Chapter 4. Images, Visual Effects, and Animation

Images are a central part of every game, and this chapter examines how we can (efficiently) load and display them, apply visual effects such as blurring, fading, and rotation, and animate them.

Image loading and processing is an area of Java which is undergoing rapid change, mainly driven by the wish for speed. We begin by reviewing the (rather outmoded) AWT imaging model, which is being superceded by the BufferedImage and VolatileImage classes, ImageIO, and the wide range of BufferedImageOp image operations offered by Java 2D. If these aren't enough then JAI (Java Advanced Imaging) has even more capabilities.

The application developed in this chapter is called ImagesTests, shown in Figure 1.

Figure 1. ImagesTests in Action.

The static picture in figure 1 doesn't show the changing effects (animations) being applied to each image. Eleven different visual effects are used, including 'zapping', 'teleportation', and the more familiar reddening, blurring, and flipping.

The effects are mostly Java 2D operations, such as convolution or affine transformation. Occasionally, we make use of capabilities in drawImage() (e.g. for resizing and flipping an image).

The images (in GIF, JPEG, or PNG format) are loaded by our own ImagesLoader class from a JAR file containing the application and the images. The images are loaded using ImageIO's read(), and stored as BufferedImage objects, in order to take advantage of the JVM's "managed image" features.

ImagesLoader can load individual images, image strips, and multiple image files which represent an animation sequence.

The animation effects utilized by ImagesTests falls into two categories:

1) those defined by repeatedly applying a visual effect, such as blurring, to the same image, but by an increasing amount;
2) those where the animation is represented by a series of different images displayed one after another.

Before we examine the details of ImagesTests, we'll take a look at the imaging capabilities in Java, including the AWT imaging model, BufferedImages, managed images, VolatileImages, Java 2D image processing, and a brief mention of JAI.

1. Image Formats

A game will typically use a mix of the GIF, JPEG and PNG images, popular graphics formats which have advantages and disadvantages.

A GIF (Graphics Interchange Format) image is best for cartoony-style graphics using few colours, since only a maximum of 256 colours can be represented in a file. This is due to GIF's use of a 256 element colour table to store information, allowing each pixel to hold only an index into that table, reducing the requirement for 3 bytes for red, green, and blue (RGB) information down to a single byte. One of the colour table entries can represent a transparent 'colour', which Java honours by not drawing.

GIF offers rudimentary animation, by permitting a file to contain several images. These will be drawn consecutively when the file is displayed (e.g. with drawImage() in Java). Actually, this feature isn't of much use since there is no simple way of controlling the animation from within Java.

A JPEG (Joint Photographic Experts Group) file employs 3 bytes (24 bits) per pixel (1 byte for each of the red, green, and blue components), but a lossy compression scheme reduces the space quite considerably. This may cause large areas using a single colour to appear blotchy, and sharp changes in contrast can become blurred (e.g. at the edges of black text on a white background). JPEG files are best for large photographic images, such as game backgrounds. JPEG files do not offer transparency.

The PNG format (Portable Network Graphics) is intended as a replacement for GIF. It includes an alpha channel along with the usual RGB components, which permits an image to include translucent areas. Translucency is particularly useful for gaming effects like laser beams, smoke, and ghosts (of course). Other advantages over GIF are gamma correction, which enables image brightness to be controlled across platforms, 2D interlacing, and (slightly) better compression, which is lossless. This last feature makes PNG a good storage choice while a photographic image is being edited, but JPEG is probably better for the finished image since its lossy compression achieves greater size reductions.

Some developers prefer PNG since it is an open source standard (see http://www.libpng.org/pub/png/), with no patents involved; the GIF format is owned by CompuServe.

2. The AWT Imaging Model

JDK 1.0 introduced the AWT imaging model for downloading and drawing images. Back then, it was thought that the most common use of imaging would involve applets pulling graphics down off the Web. A standard 1990's example (but using JApplet):

```
import javax.swing.*;
import java.awt.*;

public class ShowImage extends JApplet
{
  private Image im;

  public void init()
  {  im = getImage( getDocumentBase(), "ball.gif");  }

  public void paint(Graphics g)
  {  g.drawImage(im, 0, 0, this);  }
}
```

The getDocumentBase() method returns the URL of the directory holding the original Web document, and this is prepended to the image's filename to get a URL suitable for getImage().

The central problem with networked image retrieval is *speed*. Consequently, the Java designers considered it a bad idea to have an applet stop while an image slowly crawled over from the server-side. This led to a (somewhat) confusing behaviour for getImage() and drawImage().

The getImage() method is poorly named since it doesn't get (or download) the image at all. Instead it prepares an empty Image object (im) for holding the image, returning immediately after that. The downloading is actually triggered by drawImage() in paint (), which is called as the applet is loaded into the browser, after init() has finished.

The fourth argument of drawImage() is an ImageObserver (usually the applet, or JFrame in an application). It will monitor the gradual downloading of the image. As data arrives, the Component's imageUpdate() is repeatedly called. imageUpdate()'s default behaviour is to call repaint(), to redraw the image now that more data is available, and return true. However, if an error has occurred with the image retrieval then imageUpdate() will return false. imageUpdate() can be overridden and modified by the programmer.

The overall effect is that paint() will be called repeatedly as the image is downloaded, causing the image to appear gradually on-screen. This effect is only noticeable if the image is coming over the network. If the file is stored locally then it will be drawn fully almost instantaneously.

The result of this coding style means that the Image im contains no data until paint() is called, and even then may not contain complete information for several seconds or minutes. This make programming difficult: for instance, a GUI cannot easily allocate an on-screen space to the image since it has no known width or height until painting has started.

Experience since the introduction of JDK 1.0 has shown that most programs do not want graphics to be drawn incrementally during execution. For example, game sprites should be fully drawn from the very start.

The getImage() method is only for applets; there is a separate getImage() method for applications, accessible from Toolkit. For example:

```
Image im = Toolkit.getDefaultToolkit().getImage("http://....");
```

As with the getImage() method for applets, it doesn't download anything. That task is done by paint().

2.1. The MediaTracker Class

Most programs (and most games) want to preload images before drawing them. In other words, we do not want to tie downloading to painting.

One solution is the MediaTracker class: a MediaTracker object can start the download of an image and suspend execution until it has fully arrived, or an error occurs. The init() method in ShowImage class can be modified to do this:

```
public void init()
{
  im = getImage( getDocumentBase(), "ball.gif");

  MediaTracker tracker = new MediaTracker(this);
  tracker.addImage(im, 0);
  try {
    tracker.waitForID(0);
  }
  catch (InterruptedException e)
  {  System.out.println("Download Error"); }
}
```

waitForID() starts the separate download thread, and suspends until it finishes. The ID used in the MediaTracker object can be any positive integer.

This means that the applet will be slower to start since init()'s execution will be suspended while the image is retrieved. In paint(), drawImage() will now only draw the image, since there is no need to do a download. Consequently, drawImage() can be supplied with a null (empty) ImageObserver:

```
drawImage(im, 0, 0, null);
```

A common way of speeding up the downloading of multiple images is to spawn a pool of threads, each one assigned to the retrieval of a single image. Only when each thread has completed will init() return.

2.2. ImageIcon

Writing MediaTracker code in every applet/application can be trying, and so an ImageIcon class was introduced which sets up a MediaTracker itself. The name of the

class is a bit misleading: any size of image can be downloaded into an ImageIcon, not just icons.

The init() method becomes:

```
public void init()
{ im = new ImageIcon( getDocumentBase()+"ball.gif").getImage();  }
```

The ImageIcon object can be converted to an Image (as here), or be painted with ImageIcon's paintIcon() method.

2.3. The Rise of JARS

A JAR (Java ARchive) file is a way of packaging code and resources together into a single, compressed file. Resources can be just about anything, including images and sounds.

If an applet (or application) is going to utilize a lot of images, repeated network connections to download them will severely reduce execution speed. It's much better to create a single JAR file containing the applet (or application) and all the images, and have the browser (or user) download it. Then, when an image comes to be loaded, it's a fast, local load from the JAR file.

From a user's point of view, the download of the 'program' takes a bit longer, but it starts without any annoying delays caused by image loading.

At the end of this chapter, we'll explain how to package up the ImagesTests code, and the large number of images it uses, as a JAR file. The only coding change occurs in specifying the location of an image file. The previous ImageIcon example would become:

```
im = new ImageIcon( getClass().getResource("ball.gif") ).getImage();
```

getClass() gets the Class reference for the object (e.g. ShowImage), and getResource() specifies that the resource is stored in the same place as that class.

2.4. AWT Image Processing

It's not particularly easy to access the various elements of an Image object, such as its pixel data or colour model. For instance, the image manipulation features in AWT are primarily aimed at modifying individual pixels as they pass through a 'filter'. A stream of pixel data is sent out by a ImageProducer, passes through the ImageFilter, and on to an ImageConsumer (see Figure 2). This is known as the *push model*, since stream data is 'pushed' out by the producer.

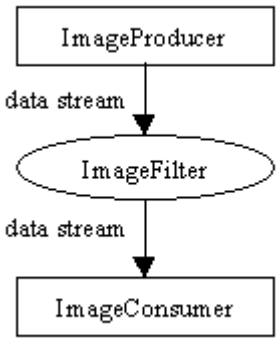

Figure 2. Image Processing in AWT.

Two pre-defined ImageFilter subclasses are CropImageFilter for cropping regions of pixels, and RGBImageFilter for processing individual pixels.

It is possible to chain filters together by making a consumer of one filter the producer of another.

This stream-view of filtering makes it difficult to process groups of pixels, especially ones which are non-contiguous. For example, a convolution operation for image smoothing would require a new subclass of ImageFilter, and a new ImageConsumer to deal with the disruption to the pixels stream.

An alternative approach is to use the PixelGrabber class to collect all the pixel data from an image into an array, where it can then be conveniently processed in its entirety. Use must also be made of MemoryImageSource to funnel the changed array's data as a stream to a specified ImageConsumer. The additional steps in the push model are shown in Figure 3.

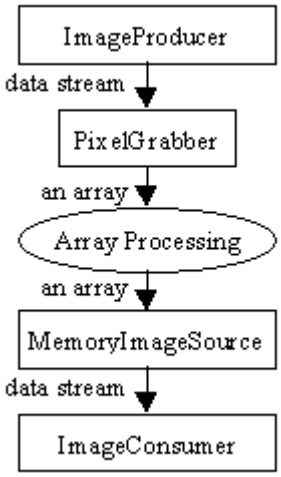

Figure 3. Processing the Image as an Array.

Modern Java code (i.e. since J2SE 1.2) can utilize the image processing capabilities of Java 2D, with its many predefined operations, and so you're unlikely to meet the push model, except in legacy code. If Java 2D is insufficient, then JAI should be considered.

3. An Overview of Java 2D

In this chapter, we'll only be using the imaging capabilities of Java 2D, which are extensive. However, Java 2D also offers a complete set of graphics features, which address the inadequacies in the older AWT graphics classes, including single pixel thickness lines, limited fonts, poor shape manipulation (e.g. no rotation), and no special fills, gradients, or patterns inside shapes.

Java 2D replaces most of the shape primitives in AWT (e.g. rectangles, arcs, lines, ellipse, polygons) with versions that can take double or floating pointing coordinates, although many people still use the old drawLine(), drawRect(), and fillRect() methods. Of more interest is the ability to create arbitrary geometric shapes by using set operations on other shapes (with union, intersection, subtraction, and exclusive or). A GeneralPath class permits a shape to be built from a series of connected lines and curves. Curves can be defined using splines.

Java 2D distinguishes between shape stroking and filling. Stroking is the drawing of lines and shape outlines, which may employ various patterns and thicknesses. Shape filling can use a solid colour (as in AWT), and patterns, colour gradients, and images acting as textures.

Affine transformations can be applied to shapes and images, including translation, rotation, scaling, and shearing, and groups of transformations can be composed

together. drawImage() can be supplied with such a transformation, which is applied before the image is rendered.

Shapes and images can be drawn together using eight different compositing rules, optionally combined with varying transparency values. Clipping can be applied, based on an arbitrary shape (not just a rectangle as in AWT).

Rendering hints can be specified, including the antialiasing of shapes and text (i.e. the smoothing of their jagged edges), image interpolation, and whether to use high-speed or high-quality rendering.

Java-based printing finally became relatively easy to control with Java 2D.

Java's top-level Web page for Java 2D is `http://java.sun.com/products/java-media/2D/`, and there is extensive documentation and a tutorial trail.

