transformation of the canvas as a property. The current transformation can be concatenated outside the canvas. The controller objects need to get and set this property. For example, a client program can allow the user to set the rotation angle in an external GUI. This program can send the rotation angle to the rotation controller object, which will generate a transformation matrix for the rotation angle selected by the user. The rotation controller will concatenate this transformation with the current transformation obtained from the image canvas. It will then set the resulting transformation to be the current transformation of the image canvas. Table 7.1 summarizes the image manipulation properties.

Besides the set and get methods for the properties shown in Table 7.1, the `ImageManipulator` canvas requires two more methods:

1. A method for applying the transformation
2. A method for resetting manipulation

Current Transformation

Client objects require the current transformation (i.e., `AffineTransform` object) for several purposes. Applying the manipulation externally is one. All the manipulation features discussed in this chapter check out the current transformation, concatenate it, and put it back. Ideally, the current transformation should be locked so that only one program updates it at a given time.

TABLE 7.1   Image Manipulation Properties

PROPERTY	TYPE	DESCRIPTION
`interpolationType`	`int`	Interpolation type; one of `AffineTransformOp.TYPE_NEAREST_NEIGHBOR` or `AffineTransformOp.TYPE_BILINEAR`
`magFactor`	`double`	Magnification factor
`panOffset`	`Point`	Translation
`rotationAngle`	`double`	Rotation
`shearFactor`	`double`	Shear
`transform`	`AffineTransform`	Current transformation
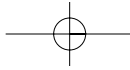
The current transformation is required for other purposes as well. For example, in region of interest (ROI) computations, the current transformation matrix is needed to convert the region (marked by the user) from user space to image space. So let's specify the current transformation as a property. The set and get methods for this property are

◆ **`public AffineTransform getTransform()`**
◆ **`public void setTransform(AffineTransform atx)`**

Resetting Manipulation

After you manipulate an image, normally you will start over again. So the client objects need a method for resetting the manipulation to a default value. Let's specify the following method for this purpose:

◆ **`public void resetManipulation();`**

Listing 7.3 shows the code for the `ImageManipulator` interface.

LISTING 7.3   The `ImageManipulator` interface

```
public interface ImageManipulator extends ImageDisplay {

    public void setPanOffset(Point panOffset);
    public Point getPanOffset();

    public void setMagFactor(double magFactor);
    public double getMagFactor();

    public void setRotationAngle(double theta);
    public double getRotationAngle();

    public void setShearFactor(double shear);
    public double getShearFactor();

    public AffineTransform getTransform();
    public void setTransform(AffineTransform at);

    public void applyTransform(AffineTransform atx);

    public void setInterpolationType(int interType);
    public int getInterpolationType();

    public void resetManipulation();
}
```

Any image-drawing component can implement `ImageManipulator.` Keep in mind that the components that implement this interface don't manipulate the image. The manipulation is actually performed by an external program. This canvas only holds the properties, applies the transformation, and paints the image.

Not all types of manipulations are required by an application. By separating the actual manipulation from the component, the image manipulation canvas does not have to carry the code to perform different types of manipulation. Instead, the manipulation code is in separate classes or beans.

Implementing a Canvas for Image Manipulation

Our goal is to build an image-viewing application that allows users to interactively manipulate images. The image manipulation functions we intend to implement are pan, zoom, rotate, flip, and shear. Clearly we need a component class or bean that renders images and provides manipulation functionality.

In Chapter 6 we implemented an image-rendering component called `ImageCanvas.` The image manipulator class, which we'll name `ImageCanvas2D,` will extend the `ImageCanvas` class. To implement the requirements described in the preceding section, `ImageCanvas2D` will implement the `ImageManipulator` interface (see Figure 7.5).

Besides all the set and get methods for the properties listed in Table 7.1, `Image Canvas2D` must implement two important methods: `paintImage()` and `apply Transform().` Listing 7.4 shows the implementation.

LISTING 7.4   The ImageCanvas2D class

```
package com.vistech.imageviewer;
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;

import java.util.*;
import java.beans.*;
import javax.swing.*;
import java.awt.geom.*;
import com.vistech.events.*;
```
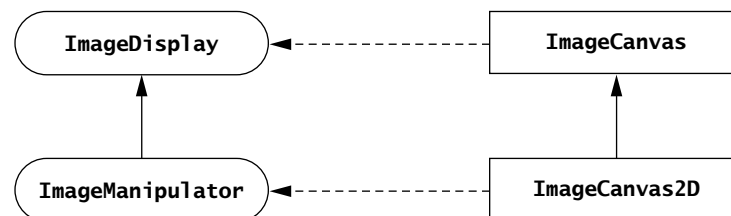
continued

FIGURE 7.5   The ImageCanvas2D class hierarchy

```java
public class ImageCanvas2D extends ImageCanvas implements
                            ImageManipulator{

   protected AffineTransform atx = new AffineTransform();
   protected AffineTransform dispModeAtx = new AffineTransform();
   protected AffineTransform flipAtx = new AffineTransform();

   // Pan variables
   protected Point panOffset = new Point(0,0);
   // Zoom variables
   protected boolean magOn = true;
   protected double magFactor = 1.0;
   protected int magCenterX = 0;
   protected int magCenterY =0;
   protected Point zoomOffset = new Point(0,0);
   // Rotation variables
   protected double rotationAngle = 0.0;
   protected boolean rotateOn = true;
   protected int rotationCenterX = 0;
   protected int rotationCenterY = 0;
   // Shear variables
   protected boolean shearOn = true;
   protected double shearFactor = 0.0;
   protected double shearX =0.0, shearY=0.0;
   // Off-screen image width and height
   private int bufImageWidth = -1;
   private int bufImageHeight = -1;
   protected int interpolationType = AffineTransformOp.TYPE_NEAREST_NEIGHBOR;

   public ImageCanvas2D(){ init();;}

   protected void init(){
      setDoubleBuffered(false);
      setBackground(Color.black);
      setForeground(Color.white);
      this.setSize(200,150);
   }

   public void setMagFactor(double magFactor){
      firePropertyChange("MagFactor",
                         new Double(this.magFactor),
                         new Double(magFactor));
      this.magFactor = magFactor;
   }
   public double getMagFactor(){ return magFactor;}
   public void setShearFactor(double shearFactor){
      firePropertyChange("ShearFactor",
                         new Double(this.shearFactor),
                         new Double(shearFactor));
      this.shearFactor = shearFactor;
   }

   public double getShearFactor(){ return shearFactor; }
   public double getShearFactorX(){ return shearX;}
   public double getShearFactorY(){ return shearY;}

   public void setRotationAngle(double rotationAngle){
      firePropertyChange("RotationAngle",
```
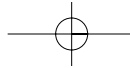
```
                        new Double(this.rotationAngle),
                        new Double(rotationAngle));
   this.rotationAngle = rotationAngle;
}
public double getRotationAngle(){ return rotationAngle; }

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

public Point getPanOffset(){ return panOffset; }

public void setInterpolationType(int interType) {interpolationType = interType;}
public int getInterpolationType() { return interpolationType;}

public AffineTransform getTransform(){ return atx; }
public void setTransform(AffineTransform at){ atx = at; }

public  void setMagOn(boolean onOff){ magOn = onOff;}
public  boolean getMagOn(){ return magOn;}

public synchronized boolean doDisplayModeAndFlip(int imageWidth, int imageHeight){
   width = this.getBounds().width;
   height = this.getBounds().height;
   double magX= (double)width/(double)imageWidth;
   double magY = (double)height/(double)imageHeight;
   int bufferWid = width, bufferHt=height;
   dispModeAtx = new AffineTransform();
   switch(displayMode){
     case DisplayMode.ORIG_SIZE:
       bufferWid = imageWidth;
       bufferHt = imageHeight;
       break;
     case DisplayMode.SCALED:
       double mag= (magY > magX)? magX:magY;
       dispModeAtx.setToScale(mag, mag);
       bufferWid = (int)(imageWidth*mag);
       bufferHt = (int)(imageHeight*mag);
       break;
   case DisplayMode.TO_FIT:
       dispModeAtx.setToScale(magX, magY);
       bufferWid = width;
       bufferHt = height;
       default:
       break;
   }
   BufferedImage bi = new BufferedImage(bufferWid, bufferHt,imageType);
   Graphics2D bigc = bi.createGraphics();
   if(originalImageType == TYPE_AWT_IMAGE)
       bigc.drawImage(awtImage, dispModeAtx, this);
   else bigc.drawImage(bufferedImage, dispModeAtx, this);

   flipAtx = createFlipTransform(flipMode, bufferWid, bufferHt);
```
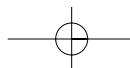
continued

```
        AffineTransformOp atop = new AffineTransformOp(flipAtx, interpolationType);
        offScrImage =  atop.filter(bi, null);
        offScrGc = offScrImage.createGraphics();
        applyTransform(offScrImage, atx);
        repaint();
        return true;
    }

    protected void applyTransform(BufferedImage bi, AffineTransform atx){
        if(offScrImage == null) return;
        if(displayImage == null) createDisplayImage();
        AffineTransformOp atop = new AffineTransformOp(atx, interpolationType);
        dispGc.setColor(Color.black);
        dispGc.fillRect(0,0,displayImage.getWidth(), displayImage.getHeight());
        dispGc.setClip(clipShape);
        if(clipShape == null) atop.filter(bi, displayImage);
        else dispGc.drawImage(bi,atx,this);
    }

    public void applyTransform(AffineTransform atx){
        applyTransform(offScrImage, atx);
        this.atx = atx;
        repaint();
    }

    public void reset(){
        panOffset = new Point(0,0);
        magCenterX = 0; magCenterY =0;
        magFactor = 1.0;
        rotationAngle = 0.0;
        shearX = 0.0; shearY = 0.0;
        atx = new AffineTransform();
        paintImage();
    }

    public void resetManipulation(){
        panOffset = new Point(0,0);
        magCenterX = 0; magCenterY =0;
        magFactor = 1.0;
        shearX = 0.0; shearY = 0.0;
        rotationAngle = 0.0;
        atx = new AffineTransform();
        paintImage();
        repaint();
    }

}
```

The `ImageCanvas2D` class holds the current value of the transform property in `atx,` which is concatenated whenever the current image is manipulated.

