8

Image Processing

IMAGE-PROCESSING tools such as Adobe Photoshop and The GIMP offer a wide variety of filters you can apply on your pictures to create various special effects (see Figure 8-1). When you are designing a user interface, it is very tempting to use those effects. For instance, you could use a filter to blur an out-of-focus element in the UI. You could also increase the brightness of an image as the user moves the mouse over a component.

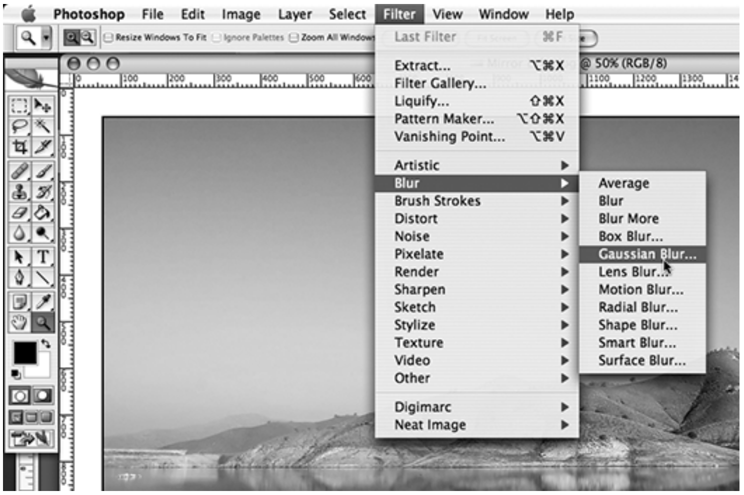

Figure 8-1 Applications like Adobe Photoshop have advanced image-processing capabilities.

Image Filters

Despite the impressive-looking results, image processing is not a difficult task to implement. Processing an image, or applying a filter, is just a matter of calculating a new color for each pixel of a source image. The information required to compute the new pixels varies greatly from one filter to another. Some filters, a grayscale filter for instance, need only the current color of a pixel; other filters, such as a sharpening filter, may also need the color of the surrounding pixels; still other filters, such as a rotation filter, may need additional parameters.

Since the introduction of Java 2D in J2SE 1.2, Java programmers have access to a straightforward image-processing model. You might have learned or read about the old producer-consumer model of Java 1.1. If you did, forget everything you know about it because the new model is much easier and more versatile. Java 2D's image-processing model revolves around the `java.awt.image.BufferedImage` class and the `java.awt.image.BufferedImageOp` interface.

A `BufferedImageOp` implementation takes a `BufferedImage` as input, called the source, and outputs another `BufferedImage`, called the destination, which is altered according to specific rules. Figure 8-2 shows how a blur filter produces the final image.

While the JDK does not offer concrete image filters, it does provide the foundations for you to create your own. If you need a sharpening or blurring filter, for example, you must know how to provide parameters to a `ConvolveOp` filter. We teach you such techniques in this chapter. Before we delve further into image-processing theory, let's see how we can use a `BufferedImageOp` to process an image.

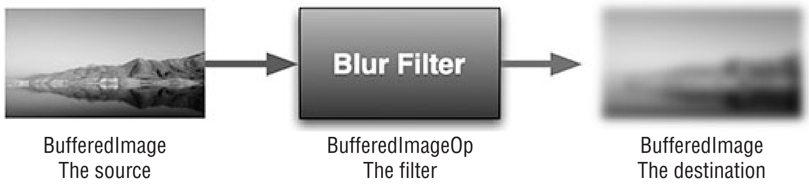

Figure 8-2 Filtering an image with Java 2D.

Processing an Image with BufferedImageOp

Filtering a `BufferedImage` can be done onscreen at painting time or offscreen. In both cases, you need a source image and an operation, an instance of `BufferedImageOp`. Processing the image at painting time is the easiest approach; here is how you might do it:

```
// createImageOp returns a useful image filter
BufferedImageOp op = createImageOp();
// loadSourceImage returns a valid image
BufferedImage sourceImage = loadSourceImage();

@Override
protected void paintComponent(Graphics g) {
  Graphics2D g2 = (Graphics2D) g;
  // Filter the image with a BufferedImageOp, then draw it
  g2.drawImage(sourceImage, op, 0, 0);
}
```

You can filter an image at painting time by invoking the `drawImage(BufferedImage, BufferedImageOp, int, int)` method in `Graphics2D` that filters the source image and draws it at the specified location.

Warning: Use Image Filters with Care. The `drawImage(BufferedImage, BufferedImageOp, int, int)` method is very convenient but often has poor runtime performance. An image filter is likely to perform at least a few operations for every pixel in the source image, which easily results in hundreds of thousands, or even millions, of operations on medium or large images. Besides, this method might have to create a temporary image, which takes time and memory. For every filter you want to use, you will have to see whether the runtime performance is acceptable or not.

Here is an example of how to preprocess an image by doing all the operations offscreen:

```
BufferedImageOp op = createImageOp();
BufferedImage sourceImage = loadSourceImage();
BufferedImage destination;

destination = op.filter(sourceImage, null);
```

Calling the `filter()` method on a `BufferedImageOp` triggers the processing of the source image and the generation of the destination image. The second parameter, set to null here, is actually the destination image, which, when set to null, tells the `filter()` method to create a new image of the appropriate size. You can, instead, pass a non-null `BufferedImage` object as this parameter to avoid creating a new one on each invocation. Doing so can save performance by reducing costly image creations.