The central Java 2D class is Graphics2D, a subclass of AWT's Graphics. paint() or paintComponent() must cast the graphics context to become a Graphics2D object before Java 2D operations can be employed, as shown in the paintComponent() method below.

```
public void paintComponent(Graphics g)
// draw a blue square
{
  super.paintComponent(g);
  Graphics2D g2d = (Graphics2D) g;  // cast the graphics context

  g2d.setPaint(Color.blue);
  Rectangle2D.Double square =
           new Rectangle2D.Double(10,10,350,350);
  g2d.fill(square);
}
```

The pen parameters are usually set first, which may set its colour, pattern, thickness, and how the drawn image will be composed with others. It's also possible to apply affine transformations to the drawing area's coordinate system, such as rotating it.

A shape object is then created, which might be a rectangle (as here), ellipse, polygon, or a GeneralPath object. The shape can be drawn in outline with draw(), or filled using the current pen settings by calling fill().

3.1. Java 2D and Active Rendering

Java 2D operations can be easily utilized in the active rendering approach described in chapters 1-3. As you may recall, a Graphics object for the off-screen buffer is obtained with getGraphics() inside gameRender(). This can be cast to a Graphics2D object.

```
// global variables for off-screen rendering
private Graphics2D dbg2D;    // was a Graphics object, dbg
private Image dbImage = null;
     :

private void gameRender()
// draw the current frame to an image buffer
{
  if (dbImage == null){  // create the buffer
    dbImage = createImage(PWIDTH, PHEIGHT);
    if (dbImage == null) {
      System.out.println("dbImage is null");
      return;
    }
    else
      dbg2D = (Graphics2D) dbImage.getGraphics();
  }

  // clear the background using Java 2D
  // draw game elements using Java 2D
  ...
  if (gameOver)
    gameOverMessage(dbg2D);
}  // end of gameRender()
```

Methods called from gameRender(), such as gameOverMessage() will now utilize the Graphics2D object.

In FSEM, the Graphics object is obtained with getDrawGraphics(), and its result can be cast:

```
private Graphics2D gScr2d;    // global, was Graphics gScr
     :

private void screenUpdate()
{ try {
    gScr2d = (Graphics2D) bufferStrategy.getDrawGraphics();
    gameRender(gScr2d);
    gScr2d.dispose();
      :
}
```

gameRender() receives a Graphics2D object, and so has the full range of Java 2D operations at its disposal.

4. The BufferedImage Class

The BufferedImage class is a subclass of Image, and so can be employed instead of Image in methods such as drawImage(). BufferedImage has two main advantages: the data required for image manipulation is easily accessible through it's methods, and BufferedImage objects are automatically converted to *managed images* by the JVM (when possible). A managed image may allow hardware acceleration to be employed when the image is being rendered.

The following code is the ShowImage applet recoded to use a BufferedImage:

```
import javax.swing.*;
import java.awt.*;
import java.io.*;
import java.awt.image.*;
import javax.imageio.ImageIO;


public class ShowImage extends JApplet
{
  private BufferedImage im;

  public void init()
  { try {
      im =  ImageIO.read( getClass().getResource("ball.gif") );
    }
    catch(IOException e) {
      System.out.println("Load Image error:");
    }
  } // end of init()

  public void paint(Graphics g)
  {  g.drawImage(im, 0, 0, this);  }
}
```

The simplest, and perhaps fastest, way of loading a BufferedImage object is with read () from the ImageIO class. Some tests suggest that it may be 10% faster than using ImageIcon, which can be significant when the image is large. They are different versions of read() for reading from a URL, InputStream, and ImageInputStream.

It's possible to optimize the BufferedImage so that it has the same internal data format and colour model as the underlying graphics device. This requires us to make a copy of the input image using GraphicsConfiguration's creatCompatibleImage(). The various steps are packaged together inside a loadImage() method given below.

```
public class ShowImage extends JApplet
{
  private GraphicsConfiguration gc;
  private BufferedImage im;

  public void init()
  {
    // get this device's graphics configuration
    GraphicsEnvironment ge =
         GraphicsEnvironment.getLocalGraphicsEnvironment();
    gc = ge.getDefaultScreenDevice().getDefaultConfiguration();
```



```
    im = loadImage("ball.gif");
  } // end of init()


   public BufferedImage loadImage(String fnm)
   /* Load the image from <fnm>, returning it as a BufferedImage
      which is compatible with the graphics device being used.
      Uses ImageIO. */
   {
     try {
       BufferedImage im = ImageIO.read(getClass().getResource(fnm));

       int transparency = im.getColorModel().getTransparency();
       BufferedImage copy =  gc.createCompatibleImage(
                               im.getWidth(), im.getHeight()),
                               transparency );

       // create a graphics context
       Graphics2D g2d = copy.createGraphics();

       // copy image
       g2d.drawImage(im,0,0,null);
       g2d.dispose();
       return copy;
     }
     catch(IOException e) {
       System.out.println("Load Image error for " + fnm + ":\n" + e);
       return null;
     }
  } // end of loadImage()


  public void paint(Graphics g)
  {  g.drawImage(im, 0, 0, this);  }

} // end of ShowImage class
```

The three argument version of createCompatibleImage() is utilized, which requires the BufferedImage's width, height, and transparency value.

The possible transparency values are Transparency.OPAQUE, Transparency.BITMASK, and Transparency.TRANSLUCENT. The BITMASK setting is applicable to GIFs which have a transparent area, while TRANSLUCENT can be employed by translucent PNG images.

There is a two argument version of createCompatibleImage() which only requires the image's width and height, but if the source image has a transparent or translucent component then it (most probably) will be copied incorrectly; for instance, the transparent areas in the source may be drawn as solid black.

Fortunately, it is quite simple to access the transparency information in the source BufferedImage, by querying its ColourModel (explained later):

```
int transparency = im.getColorModel().getTransparency();
```

The copy BufferedImage object is filled in by drawing the source image into its graphics context.

Another reason for the use of createCompatibleImage() is that it permits J2SE 1.4.2 to mark the resulting BufferedImage as a managed image, which may later be drawn to the screen using hardware acceleration. However, in J2SE 1.5. the JVM knows that anything read in by ImageIO's read() can become a managed image, so the call to createCompatibleImage() is no longer necessary for that reason. The call should still be made though, since it optimizes the BufferedImage's internals for the graphics device.

4.1. From Image to BufferedImage

Legacy code usually employs Image, and it may not be feasible to rewrite the entire code base to utilize BufferedImage. Instead, is there a way to convert an Image object to a BufferedImage object? makeBIM() makes a gallant effort:

```
private BufferedImage makeBIM(Image im, int width, int height)
// make a BufferedImage copy of im, assuming an alpha channel
{
  BufferedImage copy = new BufferedImage(width, height,
                                         BufferedImage.TYPE_INT_ARGB);
  // create a graphics context
  Graphics2D g2d = copy.createGraphics();

  // copy image
  g2d.drawImage(im,0,0,null);
  g2d.dispose();
  return copy;
}
```

It can be used in ShowImage.java:

```
public void init()
// load an imageIcon, convert to BufferedImage
{
  ImageIcon imIcon = new ImageIcon(
                       getClass().getResource("ball.gif") );

  im = makeBIM(imIcon.getImage(), imIcon.getIconWidth(),
                                  imIcon.getIconHeight());
}
```

We load an ImageIcon (to save on MediaTracker coding), and pass its Image, width, and height into makeBIM(), getting back a suitable BufferedImage object.

A niggly issue with makeBIM() is located in the BufferedImage() constructor. The constructor must be supplied with a type, and there's a lot to choose from (look at the Java documentation for BufferedImage for a complete list). A very partial list appears in Table 1.

BufferedImage Type	Description
TYPE_INT_ARGB	8-bit alpha, red, green, and blue samples packed into a 32-bit integer.
TYPE_INT_RGB	8-bit red, green and blue samples packed into a 32-bit integer.
TYPE_BYTE_GRAY	An unsigned byte grayscale image (1 pixel/byte).
TYPE_BYTE_BINARY	A byte packed binary image (8 pixels/byte).
TYPE_INT_BGR	8-bit blue, green and red samples packed into a 32-bit integer.
TYPE_3BYTE_RGB	8-bit blue, green and red samples packed into a 1 byte each.

Table 1. Some BufferedImage Types.

An image is made up of pixels (of course), and each pixel is composed from (perhaps) several *samples*. Samples hold the colour component data that combine to make the pixel's overall colour.

A standard set of colour components are red, green, and blue, RGB for short. The pixels in a transparent or translucent colour image will also include an alpha (A) component to specify the degree of transparency for the pixels.

A grayscale image only utilizes a single sample per pixel.

BufferedImage types specify how the samples that make up a pixel's data are packed together. For example, TYPE_INT_ARGB packs its four samples into 8 bits each, so that a single pixel can be stored in a single 32-bit integer. This is shown graphically in Figure 4.

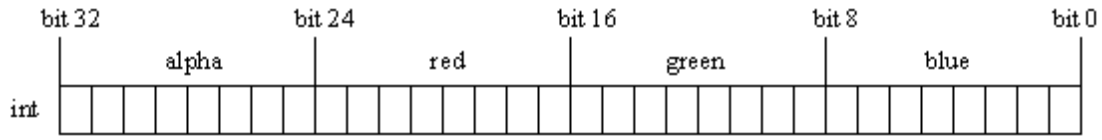

Figure 4. A TYPE_INT_ARGB Pixel.

This format is used for the BufferedImage object in makeBIM(), since it's the most general. The red, green, blue and alpha components can have 256 different values (2^8), with 255 being full on. For the alpha part, 0 means fully transparent, 255 fully opaque.

The question is whether such flexibility is always needed, for instance, when the image is opaque or a grayscale? Also, it may not be possible to accurately map an image stored using a drastically different colour model to the range of colours here. An example, would be an image using 16 bit colour components.

Nevertheless, makeBIM() deals with the normal range of image formats, GIF, JPEG, and PNG, and so is satisfactory for our needs.

A more rigorous solution is to use AWT's imaging processing capabilities to analyze the source Image object, and construct a BufferedImage accordingly. A PixelGrabber can access the pixel data inside the Image, and determine whether there is an alpha component, and whether the image is grayscale or RGB.

A third answer is to go back to basics, and ask exactly why the image is being converted to a BufferedImage object at all? A common reason is to make use of BufferedImageOp operations, but this can be achieved without a conversion. It is possible to wrap a BufferedImageOp object in a BufferedImageFilter to make it behave like an AWT ImageFilter.

4.2. The Internals of BufferedImage

The data maintained by a BufferedImage object is represented by Figure 5.

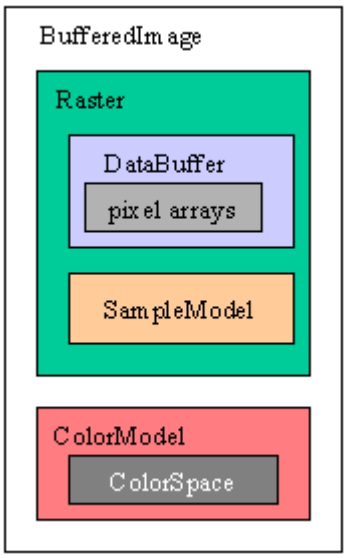

Figure 5. BufferedImage Internals.

A BufferedImage instance is made up of a Raster object which stores the pixel data, and ColorModel which contains methods for converting that data into colours.

DataBuffer holds a rectangular array of numbers that make up the data, while SampleModel explains how those numbers are grouped into the samples for each pixel.

One way of viewing the image is as a collection of *bands* (or *channels*): a band is a collection of the same samples from all the pixels. For instance, an ARGB file contains four bands for alpha, red, green, and blue.

The ColorModel object defines how the samples in a pixel are mapped to colour components, and ColorSpace specifies how the components are combined to form a renderable colour.

Java 2D supports many colour spaces, including sRGB, the standardized RGB colour space, which corresponds to the TYPE_INT_ARGB format in Figure 4. The BufferedImage method getRGB(x,y) utilizes this format: (x,y) is the pixel coordinate, and a single integer is returned which, with the help of some simple bit manipulation, can expose its 8-bit alpha, red, green, and blue components.

setRGB() updates an image pixel, and there are also get and set methods that manipulate all the pixels as an array of integers. Two of the ImagesTests visual effects use these methods (see section 14.8).

4.3. BufferedImageOp Operations

Java 2D's image processing operations are (for the most part) subclasses of the BufferedImageOp interface, which supports an *immediate imaging model*. Image processing is a filtering operation which takes a source BufferedImage as input, and produces a new BufferedImage as output. The idea is captured by Figure 6.