Recall from Chapter 6 how images are painted (see Figure 6.4). When the `setAWTImage()` or `setBufferedImage()` method is called, the original image is loaded. When the `setDisplayMode()` or `setFlipMode()` method is called, an `offScreenImage` object is created by application of the current `displayMode` and `flipMode` properties. The `offScrImage` variable holds this image.
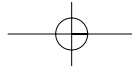
Any manipulation function will either concatenate `atx` or create a new instance of `atx` and apply it to `offScrImage.` The `applyTransform()` method discussed earlier (see Listings 7.1 and 7.2) does this. The `applyTransform()` method creates another `BufferedImage` object, which is saved in the `displayImage` variable. You may recall that `offScrImage` and `displayImage` are defined in the superclass `ImageCanvas.` Having two copies of the same image may seem wasteful, but they are needed for performance.

## Using the Affine Transformation to Set Display and Flip Modes

The `doDisplayModeAndFlip()` method overrides the `doDisplayModeAndFlip()` method in the `ImageCanvas` class. As far as the display mode is concerned, we need to apply just one type of transformation: scaling. On the basis of the requirements listed in Chapter 6, when the display modes are set, the viewport is reset. This means that we need to reset the underlying `AffineTransform` object.

The `doDisplayModeAndFlip()` method in Listing 7.4 first creates an `Affine Transform` object—`dispModeAtx`—and then checks for display mode type, which can be any one of `ORIG_SIZE`, `SCALED,` or `TO_FIT.` In the case of `ORIG_SIZE,` the affine transformation is the identity transformation itself because there is no scaling. In the case of `SCALED,` the scale factor is calculated in both the x and the y directions, and the larger factor is chosen. The `doDisplayModeAndFlip()` method then calls the `setToScale()` method to create a transformation with a specified scale factor. In this case the scale factor is the same in both the x and the y directions because of the need to preserve the aspect ratio of the image. In the `TO_FIT` case, the scale factors in both directions are taken into consideration, and an `AffineTransform` object is created.

Next the `doDisplayModeAndFlip()` method creates a `BufferedImage` instance that is as big as the image to be rendered. The original image is drawn on this `BufferedImage` object by the `drawImage()` method. The `dispModeAtx` object is passed as a parameter to `drawImage(),` thus applying the desired display mode to the original image. The image thus created may need to be flipped, depending on the flip mode selected.

The `doDisplayModeAndFlip()` method then creates a flip transformation. It uses the `filter()` method of the `AffineTransformOp` class to apply the flip transformation to the current image and creates another `BufferedImage` object. Although this image is ready for rendering, the `doDisplayModeAndFlip()` method performs one more transformation because we may use this image for image manipulation purposes. The `BufferedImage` object obtained after the flip mode operation is called `offScrImage,` and we'll make this image the base for all other image manipulation operations.

The `doDisplayModeAndFlip()` method transforms the image through the `applyTransform()` method, which we described earlier. The resulting image—`displayImage`—is the one that is rendered (see Figure 6.4). So there are two instance
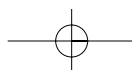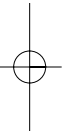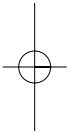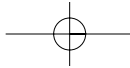
variables: `atx` and `displayImage.` The variable `atx` will be concatenated when an image is manipulated, and it will be reset when the display or flip modes are set.

Flipping

The `AffineTransform` class does not have direct methods for creating a transformation matrix for flip. We'll use the reflection transformation to create a flip matrix (see Chapter 4). You may recall that the reflection transformation creates a mirror image rather than a flipped image. To create a flip transformation, all we need to do is to translate the reflection matrix to the same quadrant as the image. Listing 7.5 shows how to do this.

LISTING 7.5   Creating a flip transformation

```
static public AffineTransform createFlipTransform(int mode,
                                                   int imageWid,
                                                   int imageHt){
    AffineTransform at = new AffineTransform();
    switch(mode){
        case FlipMode.NORMAL:
          break;
        case FlipMode.TOP_BOTTOM:
          at =  new AffineTransform(new double[] {1.0,0.0,0.0,-1.0});
          at.translate(0.0, -imageHt);
          break;
        case FlipMode.LEFT_RIGHT :
          at = new AffineTransform(new double[] {-1.0,0.0,0.0,1.0});
          at.translate(-imageWid, 0.0);
          break;
        case FlipMode.TOP_BOTTOM_LEFT_RIGHT:
          at = new AffineTransform(new double[] {-1.0,0.0,0.0,-1.0});
          at.translate(-imageWid, -imageHt);
          break;
        default:
    }
    return at;
}
```

As Listing 7.5 shows, the `createFlipTransform()` method constructs an `AffineTransform` object for reflection by passing a flat matrix array. This array contains the elements of a top left-hand 2×2 matrix, which is different for each of the flip mode cases. Once the reflection transformation has been constructed, it is translated to the original image location itself.

In the sections that follow we'll look at the different types of manipulation, starting with pan.

Pan

When an image is bigger than the viewport, we tend to scroll the image to see parts that are not visible. To pan or scroll images in a viewport, applications provide different types of user interfaces—for example, scroll bars. Some applications provide mouse-based scrolling. With mouse-based schemes, you normally hold the mouse down and drag the image so that you can view the desired portion of the image. In our design, however, we'll separate the GUI from the controller. The controller will have the logic to scroll on an image canvas, and it will accept input from any type of user interface.

Let's discuss the requirements and design an interface named `ScrollController.` Figure 7.6 shows how images are panned, or scrolled. The position at which you click the mouse is the anchor point, and as you drag, the current mouse position is the current position of the image. The pan offset is the displacement between the current position and the anchor point. The current image is drawn with this offset from the previous position.

To hold this displacement, let's specify a property named `panOffset.` The set and get methods for this property are

- **`public void setPanOffset(Point panOffset);`**
- **`public Point getPanOffset();`**

We need a method that initiates the scrolling. This method should set the anchor point. Let's call this method `startScroll():`
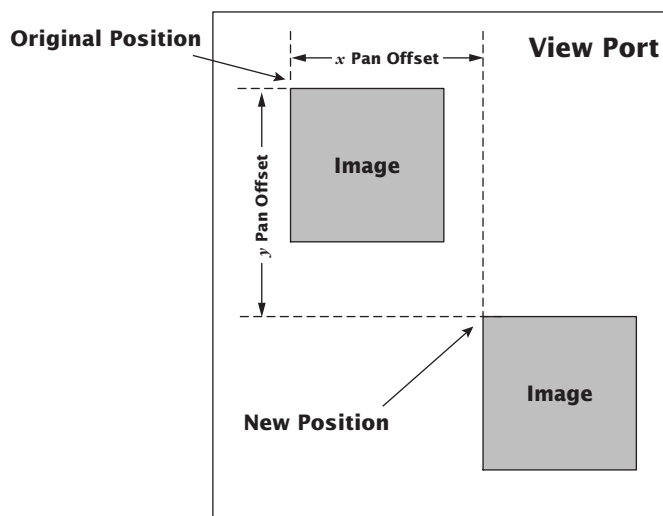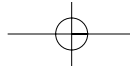

FIGURE 7.6   Panning an image

◆ **`public void startScroll(int x, int y)`**

We also need a method that stops the scrolling:

◆ **`public void stopScroll()`**

When the image is scrolled, it is painted at a new position. The `paintImage()` method we defined in the `ImageDisplay` interface will not suffice to accomplish this task. So let's specify a method for this purpose, with the displacement in the *x* and *y* directions as its inputs:

◆ **`public void scroll(int xOffset, int yOffset);`**

This method paints the current image at an offset of `xOffset` and `yOffset` from the previous position in the *x* and *y* directions, respectively.

The `ScrollController` interface, shown in Listing 7.6, contains all the methods defined here.

`LISTING 7.6`   The `ScrollController` interface

```
public interface ScrollController {
    public void setPanOffset(Point panOffset);


    public Point getPanOffset();
    public void startScroll(int x, int y);
    public void scroll(int x, int y);
    public void stopScroll();
}
```

Implementing Pan

Now that we have designed an interface to implement scrolling (panning), let's implement a mouse-driven scroll feature. It will have two classes:

1. **`Scroll`.** This class will control scrolling by implementing the `ScrollController` interface. `Scroll` will operate on an image canvas, so it needs to hold a reference to an object that implements the `ImageManipulator` interface.
2. **`ScrollGUI`.** This class will accept user input and pass it on to `Scroll Controller`. In the case of mouse-driven scrolling, `ScrollGUI` must receive `mouse` and `mouseMotion` events from `ImageManipulator`.

Figure 7.7 shows a design similar to model-view-controller (MVC) architecture[1] for scrolling. The `ScrollGUI` object receives the mouse positions through `mouse` and

---

1 If you are not familiar with MVC architecture, you may want to consult an object-oriented programming book. *Design Patterns: Elements of Reusable Object-Oriented Software* (1995), by Gamma, Helm, Johnson, and Vlissides, is a good source.

mouseMotion events. It passes the mouse positions as pan offset to the ScrollController interface, which modifies the current transformation to take the translation into account.

We'll follow the same design pattern for all the other manipulation functions. This pattern clearly separates the GUI and the application logic, so you are free to replace ScrollGUI with another class for a different type of GUI.

Two schemes are popular for scrolling images in a viewport:

1. Using scroll bars
2. Using mouse-driven interfaces

Because we use the mouse-driven interface in this book, let's look at how a mouse-driven client object must interact with a Scroll object.

The Scroll object must remember two positions: the anchor point and the current position of the image. When the mouse is pressed, the mousePressed() method of a client object must call startScroll(int x, int y) and pass the current coordinates of the mouse as the input parameters. The startScroll() method must save this position as the anchor point. Whenever the mouse is dragged, the mouseDragged() method of the client object must call scroll(int x, int y) and pass the current position of the mouse as input parameters. The scroll() method must compute the displacement between the current position of the mouse and the anchor position. It must then pass this displacement to the ImageManipulator object so that it can paint the image at the current mouse position. When the mouse button is released, the mouseReleased() method of the client object must call stopScroll().

Listing 7.7 shows the code for Scroll.