The following code example shows how you can optimize a routine applying the same filter on several images of the same size:

```
BufferedImageOp op = createImageOp();
BufferedImage[] sourceImagesArray = loadImages();
BufferedImage destination = null;

for (BufferedImage sourceImage : sourceImagesArray) {
  // on the first pass, destination is null
  // so we need to retrieve the reference to
  // the newly created BufferedImage
  destination = op.filter(sourceImage, destination);
  saveImage(destination);
}
```

After the first pass in the loop, the destination will be non-null and `filter()` will not create a new `BufferedImage` when invoked. By doing so, we also make sure that the destination is in a format optimized for the filter, as it is created by the filter itself.

Processing an image with Java 2D is an easy task. No matter which method you choose, you will need to write only one line of code. But we haven't seen any concrete `BufferedImageOp` yet and have just used an imaginary `createImageOp()` method that was supposedly returning a useful filter. As of Java SE 6, the JDK contains five implementations of `BufferedImageOp` we can rely on to write our own filters: `AffineTransformOp`, `ColorConvertOp`, `ConvolveOp`, `LookupOp`, and `RescaleOp`.

You can also write your own implementation of `BufferedImageOp` from scratch if the JDK does not fulfill your needs. Before learning how to write your own, let's take a closer look at what the JDK has to offer. Each filter we investigate will be applied to the sample picture shown in Figure 8-3 to give you a better idea of the result.

ONLINE DEMO The complete source code for all the examples can be found on this book's Web site in the project named `ImageOps`.

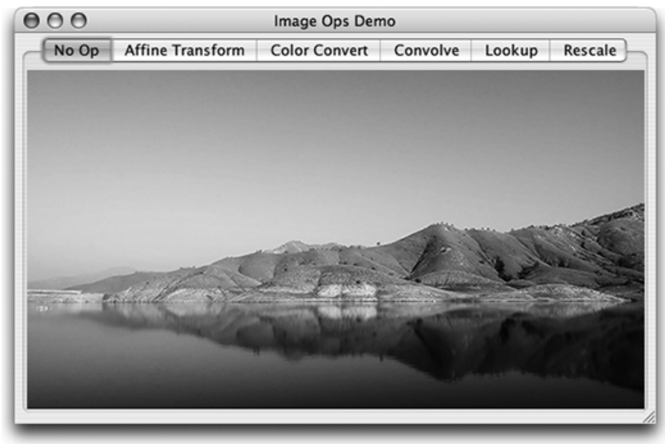

Figure 8-3 The image used in our filter examples.

AffineTransformOp

An `AffineTransformOp` is a geometry filter. It does not work on the actual color of the pixels but on the shape of the picture. As its name suggests, however, it is not meant to perform any kind of geometry transformation. Instead, it is limited to linear mapping from 2D coordinates in the source image to 2D coordinates in the destination image.

This kind of filter is created with an `AffineTransform` instance, which you should be familiar with if you have worked with the `Graphics2D` class (this class is also discussed in Chapter 3, "Graphics Fundamentals"). An `AffineTransform` can be used to rotate, scale, translate, and shear objects in a 2D space.

The following code illustrates how to divide the size of an image by two using an `AffineTransformOp`:

```
BufferedImage dstImage = null;
AffineTransform transform =
    AffineTransform.getScaleInstance(0.5, 0.5);
AffineTransformOp op = new AffineTransformOp(transform,
    AffineTransformOp.TYPE_BILINEAR);
dstImage = op.filter(sourceImage, null);
```

The `AffineTransformOp` constructor used in this example takes two parameters: an `AffineTransform`, in this case a scale operation of 50 percent on both axes,

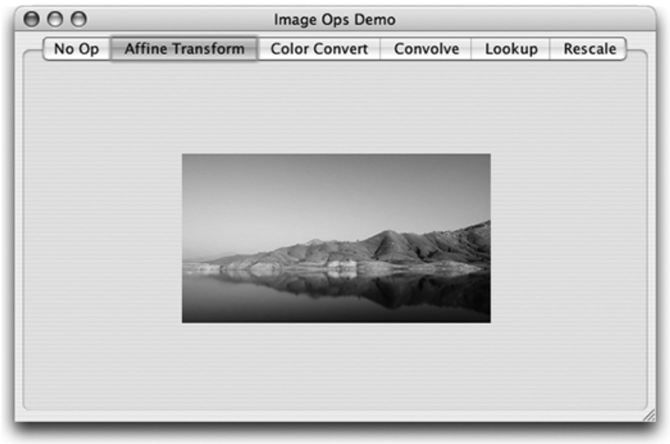

Figure 8-4 The size of the original image is reduced by 50 percent with an `AffineTransformOp`.

and an interpolation type, which is equivalent to the interpolation rendering hint you can find in the `RenderingHints` class (see Chapter 3). You can also pass a `RenderingHints` instance instead of the interpolation type, in which case the interpolation rendering hint will be used.

Figure 8-4 shows the result of our scaling operation.

ColorConvertOp

This `BufferedImageOp` implementation performs a pixel-by-pixel color conversion of the source image into the destination image. This particular image-processing operation has an interesting feature: It transforms a given pixel from one color model to another. To do this, the filter needs the color value of only this single pixel, which means that it is possible to use the same image as both the source and the destination.