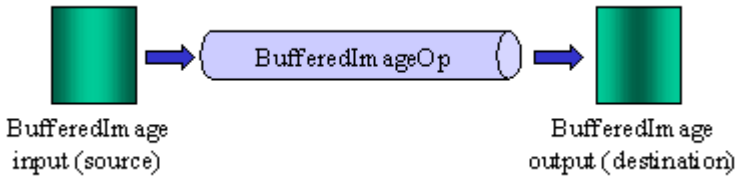

Figure 6. The BufferedImageOp Imaging Model.

This doesn't appear to be much different from the ImageFilter idea in Figure 2. The differences are in the expressibility of the operations which can, for instance, manipulate groups of pixels, and affect the colour space. This is due to the data model offered by BufferedImage.

The code fragment below shows the creation of a destination BufferedImage, by manipulating a source BufferedImage using RescaleOp, which implements the BufferedImageOp interface.

```
RescaleOp negOp = new RescaleOp(-1.0f, 255f, null);
BufferedImage destination = negOp.filter(source, null);
```

The filter() method does the work, taking the source image as input, and returning the resulting image.

Certain image processing operations can be carried out *in place*, which means that the destination BufferedImage can be the source.

Another common way of using a BufferedImageOp is as an argument to drawImage(): the image will be processed, and the result drawn straight to the screen:

```
g2d.drawImage(source, negOp, x, y);
```

The (x,y) coordinate is where the top-left corner of the resulting image will be placed.

The predefined BufferedImageOp image processing classes are listed in Table 2.

Class Name	Description	Some Possible Effects	In Place?
AffineTransformOp	A geometric transformation is applied to the image's coords.	Scaling, rotating, shearing.	No.
BandCombineOp	Combine bands in the image's Raster	Change the mix of colours.	Yes.
ColorConvertOp	ColorSpace conversion.	Convert RGB to grayscale.	Yes.
ConvolveOp	Combine groups of pixel values to obtain a new pixel value.	Blurring, sharpening, edge detection.	No.
LookupOp	Modify pixel values based on a table lookup.	Colour inversion, reddening, brightening, darkening.	Yes.
RescaleOp	Modify pixel values based on a linear equation.	Mostly the same as LookupOp.	Yes.

Table 2. Image Processing Classes.

Various examples of these, together with more detailed explanations of the operations, will be given when we discuss ImagesTests.

5. Managed Images

A managed image is automatically cached in VRAM (video memory) by the JVM. When drawImage() is applied to its original version located in RAM (system memory), the JVM uses the VRAM cache instead, and employs a hardware copy (blit) to draw it to the screen. The payoff is speed, since a hardware blit will be faster than a software-based copy from RAM to the screen. This idea is illustrated by Figure 7.

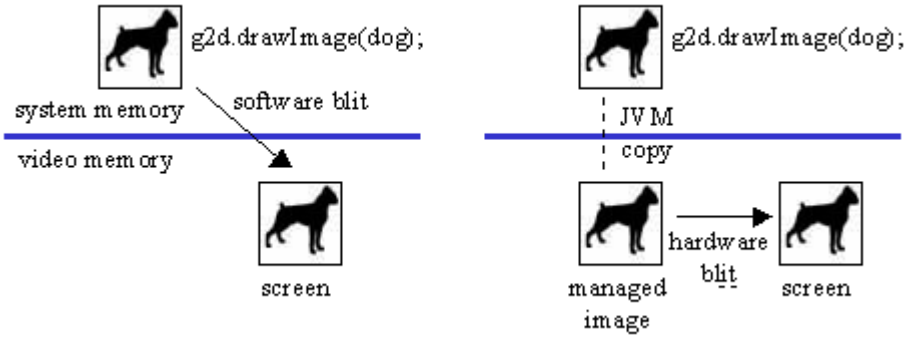

Figure 7. Drawing Images and Managed Images.

A managed image is not explicitly created by the programmer – there is *no* ManagedImage class which can be used to instantiate suitable objects. Managed images are created at the whim of the JVM, although the programmer can 'encourage' the JVM to make them.

Image, ImageIcon, and BufferedImage objects qualify to become managed images, if they have been created with createImage(), createCompatibleImage(), read in with getImage() or ImageIO's read(), or created with the BufferedImage() constructor. Opaque images and images with BITMASK transparency (e.g. GIF files) can be managed. Translucent images can also be managed, but require property flags to be set, which vary between MS Windows and Linux/Solaris.

The JVM will copy an image to VRAM when it detects that the image has not been changed/edited for a 'significant' amount of time. Typically, this means when two consecutive drawImage() calls have used the same image. The VRAM copy will be scrapped if the original image is manipulated by an operation which is not hardware accelerated, and the next drawImage() will switch back to the system memory version.

Exactly which operations are hardware accelerated depends on the OS. Virtually nothing aside from image translation is accelerated in MS Windows; this is not due to inadequacies in DirectDraw, but rather to the Java interface. The situation is a lot better on Linux/Solaris where all affine transformations, composites and clips will be accelerated. However, these features depend on underlying OS support for a version of OpenGL that offers pbuffers. A pbuffer is a kind of offscreen rendering area, somewhat like a pixmap but with support for accelerated rendering.

Bearing in mind how the JVM deals with managed images, it is inadvisable to excessively modify them at run time since their hardware acceleration will probably be lost, at least for a short time.

In some older documentation, managed images are known as *automated images*.

6. VolatileImage

Whereas managed images are created by the JVM, the VolatileImage class allows the programmer to create and manage their own hardware-accelerated images. In fact, a VolatileImage object exists only in VRAM, it has no system memory copy at all (see Figure 8).

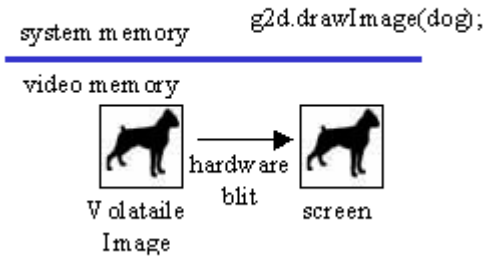

Figure 8. A VolatileImage Object.

VolatileImage objects stay in VRAM, and so get the benefits of hardware blitting all the time. Well, that's sort of true, but it depends on the underlying OS. On MS Windows, VolatileImage is implemented using DirectDraw which manages the image in video memory, and may decide to grab the memory back to give to another task,

such as a screensaver or new foreground process. This means that the programmer must keep checking his VolatileImage objects to see if they're still around. If a VolatileImage's memory is lost, then the programmer has to recreate the object.

The situation is better on Linux/Solaris since VolatileImage is implemented with OpenGL pbuffers, which cannot be deallocated by the OS.

Another drawback with VolatileImages is that any processing of an image must be done in VRAM, which is generally much slower to do as a *software* operation than similar calculations in system memory. Of course, if the manipulation (e.g. applying an affine transform such as a rotation) can be done by the VRAM *hardware*, then it will be must faster than in system memory. Unfortunately, the mix of software/hardware-based operations depends on the OS, as explained above for managed images.

Bearing in mind the issues surrounding VolatileImage, when exactly it is useful? Its key benefit over managed images is that the programmer is in charge rather than the JVM. The programmer can decide when to create, update, delete an image.

However, managed image support is becoming so good in the JVM that most programs probably do not need the complexity that VolatileImage adds to the code. ImagesTests only uses managed images, which it encourages by only creating BufferedImages.

7. Java 2D Speed

The issues over the speed of Java 2D operations mirror our discussion about the use of managed images and VolatileImages, since speed depends on which operations are hardware accelerated, and that depends on the OS.

On MS Windows, hardware acceleration is mostly restricted to the basic 2D operations such as filling, copying rectangular areas, line drawing (vertical and horizontal only), and basic text rendering. Unfortunately, the fun parts of Java 2D such as curves, antialiasing, compositing, all use software rendering.

In Linux/Solaris, so long as OpenGL buffers are supported, then most elements of Java 2D are accelerated.

The situation described here is for J2SE 1.5, and will undoubtedly improve. The best check is to profile your code. General profiling techniques are explained in Appendix ??, but a Java 2D-specific approach is described later (switching on Java 2D low-level operation logging).

8. Portability and Java 2D

The current situation with Java 2D's hardware acceleration, exposes a rather nasty portability problem with Java. Graphics, especially gaming graphics, require speed, and the Java implementers have taken a two-track approach. The MS Windows-based version of Java utilize DirectX and other Windows features, while on other platforms the software underlying Java 2D relies on OpenGL.

This approach seems like a unnecessary duplication of effort, and a wonderful source of confusion to programmers. It is not surprising though, since the same situation exists for Java 3D, as described in chapters 8 ?? and beyond.

My opinion is that Java should restrict itself to OpenGL, an open standard that is under active development by many talented people around the world. In fact, this view may already be prevailing inside Sun, indicated by its promotion of JOGL (https://jogl.dev.java.net/), a Java/OpenGL binding.

9. JAI

Java Advanced Imaging (JAI) offers extended image processing capabilities beyond those found in Java 2D. For example, geometric operations include translation, rotation, scaling, shearing, *and* transposition and warping. Pixel-based operations utilize lookup tables and rescaling equations, but can be applied to multiple sources, combined to get a single outcome. Modifications can be restricted to regions in the source, statistical operations are available (e.g. mean and median), and frequency domains can be employed.

An intended application domain for JAI is the manipulation of images too large to be loaded into memory in their entirety. A TiledImage class supports pixel editing based on tiles, which can be processed and displayed independently of their overall image.

Image processing can be distributed over a network by using RMI to farm out areas of the image to servers, with the results returned to the client for displaying.

JAI employs a *pull imaging model*, where an image is constructed from a series of source images, arranged into a graph. Only when a particular pixel (or tile) is required, will the image request data from its sources.

These kinds of extended features are not usually required for gaming, and are not used here.

More information on JAI can be found at its home page at http://java.sun.com/products/java-media/jai/.

10. ImagesTests Overview

The ImagesTests application is shown in Figure 1, and again in Figure 9 below. The screenshot in Figure 9 includes the name of the images, for ease of reference later.

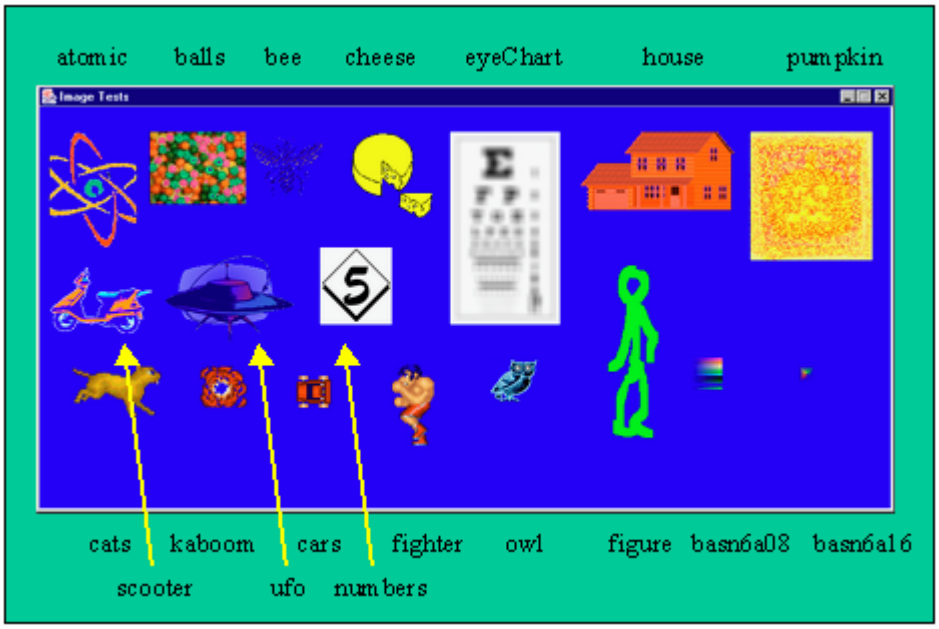

Figure 9. ImagesTests and Image Names.

Table 3 lists the image names against the visual effect they demonstrate.

Image Name	Visual Effect
atomic	rotation
balls, basn6a08	mixed colours
bee	teleportation (uneven fading)
cheese	horizontal/vertical flipping
eyeChart	progressive blurring
house	reddening
pumpkin	zapping (red/yellow pixels)
scooter	brightening
ufo	fading
owl	negation
basn6a16	resizing
cars, kaboom, cats, figure	numbered animation
fighter	named animation
numbers	callback animation

Table 3. Images Names and their Visual Effects.

The majority of the images are GIFs with a transparent background; balls.jpg is the only JPEG. The PNG files are: owl.png, pumpkin.png, basn6a08.png, and basn6a16.png. The latter two use translucency, and come from the PNG suite maintained by Willem van Schaik at http://www.schaik.com/pngsuite/pngsuite.html.