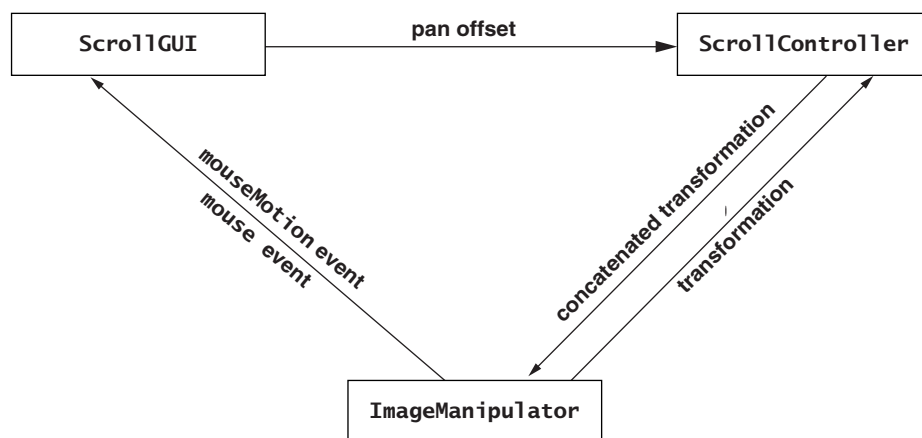**FIGURE 7.7**   The data flow for scrolling

LISTING 7.7   The Scroll class

```java
package com.vistech.imageviewer;
import java.io.*;
import java.awt.*;
import java.awt.image.*;
import java.util.*;
import java.awt.geom.*;

public class Scroll implements ScrollController  {
    protected AffineTransform atx = new AffineTransform();
    protected Point panOffset = new Point(0,0);
    private Point diff = new Point(0,0);
    private Point scrollAnchor  = new Point(0,0);
    protected ImageManipulator imageCanvas;

    public Scroll() {}

    public Scroll(ImageManipulator imageCanvas) {
        this.imageCanvas = imageCanvas;
    }
    public void setImageManipulator(ImageManipulator imageCanvas){
        this.imageCanvas = imageCanvas;
    }

    public void setPanOffset(Point panOffset){
        this.panOffset = panOffset;
        imageCanvas.setPanOffset(panOffset);
    }

    public Point getPanOffset(){return panOffset; }

    public void translateIncr(double incrx, double incry) {
        atx.translate(incrx, incry);
        imageCanvas.applyTransform(atx);
    }

    public void translate(double diffx, double diffy) {
        double dx = diffx -panOffset.x;
        double dy = diffy -panOffset.y;
        panOffset.x = (int)diffx;
        panOffset.y = (int)diffy;
        translateIncr(dx,dy);
    }

    public void resetAndTranslate(int dx, int dy) {
        atx.setToTranslation((double)dx, (double)dy);
        imageCanvas.applyTransform(atx);
    }

    public void scroll(int x, int y){
        if((x <0 )|| (y<0)) return;
        try {
            Point2D xy = null;
            xy = atx.inverseTransform((Point2D)(new Point(x,y)), xy);
            double ix  = (xy.getX()-scrollAnchor.x);
            double iy =  (xy.getY()-scrollAnchor.y);
```

```
            translateIncr(ix,iy);
         }catch(Exception e) {System.out.println(e);}
      }

      public void startScroll(int x, int y){
         atx = imageCanvas.getTransform();
         // Create a new anchor point so that every time mouse button is clicked,
         // the image does not move, but instead the anchor point moves.
         try {
             Point2D xy = null;
             xy = atx.inverseTransform((Point2D)(new Point(x,y)), xy);
             scrollAnchor = new Point((int)(xy.getX()),(int)(xy.getY()));
             imageCanvas.setCursor(Cursor.getPredefinedCursor(Cursor.HAND_CURSOR));
          }catch(Exception e) {System.out.println(e);}
      }

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

      public void reset() {
         scrollAnchor = new Point(0,0);
         diff = new Point(0,0);
      }
}
```

The `Scroll` class implements the `ScrollController` interface. Note that `Scroll`'s constructor takes the `ImageManipulator` object as an input. This means you can pass any component class that implements the `ImageManipulator` interface as a parameter to this constructor.

The `Scroll` object gets the current transformation from the canvas, concatenates it with the transformation generated for the translation, and sets this concatenated transformation as the current transformation of the canvas. Let's look at `Scroll`'s methods.

Scrolling

The `startScroll()` method first computes the anchor position and assigns it to the variable `scrollAnchor.` The position passed as an input parameter to `startScroll()` is in the user coordinate space. These coordinates need to be converted to the image space because the displayed image might have already undergone some transformations. User space is converted to image space through the inverse transformation.

What would happen if this transformation were not done? You would get different results, depending on the transformation(s) already applied to the image. If the image were flipped right to left, it would move in the opposite direction along the *x*-axis; that is, if you tried to pan the image left to right, it would move right to left. If the image were rotated, it would move in the direction of the rotation when you tried to pan along the *x*- or *y*-axis. Inverse transformation solves this problem. The `startScroll()` method also launches a new cursor to indicate that the image is being panned.
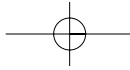
The `scroll()` method converts the input coordinates to the image space and computes the displacement from the anchor point. This displacement is passed to the `translateIncr()` method, which performs the actual translation of the image. The `stopScroll()` method is called when the mouse is released, and it returns the original cursor.

Next we'll look at the translation-related methods in more detail.

Translation

Applying translation makes the image move. The `translateIncr()` method takes translation increments as input parameters. For instance, if an image is moved to position $P_{(x',y')}$ from $P_{(x,y)}$, the translation parameters are the displacements from $P_{(x,y)}$ to $P_{(x',y')}$. The reason is that the current transformation, which is contained in the `atx` variable, has already taken into account the translation to $P_{(x,y)}$ from the original position.

As stated earlier, if the translation parameters are the coordinates from the user space (i.e., viewport), the inverse transformation needs to be applied to these coordinates so that they are in `atx` space.

The `translate()` method does the same thing as the `translateIncr()` method, but it takes the absolute translation from the anchor point. This method resets `atx` and sets the translation to the parameters specified in the inputs. It calls the `setToTranslation()` method of the `AffineTransform` class to reset `atx` and then translates the images to (d_x, d_y). Note that the input arguments are of type `int.` The reason is that the coordinates in the user space are integer types, and these coordinates are expected to be passed directly to this method.

User Interface for Scrolling

As stated earlier, panning in our implementation is performed through `mouse` and `mouseMotion` events. The `ScrollGUI` class must capture these events, which means it needs to implement the `MouseListener` and `MouseMotionListener` interfaces. Listing 7.8 shows the implementation.

LISTING 7.8 The `ScrollGUI` class

```
package com.vistech.imageviewer;
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ScrollGUI implements  MouseListener,MouseMotionListener{
    protected ScrollController scrollController;
    protected boolean scrollOn = true;
    protected boolean mousePressed = false;
```
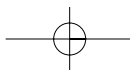
```
public ScrollGUI(ScrollController c){scrollController = c;}
public  void setScrollOn(boolean onOff){ scrollOn = onOff;}
public void init() { setScrollOn(true);}

public  boolean getScrollOn(){ return scrollOn; }
public void startScroll(){ scrollOn = true;}

public void reset(){
    if(scrollController != null)scrollController.stopScroll();
    setScrollOn(false);
}

public void mousePressed(MouseEvent e) {
    if(!scrollOn) return;
    if(mousePressed) return;
    mousePressed = true;
    if(SwingUtilities.isLeftMouseButton(e)){
        scrollController.startScroll(e.getX(), e.getY());
    }
}

public void mouseReleased(MouseEvent e) {
    scrollController.stopScroll();
    mousePressed = false;
}
public void mouseClicked(MouseEvent e){}
public void mouseEntered(MouseEvent e){}
public void mouseExited(MouseEvent e){}

public void mouseDragged(MouseEvent e){
    if(SwingUtilities.isLeftMouseButton(e)){
        if(scrollOn) scrollController.scroll(e.getX(), e.getY());
    }
}
public void mouseMoved(MouseEvent e){}
}
```

Note that the `ScrollGUI` constructor takes `ScrollController` as the input. It communicates with the `ImageManipulator` object through `Scroll` (see Figure 7.7).

In order for `ScrollGUI` to receive `mouse` and `mouseMotion` events, a separate object must register `ImageManipulator` with `ScrollGUI`. The reason is that both the `ImageManipulator` and the `ScrollController` objects are unaware of the `ScrollGUI` object. This registration needs to be done when `ScrollGUI` is constructed by an application, as shown here:

```
ImageManipulator imageCanvas = new ImageCanvas2D();

ScrollController scroll = new Scroll(imageCanvas);
ScrollGUI scrollUI = new ScrollGUI(scroll);

imageCanvas.addMouseListener(scrollUI);
```

To refresh your memory, `Scroll` implements the `ScrollController` interface, and `ImageCanvas2D` implements the `ImageManipulator` interface. These objects can be used as `ScrollController` and `ImageManipulator` objects, respectively.

Let's construct a typical sequence of method calls with `ScrollGUI`, `Scroll Controller`, and `ImageManipulator` objects:

1. The application that needs the scroll feature registers `ScrollGUI` to receive `mouse` and `mouseMotion` events with the `ImageManipulator` object.

2. The user presses the left mouse button. The `ImageManipulator` object, which is the mouse event source, fires the `mousePressed` event to the `ScrollGUI` object. The `mousePressed()` method is invoked when any mouse button is clicked. Because the `Scroll` class uses only the left mouse button, the `isLeftMouse Button()` method of the `SwingUtilities` class checks for this condition; that is, it returns `true` only when the left mouse button is pressed.

3. The `mousePressed()` method calls the `startScroll()` method in the `ScrollController` object, which saves the position at which the mouse was pressed as the anchor point (in the variable called `anchorPoint`).

4. The user drags the mouse while holding the left mouse down. The `ImageManipulator` object fires the `mouseDragged` event, which is received by the `mouseDragged()` method in `ScrollGUI.` The `mouseDragged()` method is called repeatedly as the user drags the mouse.

5. The `mouseDragged()` method calls the `scroll()` method in the `Scroll Controller` object and passes the current mouse coordinates as its parameters.

6. The `ImageManipulator` object paints the current image (i.e., `displayImage`) at the new position.

7. The user releases the mouse button, prompting the `ImageManipulator` object to fire a `mouseReleased` event. This event is received by the `mouseReleased()` method.