Converting an image from one color model to another has little practical use if you are not building an advanced imaging tool. And it is definitely useless if terms like "CMYK," "sRGB," and "Adobe RGB 1998 color profile" mean nothing to you. Color spaces are very useful, but describing them and their applications goes way beyond the scope of this book. Even so, we can use a `ColorConvertOp` to create something more basic and potentially useful to us: a grayscale version of a source image.

You first need to create a `ColorSpace` instance that represents the color model to which you want to convert your image. A `ColorSpace` can be instantiated by invoking `ColorSpace.getInstance(int)` and passing one of the five following constants:

```
ColorSpace.CS_CIEXYZ
ColorSpace.CS_GRAY
ColorSpace.CS_LINEAR_RGB
ColorSpace.CS_PYCC
ColorSpace.CS_sRGB
```

You might have already guessed which one is best suited to our purpose of performing a grayscale conversion:

```
BufferedImage dstImage = null;
ColorSpace colorSpace = ColorSpace.getInstance(
    ColorSpace.CS_GRAY);
ColorConvertOp op = new ColorConvertOp(colorSpace, null);
dstImage = op.filter(sourceImage, null);
```

Similar to `AffineTransformOp`, `ColorConvertOp` can use a set of `RenderingHints` to control the quality of the color conversion and the dithering. Figure 8-5 shows the result of the our color conversion.

Last but not least, it is important to know that performing such a conversion on an image may make it incompatible with your graphics display hardware, thus hurting the performance if you need to paint the filtered image to the Swing window.

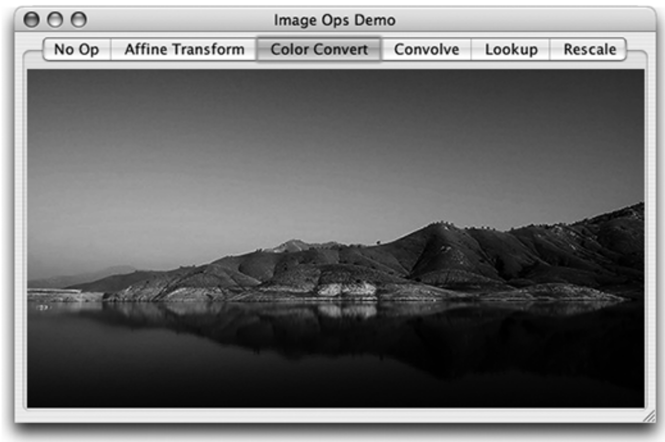

Figure 8-5 `ColorConvertOp` can be used to create a grayscale version of an image.

You may want to convert the image to be a compatible image instead for general usage. See Chapter 3 for more information.

ConvolveOp

The `ConvolveOp` is the most complicated `BufferedImageOp` but also the most versatile. It is the only `BufferedImageOp` you should master and know by heart. A `ConvolveOp` is used to perform a convolution from the source image to the destination. If you never took any math courses, or more likely, if you forgot everything about what you learned during those classes, a convolution is a spatial operation that computes the destination pixel by multiplying the source pixel and its neighbors by a convolution kernel.[1] Don't be frightened: You will soon understand what this gobbledygook means.

Any convolution operation relies on a convolution kernel, which is just a matrix of numbers. Here is an example:

$$kernel = \begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

The kernel defined here represents a 3 × 3 matrix of floating-point numbers. When you perform a convolution operation, this matrix is used as a sliding mask over the pixels of the source image. For instance, to compute the result of the convolution for a pixel located at the coordinates (x, y) in the source image, the center of the kernel is positioned at these coordinates. In the case of a 3 × 3 kernel, here are the coordinates, in the source image, of the pixels that each corresponding kernel value is applied to:

$$kernel\ coordinates = \begin{bmatrix} x-1, y-1 & x, y-1 & x+1, y-1 \\ x-1, y & x, y & x+1, y \\ x-1, y+1 & x, y+1 & x+1, y+1 \end{bmatrix}$$

1. And if this does not make sense to you, consider the mathematical definition from the Wikipedia: "Convolution is a mathematical operator which takes two functions, *f* and *g*, and produces a third function that in a sense represents the amount of overlap between *f* and a reversed and translated version of *g*." At least my version talks about pixels.

To compute the value of the destination pixel at (*x*, *y*), Java 2D multiplies the kernel values with their corresponding color values in the source image. Imagine a 3 × 3 white image with a single black pixel in its center, as suggested in Figure 8-6.

To convolve the black pixel with our 3 × 3 kernel, we must start by placing the matrix over the pixels, as shown in Figure 8-7.

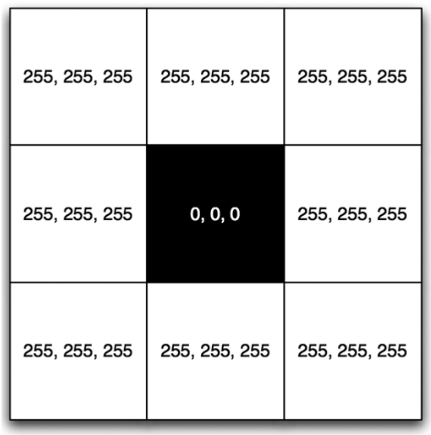

Figure 8-6 A black pixel surrounded by white pixels. The numbers show the RGB value of each pixel.