I've utilized several images from the excellent SpriteLib sprite library by Ari Feldman, available at http://www.arifeldman.com/games/spritelib.html, notably for the 'cats', 'kaboom', 'cars', and 'fighter' animations.

10.1. UML Diagrams for ImagesTests

Figure 10 shows the UML diagrams for the classes in the ImagesTests application. The class names, public methods and constants are shown.

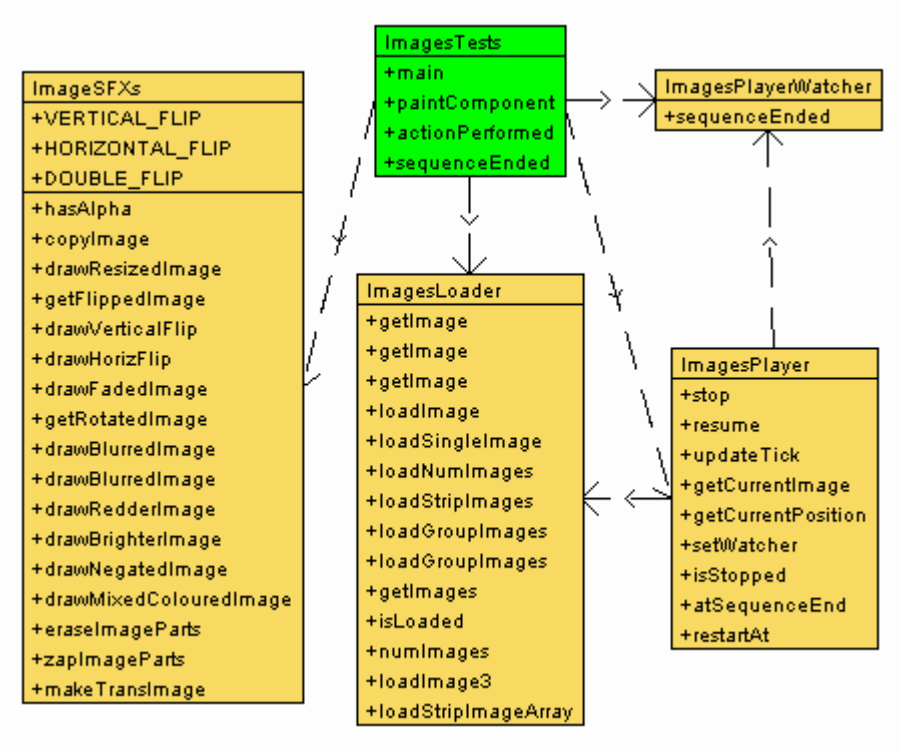

Figure 10. UML Class Diagrams for ImagesTests.

ImagesTests creates a JFrame and the JPanel where the images are drawn, and starts a Swing timer to update its images every 0.1 second.

ImagesTests employs an ImagesLoader object to load the images named in a configuration file (imsInfo.txt in the Images/ subdirectory).

The visual effects methods, such as blurring, are grouped together in ImagesSFXs. Animations represented by sequences of images (e.g. 'numbers', 'cars', 'kaboom', 'cats', and 'figure') are controlled by ImagesPlayer objects. A sequence may be shown repeatedly, once only, be stopped and restarted.

A completed animation sequence can call sequenceEnded() in an object which implements the ImagesPlayerWatcher interface. ImagesTests implements ImagesPlayerWatcher, and is used as a callback by the 'numbers' sequence.

11. The ImagesLoader Class

The ImagesLoader class can load images in four different formats, which we call 'o', 'n', 's', and 'g' images.

The images are assumed to be in a local JAR file, in a subdirectory Images/ below ImagesLoader. They are loaded as BufferedImages using ImageIO's read(), so they can become managed images.

The typical way of using an ImagesLoader object is to supply it with a configuration file containing the filenames of the required images, to be loaded before game play begins. However, it is possible to call ImagesLoader's load methods at any time during execution.

The imsInfo.txt configuration file used in the ImagesTests example is listed below.

```
// imsInfo.txt images

o atomic.gif
o balls.jpg
o bee.gif
o cheese.gif
o eyeChart.gif
o house.gif
o pumpkin.png
o scooter.gif
o ufo.gif
o owl.png

n numbers*.gif  6
n figure*.gif 9

g fighter  left.gif right.gif still.gif up.gif

s cars.gif 8
s cats.gif 6
s kaboom.gif 6

o basn6a08.png
o basn6a16.png
```

Blank lines, and lines beginning with '//', are ignored by the loader. The syntax for the four image formats are:

```
o <fnm>
n <fnm*.ext> <number>
s <fnm> <number>
g <name> <fnm> [ <fnm> ]*
```

An 'o' line causes a single filename, called <fnm>, to be loaded from Images/.

A 'n' line loads a series of numbered image files, whose filenames use the numbers 0 - <number>-1 in place of the '*' character in the filename. For example:

```
n numbers*.gif  6
```

means that the files numbers0.gif, numbers1.gif, up to numbers5.gif should be loaded.

A 's' line loads a *strip file* (called fnm) containing a single row of <number> images. After the file's graphic has been loaded, it is automatically divided up into the component images. For instance:

```
s kaboom.gif 6
```

refers to the strip file "kaboom.gif" containing a row of six images, as shown in Figure 11.

Figure 11. The kaboom.gif Strip File.

A 'g' line specifies a group of files with different names. After being loaded, the images will be accessible using a positional notation or by means of their filenames (minus the extension). For example, the 'fighter' 'g' images are defined as:

```
g fighter  left.gif right.gif still.gif up.gif
```

Subsequently, the image in right.gif can be accessed using the number 1 or the string "right".

11.1. Internal Data Structures

The ImagesLoader object creates two main data structures as it loads the images, both of them HashMaps:

```
private HashMap  imagesMap, gNamesMap;
```

The imagesMap key is the image's 'name', and its value is an ArrayList of BufferedImage objects associated with that name. The exact meaning of 'name' varies depending on the type of image that was loaded.

For an 'o' image (e.g. "o atomic.gif"), the name is the filename minus its extension (i.e. "atomic"), and the ArrayList holds just a single image.

For a 'n' image (e.g. "n numbers*.gif 6"), the name is the part of the filename before the * (i.e. "numbers"), and the ArrayList holds several images (6 in this case).

For a 's' image (e.g. "s cars.gif 8"), the name is the filename minus the extension (i.e. "cars"), and the ArrayList holds the images pulled from the strip graphic (8 for this example).

For a 'g' image (e.g. "g fighter left.gif right.gif still.gif up.gif"), the name is the string after the 'g' character (i.e. "fighter"), and the ArrayList is as large as the sequence of filenames given (4).

The loading of 'g' images also causes updates to the gNamesMap HashMap. Its key is the 'g' name (e.g. "fighter"), but its value is an ArrayList of filename *Strings* (minus their extensions). For instance, the 'fighter' name has an ArrayList associated with it holding the strings "left", "right", "still", and "up".

11.2. Getting an Image

The image accessing interface is surprisingly uniform, independent of whether 'o', 'n', 's', or 'g' images are being accessed.

There are three public getImage() methods in ImagesLoader, and also getImage**s**(). Their prototypes:

```
BufferedImage getImage(String name);
BufferedImage getImage(String name, int posn);
BufferedImage getImage(String name, String fnmPrefix);

ArrayList getImages(String name);
```

The single argument getImage() returns the image associated with name, and is intended primarily for accessing 'o' images (which only have a single image). If a 'n', 's', or 'g' image is accessed then the first image in the ArrayList is returned.

The two-argument getImage() which takes an integer position argument is more useful for accessing 'n', 's', and 'g' names (which will have multiple images in their ArrayLists). If the supplied number is negative then the first image is returned. If the number is too large then it is reduced modulo the ArrayList size.

The third getImage() method takes a String argument, and is aimed at 'g' images. The String should be a filename, which is used to index into the ‘g’ name’s ArrayList.

The getImages() method returns the entire ArrayList for the given name.

11.3. Using ImagesLoader

ImagesTests employs ImagesLoader by supplying it with a images configuration file:

```
ImagesLoader imsLoader = new ImagesLoader("imsInfo.txt");
```

The ImagesLoader constructor assumes the file (and all the images) are in the Images/ subdirectory below the current directory, and that everything is packed inside a JAR. Details about creating such a JAR are given at the end of this chapter.

Loading 'o' images is straightforward:

```
BufferedImage atomic = imsLoader.getImage("atomic");
```

Loading 'n', 's', and 'g' images usually requires a numerical value:

```
BufferedImage cats1 = imsLoader.getImage("cats", 1);
```

A related method is numImages(), which returns the number of images associated with a given name:

```
int numCats = imsLoader.numImage("cats");
```

'g' images can be accessed using a filename prefix:

```
BufferedImage leftFighter = imsLoader.getImage("fighter", "left");
```

If a requested image cannot be found, then null is returned by the loader.

An alternative way of using ImagesLoader is to create an 'empty' one (no configuration file is supplied to the constructor). Then public methods for loading 'o', 'n', 's', and 'g' images can be called by the application:


```
ImagesLoader imsLoader = new ImagesLoader();

imsLoader.loadSingleImage("atomic.gif");
imsLoader.loadNumImages("numbers*.gif", 6);
imsLoader.loadStripImages("kaboom.gif", 6);

String[] fnms = {"left.gif", "right.gif", "still.gif", "up.gif"};
imsLoader.loadGroupImages("fighter", fnms );
```

11.4. Implementation Details

A large part of ImagesLoader is given over to parsing and error checking. The top-level method for parsing the configuration file is loadImagesFile().

```
private void loadImagesFile(String fnm)
{
  String imsFNm = IMAGE_DIR + fnm;
  System.out.println("Reading file: " + imsFNm);
  try {
    InputStream in = this.getClass().getResourceAsStream(imsFNm);
    BufferedReader br = new BufferedReader(
                              new InputStreamReader(in));
    String line;
    char ch;
    while((line = br.readLine()) != null) {
      if (line.length() == 0)  // blank line
        continue;
      if (line.startsWith("//"))   // comment
        continue;
      ch = Character.toLowerCase( line.charAt(0) );
      if (ch == 'o')  // a single image
        getFileNameImage(line);
      else if (ch == 'n')  // a numbered sequence of images
        getNumberedImages(line);
      else if (ch == 's')  // an images strip
        getStripImages(line);
      else if (ch == 'g')  // a group of images
        getGroupImages(line);
      else
        System.out.println("Do not recognize line: " + line);
    }
    br.close();
  }
  catch (IOException e)
  { System.out.println("Error reading file: " + imsFNm);
    System.exit(1);
  }
}  // end of loadImagesFile()
```

A line is read in at a time, and a multi-way branch decides which syntactic form should be processed depending on the first character on the input line.

The input stream coming from the configuration file is created using getResourceAsStream(), which is needed when the application and all the resources all wrapped up inside a JAR.

getFileNameImage() is quite typical: it extracts the tokens from the line, processing them by calling loadSingleImage().

```
private void getFileNameImage(String line)
// format is   o <fnm>
{ StringTokenizer tokens = new StringTokenizer(line);

  if (tokens.countTokens() != 2)
    System.out.println("Wrong no. of arguments for " + line);
  else {
    tokens.nextToken();    // skip command label
    System.out.print("o Line: ");
    loadSingleImage( tokens.nextToken() );
  }
}
```

loadSingleImage() is the public method for loading an 'o' image. If an entry for the image's name does not already exist then imagesMap is extended with a new key (holding name), and an ArrayList containing a single BufferedImage.

```
public boolean loadSingleImage(String fnm)
{
  String name = getPrefix(fnm);

  if (imagesMap.containsKey(name)) {
    System.out.println( "Error: " + name + "already used");
    return false;
  }

  BufferedImage bi = loadImage(fnm);
  if (bi != null) {
    ArrayList imsList = new ArrayList();
    imsList.add(bi);
    imagesMap.put(name, imsList);
    System.out.println("  Stored " + name + "/" + fnm);
    return true;
  }
  else
    return false;
}  // end of loadSingleImage()
```

11.5. Image Loading

We finally arrive at the image loading method, loadImage(), which is also at the heart of the processing of 'n' and 'g' lines. Its implementation is almost identical to the loadImage() method described back in section 4.

```
public BufferedImage loadImage(String fnm)
{
  try {
    BufferedImage im =  ImageIO.read(
              getClass().getResource(IMAGE_DIR + fnm) );

    int transparency = im.getColorModel().getTransparency();
    BufferedImage copy =  gc.createCompatibleImage(
```



```
                                im.getWidth(), im.getHeight(),
                                transparency );

      // create a graphics context
      Graphics2D g2d = copy.createGraphics();

      // reportTransparency(IMAGE_DIR + fnm, transparency);

      // copy image
      g2d.drawImage(im,0,0,null);
      g2d.dispose();
      return copy;
    }
    catch(IOException e) {
      System.out.println("Load Image error for " +
                    IMAGE_DIR + "/" + fnm + ":\n" + e);
      return null;
    }
} // end of loadImage() using ImageIO
```

reportTransparency() is a utility for printing out the transparency value of the loaded image. It is useful for checking whether the transparency/translucency of the image has been detected.