8. The `mouseReleased()` method calls `stopScroll()` to halt scrolling.

Zoom

As with pan, different applications provide different types of user interfaces for zooming images. Some applications allow you to zoom in or zoom out by clicking on the image. When the image is magnified or reduced, the point on the image where you clicked remains at the same position on the viewport. In some applications you are asked to mark a rectangular region over the image. The image within this rectangular region is then zoomed onto the viewport.
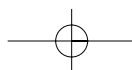
As we did with `ScrollController` to implement the pan operation, let's create an interface called `ZoomController` for implementing GUI-independent zoom logic and then build a GUI-specific class for a zoom feature. The `ZoomController` interface will have a property called `magFactor` whose set and get methods are

◆ **public void setMagFactor(double magFactor);**
◆ **public double getMagFactor();**

We'll design two types of magnification methods:

1. A method for magnifying the original image by a specified factor
2. A method for magnifying a displayed image by a specified factor and then setting the magnified image as the new displayed image

Both types of methods are required in practice.

Magnifying the Original Image

Let's specify the following method for magnifying the original image:

◆ **public boolean magnify(int magCenterX, int magCenterY, double mag);**
This method will reset any other transformation that has been performed on the image, except when the image is transformed to set the `displayMode` and `flipMode` properties. It magnifies the image with (`magCenterX`, `magCenterY`) as the coordinates of the center of magnification. This means that if you click at position $P_{(magCenterX,magCenterY)}$ on the viewport, the corresponding point on the magnified image will be located at the same position (see Figure 7.8).

Let's define one more flavor of the `magnify()` method for implementing the zoom feature:

◆ **public boolean magnify(int magCenterX, int magCenterY);**
This method is same as the preceding `magnify()` method, except that it gets the magnification factor from the `magFactor` property.

Magnifying the Displayed Image

It may be convenient in some cases to magnify the displayed image itself. To meet this requirement, let's define one more `paintImage()` method:

◆ **public void paintImage(int magCenterX, int magCenterY, double mag);**
When this method is executed, the magnified image becomes the new `displayImage` object.

The `ZoomController` interface, shown in Listing 7.9, contains all the methods defined here.

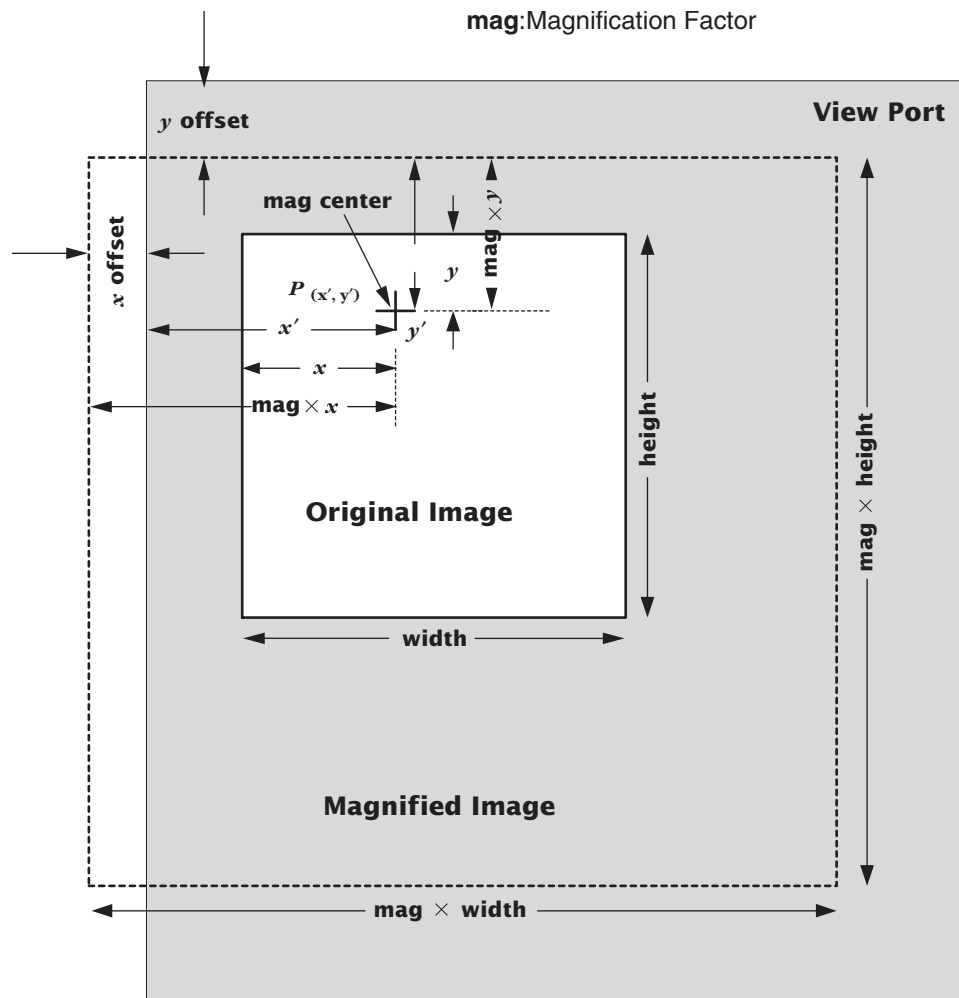**FIGURE 7.8**   Zooming an image

LISTING 7.9   The ZoomController interface

```
public interface ZoomController {
    public void setMagFactor(double magFactor);
    public double getMagFactor();
    public void magnify(int magCenterX, int magCenterY, double mag);
    public void magnify(int magCenterX, int magCenterY);
    public void paintImage(int magCenterX, int magCenterY, double mag);
 }
```

Implementing Zoom

The ZoomController interface specifies several methods for magnification as well. The design of the zoom feature is similar to the scroll feature. We'll define two classes: one for controlling zoom, and the other for the GUI:

1. **Zoom.** This class will control zoom by implementing the ZoomController interface.

2. **ZoomGUI.** This class will implement the zoom feature. It will contain the mouse user interface.

The zoom feature follows the same design pattern as scroll (see Figure 7.9). Listing 7.10 shows the code for the Zoom class.

LISTING 7.10   The Zoom class

```
package com.vistech.imageviewer;
import java.io.*;
import java.awt.*;
import java.awt.image.*;
import java.awt.geom.*;
public class Zoom implements ZoomController  {
   protected AffineTransform atx = new AffineTransform();
   protected boolean magOn = true;
   protected double magFactor = 1.0;
   protected int magCenterX = 0;
   protected int magCenterY =0;
   protected Point zoomOffset = new Point(0,0);
   protected ImageManipulator imageCanvas;

   public Zoom(){}
   public Zoom(ImageManipulator imageCanvas){ this.imageCanvas = imageCanvas;}
```
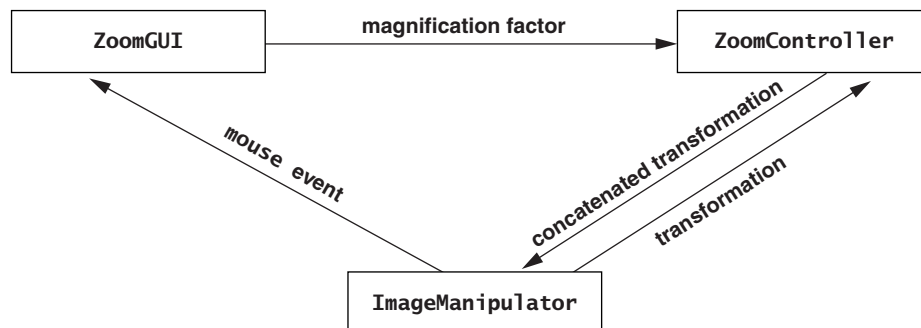
FIGURE 7.9   The data flow for zoom

```
public void setImageManipulator(ImageManipulator imageCanvas){
   this.imageCanvas = imageCanvas;
}

public void setMagOn(boolean onOff){ magOn = onOff;}
public boolean getMagOn(){ return magOn; }

public void setMagFactor(double magFactor){
   this.magFactor = magFactor;
   imageCanvas.setMagFactor(magFactor);
}

public double getMagFactor(){ return magFactor;}

public void magnify(int magCenterX, int magCenterY){
    magnify(magCenterX, magCenterY, magFactor);
}

public void magnify(int magCenterX, int magCenterY, double magFac){
   setMagFactor(magFac);
   this.magCenterX = magCenterX;
   this.magCenterY = magCenterY;
   Point panOffset = imageCanvas.getPanOffset();
   int x = (int)((magCenterX-panOffset.x)*magFactor)-magCenterX;
   int y = (int)((magCenterY-panOffset.y)*magFactor)-magCenterY;
   atx = imageCanvas.getTransform();
   atx.setToTranslation(-x, -y);
   atx.scale(magFactor, magFactor);
   applyTransform(atx);
}

public void paintImage(int magCenterX, int magCenterY, double mag){
   setMagFactor(this.magFactor *mag);
   int dx = this.magCenterX -magCenterX;
   int dy = this.magCenterY-magCenterY;
   this.magCenterX = magCenterX;
   this.magCenterY = magCenterY;
   try {
     Point2D mgp = null;
     atx = imageCanvas.getTransform();
     mgp =
     atx.inverseTransform((Point2D)(new Point(magCenterX, magCenterY)),
                                               (Point2D)mgp);
     double x = (mgp.getX()*mag)-mgp.getX();
     double y = (mgp.getY()*mag)-mgp.getY();
     scale(-x,-y, mag);
   }catch (Exception e) {System.out.println(e); }
}

public void scale(double magOffsetX, double magOffsetY, double mag){
   atx.translate(magOffsetX,magOffsetY);
   atx.scale(mag,mag);
   applyTransform(atx);
}
```

```
public void resetAndScale(double magOffsetX, double magOffsetY, double mag){
    atx.setToTranslation(magOffsetX,magOffsetY);
    atx.scale(mag,mag);
    applyTransform(atx);
}

public void applyTransform(AffineTransform atx) {
    imageCanvas.applyTransform(atx);
}

public void reset() {
    magCenterX = 0;
    magCenterY = 0;
    magFactor = 1.0;
}
}
```

The `paintImage()` method is the key method for implementation of the zoom feature. The `mag` input parameter holds the zoom increment or decrement value. Just as with the `scroll()` method in the scroll feature, `paintImage()` converts the point at which the mouse is clicked to the image space by using the inverse transformation. Once that point has been obtained, `paintImage()` computes the amount of translation that needs to be applied. As Figure 7.8 shows, translations in the x and y directions are calculated as follows:

$$T_x = (\text{mag} \times x) - x$$

$$T_y = (\text{mag} \times y) - y$$

The `scale()` method takes these translations and the `mag` increment value as inputs.