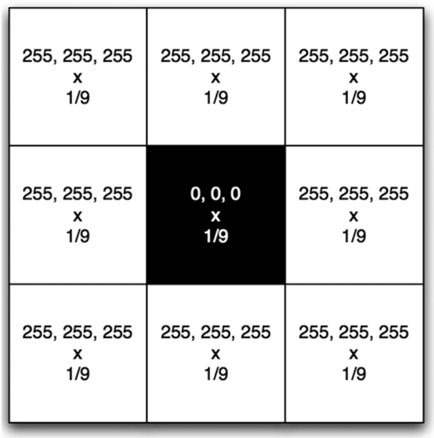

Figure 8-7 Each color value is multiplied by the corresponding value of the kernel.

Now we can compute all the multiplications, add up the results, and get the color value of the destination pixel:

$$R = 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 0 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{8}{9}$$

$$B = 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 0 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{8}{9}$$

$$G = 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 0 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{1}{9} + 255 \cdot \frac{8}{9}$$

The destination pixel is therefore a light gray; its RGB value is (227, 227, 227), or #E3E3E3. By now, you might have guessed what this kernel does: It replaces each pixel by the average color of its surroundings. Such a convolution operation is commonly known as a blur. We discuss blurring filters in more detail in Chapter 16, "Static Effects."

Constructing a Kernel

There are no particular restrictions about the size and contents of the kernels you can use with Java 2D. However, you should be aware of several important characteristics of kernels.

First, the values of a kernel should add up to 1.0 in the typical case, as in the previous example where all nine entries have the value 1/9. If these values do not add up to 1.0, the luminosity of the picture will not be preserved. This can, however, be turned to your advantage. For instance, you can increase the luminosity of a picture by 10 percent with a 1 × 1 kernel containing the value 1.1. Similarly, you can darken a picture by 10 percent with a 1 × 1 kernel containing the value 0.9. When dealing with larger kernels, the sum of the values defines the new luminosity. For instance, if the sum equals 0.5, then the luminosity will be cut in half. Keep that in mind when creating a kernel.

The size of a kernel defines the strength of a filter. For instance, a 3 × 3 blurring kernel produces a slightly blurry picture, whereas a 40 × 40 blurring kernel produces an indistinguishable blob from the original image.

The dimensions of the kernel are equally important. Kernels are usually odd-sided. While it is perfectly safe to use a 4 × 4 or a 12 × 12 kernel, it is not recommended. An even-sided kernel will not be centered over the source pixel and might give unbalanced visual results, which you should avoid. Also, it is easier for code readers to understand how an odd-sided kernel will behave. The Java 2D documentation defines the value of the matrix used as the center of the kernel as being the one at the coordinates (w – 1)/2, (h – 1)/2. This definition makes it harder to know which value is used as the center.

Your kernels do not have to be square shaped. Vertical kernels, for example with a 1 × 5 matrix, and horizontal kernels, for example with a 5 × 1 matrix, can be used to apply effects that work in only one direction. Chapter 16 presents examples of such kernels.

Last but not least, avoid using large kernels. When convolving a picture with a 3 × 3 kernel, Java 2D performs at least 17 operations (9 multiplications and 8 additions) per color component per pixel. Convolving a 640 × 480 picture requires at least 640 × 480 × 3 × 17 = 15,667,200 operations! That's quite a lot.[2] And this number does not even include the operations of reading and writing the actual pixel values from and to the source and destination pictures. We therefore strongly advise you not to perform convolve operations at painting time. Instead perform the operations once prior to painting and cache the results instead.

No matter what kernel you create, writing the code to perform the convolution is simple:

```
BufferedImage dstImage = null;
float[] sharpen = new float[] {
     0.0f, -1.0f,  0.0f,
    -1.0f,  5.0f, -1.0f,
     0.0f, -1.0f,  0.0f
};
Kernel kernel = new Kernel(3, 3, sharpen);
ConvolveOp op = new ConvolveOp(kernel);
dstImage = op.filter(sourceImage, null);
```

In Java 2D, a kernel is an array of floats and two dimensions. In this case, we use a 3 × 3 sharpening kernel to create an array of nine floats and tell the `Kernel` class that we want this array to be treated as a 3 × 3 matrix.

Figure 8-8 shows the result of the convolution with the 3 × 3 sharpening kernel shown in the previous code example.

Working on the Edge

Everything is not perfect yet. Take a close look at the generated result: You should see a black border surrounding the picture. During the convolve operation, Java

2. Even with today's CPU, it's still a lot. Really. And we are talking about convolving a small picture with a small kernel.

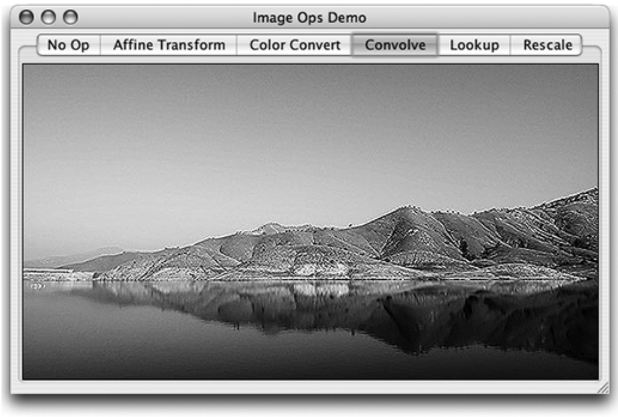

Figure 8-8 The sharpened picture shows enhanced details.