ImagesLoader also contains two other versions of loadImages(), imaginatively called loadImages2() and loadImages3(). They play no part in the functioning of the class, and are only included to show how BufferedImages can be loaded using ImageIcon or Image's getImage(). The ImageIcon code in loadImages2() uses:

```
ImageIcon imIcon = new ImageIcon(
                    getClass().getResource(IMAGE_DIR + fnm) );
```

and then calls makeBIM() to convert its Image into a BufferedImage. makeBIM() is described in section 4.1.

The Image code in loadImage3() uses a MediaTracker to delay execution until the image is fully loaded (see section 2.1), and then calls makeBIM() to obtain a BufferedImage.

11.6. Loading Strip File Images

The images from a strip file are obtained in steps: first the entire graphic is loaded from the file, then cut into pieces, and each resulting image is placed in an array. This array is subsequently stored as an ArrayList in imagesMap under the 's' name.

```
public BufferedImage[] loadStripImageArray(String fnm, int number)
{
  if (number <= 0) {
    System.out.println("number <= 0; returning null");
    return null;
  }

  BufferedImage stripIm;
  if ((stripIm = loadImage(fnm)) == null) {
    System.out.println("Returning null");
    return null;
  }
```



```
    int imWidth = (int) stripIm.getWidth() / number;
    int height = stripIm.getHeight();
    int transparency = stripIm.getColorModel().getTransparency();

    BufferedImage[] strip = new BufferedImage[number];
    Graphics2D stripGC;

    // each BufferedImage from the strip file is stored in strip[]
    for (int i=0; i < number; i++) {
      strip[i]=gc.createCompatibleImage(imWidth,height,transparency);

      // create a graphics context
      stripGC = strip[i].createGraphics();

      // copy image
      stripGC.drawImage(stripIm,
                  0,0, imWidth,height,
                  i*imWidth,0, (i*imWidth)+imWidth,height,
                  null);
      stripGC.dispose();
    }
    return strip;
  } // end of loadStripImageArray()
```

drawImage() is used to clip the images out of the strip.

An alternative approach would be to use a CropImageFilter combined with a FilteredImageSource. This seems like too much work for images which are positioned so simply.

12. The ImagesTests Class

Perhaps a surprising thing about ImagesTests is that it uses a Swing timer to animate its image effects rather than the active rendering approach developed in early chapters. This is purely a matter of keeping the code simple, since the high accuracy offered by active rendering is not required. The visual effects employed here are generally composed from 5-10 distinct frames, displayed over the course of 1-2 seconds; this implies a need for 10 FPS at most, which is within the capabilities of the Swing timer.

If necessary, the effects techniques can be easily translated to an active rendering setting.

The timer-driven framework is illustrated by Figure 12. The details of actionPerformed() and paintComponent() will be explained below.

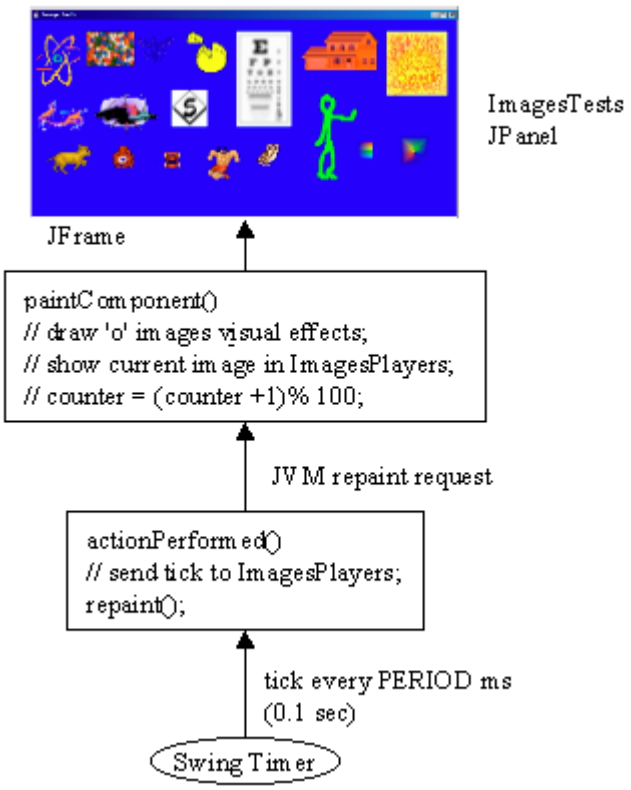

Figure 12. ImagesTests and the Swing Timer.

ImagesTests maintains a global variable counter, which starts at 0, and is incremented at the end of each paintComponent() call, modulo 100. The modulo operation isn't significant: it's used to keep the counter value from becoming excessively large. counter is used in many places in the code, often to generate input arguments to the visual effects.

12.1. Starting ImagesTests

The main() method for ImagesTests creates a simple JFrame and adds the ImagesTests JPanel to it.


```
 public static void main(String args[])
 {
   // switch on translucency acceleration in Windows
   System.setProperty("sun.java2d.translaccel", "true");
   System.setProperty("sun.java2d.ddforcevram", "true");

   // switch on hardware acceleration if using OpenGL with pbuffers
   // System.setProperty("sun.java2d.opengl", "true");

   ImagesTests ttPanel = new ImagesTests();

   // create a JFrame to hold the test JPanel
   JFrame app = new JFrame("Image Tests");
   app.getContentPane().add(ttPanel, BorderLayout.CENTER);
   app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

   app.pack();
   app.setResizable(false);
   app.show();
} // end of main()
```

Of more interest are the calls to setProperty(). If we require hardware acceleration of translucent images on MS Windows (e.g. for the PNG files basn6a08.png and basn6a16.png) then the Java 2D 'translaccel' and 'ddforcevram' flags should be switched on. They also accelerate alpha composite operations.

On Linux/Solaris, only the 'opengl' flag is required for hardware acceleration but, as mentioned earlier, the version of OpenGL must support pbuffers.

The ImagesTests constructor initiates image loading, creates the ImageSFXs visual effects object, obtains references to the 'o' images, and starts the timer.

```
// various globals
private ImagesLoader imsLoader;   // the image loader
private int counter;
private boolean justStarted;
private ImageSFXs imageSfx;       // the visual effects class

private GraphicsDevice gd;      // for reporting accl. memory usage
private int accelMemory;
private DecimalFormat df;
      :

public ImagesTests()
{
  df = new DecimalFormat("0.0");  // 1 dp

  GraphicsEnvironment ge =
         GraphicsEnvironment.getLocalGraphicsEnvironment();
  gd = ge.getDefaultScreenDevice();

  accelMemory = gd.getAvailableAcceleratedMemory();  // in bytes
  System.out.println("Initial Acc. Mem.: " +
        df.format( ((double)accelMemory)/(1024*1024) ) + " MB" );

  setBackground(Color.white);
```



```
    setPreferredSize( new Dimension(PWIDTH, PHEIGHT) );

    // load and initialise the images
    imsLoader = new ImagesLoader(IMS_FILE);   // "imsInfo.txt"
    imageSfx = new ImageSFXs();
    initImages();

    counter = 0;
    justStarted = true;

    new Timer(PERIOD, this).start();    // PERIOD = 0.1 sec
  } // end of ImagesTests()
```

The getAvailableAcceleratedMemory() call returns the current amount of available hardware accelerated memory. The application continues to report this value, as it changes, to give an indication of when BufferedImage objects become managed images.

12.2. Initializing Images

initImages() does three tasks: it stores references to the 'o' images as global variables, creates ImagesPlayers objects for the 'n' and 's' images, and references the first 'g' 'fighter' image, its "left" image.

```
// global variables
// hold the single 'o' images
private BufferedImage atomic, balls, bee, cheese, eyeChart,
                      house, pumpkin, scooter,
                      fighter, ufo, owl, basn8, basn16;

// for manipulating the 'n' and 's' images
private ImagesPlayer numbersPlayer, figurePlayer, carsPlayer,
                     catsPlayer, kaboomPlayer;
       :

private void initImages()
{
  // initialize the 'o' image variables
  atomic = imsLoader.getImage("atomic");
  balls = imsLoader.getImage("balls");
  bee = imsLoader.getImage("bee");
  cheese = imsLoader.getImage("cheese");
  eyeChart = imsLoader.getImage("eyeChart");
  house = imsLoader.getImage("house");
  pumpkin = imsLoader.getImage("pumpkin");
  scooter = imsLoader.getImage("scooter");
  ufo = imsLoader.getImage("ufo");
  owl = imsLoader.getImage("owl");
  basn8 = imsLoader.getImage("basn6a08");
  basn16 = imsLoader.getImage("basn6a16");

  /* Initialize ImagesPlayers for the 'n' and 's' images.
     The 'numbers' sequence is not cycled, the other are.
  */
  numbersPlayer =
       new ImagesPlayer("numbers", PERIOD, 1, false, imsLoader);
  numbersPlayer.setWatcher(this);
```



```
          // report the sequence's finish back to ImagesTests

    figurePlayer =
         new ImagesPlayer("figure", PERIOD, 2, true, imsLoader);
    carsPlayer =
         new ImagesPlayer("cars", PERIOD, 1, true, imsLoader);
    catsPlayer =
         new ImagesPlayer("cats", PERIOD, 0.5, true, imsLoader);
    kaboomPlayer =
         new ImagesPlayer("kaboom", PERIOD, 1.5, true, imsLoader);

    // the 1st 'g' image for 'fighter' is set using a filename prefix
    fighter = imsLoader.getImage("fighter", "left");
  }  // end of initImages()
```

The ImagesPlayer class wraps up code for playing a sequence of images. ImagesTests uses ImagesPlayer objects for animating the 'n' and 's' 'figure', cars', 'kaboom', and 'cats' images. Each sequence is shown repeatedly.

'numbers' is also an 'n' name, so made up of several images, but its ImagesPlayer is set up a little differently. The player will call sequenceEnded() in ImagesTests when the end of the sequence is reached, and it doesn't play the images again. The callback requires that ImagesTests implements the ImagesPlayerWatcher interface:

```
public class ImagesTests extends JPanel
               implements ActionListener, ImagesPlayerWatcher
{  ...

  public void sequenceEnded(String imageName)
  // called by ImagesPlayer when its images sequence has finished
  {  System.out.println( imageName + " sequence has ended");  }

}
```

The name of the sequence (i.e. 'numbers') is passed as an argument to sequenceEnded () by its player. The implementation in ImagesTests only prints out a message, but it could do something more fanciful.

12.3. Updating the Images

Image updating is carried out by imagesUpdate() when actionPerformed() is called (i.e. every 0.1 second).

```
  public void actionPerformed(ActionEvent e)
  // triggered by the timer: update, repaint
  {
    if (justStarted)   // don't do updates the first time through
      justStarted = false;
    else
      imagesUpdate();

    repaint();
  } // end of actionPerformed()


  private void imagesUpdate()
```



```
{
  // numbered images ('n' images); using ImagesPlayer
  numbersPlayer.updateTick();
  if (counter%30 == 0)     // restart image sequence periodically
    numbersPlayer.restartAt(2);

  figurePlayer.updateTick();

  // strip images ('s' images); using ImagesPlayer
  carsPlayer.updateTick();
  catsPlayer.updateTick();
  kaboomPlayer.updateTick();

  // grouped images ('g' images)
  // The 'fighter' images are the only 'g' images in this example.
  updateFighter();
} // end of imagesUpdate()
```

imagesUpdate() does nothing to the 'o' images, since they are processed by paintComponent(); instead, it concentrates on the 'n', 's', and 'g' images.

updateTick() in called in all of the ImagesPlayers (i.e. for 'numbers', 'figure', 'cars', 'cats', and 'kaboom'). This informs the players that another animation period has passed in ImagesTests. This is used to calculate timings, and determine which of the images in a sequence is the current one.

The 'n' 'numbers' images are utilized in a slightly different way: when the counter value reaches a multiple of 30, the sequence is restarted at image number 2:

```
if (counter%30 == 0)
   numbersPlayer.restartAt(2);
```

The on-screen behaviour of 'numbers' is to step through its 6 images (pictures of the numbers 0 to 5) and stop after calling sequenceEnded() in ImagesTests. Later, when ImagesTests' counter reaches a multiple of 30, the sequence will restart at picture 2, step through to picture 5 and stop again (after calling sequenceEnded() again). This behaviour will repeat whenever the counter reaches a multiple of 30.