Scaling

The `scale()` method of the `Zoom` class does the uniform scaling. The scaling is performed at a reference point (the magnification center) whose coordinates are specified by `magCenterX` and `magCenterY`. The `scale()` method of the `AffineTransform` class performs scaling with the origin as the reference point. In most applications, however, the reference point is somewhere in the viewport. This can be a point where the mouse is clicked, or it can be the center of a rectangle that was drawn over the image. To achieve scaling at a reference point, the image is translated by a certain amount. So the `scale()` method of the `Zoom` class first translates the image to the reference point specified by (`magOffsetX`, `magOffsetY`) and then scales it by `mag.`

The `resetAndScale()` method performs absolute scaling. It resets `atx` and translates the images to (`magOffsetX`, `magOffsetY`) and then applies scaling.
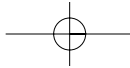
User Interface for Zoom

Let's design a GUI interface that zooms an image in or out at a position where the mouse is clicked. The zoom operator has two modes: zoom in and zoom out. If the zoom-in mode is selected, the magnification of the image increases every time the user clicks the mouse. If the zoom-out mode is selected, the opposite happens.

The ZoomGUI class implements the MouseListener interface. That means it captures only mouse events, and not mouseMotion events. Listing 7.11 shows the code for ZoomGUI.

LISTING 7.11 The ZoomGUI class

```java
package com.vistech.imageviewer;
import java.io.*;
import java.awt.*;
import javax.swing.*;

public class ZoomGUI implements MouseListener{
    protected ZoomController zoomController;
    protected final static double baseZoomFactor = 1.0;
    protected boolean zoomOn = false;
    protected double zoomFactor = 1.0;
    protected double increment = 0.1;
    protected boolean zoomOut = false;
    protected boolean mousePressed = false;

    public ZoomGUI(ZoomController c){ zoomController = c;}

    public  void setZoomOn(boolean onOff){zoomOn = onOff;}
    public  boolean getZoomOn(){ return zoomOn;}
    public  void setZoomOut(boolean outIn){zoomOut = outIn;}
    public  boolean getZoomOut(){ return zoomOut; }
    public void setZoomfactor(double mag){ zoomFactor = mag; }
    public double getZoomFactor(){ return zoomFactor; }
    public void setZoomIncrement(double incr){ increment = incr; }
    public double getZoomIncrement(){ return increment;}

    public void zoom(int x, int y, double zoomfact){
       if(zoomOut) {
          zoomController.paintImage(x,y,baseZoomFactor-increment);
          zoomFactor *= baseZoomFactor-increment;
       }
       else {
          zoomController.paintImage(x,y,baseZoomFactor+increment);
          zoomFactor *= baseZoomFactor+increment;
       }
    }
    public void reset() {
       setZoomOn(false);
       zoomFactor = 1.0;
    }
    public void mousePressed(MouseEvent e) {
       if(mousePressed) return;
       mousePressed = true;
       if(!zoomOn) return;
```

```
        if(SwingUtilities.isLeftMouseButton(e)){
           zoom(e.getX(), e.getY(), zoomFactor);
        }
    }
    public void mouseReleased(MouseEvent e){ mousePressed = false;}
    public void mouseClicked(MouseEvent e){}
    public void mouseEntered(MouseEvent e){}
    public void mouseExited(MouseEvent e){}
}
```

The properties of the zoom operator include `zoomOn, zoomOut, zoomFactor,` and `zoomIncrement.` When you reset the zoom feature, the `zoomFactor` property starts with 1.0—that is, no magnification. When the mouse is clicked, the `mousePressed()` method calls `zoom(),` which increments or decrements `zoomFactor` depending on the current zoom mode. The `zoom()` method then calls the `paintImage()` method of `ZoomController` and passes the current position of the mouse and `magFactor.`

The following code fragment shows how to use the zoom-related classes:

```
ImageManipulator  imageCanvas = new  ImageCanvas2D();

ZoomController zoom = new Zoom(imageCanvas);
ZoomGUI  zoomUI = new ZoomGUI(zoom);

imageCanvas.addMouseListener(zoomUI);
```

Implementing Pan and Zoom Together

Now let's use both the scroll (pan) and the zoom features to implement a combined pan-zoom operator and use it in the image viewer described in Chapter 6. The `Scroll` and `Zoom` operator classes provide no GUIs. But we need a GUI interface that indicates to `Scroll` and `Zoom` objects when to zoom and when to pan. So let's provide a pop-up menu that is launched by right-clicking on the image. This pop-up menu will have the following three menu items:

1. **Zoom in**
2. **Zoom out**
3. **Pan**

You can choose any of these options at any time.

The `PanZoom` class (see Listing 7.12) combines the pan and zoom features.

LISTING 7.12   The `PanZoom` class

```
package com.vistech.imageviewer;
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

```java
public class PanZoom extends MouseAdapter{
    protected ImageManipulator imageCanvas;
    protected ScrollGUI pan;
    protected Scroll scroll;
    protected ZoomGUI zoomGUI;
    protected Zoom zoom;
    protected boolean panOn,zoomOn, zoomOut, panZoomOn = true;

    public PanZoom(ImageManipulator manip) {
        imageCanvas = manip;
        scroll = new Scroll(imageCanvas);
        pan = new ScrollGUI(scroll);

        zoom = new Zoom(imageCanvas);
        zoomGUI = new ZoomGUI(zoom);
        init();
    }
    protected void init() {
        if(imageCanvas == null) return;
        imageCanvas.addMouseListener(pan);
        imageCanvas.addMouseListener(zoomGUI);
        imageCanvas.addMouseListener(this);
        imageCanvas.addMouseMotionListener(pan);
        panOn = true;
        zoomOn = false;
        zoomOut = false;
    }

    protected void setStates() {
        pan.setScrollOn(panOn);
        zoomGUI.setZoomOn(zoomOn);
        zoomGUI.setZoomOut(zoomOut);
    }
    public void setPanZoomOn(boolean onOrOff){
        if(panZoomOn == onOrOff) return;
        panZoomOn = onOrOff;
        if(panZoomOn){
            imageCanvas.addMouseListener(pan);
            imageCanvas.addMouseListener(zoomGUI);
            imageCanvas.addMouseListener(this);
            imageCanvas.addMouseMotionListener(pan);
        }else {
            imageCanvas.removeMouseListener(pan);
            imageCanvas.removeMouseListener(zoomGUI);
            imageCanvas.removeMouseListener(this);
            imageCanvas.removeMouseMotionListener(pan);
        }
    }
    public boolean getZoomOut(){ return zoomOut; }
    public boolean getZoomOn(){ return zoomOn; }
    public double getZoomFactor(){
        return zoomGUI.getZoomFactor();
    }

    public boolean getPanOn(){ return panOn; }
    public void reset(){
        zoomGUI.reset();
```

```
            pan.reset();
        }
    protected void popupMenu(JComponent comp,int x, int y){
        JPopupMenu jp = new JPopupMenu("");
        jp.setLightWeightPopupEnabled(true);
        comp.add(jp);
        JMenuItem zout = new JMenuItem("Zoom out");
        zout.addActionListener(
            new ActionListener(){
                public void actionPerformed(ActionEvent e){
                    zoomOut = true; zoomOn = true; panOn = false;
                    setStates();
                }
            }
        );
        JMenuItem zin = new JMenuItem("Zoom in");
        zin.addActionListener(
            new ActionListener(){
                public void actionPerformed(ActionEvent e){
                    zoomOut = false; zoomOn = true; panOn = false;
                    setStates();
                }
            }
        );

        JMenuItem pn = new JMenuItem("Pan");
        pn.addActionListener(
            new ActionListener(){
                public void actionPerformed(ActionEvent e){
                    zoomOn = false; panOn = true;
                    setStates();
                }
            }
        );
        jp.add(zin);
        jp.add(zout);
        jp.add(pn);
        jp.show(comp,x,y);
    }

    public void mousePressed(MouseEvent e) {
        if(!SwingUtilities.isLeftMouseButton(e)){
            popupMenu((JComponent)e.getSource(), e.getX(), e.getY());
        }
    }
}
```

The `PanZoom` class extends the `java.awt.events.MouseAdapter` class, which implements the `MouseListener` interface. This interface is required to launch the pop-up menu. The constructor for the `PanZoom` class takes the `ImageManipulator` interface as input. It creates `Pan` and `Zoom` objects by passing the `ImageManipulator` parameter to their constructor and calls the `init()` method. The `init()` method registers the `Pan` object to receive `mouse` and `mouseMotion` events.

When pan mode is selected, the position where the mouse is pressed is the anchor for pan. While the mouse is being dragged, the difference in distance from the

current position to the anchor is computed and passed to the `ImageManipulator` object to paint the image at the new position. The `init()` method registers the `Zoom` object to receive `mouse` events and then sets the default values for the three state variables: `panOn`, `zoomOn`, and `zoomOut`.

The `popupMenu()` method takes *x* and *y* coordinates of the position at which the mouse is clicked as the input parameters and uses them to show the pop-up menu at that location. This method creates three menu items—**Pan, Zoom out,** and **Zoom in**—each of which has an anonymous inner class to handle action events. These `actionPerformed()` methods set the member variables `panOn`, `zoomOn`, and `zoomOut` and call the appropriate `setStates()` methods in the `Pan` and `Zoom` objects, depending on the menu item selected.

The screen shot in Figure 7.10 shows an image viewer that implements the combined pan-zoom feature. In this image viewer, you can right-click on the image to select the operation you want to perform by choosing the appropriate item from the
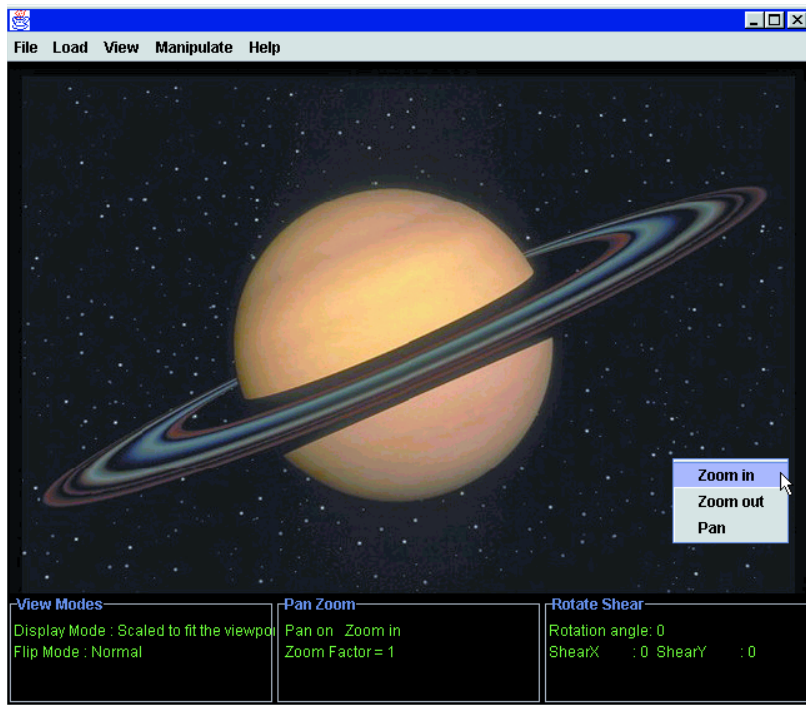
© Antonio M. Rosario/The Image Bank

FIGURE 7.10   An image viewer with the pan-zoom feature

pop-up menu. This image viewer is an extension of the one in Chapter 6. As the figure shows, the status bar displays the image manipulation status messages. Regarding pan-zoom, the status bar tells you which feature is currently active and the current zoom factor.

Now let's implement a special feature called `Lens` using the zoom API.

Implementing a Lens

A moving lens over an image is quite a useful feature in many applications where visual inspection is an important factor in analyzing images. In addition to the `AffineTransform` class, the lens feature, which is implemented by the `Lens` class, uses many aspects of the `Graphics2D` class. Listing 7.13 shows the code for `Lens.`

LISTING 7.13   The `Lens` class