2D always matches the center of the kernel with one pixel of the source image. This works well for every pixel except the ones on the edges of the picture. Try to line up a 3 × 3 kernel with any pixel on the edge of an image and you will see that some parts of the kernel lie outside of the image. To work around this problem, Java 2D replaces the pixels it cannot compute with black pixels, which results in darkened edges because of the extra black introduced into the convolve operations for these edge pixels. To avoid this result, you can instruct Java 2D to do nothing and to keep the original color:

```
// the default is ConvolveOp.EDGE_ZERO_FILL
// the last parameter is the RenderingHints set
ConvolveOp op = new ConvolveOp(kernel,
    ConvolveOp.EDGE_NO_OP, null);
```

Unfortunately, neither of these solutions generates good-looking results. To get rid of any problem on the edges, you can simply increase the size of the original picture, as follows:

```
int kernelWidth = 3;
int kernelHeight = 3;

int xOffset = (kernelWidth - 1) / 2;
int yOffset = (kernelHeight - 1) / 2;

BufferedImage newSource = new BufferedImage(
  sourceImage.getWidth() + kernelWidth - 1,
  sourceImage.getHeight() + kernelHeight - 1,
  BufferedImage.TYPE_INT_ARGB);
Graphics2D g2 = newSource.createGraphics();
```

```
g2.drawImage(sourceImage, xOffset, yOffset, null);
g2.dispose();

ConvolveOp op = new ConvolveOp(kernel,
    ConvolveOp.EDGE_NO_OP, null);
dstImage = op.filter(newSource, null);
```

The original image is drawn centered into a new, larger, transparent image. Because we added enough transparent pixels on each side of the original image, the convolution operation will not affect the pixels of the original image. It is important to use the `ConvolveOp.EDGE_NO_OP` edge condition so you will keep the pixels transparent around the image. This technique of adding transparent pixels on the sides provides better-looking results, but you have to take the extraneous pixels into account.

LookupOp

A `LookupOp` maps the color values of the source to new color values in the destination. This operation is achieved with a lookup table that contains the destination values for each possible source value.

Lookup operations can be used to generate several common filters, such as negative filters, posterizing filters, and thresholding filters. Negative filters are interesting because they help illustrate how lookup tables work. Pixel colors are usually represented using three components (red, green, and blue) stored in 8 bits each. As a result, the color values of a negative image are the 8 bits' complements of the source image color values:

```
dstR = 255 - srcR;
dstG = 255 - srcG;
dstB = 255 - srcB;
```

To apply such a conversion to the source image, you must create a lookup table that associates all the values in the 8 bits range (from 0 to 255) to their complements:

```
short[] data = new short[256];
for (short i = 0; i < 256; i++) {
    data[i] = 255 - i;
}

BufferedImage dstImage = null;
LookupTable lookupTable = new ShortLookupTable(0, data);
LookupOp op = new LookupOp(lookupTable, null);
dstImage = op.filter(sourceImage, null);
```

Figure 8-9 shows the result of this negative filter.

The `LookupTable` from this example contains only one lookup array, used for all of the color components of the source image, resulting in the same conversion of all of the color components.

To perform a different conversion for each component, you simply need to create one lookup array per color component in the source image. Since the example relies on an RGB picture, we can create a filter that inverts only the red component by defining three lookup arrays:

```
short[] red = new short[256];
short[] green = new short[256];
short[] blue = new short[256];

for (short i = 0; i < 256; i++) {
    red[i] = 255 - i;
    green[i] = blue[i] = i;
}

short[][] data = new short[][] {
    red, green, blue
};

BufferedImage dstImage;
LookupTable lookupTable = new ShortLookupTable(0, data);
dstImage = op.filter(sourceImage, null);
LookupOp op = new LookupOp(lookupTable, null);
```

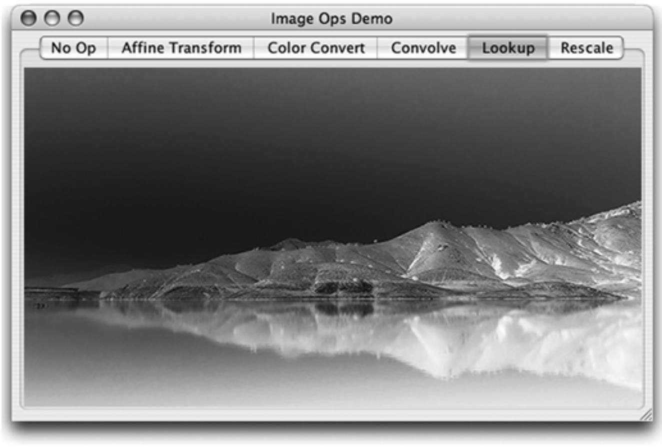

Figure 8-9 A simple lookup operation can be used to produce a negative image.

You do not need to provide a lookup array for the alpha channel of your picture, if present. In this case, Java 2D will simply preserve the original alpha values. Whenever you create a new `LookupOp`, ensure that the number and size of your lookup arrays match the source image structure.