Admittedly, this behaviour is somewhat strange, but it illustrates that ImagesPlayer can do more than just endlessly cycle through image sequences.

updateFighter() deals with the 'g' 'fighter' images, defined in "imsInfo.txt" as:

```
g fighter  left.gif right.gif still.gif up.gif
```

Back in initImages(), the global BufferedImage variable, fighter, was set to refer to the "left" image. updateFighter() cycles through the other images using the counter value, modulo 4.

```
private void updateFighter()
/* The images are shown using their filename prefixes (although a
   positional approach could be used, which would allow an
   ImagesPlayer to be employed.
*/
{ int posn = counter % 4;  // number of fighter images;
        // could use  imsLoader.numImages("fighter")
  switch(posn) {
```



```
      case 0:
        fighter = imsLoader.getImage("fighter", "left");
        break;
      case 1:
        fighter = imsLoader.getImage("fighter", "right");
        break;
      case 2:
        fighter = imsLoader.getImage("fighter", "still");
        break;
      case 3:
        fighter = imsLoader.getImage("fighter", "up");
        break;
      default:
        System.out.println("Unknown fighter group name");
        fighter = imsLoader.getImage("fighter", "left");
        break;
    }
  }  // end of updateFighter()
```

This code only updates the fighter reference; the image is not actually displayed until paintComponent() is called.

12.4. Painting the Images

paintComponent() has four main jobs:

1) it applies a visual effect to each 'o' image and displays the result;
2) it requests the current image from each ImagesPlayer and displays it;
3) it displays any change in the amount of hardware accelerated memory (VRAM);
4) it increments the counter (modulo 100).

```
public void paintComponent(Graphics g)
{
  super.paintComponent(g);
  Graphics2D g2d = (Graphics2D)g;

  //antialiasing
  g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                       RenderingHints.VALUE_ANTIALIAS_ON);

  // smoother (and slower) image transforms  (e.g. for resizing)
  g2d.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
                    RenderingHints.VALUE_INTERPOLATION_BILINEAR);

  // clear the background
  g2d.setColor(Color.blue);
  g2d.fillRect(0, 0, PWIDTH, PHEIGHT);

  // ------------------ 'o' images ---------------------
  /* The programmer must manually edit the code here in order to
     draw the 'o' images with different visual effects. */

  // drawImage(g2d, atomic, 10, 25);   // only draw the image

  rotatingImage(g2d, atomic, 10, 25);
  mixedImage(g2d, balls, 110, 25);
```



```
    teleImage = teleportImage(g2d, bee, teleImage, 210, 25);
    flippingImage(g2d, cheese, 310, 25);
    blurringImage(g2d, eyeChart, 410, 25);
    reddenImage(g2d, house, 540, 25);
    zapImage = zapImage(g2d, pumpkin, zapImage, 710, 25);
    brighteningImage(g2d, scooter, 10, 160);
    fadingImage(g2d, ufo, 110, 140);
    negatingImage(g2d, owl, 450, 250);
    mixedImage(g2d, basn8, 650, 250);
    resizingImage(g2d, basn16, 750, 250);

    // --------------- numbered images -------------------
    drawImage(g2d, numbersPlayer.getCurrentImage(), 280, 140);
    drawImage(g2d, figurePlayer.getCurrentImage(), 550, 140);

    // --------------- strip images ----------------------
    drawImage(g2d, catsPlayer.getCurrentImage(), 10, 235);
    drawImage(g2d, kaboomPlayer.getCurrentImage(), 150, 250);
    drawImage(g2d, carsPlayer.getCurrentImage(), 250, 250);

    // --------------- grouped images --------------------
    drawImage(g2d, fighter, 350, 250);

    reportAccelMemory();
    counter = (counter + 1)% 100;    // 0-99 is a large enough range
  } // end of paintComponent()
```

The calls to setRenderingHint() show how Java 2D can make rendering requests, based around a key and value scheme.

The use of antialiasing has no appreciable effect in this example, since there are no lines, shapes, or text drawn in the JPanel. It might be better not to bother with this hint, and so gain a little extra speed.

The interpolation hint is more useful, especially for the resizing operation. For instance, there is a noticeable improvement in the resized smoothness of 'basn6a16' with the hint compared to when the hint is absent.

The eleven visual effects applied to the 'o' images are explained below. However, all the methods have a similar interface, requiring a reference to the graphics context, the name of the image, and the (x,y) coordinate where the modified image will be drawn.

The 'n' and 's images are managed by ImagesPlayer objects, so the current image is obtained by calling the objects' getCurrentImage() method. The returned image reference is passed to drawImage(), which wraps a little extra error processing around Graphics' drawImage().

```
  private void drawImage(Graphics2D g2d, BufferedImage im,
                                                  int x, int y)
  /* Draw the image, or a yellow box with ?? in it if
     there is no image. */
  {
    if (im == null) {
      // System.out.println("Null image supplied");
      g2d.setColor(Color.yellow);
```



```
      g2d.fillRect(x, y, 20, 20);
      g2d.setColor(Color.black);
      g2d.drawString("??", x+10, y+10);
    }
    else
      g2d.drawImage(im, x, y, this);
  }
```

12.5. Information on Accelerated Memory

reportAccelMemory() prints the total amount of VRAM left, and the size of the change since the last report. This method is called at the end of every animation loop, but only writes output if there has been a change in the VRAM quantity.

```
private void reportAccelMemory()
// report any change in the amount of accelerated memory
{
  int mem = gd.getAvailableAcceleratedMemory();   // in bytes
  int memChange = mem - accelMemory;

  if (memChange != 0)
    System.out.println(counter + ". Acc. Mem: " +
              df.format( ((double)accelMemory)/(1024*1024) ) +
                         " MB; Change: " +
              df.format( ((double)memChange)/1024 ) + " K");
  accelMemory = mem;
}  // end of reportAcceleMemory()
```

A typical run of ImagesTests produces the following stream of messages (edited to emphasize the memory related prints):

```
DirectDraw surfaces constrained to use vram
Initial Acc. Mem.: 179.6 MB
Reading file: Images/imsInfo.txt
    // many information lines printed by the loader
          :
0. Acc. Mem: 179.6 MB; Change: -1464.8 K
1. Acc. Mem: 178.1 MB; Change: -115.5 K
3. Acc. Mem: 178.0 MB; Change: -113.2 K
4. Acc. Mem: 177.9 MB; Change: -16.3 K
5. Acc. Mem: 177.9 MB; Change: -176.8 K
numbers sequence has ended
6. Acc. Mem: 177.7 MB; Change: -339.0 K
7. Acc. Mem: 177.4 MB; Change: -99.0 K
     // 9 similar accelerated memory lines edited out
          :
18. Acc. Mem: 176.6 MB; Change: -16.2 K
19. Acc. Mem: 176.6 MB; Change: -93.9 K
21. Acc. Mem: 176.5 MB; Change: -48.8 K
25. Acc. Mem: 176.4 MB; Change: -60.0 K
numbers sequence has ended
numbers sequence has ended
numbers sequence has ended
numbers sequence has ended
          :
```


The images use about 120K in total, and appear to be moved into VRAM at load time, together with space for other rendering tasks (see line number 0). The large additional allocation is probably caused by Swing, which uses VolatileImage for its double buffering.

The later VRAM allocations are due to the rendering carried out by the visual effect operations, and stop occurring after the counter reaches 25 (or thereabouts). Since each loop takes about 0.1 seconds, this means that new VRAM allocations cease after about 2.5 seconds. VRAM is not claimed in every animation loop; for instance, no VRAM change is reported when the counter is 20, 22, and 24.

This behaviour can be understood by considering how the visual effects methods behave. Typically, every few animation frames they generate new images based on the original 'o' images. The operations are cyclic: after a certain number of frames they start over. The longest running cyclic is the fade method, which completes one cycle after 25 frames (2.5. seconds). Some of the operations write directly to the screen, and so will not require additional VRAM, while others use temporary BufferedImage variables; it is these which probably trigger the VRAM allocations. Once these claims have been granted, the space can be reused by the JVM when the methods restart there image processing cycle.

If the 'ddforcevram' flag is commented out from main() in ImagesTests:

```
// System.setProperty("sun.java2d.ddforcevram", "true");
```

Then only the first reduction to VRAM occurs (of about 1.4 MB). The subsequent requests are never made.

More information can be obtained about the low-level workings of Java 2D by turning on logging. For instance:

```
> java -Dsun.java2d.trace=log,count,out:log.txt ImagesTests
```

This will record all the internal calls made by Java 2D, together with a count of the calls, to the text file log.txt. Unfortunately, the sheer volume of data can be overwhelming, and it's probably better to utilize real profiling if this level of detail is required (see Appendix ??). However, if only the call counts are recorded, then the data is more manageable:

```
> java -Dsun.java2d.trace=count,out:log.txt ImagesTests
```

The vast majority of the calls, about 92% of them, are software rendering operations for drawing filled blocks of colour (the MaskFill() function). The percentage of hardware-assisted copies (blits) is greater when the 'ddforcevram' flag is switched on. These operations have "Win32", "DD", or "D3D" in their names. Nevertheless, it only increases from a paltry 0.5% to 2.3%.

The comparatively few hardware-based operations in the log is a reflection of Java's lack of support for image processing operations *in MS Windows*. Undoubtedly this will improve in later versions of Java and, in any case, greatly depends on the mix of operations that an application utilizes.

It may be worth moving the application to FSEM, since VolatileImages are automatically utilized for page flipping.

13. The ImagesPlayer Class

ImagesPlayer is aimed at displaying the sequence of images making up a 'n', 's', or 'g' set of images.

The ImagesPlayer constructor is supplied with the duration required for showing the entire sequence (seqDuration). This is used to calculate showPeriod, the amount of time each image will be the current one before the next image takes its place.

The animation period (animPeriod) argument of the ImagesPlayer constructor states how often the ImagesPlayer's updateTick() method will be called. The intention is that updateTick() will be called periodically by the update() method in the top-level animation framework (in our code, this is imagesUpdate() in ImagesTests).

The current animation time is calculated when updateTick() is called, and used to calculate imPosition, imPosition specifies which image should be returned when getCurrentImage() is called.

This situation is illustrated by Figure 13.

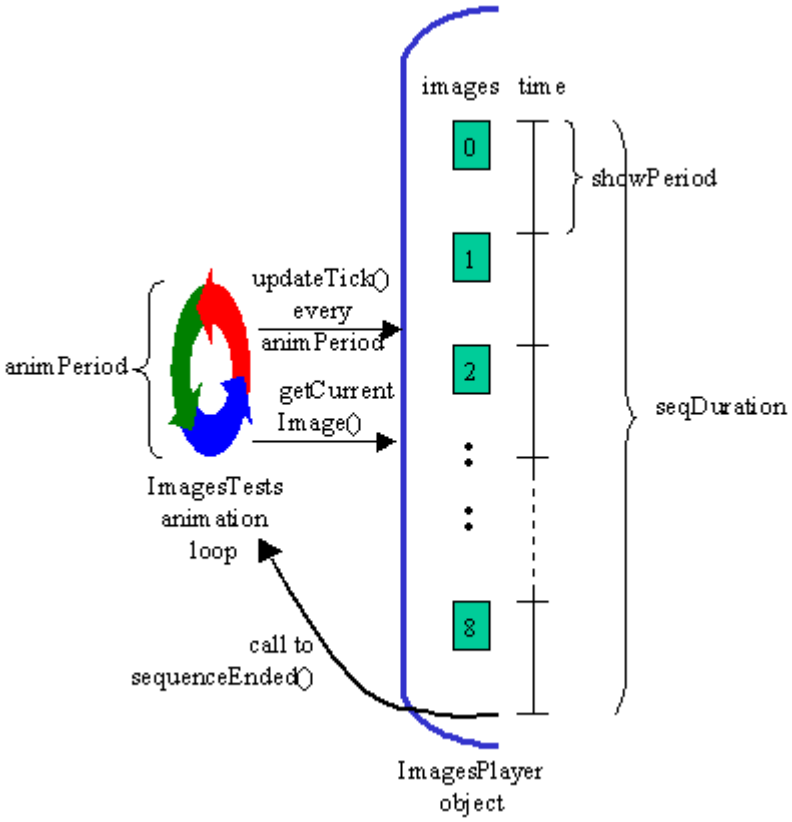

Figure 13. ImagesPlayer in Use.