```
package com.vistech.imageviewer;
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Lens implements MouseListener, MouseMotionListener{
   protected ImageManipulator imageCanvas;
   protected Dimension lensSize = new Dimension(60,80);
   protected Point prevPoint = new Point(0,0);
   protected boolean lensOn = false;
   protected int sizeIncrement = 10;
   protected double magIncrement = 0.5;
   protected double lensMag = 2.0;
   protected Zoom zoom;

   public Lens(ImageManipulator c){
      imageCanvas = c;
      zoom = new Zoom(imageCanvas);
   }
   public void init() { setLensOn(true);}

   public void setLensSize(Dimension d){ lensSize = d; }

   public Dimension getLensSize(){ return lensSize; }

   public void setLensMag(double mag){ lensMag = mag; }

   public double getLensMag(){ return lensMag; }

   public void setLensMagIncrement(int incr){ magIncrement = incr; }

   public double getLensMagIncrement(){ return magIncrement; }

   public void setLensSizeIncrement(int incr){ sizeIncrement = incr; }
```
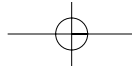
```java
public int getLensSizeIncrement(){ return sizeIncrement; }

public  void setLensOn(boolean onOff){ lensOn = onOff; }

public  boolean getLensOn(){ return lensOn; }

public void drawLens(int x, int y){
   int wid = lensSize.width; int ht = lensSize.height;
   Shape lens = new Ellipse2D.Float(prevPoint.x-wid,prevPoint.y-ht, wid,ht);
   imageCanvas.setClip(lens);
   zoom.magnify(0,0, 1);
   Shape ch = new Ellipse2D.Float(x-wid,y-ht, wid,ht);
   imageCanvas.setClip(ch);
   zoom.magnify((x-wid/2), (y-ht/2), lensMag);

   imageCanvas.draw(ch);
   prevPoint = new Point(x,y);
}

public void incrementLensSize(){
   lensSize.width += sizeIncrement;
   lensSize.height += sizeIncrement;
}
public void decrementLensSize(){
   lensSize.width -= sizeIncrement;
   lensSize.height -= sizeIncrement;
}

public void incrementLensMag(){
   lensMag += magIncrement;
}

public void decrementLensMag(){
    lensMag -= magIncrement;
}

public void reset() { setLensOn(false); }

public void mousePressed(MouseEvent e) {
   if(!lensOn) return;
   if(SwingUtilities.isLeftMouseButton(e)){
      drawLens(e.getX(), e.getY());
    }
}
public void mouseClicked(MouseEvent e){}
public void mouseEntered(MouseEvent e){}
public void mouseExited(MouseEvent e){}

public void mouseReleased(MouseEvent e) {
   if(!lensOn) return;
   imageCanvas.setClip(null);
   zoom.magnify(0,0,1);
}

public void mouseDragged(MouseEvent e){
   if(SwingUtilities.isLeftMouseButton(e)){
      if(lensOn) drawLens(e.getX(), e.getY());
    }
```

```
    }
    public void mouseMoved(MouseEvent e){}
}
```

The `Lens` class constructor takes `ImageManipulator` as an input parameter. This means that we can pass the `ImageCanvas2D` object, which implements the `ImageManipulator` interface, as this parameter.

The `Lens` class implements both the `mouse` and the `mouseMotion` event–handling methods. A third-party object registers `Lens` with `ImageManipulator` to receive these events. Both of these events are needed for dragging the lens over the image.

When the mouse is pressed, a `Lens` object is created at the mouse position, and when the mouse is released the object is destroyed. When the mouse is dragged, the lens is moved. Figure 7.11 shows a screen shot of a lens overlaid on an image.

The principle of drawing the lens is simple. The shape of the lens is oval, so we use the `Ellipse2D` class for drawing it. Before the lens is drawn, the graphical context is clipped over a region covering the lens. For this reason, the same shape that is
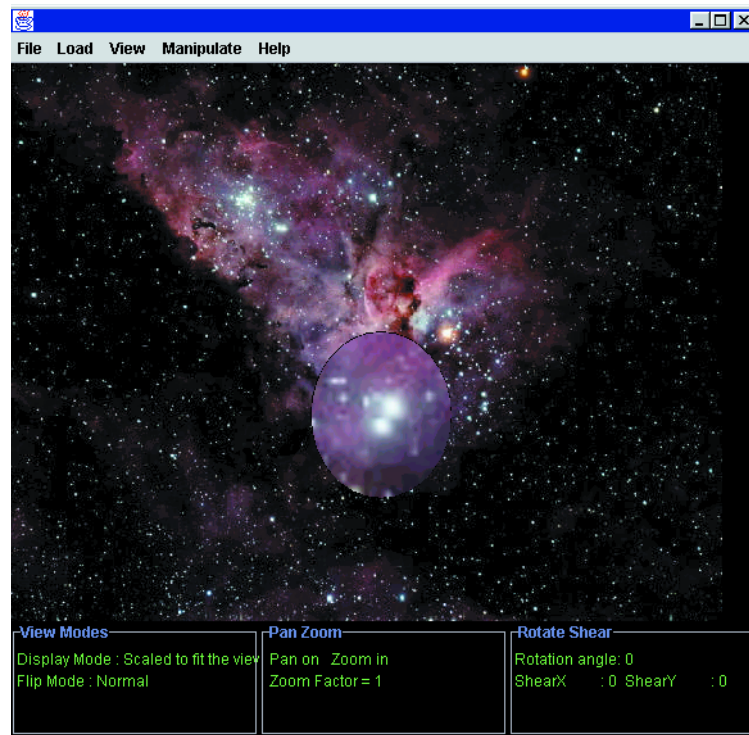
© World Perspectives/Stone

FIGURE 7.11   An interactive lens over an image

used for creating the lens is also used for clipping the graphical context. The image is then magnified and is drawn over the entire viewport. Because the graphical context is clipped, the image is drawn only over the region where the clipping is in effect.

When you drag the mouse, the lens is drawn at the previous position to erase that image. Recall from Chapter 5 that the XOR paint mode erases the graphical object if it is drawn twice. Drawing the original image at the previous clip area does exactly this. When the mouse is released, the clip region is reset, by the passing of `null` to the `setClip()` method.

The `Lens` class has several properties, including lens size and magnification factor. These properties allow the user to set the desirable lens and magnification attributes. Figure 7.12 shows a screen shot of a simple panel we created to add to our image viewer. When the `Lens` feature is invoked, this panel is launched. Listing 7.14 shows the code for the `LensPanel` class.

LISTING 7.14   The `LensPanel` class

```
public class LensPanel extends JPanel {
   protected ImageManipulator imageCanvas;
   protected Lens lens;
   protected boolean lensOn = false;
   public LensPanel(ImageManipulator manip) {
      imageCanvas = manip;
      if(manip != null) {
         manip.resetManipulation();
         lens = new Lens(manip);
         createUI();
      }
   }

   public void setLensOn(boolean onOrOff){
      if(lensOn ==  onOrOff) return;
      lensOn = onOrOff;
      if(lensOn) {
         imageCanvas.addMouseListener(lens);
         imageCanvas.addMouseMotionListener(lens);
      } else {
         imageCanvas.removeMouseListener(lens);
         imageCanvas.removeMouseMotionListener(lens);
      }
      lens.setLensOn(lensOn);
   }
```
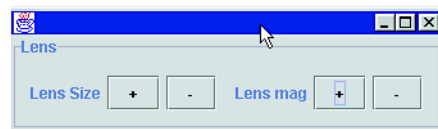

FIGURE 7.12   The **Lens** panel
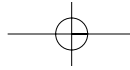
```
    private void createUI() {
        // Code that creates lens panel
    }