RescaleOp

`RescaleOp` does not scale the size of an image as you would expect it to. Instead, `RescaleOp` performs a rescaling operation by multiplying the color value of each pixel in the source image by a scale factor and then adding an offset. Here is the formula applied to each color component of the source pixels:

```
dstR = (srcR * scaleFactor) + offset
dstG = (srcG * scaleFactor) + offset
dstB = (srcB * scaleFactor) + offset
```

Rescaling operations can be used to brighten, darken, or tint images. The following code example increases the overall brightness of the picture by 10 percent:

```
BufferedImage dstImage = null;
RescaleOp op = new RescaleOp(1.1f, 0.0f, null);
dstImage = op.filter(sourceImage, null);
```

The first two parameters of the `RescaleOp` constructor are respectively the scale factor and the offset. Note that a `RescaleOp` with an offset of 0 is no different from a `ConvolveOp` with a 1 × 1 kernel. You can also adjust each color component independently:

```
BufferedImage dstImage = null;
float[] factors = new float[] {
    1.4f, 1.4f, 1.4f
};
float[] offsets = new float[] {
    0.0f, 0.0f, 30.0f
};
RescaleOp op = new RescaleOp(factors, offsets, null);
dstImage = op.filter(sourceImage, null);
```

In this case, the overall brightness is increased by 40 percent, and all of the pixel colors are shifted toward the blue color. The offset of 30 increases the blue component of each pixel by 12 percent (30/256). Remember, the offset is added to the color value and must therefore be a value between 0 and 255, as opposed to the scale factor, which acts as a percentage.

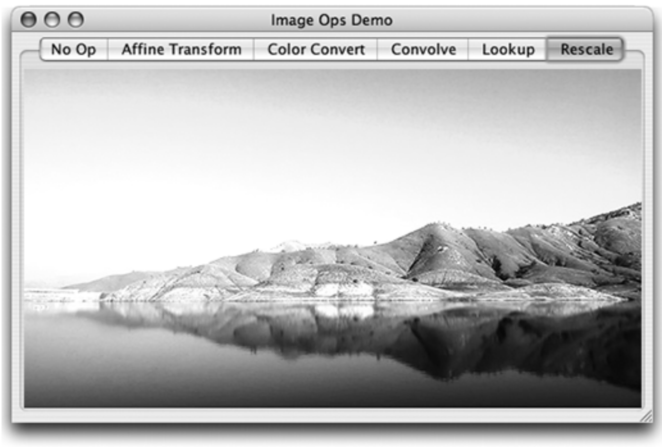

Figure 8-10 The image is brighter and the blues are bluer after processing.

Figure 8-10 shows the result produced by a `RescaleOp` with a scale factor of 1.4 for each component and an offset of 30 for the blue component.

Just as in `LookupOp`, the number of values used in the scale factors and offset arrays depend on the number of components in the source image. Working on `TYPE_INT_RGB` or `TYPE_INT_ARGB` pictures is therefore easier than working on other types of `BufferedImage`. When the source image contains an alpha channel, you do not need to specify a factor and an offset for the alpha component. Java 2D automatically preserves the original values.

These five `BufferedImageOps` will probably be all you need for most situations. Nevertheless, you might want to create your own specialized `BufferedImageOp` to create advanced graphical effects.

Custom BufferedImageOp

Creating a new filter from scratch is not a very complicated task. To prove it, we show you how to implement a color tint filter. This kind of filter can be used to mimic the effect of the colored filters photographers screw in front of their lenses. For instance, an orange color tint filter gives a sunset mood to a scene, while a blue filter cools down the tones in the picture.

You first need to create a new class that implements the `BufferedImageOp` interface and its five methods. To make the creation of several filters easier, we first define a new abstract class entitled AbstractFilter. As you will soon discover, all filters based on this class are nonspatial, linear color filters. That means that they will not affect the geometry of the source image and that they assume the destination image has the same size as the source image.

ONLINE DEMO

The complete source code of our custom BufferedImage is available on this book's Web site in the project entitled CustomImageOp.

Base Filter Class

AbstractFilter implements all the methods from BufferedImageOp except for filter(), which actually processes the source image into the destination and hence belongs in the subclasses:

```
public abstract class AbstractFilter
    implements BufferedImageOp {
  public abstract BufferedImage filter(
    BufferedImage src, BufferedImage dest);

  public Rectangle2D getBounds2D(BufferedImage src) {
    return new Rectangle(0, 0, src.getWidth(),
                         src.getHeight());
  }

  public BufferedImage createCompatibleDestImage(
    BufferedImage src, ColorModel destCM) {
    if (destCM == null) {
        destCM = src.getColorModel();
    }

    return new BufferedImage(destCM,
        destCM.createCompatibleWritableRaster(
          src.getWidth(), src.getHeight()),
        destCM.isAlphaPremultiplied(), null);
  }

  public Point2D getPoint2D(Point2D srcPt,
                            Point2D dstPt) {
    return (Point2D) srcPt.clone();
  }

  public RenderingHints getRenderingHints() {
    return null;
  }
}
```

The `getRenderingHints()` method must return a set of `RenderingHints` when the image filter relies on rendering hints. Since this will probably not be the case for our custom filters, the abstract class simply returns null.

The two methods `getBounds2D()` and `getPoint2D()` are very important for spatial filters, such as `AffineTransformOp`. The first method, `getBounds2D()`, returns the bounding box of the filtered image. If your custom filter modifies the dimension of the source image, you must implement this method accordingly. The implementation proposed here makes the assumption that the filtered image will have the same size as the source image.

The other method, `getPoint2D()`, returns the corresponding destination point given a location in the source image. As for `getBounds2D()`, `AbstractFilter` makes the assumption that no geometry transformation will be applied to the image, and the returned location is therefore the source location.

`AbstractFilter` also assumes that the only data needed to compute the pixel for (*x*, *y*) in the destination is the pixel for (*x*, *y*) in the source.