This approach relies on the animation loop calling updateTick() regularly at a fixed time interval, which is true for ImagesTests, and the animation frameworks developed in earlier chapters.

Another implicit assumption is that showPeriod will be larger than the animPeriod. For example, showPeriod might be in tenths of seconds while animPeriod may be in milliseconds.

If showPeriod is less than animPeriod then rendering progresses too slowly to display all the images within the required seqDuration time, and images (frames) will be skipped.

When the sequence finishes, a callback, sequenceEnded(), can be invoked in a specified object implementing the ImagesPlayerWatcher interface. This is done for the 'n' 'numbers' images:

```
numbersPlayer =
     new ImagesPlayer("numbers", PERIOD, 1, false, imsLoader);
numbersPlayer.setWatcher(this);
        // report sequence's finish to ImagesTests
```

The ImagesPlayer constructor takes the name of the images, the animPeriod value, a seqDuration value, a boolean indicating if the sequence should repeat, and a reference to the ImagesLoader.

In the case of 'numbers', animPeriod is PERIOD (0.1 sec), seqDuration is 1 second, and the sequence will not repeat. Since there are 6 'numbers' files, showPeriod will be about 0.17 seconds, so (just) greater than the animPeriod.

ImagesPlayer includes public methods for stopping, resuming and restarting an animation at a given image position.

'numbers' uses the reStartAt() method to restart the animation at the third image

```
numbersPlayer.restartAt(2);
```

13.1. Implementation Details

The ImagesPlayer object maintains an animTotalTime variable, which holds the current time (in milliseconds) since the object was created. It is incremented when updateTick() is called.

```
public void updateTick()
// We assume that this method is called every animPeriod ms
{
  if (!ticksIgnored) {
    // update total animation time, modulo seq duration
    animTotalTime = (animTotalTime + animPeriod) %
                                (long)(1000 * seqDuration);

    // calculate current displayable image position
    imPosition = (int) (animTotalTime / showPeriod);

    if ((imPosition == numImages-1) && (!isRepeating)) {  //seq end
```



```
        ticksIgnored = true;   // stop at this image
        if (watcher != null)
          watcher.sequenceEnded(imName);   // call callback
      }
    }
  }
```

imPosition holds the index into the sequence of images. showPeriod is defined as:

```
showPeriod = (int) (1000 * seqDuration / numImages);
```

This means that imPosition can only be a value between 0 and numImages-1.

getCurrentImage() uses imPosition to access the relevant image in the loader:

```
public BufferedImage getCurrentImage()
{ if (numImages != 0)
    return imsLoader.getImage(imName, imPosition);
  else
    return null;
}
```

getCurrentImage()'s test of numImages is used to detect problems which may have arisen when the ImagesPlayer was created, typically that the image name (imName) is unknown to the loader.

The ticksIgnored boolean is employed to stop the progression of a sequence. In updateTick(), if ticksIgnored is true then the internal time counter, animTotalTime, is not incremented. It is controlled by the stop(), resume() and restartAt() methods. stop() is simply:

```
public void stop()
{  ticksIgnored = true;  }
```

14. ImagesTests 'o' Image Visual Effects

A quick look at Table 3 at the beginning of section 10 shows that ImagesTests utilizes a large number of visuals effects. These can be classified into two groups:

1) animations of image sequences, carried out by ImagesPlayer objects;
2) image processing operations applied to 'o' images.

We have already described the first group, which leaves a total of eleven effects. These are applied to the 'o' images inside paintComponent() of ImagesTests. The relevant code fragment is:

```
        :
// ------------------ 'o' images ---------------------
/* The programmer must manually edit the code here in order to
   draw the 'o' images with different visual effects. */

// drawImage(g2d, atomic, 10, 25); // only draw the image
```



```
rotatingImage(g2d, atomic, 10, 25);
mixedImage(g2d, balls, 110, 25);
teleImage = teleportImage(g2d, bee, teleImage, 210, 25);
flippingImage(g2d, cheese, 310, 25);
blurringImage(g2d, eyeChart, 410, 25);
reddenImage(g2d, house, 540, 25);
zapImage = zapImage(g2d, pumpkin, zapImage, 710, 25);
brighteningImage(g2d, scooter, 10, 160);
fadingImage(g2d, ufo, 110, 140);
negatingImage(g2d, owl, 450, 250);
mixedImage(g2d, basn8, 650, 250);
resizingImage(g2d, basn16, 750, 250);
        :
```

All the methods have a similar interface, requiring a reference to the graphics context, the name of the image, and the (x,y) coordinate where the modified image will be drawn.

The eleven operations can be grouped into eight categories, shown in Table 4.

1.drawImage() Based	
resizingImage()	The image grows.
flippingImage()	Keep flipping the image horizontally and vertically.
2. Alpha Compositing	
fadingImage()	The image smoothly fades away to nothing.
3. Affine Transforms	
rotatingImage()	Spin the image in a clockwise direction.
4. ConvolveOp	
blurringImage()	Make the image increasingly more blurred.
5. LookupOp	
reddenImage()	Turn the image ever more red.
6. RescaleOp	
reddenImage()	Turn the image ever more red (again).
brighteningImage()	Keep turning up the image's brightness.
negatingImage()	Keep switching between the image and its negative.
7. BandCombineOp	
mixedImage()	Keep mixing up the colours of the image.
8. Pixel Effects	
teleportImage()	Make the image fade, groups of pixels at a time.
zapImage()	Change the image to a mass of red and yellow pixels.

Table 4. Visual Effect Operations by Category.

The following subsections are organized according to the eight categories, with the operations explained in their relevant category. However, some general comments can be made about them here.

The methods in ImagesTest do *not* do image processing. Their main task is to use the current counter value, modulo some constant, to generate suitable arguments to image processing methods located in ImageSFXs.

The use of the modulo operator means that the effects will repeat as the counter progresses. For example, resizingImage() makes the image grow bigger for 6 frames, at which point the image is redrawn back at its starting size, and growth begins again.

The image processing methods in ImagesSFXs do not change the original 'o' images. Some of the methods write directly to the screen, by calling drawImage() with an image processing operator. Other methods generate a temporary BufferedImage object which is subsequently drawn to the screen. The object exists only until the end of the method.

teleportImage() and zapImage() are different in that their images *are* stored globally in ImagesTests, in the variables teleImage and zapImage. This means that method processing can be *cumulative*, since earlier changes will be stored/remembered in the global variables. Note that even these operations do not modify the original 'o' images, only teleImage and zapImage.

The main reason for not changing the original images is to allow them to be reused as the effects cycles repeat. Another reason is that any changes to the images will cause the JVM to drop them from VRAM. This would make their future rendering slower, at least for a short time.

Where possible, image operations should be applied through drawImage() directly to the screen, as this will make hardware acceleration more likely to occur.

If a temporary variable is absolutely necessary, then it might be a good idea to apply the image operation to a copy of the graphic in a VolatileImage object, forcing processing to be carried out in VRAM. There is a chance that this will allow the operation to be accelerated, but it may also slow things down!

On MS Windows, the ddforcevram flag appears to force the creation of managed images for temporary BufferedImage variables, and so the VolatileImage approach is unnecessary.

The main drawback with image processing operations is their potentially adverse effect on speed. On Windows, none of the operations, except perhaps for those using drawImage() resizing and flipping, will be hardware accelerated. The situation should be considerably better on Solaris/Linux, although I have not tested it.

In general, visual effects based around image processing operations should be used sparingly, due to their poor performance. In many cases, alternatives using image sequences can be employed; rotation is an example. The 's' 'cars' images display a series of rotated car images, which may all be in VRAM since the images are never modified internally. By comparison, the rotatingImage() method applied to the 'atomic' 'o' image also makes it rotate, but this is achieved by generating new images at run time using affine transformations. On Windows, none of these images would be hardware accelerated.

One way of viewing this suggestion is that graphical effects should be pre-calculated outside of the application, and stored as ready-to-use images. The cost/complexity of image processing is therefore separated from the executing game.

14.1. drawImage() Based Processing

There are several variants of drawImage(), useful for visual effects such as scaling and flipping, and much faster than the BufferedImageOp operations.

The version of drawImage() relevant for resizing is:

```
boolean drawImage(Image im, int x, int y,
                  int width, int height, ImageObserver imOb)
```

The width and height arguments scale the image so it has the required dimensions. By default, scaling uses a nearest neighbour algorithm: the colour of a pixel on-screen is based on the scaled image pixel that is nearest to the on-screen one. This tends to make an image blocky-looking, if it is enlarged excessively. A smoother appearance, although slower to calculate, can be achieved with *bilinear interpolation*. The colour of an on-screen pixel is derived from a combination of all the scaled image pixels that overlap the on-screen one. Bilinear interpolation can be requested at the start of paintComponent():

```
g2d.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
                     RenderingHints.VALUE_INTERPOLATION_BILINEAR);
```

resizingImage() in ImagesTests:

```
private void resizingImage(Graphics2D g2d, BufferedImage im,
                                                int x, int y)
{ double sizeChange = (counter%6)/2.0 + 0.5;    // gives 0.5 -- 3
  imageSfx.drawResizedImage(g2d, im, x, y,sizeChange,sizeChange);
}
```

The sizeChange value is calculated from the counter value in such a way that it increases from 0.5 to 3.0 in steps of 0.5, then restarts. This causes the image ('basn6a16') to start at half size and gradually grow to 3 times its actual dimensions.

The two sizeChange values become widthChange and heightChange in drawResizedImage() in ImageSFXs, and after some error-checking, its resizing code is:

```
int destWidth = (int) (im.getWidth() * widthChange);
int destHeight = (int) (im.getHeight() * heightChange);

// adjust top-left (x,y) coord of resized image so remains centered
int destX = x + im.getWidth()/2 - destWidth/2;
int destY = y + im.getHeight()/2 - destHeight/2;

g2d.drawImage(im, destX, destY, destWidth, destHeight, null);
```

The drawing coordinate (destX,destY) is adjusted so that the image's center point does not move on-screen when the image is resized.

The version of drawImage() suitable for image flipping is:

```
boolean drawImage(Image im, int dx1, int dy1, int dx2, int dy2,
                            int sx1, int sy1, int sx2, int sy2,
                          ImageObserver imOb)
```

The eight integers represent 4 coordinates: (sx1,sy1) and (sx2,sy2) are the top-left and bottom-right corners of the image, and (dx1,dy1) and (dx2,dy2) are the top-left and bottom-right corners of a rectangle somewhere on-screen where those points will be drawn. The idea is shown in Figure 14.

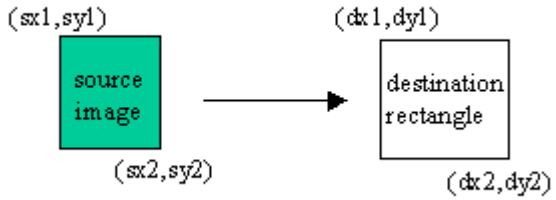

Figure 14. Drawing an Image into an On-Screen.

Usually, the image coordinates are (0,0) and (width, height) so the entire image is drawn. The versatility comes in the range of possibilities for the on-screen rectangle: it can be used to scale, stretch, and flip (as here).

flippingImage() in ImagesTests calls getFlippedImage() in ImageSFXs with an ImageSFXs flipping constant.

```
  private void flippingImage(Graphics2D g2d, BufferedImage im,
                                              int x, int y)
  { BufferedImage flipIm = null;
    if (counter%4 == 0)
      flipIm = im;     // no flipping
    else if (counter%4 == 1)
      flipIm =imageSfx.getFlippedImage(im,ImageSFXs.HORIZONTAL_FLIP);
    else if (counter%4 == 2)
      flipIm = imageSfx.getFlippedImage(im, ImageSFXs.VERTICAL_FLIP);
    else
      flipIm = imageSfx.getFlippedImage(im, ImageSFXs.DOUBLE_FLIP);

    drawImage(g2d, flipIm, x, y);
  }
```

The counter value is manipulated so that the image ('cheese') will be repeatedly flipped horizontally, vertically, both ways, and not at all.