}
```

In our image viewer we added a new menu called **Manipulate** for selecting ma-nipulation-related features. To invoke the lens feature, select this menu and click on **Lens.** The panel shown in Figure 7.12 will appear. Invoking `Lens` will reset all the manipulation operations but will preserve the display mode and flip mode settings.

Other Zooming Techniques

If your application requires a simple zoom feature, you can try other ways of zoom-ing. We already mentioned one technique in Chapter 6: manipulating `drawImage()` parameters.
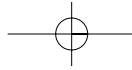
Using the Image Class

You need to use the following method to create a scaled instance of an image:

♦ **public Image getScaledInstance(int width, int height, int hints)**

Set the `width` and `height` parameters to suit the scale factor. If S_x is the desired scale factor in the x direction (magnified image width divided by original image width), then magnified width $= S_x \times$ image width. Likewise, if S_y is the desired scale factor in the y direction, then magnified height $= S_y \times$ image height.

The `hints` parameter is similar to the `renderingHints` attribute we saw in Chapter 5. This parameter can take any one of the following values:

- ♦ **SCALE_DEFAULT.** With this option the default scaling algorithm is used.
- ♦ **SCALE_FAST.** This option produces the image fast rather than smoothly. In this case scaling might use the nearest-neighbor algorithm.
- ♦ **SCALE_SMOOTH.** This option produces a smoother scaled image. Here the choice is image quality rather than speed. In this case the scaling algorithm used could be bilinear or bicubic, depending on the availability of the implementation in a given platform.
- ♦ **SCALE_REPLICATE.** With this option the scaling algorithm provided in the `ReplicateScaleFilter` class is used.
- ♦ **SCALE_AREA_AVERAGING.** With this option the area-averaging scaling algorithm provided in the `AreaAverageScaleFilter` class is used.

Using Filters

Two filter classes are available for scaling: `ReplicateScaleFilter` and `AreaAverage ScaleFilter.` Listing 7.15 shows an example using the latter:

LISTING 7.15   Scaling using filters

```
public Image scale(Image image, double magfactor) {
    int imageWidth = image.getWidth(imObs);
    int imageHeight = image.getHeight(imObs);
    AreaAverageScaleFilter scalefilter =
        new AreaAverageScaleFilter(imageWidth*magfactor,
                            imageHeight*magfactor);
    ImageProducer ip =
        new FilteredImageSource(image.getSource(), scalefilter);
    return Toolkit.getDefaultToolkit().createImage(ip);
}
```

If you want to obtain the width and height of the original image, the input for the method shown in Listing 7.15 should pertain to an already loaded image.

Rotation, Flip, and Shear

Now that we have described the pan and zoom features in detail, it's time to look at other manipulation functions, which include rotate, flip, and shear. For the sake of convenience, instead of building individual interfaces for each of these functions, we'll build a common interface called `GeomManipController` for all of them. Again, we'll use the same design pattern we used for scroll and zoom. As mentioned earlier, with this design pattern the GUI is separated from control logic. Let's look at the manipulation operations one by one.

Rotation

Rotation may not be used as frequently as pan and zoom. Nevertheless, it is important to have it in your image viewer. Typically, rotation is performed around the midpoint of the image, but our interface can provide a method that specifies the rotation center. First, let's specify the get and set methods for the rotation angle property:

- **public double getRotationAngle();**
- **public void setRotationAngle(double theta);**

Here are the methods that perform rotation:

- **public void rotate(double theta);**
- **public void rotate(double theta, int rotCenterX, int rotCenterY);**
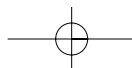
The first method rotates the image around the midpoint of the image. The second one rotates the image around a point specified by the input parameters.

Flip

We discussed the fundamentals of flip in Chapter 4. The method for flipping an image at any position or at any state is

◆ `public void flip(int flipMode)`

The `flipMode` parameter can be any one of the four values defined in the `FlipMode` class (see Chapter 6).

Shear

Just as we specified the `magFactor` property for the scaling operation, for shear we'll specify the `shearFactor` property. The get and set methods for this property are

◆ `public double getShearFactor();`
◆ `public void setShearFactor(double shear);`

The method for shearing an image at any stage of the manipulation is

◆ `public void shear(double shx, double shy);`

The `shx` and `shy` parameters are the shear factors in the x and y directions, respectively.

Implementing Rotation, Flip, and Shear

Let's develop the `GeomManip` class in a manner similar to what we did for pan and zoom. We'll build the controller class first and then the GUI class. Here are the two classes:

1. **GeomManip.** This class implements the `GeomManipController` interface.
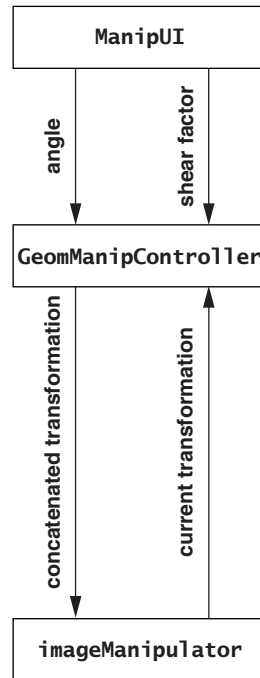2. **ManipUI.** This class implements the slider interfaces for adjusting the rotate and shear values.

Figure 7.13 shows the model-view-controller architecture for the rotation and shear features, which resembles the design for scroll and for zoom. Listing 7.16 shows the code for `GeomManip.`

LISTING 7.16   The GeomManip class

```
package com.vistech.imageviewer;
import java.io.*;
import java.awt.*;
```

```
                    ┌─────────────────────┐
                    │      ManipUI        │
                    └─────────────────────┘
                      │                 │
                   angle          shear factor
                      ▼                 ▼
                    ┌─────────────────────┐
                    │ GeomManipController │
                    └─────────────────────┘
                      │                 ▲
            concatenated            current
            transformation       transformation
                      ▼                 │
                    ┌─────────────────────┐
                    │   imageManipulator  │
                    └─────────────────────┘
```

FIGURE 7.13   The data flow for rotation and shear

```java
import java.awt.event.*;
import java.awt.image.*;
import java.awt.geom.*;

public class GeomManip  implements GeomManipController  {
   protected AffineTransform atx = new AffineTransform();
   // Rotation variables
   protected double rotationAngle = 0.0;
   protected boolean rotateOn = true;
   protected int rotationCenterX = 0;
   protected int rotationCenterY = 0;
   // Shear variables
   protected boolean shearOn = true;
   protected double shearFactor = 0.0;
   protected double shearX =0.0, shearY=0.0;
   protected ImageManipulator imageCanvas;
   protected int flipMode = 0;

   public GeomManip(){}
   public GeomManip(ImageManipulator imageCanvas){ this.imageCanvas = imageCanvas;}

   public void setImageManipulator(ImageManipulator imageCanvas){
      this.imageCanvas = imageCanvas;
   }
```

```
public synchronized void setFlipMode(int mode){
   if(mode == flipMode) return;
   int oldmode = flipMode;
   flipMode = mode;
}

public int getFlipMode(){ return flipMode;}

public void setShearFactor(double shearFactor){
   this.shearFactor = shearFactor;
   imageCanvas.setShearFactor(shearFactor);
}
public double getShearFactor(){ return shearFactor;}

public double getShearFactorX(){ return shearX;}
public double getShearFactorY(){return shearY;}

public void setRotationAngle(double rotationAngle){
   this.rotationAngle = rotationAngle;
   imageCanvas.setRotationAngle(rotationAngle);
}

public double getRotationAngle(){ return rotationAngle;}

public void rotate(double theta){
   double ang = this.rotationAngle -theta;
   Dimension dim = imageCanvas.getImageSize();
   int wid = dim.width;
   int ht  = dim.height;
   setRotationAngle(theta);
   atx = imageCanvas.getTransform();
   atx.rotate(ang, wid/2, ht/2);
   imageCanvas.applyTransform(atx);
}

public void rotate(double theta, int rotCenterX, int rotCenterY){
   double ang = this.rotationAngle -theta;
   setRotationAngle(theta);
   atx = imageCanvas.getTransform();
   atx.rotate(ang, rotCenterX, rotCenterY);
   imageCanvas.applyTransform(atx);
}

public void resetAndRotate(double theta) {
   BufferedImage image = imageCanvas.getOffScreenImage();
   int wid = image.getWidth();
   int ht  = image.getHeight();
   setRotationAngle(theta);
   atx.setToRotation(theta, wid/2, ht/2);
   imageCanvas.applyTransform(atx);
}

public void shear(double shx, double shy){
   double shxIncr = shearX -shx;
   double shyIncr = shearY -shy;
   setShearFactor(shx);
```

continued

```
        this.shearX = shx;
        this.shearY = shy;
        atx = imageCanvas.getTransform();
        atx.shear(shxIncr, shyIncr);
        imageCanvas.applyTransform(atx);
    }

    public void shearIncr(double shxIncr, double shyIncr){
        shearX += shxIncr;
        shearY += shyIncr;
        setShearFactor(shearX);
        atx.shear(shxIncr, shyIncr);
        imageCanvas.applyTransform(atx);
    }

    public void resetAndShear(double shx, double shy){
        shearX =shx; shearY = shy;
        atx.setToShear(shx,shy);
        setShearFactor(shearX);
        imageCanvas.applyTransform(atx);
    }

    public static AffineTransform createFlipTransform(int mode,
                                              int imageWid,
                                              int imageHt){
        AffineTransform at = new AffineTransform();
        switch(mode){
            case FlipMode.NORMAL:
              break;
            case FlipMode.TOP_BOTTOM:
              at =  new AffineTransform(new double[] {1.0,0.0,0.0,-1.0});
              at.translate(0.0, -imageHt);
              break;
            case FlipMode.LEFT_RIGHT :
              at = new AffineTransform(new double[] {-1.0,0.0,0.0,1.0});
              at.translate(-imageWid, 0.0);
              break;
            case FlipMode.TOP_BOTTOM_LEFT_RIGHT:
              at = new AffineTransform(new double[] {-1.0,0.0,0.0,-1.0});
              at.translate(-imageWid, -imageHt);
              break;
            default:
        }
        return at;
    }

    public void flip(int mode){
        Dimension dim = imageCanvas.getImageSize();
        int wid = dim.width;
        int ht  = dim.height;
        AffineTransform flipTx = createFlipTransform(mode,wid,ht);
        atx = imageCanvas.getTransform();
        atx.concatenate(flipTx);
        imageCanvas.applyTransform(atx);
    }

    public void resetAndFlip(int mode){
        atx = new AffineTransform();
```
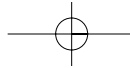
```
        flip(mode);
    }