The last implemented method is `createCompatibleDestImage()`. Its role is to produce an image with the correct size and number of color components to contain the filtered image. The implementation shown in the previous source code creates an empty clone of the source image; it has the same size and the same color model regardless of the source image type.

Color Tint Filter

The color tint filter, cleverly named `ColorTintFilter`, extends `AbstractFilter` and implements `filter()`, the only method left from the `BufferedImageOp` interface. Before we delve into the source code, we must first define the operation that the filter will perform on the source image. A color tint filter mixes every pixel from the source image with a given color. The strength of the mix is defined by a mix value. A mix value of 0 means that all of the pixels remain the same, whereas a mix value of 1 means that all of the source pixels are replaced by the tinting color. Given those two parameters, a color and a mix percentage, we can compute the color value of the destination pixels:

```
dstR = srcR * (1 – mixValue) + mixR * mixValue
dstG = srcG * (1 – mixValue) + mixG * mixValue
dstB = srcB * (1 – mixValue) + mixB * mixValue
```

If you tint a picture with 40 percent white, the filter will retain 60 percent (1 or 1 – `mixValue`) of the source pixel color values to preserve the overall luminosity of the picture.

The following source code shows the skeleton of `ColorTintFilter`, an immutable class.

Note: Immutability. It is very important to ensure that your filters are immutable to avoid any problem during the processing of the source images. Imagine what havoc a thread could cause by modifying one of the parameters of the filter while another thread is filtering an image. Rather than synchronizing code blocks or spending hours in a debugger, go the easy route and make your `BufferedImageOp` implementations immutable.

```
public class ColorTintFilter extends AbstractFilter {
  private final Color mixColor;
  private final float mixValue;

  public ColorTintFilter(Color mixColor, float mixValue) {
    if (mixColor == null) {
      throw new IllegalArgumentException(
          "mixColor cannot be null");
    }

    this.mixColor = mixColor;
    if (mixValue < 0.0f) {
      mixValue = 0.0f;
    } else if (mixValue > 1.0f) {
      mixValue = 1.0f;
    }
    this.mixValue = mixValue;
  }

  public float getMixValue() {
    return mixValue;
  }

  public Color getMixColor() {
    return mixColor;
  }

  @Override
  public BufferedImage filter(BufferedImage src,
                              BufferedImage dst) {
    // filters src into dst
  }
}
```

The most interesting part of this class is the implementation of the `filter()` method:

```
@Override
public BufferedImage filter(BufferedImage src,
                            BufferedImage dst) {
  if (dst == null) {
      dst = createCompatibleDestImage(src, null);
  }

  int width = src.getWidth();
  int height = src.getHeight();

  int[] pixels = new int[width * height];
  GraphicsUtilities.getPixels(src, 0, 0, width,
                              height, pixels);
  mixColor(pixels);
  GraphicsUtilities.setPixels(dst, 0, 0, width,
                              height, pixels);

  return dst;
}
```

The first few lines of this method create an acceptable destination image when the caller provides none. The `javadoc` of the `BufferedImageOp` interface dictates this behavior: "If the destination image is null, a `BufferedImage` with an appropriate `ColorModel` is created."

Instead of working directly on the source and destination images, the color tint filter reads all the pixels of the source image into an array of integers. The implications are threefold. First, all of the color values are stored on four ARGB 8-bit components packed as an integer. Then, the source and the destination can be the same, since all work will be performed on the array of integers. Finally, despite the increased memory usage, it is faster to perform one read and one write operation on the images rather than reading and writing pixel by pixel. Before we take a closer look at `mixColor()`, where the bulk of the work is done, here is the code used to read all the pixels at once into a single array of integers:

```
public static int[] getPixels(BufferedImage img,
                              int x, int y,
                              int w, int h,
                              int[] pixels) {
  if (w == 0 || h == 0) {
    return new int[0];
  }
```

```
    if (pixels == null) {
      pixels = new int[w * h];
    } else if (pixels.length < w * h) {
      throw new IllegalArgumentException(
          "pixels array must have a length >= w*h");
    }

    int imageType = img.getType();
    if (imageType == BufferedImage.TYPE_INT_ARGB ||
        imageType == BufferedImage.TYPE_INT_RGB) {
        Raster raster = img.getRaster();
        return (int[]) raster.getDataElements(x, y, w, h, pixels);
    }

    return img.getRGB(x, y, w, h, pixels, 0, w);
  }
```

There are two different code paths, depending on the nature of the image from which the pixels are read. When the image is of type `INT_ARGB` or `INT_RGB`, we know for sure that the data elements composing the image are integers. We can therefore call `Raster.getDataElements()` and cast the result to an array of integers. This solution is not only fast but preserves all the optimizations of managed images performed by Java 2D.

When the image is of another type, for instance `TYPE_3BYTE_BGR`, as is often the case with JPEG pictures loaded from disk, the pixels are read by calling the `BufferedImage.getRGB(int, int, int, int, int[], int, int)` method. This invocation has two major problems. First, it needs to convert all the data elements into integers, which can take quite some time for large images. Second, it throws away all the optimizations made by Java 2D, resulting in slower painting operations, for instance. The picture is then said to be unmanaged. To learn more details about managed images, please refer to the Chapter 5, "Performance."