    public void resetManipulation(){
        shearX = 0.0; shearY = 0.0;
        rotationAngle = 0.0;
        atx = new AffineTransform();
    }
}
```

Typically, images are rotated about their midpoint. In some situations, however, an image can be rotated around any arbitrary point. The `rotate()` methods in the `GeomManip` class meet both of these requirements.

The `rotate(theta)` method rotates the image about its midpoint, irrespective of where the image is positioned in the viewport. The input parameter `theta` is the angle of rotation from the original position of the image. When this method is called, the image might have been rotated already. This method therefore computes the difference between current rotation angle and the input. Then it gets the midpoint of the current image on the canvas. It then calls the `rotate()` method of the `AffineTransform` class with the difference in angles and the center point of the image as its inputs.

The `rotate(theta, rotCenterX, rotCenterY)` method rotates the image about any arbitrary point. The `resetAndRotate()` method implements absolute rotation about the midpoint. The `setToRotation()` method resets `atx` and rotates it by a fixed amount.

The shear methods are similar to the scale and rotate methods.

The `AffineTransform` class doesn't have an explicit flip transformation. We discussed how to implement flip in the preceding section and defined a method called `createFlipTransform()`. The `flip()` method uses `createFlipTransform()` to flip the image at any stage of image manipulation. Note that here we use the `concatenate()` method to concatenate the flip transformation to the current transformation `atx`. The `resetAndFlip()` method resets the current transformation and flips the image in the direction specified in the input.

User Interface for Rotation and Shear

Implementing rotation and shear is straightforward. We can directly use the transformation methods for rotation and shear in the `GeomManip` class. But first we need a GUI to operate on the `GeomManip` class. Figure 7.14 shows a screen shot of a panel that has two tabs: **Rotate** and **Shear.** To launch this panel, select the **Rotate/Shear** option in the **Manipulate** menu. You can either use the slider or enter a value in the text field. Notice that in the **Rotate** tab, the angles vary from –360 to +360, thereby enabling rotation in the clockwise or counterclockwise direction. When you move the slider, the text field reflects the slider's current value.
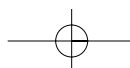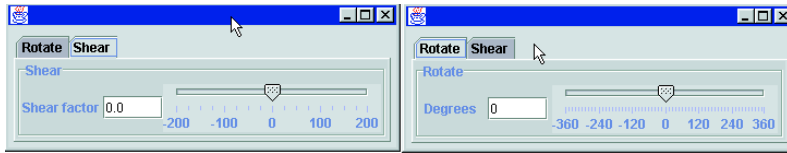
FIGURE 7.14   The **Rotate** and **Shear** panel

We'll provide only some code snippets because implementing the GUI involves a large chunk of code. Listing 7.17 shows the code for `ManipUI`.

LISTING 7.17   The `ManipUI` class

```java
public class ManipUI extends JPanel{
    protected JTabbedPane jtp;
    protected  int startValue = 0;
    protected  int minValue = -360, maxValue = 360;
    protected  JTextField rotateValueField;
    protected  double rotateValue = 0.0;
    protected  int rotateStartValue = 0;
    protected  JSlider rotateSlider;

    protected int shearMinValue = -200;
    protected int shearMaxValue = 200;
    protected JTextField shearValueField;
    protected int shearValue = 0;
    protected int shearStartValue = 0;
    protected JSlider shearSlider;
    protected GeomManipController imageViewer;

    public ManipUI(GeomManipController manip) {
        imageViewer = manip;
        createUI();
    }

    private void createUI() {
        JPanel rpanel = createRotatePanel();
        JPanel spanel = createShearPanel();
        jtp = new JTabbedPane();
        jtp.addTab("Rotate", rpanel);
        jtp.addTab("Shear", spanel);
        add(jtp);
    }

    protected JPanel createRotatePanel() {
        JButton resetButton = new JButton("Reset Rotate");
        resetButton.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent e){
                    rotateSlider.setValue(rotateStartValue);
                    rotateValueField.setText((new Double(rotateStartValue)).toString());
                    imageViewer.rotate(0.0);
```

```java
                }
            }
        );

        rotateValueField = new JTextField(5);
        rotateValueField.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent e){
                    try {
                        String str = ((JTextField)e.getSource()).getText();
                        rotateValue = (Double.valueOf(str)).doubleValue();

                        rotateSlider.setValue((int)rotateValue);
                        imageViewer.rotate(rotateValue*(Math.PI/180.0));
                    } catch (Exception e1){}
                }
            }
        );

        rotateValueField.setText(Integer.toString(rotateStartValue));
        JLabel rotateLabel = new JLabel("Rotate");

        rotateSlider = new JSlider(minValue,maxValue,startValue);
        rotateSlider.setMajorTickSpacing(120);
        rotateSlider.setMinorTickSpacing(12);
        rotateSlider.setExtent(12);
        rotateSlider.setPaintTicks(true);
        rotateSlider.setPaintLabels(true);

        rotateSlider.addChangeListener(
            new ChangeListener() {
                public void stateChanged(ChangeEvent e){
                    double rotateValueOld = rotateValue;
                    rotateValue = ((JSlider)(e.getSource())).getValue();
                    imageViewer.rotate(rotateValue*(Math.PI/180.0));
                    rotateValueField.setText(Integer.toString((int)rotateValue));
                }
            }
        );

        JPanel rotatepan = new JPanel();
       // Grid layout
      return rotatepan;
    }

    protected JPanel createShearPanel() {

        // Code similar to createRotatePanel
    }

    public void resetManipulation(){
        shearSlider.setValue((int)(shearStartValue*100));
        shearValueField.setText(Double.toString(shearStartValue));
        rotateValueField.setText(Integer.toString((int)rotateStartValue));
        rotateSlider.setValue((int)rotateStartValue);
    }
}
```

Most of the code in Listing 7.17 is self-explanatory. The `rotateSlider` event–handling method calls the `rotate()` method with the current `rotateSlider` value. Likewise, the `shearSlider` event–handling method calls the `shear()` method with the current `shearSlider` value.

The code snippets that follow, which are from the `ImageManip2D` application, show how to use the `GeomManip` and `ManipUI` classes in an application:

```java
ImageManipulator   viewer = new ImageCanvas2D();
GeomManipController manip = new GeomManip(viewer);


 rot.addActionListener(
    new ActionListener() {
      public void actionPerformed(ActionEvent e){
        if(manipFrame == null) {
           manipFrame = new JFrame();
           ManipUI manipui = new ManipUI(manip);
           manipFrame.getContentPane().add(manipui);
           manipFrame.pack();
        }
        manipFrame.show();
      }
    }
 );
```
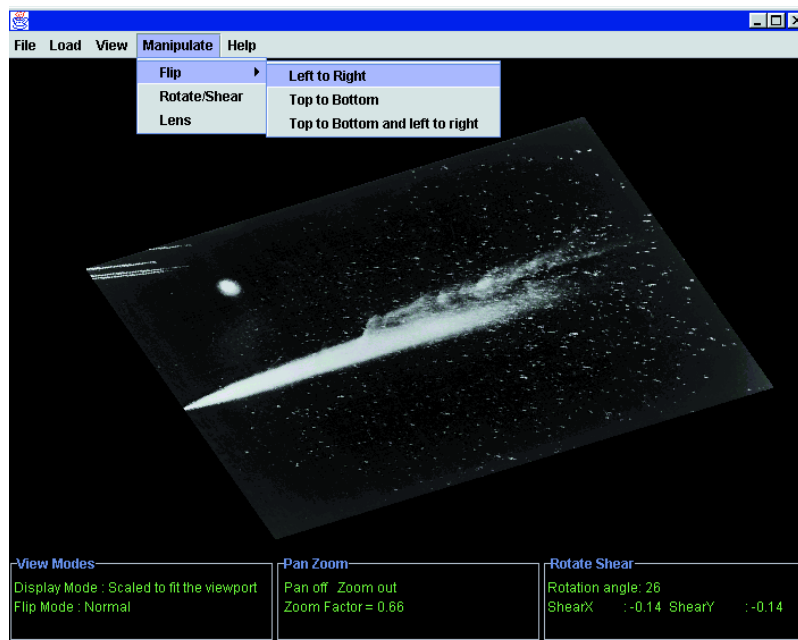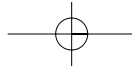

FIGURE 7.15   Performing pan, zoom, flip, rotate, and shear together

The `rot` parameter represents the **Rotate/Shear** option on the **Manipulate** menu. The code snippet is the event handler that launches the frame shown in Figure 7.14.

The screen shot in Figure 7.15 shows the image viewer when pan, zoom, rotate, and shear have all been performed. Besides the **Lens** and **Rotate/Shear** options, the **Manipulate** menu also has the option **Flip.**

Running the Image Viewer

To run the `ImageManip2D` application, type "java app.ImageManip2D" on the command line. When the application frame comes up, select **Load | List Select** to launch the `MultiImageLoader` bean. When this bean is launched, you can either enter the desired directory or click on the **Browse** option. Once you have selected an image, click on **Load Image** to display it.

To pan and zoom the displayed image, right-click on the viewport to launch a pop-up menu with three items: **Pan, Zoom in,** and **Zoom out** (see Figure 7.10). If you select **Pan,** drag the mouse to pan the image. If you select **Zoom in** or **Zoom out,** position the cursor at a desired position and click. If **Zoom in** is selected, the displayed image will be magnified with the position of the mouse as the center of the image. Likewise, **Zoom out** will shrink the displayed image whenever you click on the image.

To rotate and shear the image, select the **Manipulate** menu, which has three menu items: **Flip, Rotate/Shear,** and **Lens** (see Figure 7.15). If you select **Lens,** the image viewer application will launch a **Lens** panel as shown in Figure 7.12. Note that the lens feature resets all manipulations. To use the lens feature, drag the mouse over the image. You can change the size of the lens with the options in the **Lens** panel.
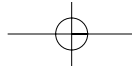
If you select **Rotate/Shear,** the panel shown in Figure 7.14 will be launched. To rotate the image, select the **Rotate** tab and adjust the slider. You can see the image rotating at its midpoint. Likewise, to shear the image, adjust the **Shear** slider.

The **Flip** menu item has three subitems: **Left to Right, Top to Bottom,** and **Top to Bottom and left to right** (see Figure 7.15). Select any one of these options to flip the image in the desired direction.

To reset manipulation, select **View | Display Mode** and click on the desired display mode. This action will move the image to the default position with the selected display mode.

Conclusion

In this chapter we discussed how to implement different types of image manipulation operations. Implementation of manipulation functions is much easier with the

`AffineTransform` class, which allows the functions to be combined in any order without much coding.

The image viewer application that we built in this chapter incorporates different types of image manipulations. While running these operations, you may have noticed that performance is not at all a problem.

With the design described in this chapter, the manipulation operations are built like building blocks. Applications need to construct only the functions they require at a given time. In addition, the manipulation logic is separated from the GUI, allowing application developers to choose any GUI they like for generating image manipulation parameters. As we'll see in Part III (the JAI chapters), the manipulation classes described in this chapter are reused in other applications.