Note: Performance and `getRGB()`. The class `BufferedImage` offers two variants of the `getRGB()` method. The one discussed previously has the following signature:

```
int[] getRGB(int startX, int startY, int w, int h,
             int[] rgbArray, int offset, int scansize)
```

This method is used to retrieve an array of pixels at once, and invoking it will punt the optimizations made by Java 2D. Consider the second variant of `getRGB()`:

```
int getRGB(int x, int y)
```

This method is used to retrieve a single pixel and does not throw away the optimizations made by Java 2D. Be very careful about which one of these methods you decide to use.

The `setPixels()` method is very similar to `getPixels()`:

```
public static void setPixels(BufferedImage img,
                             int x, int y,
                             int w, int h,
                             int[] pixels) {
  if (pixels == null || w == 0 || h == 0) {
      return;
  } else if (pixels.length < w * h) {
    throw new IllegalArgumentException(
        "pixels array must have a length >= w*h");
  }

  int imageType = img.getType();
  if (imageType == BufferedImage.TYPE_INT_ARGB ||
      imageType == BufferedImage.TYPE_INT_RGB) {
      WritableRaster raster = img.getRaster();
      raster.setDataElements(x, y, w, h, pixels);
  } else {
      img.setRGB(x, y, w, h, pixels, 0, w);
  }
}
```

Performance Tip: Working on a `TYPE_INT_RGB` or `TYPE_INT_ARGB` results in better performance, since no type conversion is required to store the processed pixels into the destination image.

Reading and writing pixels from and to images would be completely useless if we did not process them in between operations. The implementation of the color tint equations is straightforward:

```
private void mixColor(int[] inPixels) {
  int mix_a = mixColor.getAlpha();
  int mix_r = mixColor.getRed();
  int mix_b = mixColor.getBlue();
  int mix_g = mixColor.getGreen();

  for (int i = 0; i < inPixels.length; i++) {
      int argb = inPixels[i];

      int a = argb & 0xFF000000;
      int r = (argb >> 16) & 0xFF;
      int g = (argb >>  8) & 0xFF;
      int b = (argb      ) & 0xFF;
```

```
            r = (int) (r * (1.0f - mixValue) + mix_r * mixValue);
            g = (int) (g * (1.0f - mixValue) + mix_g * mixValue);
            b = (int) (b * (1.0f - mixValue) + mix_b * mixValue);

            inPixels[i] = a << 24 | r << 16 | g << 8 | b;
        }
    }
```

Before applying the equations, we must split the pixels into their four color components. Some bit shifting and masking is all you need in this situation. Once each color component has been filtered, the destination pixel is computed by packing the four modified color components into a single integer. Figure 8-11 shows a picture tinted with 50 percent red.

ONLINE DEMO

The above implementation works well but can be vastly improved performance-wise. The `ColorTintFilter` class in the `CustomImageOp` project on this book's Web site offers a better implementation that uses a few tricks to avoid doing all of the computations in the loop.

Note: As an exercise, you can try to improve this implementation on your own before looking at the final version. (Hint: You can use lookup arrays.)

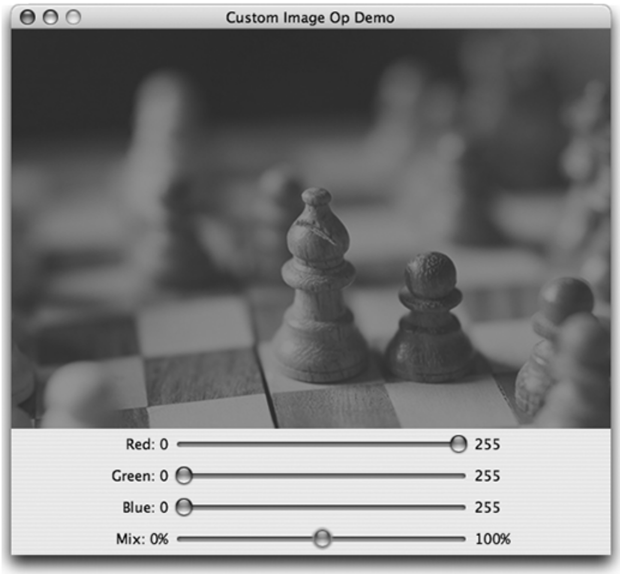

Figure 8-11 A red-tinted picture.

A Note about Filters Performance

Image filters perform a lot of operations on images, and performance can easily degrade if you do not pay attention to a few details. Whenever you write a filter assuming the source image will be of type `INT_RGB` or `INT_ARGB`, make sure the source image is actually of that type.

Usually, compatible images (images created with `GraphicsConfiguration.createCompatibleImage()`), which are designed to be in the same format as the screen, are stored as integers. It is often the case that the user's display is in 32-bit format and not the older 8-, 16-, and 24-bit formats. Therefore, it is a good idea to always load your images as compatible images.

The `CustomImageOp` demo loads a JPEG picture, which would normally be of type `3BYTE_BGR`, and turns it into a compatible image of type `INT_RGB`. You can look for the call to `GraphicsUtilities.loadCompatibleImage()` in the source code of the demo and replace it with `ImageIO.read()` to see the difference when moving the sliders of the user interface. As a rule of thumb, do not hesitate to use the various methods from the `GraphicsUtilities` class to always use compatible images.

Summary

Java 2D offers several powerful facilities to perform image processing on your pictures. The built-in `BufferedImageOp` implementations let you write your own custom filters very quickly. And if you need more flexibility, you can even create a new `BufferedImageOp` implementation from scratch.