The image returned from getFlippedImage() is drawn by drawImage(). This code does not make further use of flipIm, but it might be useful to store flipped copies of images for use later.

getFlippedImage() creates an empty copy of the source BufferedImage, and then writes a flipped version of the image into it by calling renderFlip().

```
  public BufferedImage getFlippedImage(BufferedImage im,int flipKind)
```



```
{
  if (im == null) {
    System.out.println("getFlippedImage: input image is null");
    return null;
  }

  int imWidth = im.getWidth();
  int imHeight = im.getHeight();
  int transparency = im.getColorModel().getTransparency();

  BufferedImage copy =
      gc.createCompatibleImage(imWidth, imHeight, transparency);
  Graphics2D g2d = copy.createGraphics();

  // draw in the flipped image
  renderFlip(g2d, im, imWidth, imHeight, flipKind);
  g2d.dispose();

  return copy;
} // end of getFlippedImage()
```

renderFlip() is a multi-way branch based on the flipping constant supplied in the top-level call.

```
private void renderFlip(Graphics2D g2d, BufferedImage im,
                        int imWidth, int imHeight, int flipKind)
{
  if (flipKind == VERTICAL_FLIP)
    g2d.drawImage(im, imWidth, 0,  0, imHeight,
                      0, 0,  imWidth, imHeight, null);
  else if (flipKind == HORIZONTAL_FLIP)
    g2d.drawImage(im, 0, imHeight,  imWidth, 0,
                      0, 0,  imWidth, imHeight, null);
  else    // assume DOUBLE_FLIP
    g2d.drawImage(im, imWidth, imHeight,  0, 0,
                      0, 0,  imWidth, imHeight, null);
}
```

To illustrate how the flipping works, consider the vertical flip shown in Figure 15 (w = imWidth, h = imHeight).

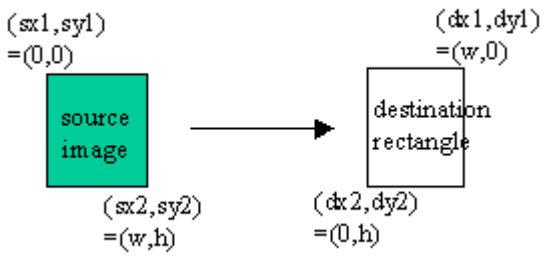

Figure 15. A Vertical Flip.

ImageSFXs also contains two flipping methods which draw directly to the screen: drawVerticalFlip() and drawHorizFlip(); they are not currently used by ImagesTests.

14.2. Alpha Compositing

Compositing is the process of combining two images together. The existing image (often the screen's drawing surface) is called the *destination*, and the image being rendered onto it is the *source*. Java 2D offers eight compositing rules which specify varies ways that the source can be combined with the destination; the most useful is probably SRC_OVER (source over destination), the others include DST_OVER (destination over source), and SRC_IN which clips the source to be visible only inside the boundaries of the destination.

Java 2D's AlphaComposite class adds another element to the compositing rules: the alpha values for the source and destination. This can be somewhat confusing, especially when both images have alpha channels. However, for the SRC_OVER case, when the destination image is opaque (e.g. the on-screen background), the alpha effectively applies to the source image only. An alpha value of 0.0f makes the source disappear, while 1.0f makes it completely opaque, and various degrees of translucency exist between.

fadingImage() in ImagesTests hacks together a alpha value based on the counter, such that as the counter increases towards 25, the alpha value goes to 0. The result is that the image ('ufo' in ImagesTests) will gradually fade away over a period of 2.5 seconds (25 frames, each of 0.1 second), then spring back into view as the process starts again.

```
private void fadingImage(Graphics2D g2d, BufferedImage im,
                                                  int x, int y)
{ float alpha = 1.0f - (((counter*4)%100)/100.0f);
  imageSfx.drawFadedImage(g2d, ufo, x, y, alpha);
}
```

drawFadedImage() in ImageSFXs does various forms of error checking, and then creates an AlphaComposite object using SRC_OVER and the alpha value:

```
Composite c = g2d.getComposite();  // backup the old composite

g2d.setComposite( AlphaComposite.getInstance(
                        AlphaComposite.SRC_OVER, alpha) );
g2d.drawImage(im, x, y, null);

g2d.setComposite(c);
 // restore old composite so it doesn't mess up future rendering
```

g2d is the screen's graphics context, and its composite is modified prior to calling drawImage(). Care must be taken to backup the existing composite, so it can be restored after the draw.

14.3. Affine Transforms

rotatingImage() in ImagesTests rotates the image ('atomic') in steps of 10 degrees in a clockwise direction, using the image's center as the center of rotation.

The ImageSFXs method getRotatedImage() utilizes an AffineTransform operation to rotate a copy of the image, which is returned to rotatingImage() and drawn.

```
private void rotatingImage(Graphics2D g2d, BufferedImage im,
                                                  int x, int y)
{ int angle = (counter * 10) % 360;
  BufferedImage rotIm = imageSfx.getRotatedImage(im, angle);
  drawImage(g2d, rotIm, x, y);
}
```

getRotatedImage() makes a new BufferedImage, called dest. An AffineTransform object is created which rotates dest's coordinate space by angle degrees *anti-clockwise* around its center. The source image is then copied in, which makes it appear to be rotated by angle degrees *clockwise* around the center of dest.

```
public BufferedImage getRotatedImage(BufferedImage src, int angle)
{
  if (src == null) {
    System.out.println("getRotatedImage: input image is null");
    return null;
  }

  int transparency = src.getColorModel().getTransparency();
  BufferedImage dest =  gc.createCompatibleImage(
                src.getWidth(), src.getHeight(), transparency );
  Graphics2D g2d = dest.createGraphics();

  AffineTransform origAT = g2d.getTransform();  // save original

  // rotate the coord. system of the dest. image around its center
  AffineTransform rot = new AffineTransform();
  rot.rotate( Math.toRadians(angle),
                src.getWidth()/2, src.getHeight()/2);
  g2d.transform(rot);

  g2d.drawImage(src, 0, 0, null);   // copy in the image

  g2d.setTransform(origAT);    // restore original transform
  g2d.dispose();

  return dest;
}
```

The single AffineTransform object (e.g. rot) can be composed from multiple transforms, such as translations, scaling, and shearing, by simply applying more operations to it. For instance, translate(), scale(), and shear() applied to rot will be cumulative in effect. Ordering is important: a translation followed by a rotation is generally not the same as a rotation followed by a translation.

The main problem with this approach is that the image is transformed within the image space of dest, which acts as a clipping rectangle. Thus, if the image is

translated/rotated/sheared outside dest's boundaries, for example beyond the bottom-right corner, then the image will be clipped or perhaps 'disappear' completely

This problem can occur even with rotations around dest's center; a look at the rotating 'atomic' image highlights the problem.

The simplest solution is a careful design of the graphic, to ensure that its opaque areas all fall within a 'rotation' circle placed at the center of the image file, with a radius constrained by the file's dimensions. For example, the image in Figure 16a is safe to rotate (around the file's center point), the image in Figure 16b is not.

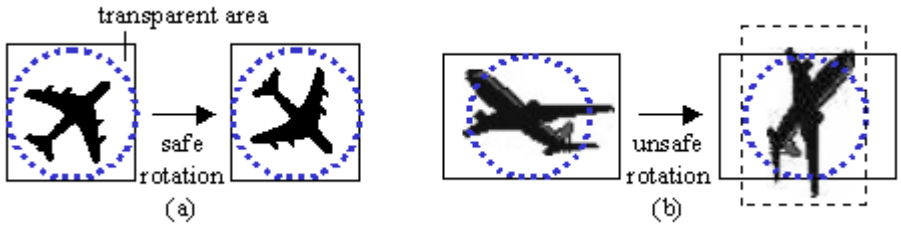

Figure 16. Safe and Unsafe Rotations.

The rotation circle can also be employed as an image boundary when carrying out collision detection between the image and other sprites. Two images have 'collided' if their rotation circles intersect.

When an image is rotated, there will often be areas in the destination image which do not correspond to pixels in the source. For instance, in Figure 16b, there are strips on the left and right of the rotated image which do not correspond to pixels in the original. Fortunately, these are drawn transparently if the original image has an alpha channel. However, if the original image is opaque (e.g. a JPEG), then the pixels are coloured black.

14.4. ConvolveOp Processing

A convolution operator calculates the colour of each pixel in a destination image in terms of a combination of the colours of the corresponding pixel in the source image, *and* its neighbours. A matrix (called a *kernel*) specifies the neighbours and gives weights for how their colours should be combined with the source pixel to give the destination pixel value. The kernel must have an odd number of rows and columns (e.g. 3 by 3) so that the central cell can represent the source pixel (e.g. cell (1,1)), and the surrounding cells its neighbours.

Convolution is carried out by applying the kernel to every pixel in the source, generating destination pixels as it traverses the image. The example in Figure 17 is using a 3 by 3 kernel.

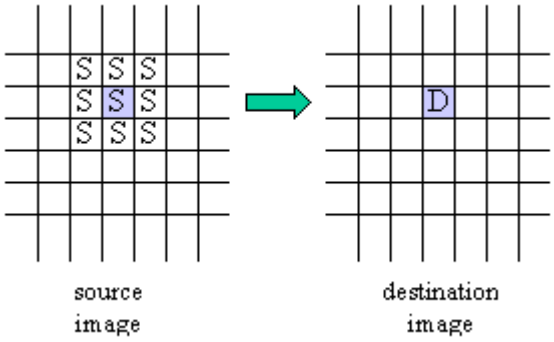

Figure 17. Convolution from Source to Destination.

A typical 3 by 3 kernel:

$$\begin{pmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{pmatrix}$$

The 1/9 values are the weights. This kernel combines the source pixel and its eight neighbours using equal weights, which causes the destination pixel to be a combination of all those pixel's colours, resulting in an overall blurry image.

The weights should add up to 1 in order to maintain the brightness of the destination image. A total weight of more than 1 will make the image brighter, less than 1 will darken it. Also, the resulting pixel colour values are constrained to be between 0 and 255 (since the sRGB format assigns 8 bits to each colour).

One tricky aspect is what to do at the edges of the image, for example, for the source pixel at (0,0) which has no left and top neighbours? In most image processing packages, the solution is to treat the graphic as a wrap-around so that the pixels at the bottom of the image are used as the top neighbours, and the pixels at the right edge as left neighbours. Unfortunately, Java 2D is a little lacking in this area, since its edge behaviours are quite simple. Either the destination pixel (e.g. (0,0)) is automatically

filled with black, or set to contain the source pixel value unchanged. These possibilities are denoted by the ConvolveOp constants EDGE_ZERO_FILL and EDGE_NO_OP.

Kernels for edge detection and sharpening are given in Figure 18.

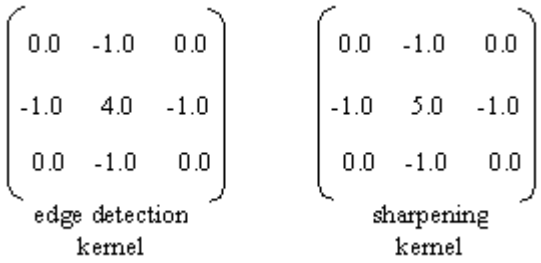

Figure 18. Edge Detection and Sharpening Kernels.

The edge detection kernel highlights the places where the colours in the image change sharply (usually at the boundaries between parts of the images), drawing them in white or gray. Meanwhile, large blocks of similar colour will be cast into gloom. The result is a destination image showing only the edges between areas in the original picture.

The sharpening kernel is actually a variant of the edge detection matrix, with more weight applied to the source pixel, making the overall weight 1.0 so that the destination image's brightness is maintained. The result is that the original image will still be visible but edges will be thicker and brighter.

ImageSFXs contains a drawBluredImage() method which applies a precalculated blurring kernel:

```
private ConvolveOp blurOp;    // global for image blurring
  :

private void initEffects()
// Create pre-defined ops for image negation and blurring.
{ // image negative, explained later...

  // blur by convolving the image with a matrix
  float ninth = 1.0f / 9.0f;

  float[] blurKernel = {      // the 'hello world' of Image Ops :)
      ninth, ninth, ninth,
      ninth, ninth, ninth,
      ninth, ninth, ninth
  };
  blurOp = new ConvolveOp(
     new Kernel(3, 3, blurKernel), ConvolveOp.EDGE_NO_OP, null);
}


public void drawBlurredImage(Graphics2D g2d,
                        BufferedImage im, int x, int y)
// blurring with a fixed convolution kernel
{ if (im == null) {
```



```
      System.out.println("getBlurredImage: input image is null");
      return;
    }
    g2d.drawImage(im, blurOp, x, y);   // use predefined ConvolveOp
  }  // end of drawBlurredImage()
```

When the ImageSFXs object is created, initEffects() is called to initialise the blurOp ConvolveOp object. A 3 by 3 array of floats is used to create the kernel. The EDGE_NO_OP argument states that pixels at the edges of the image will be unaffected by the convolution process.

Note that drawBlurredImage() uses the version of drawImage() which takes a BufferedImageOp argument, so the modified image is written directly to the screen.

This coding is quite adequate, but we require an image to become *increasingly* blurry over a period of several frames (see 'eyeChart' in ImagesTests).

One solution would be to store the destination image at the end of the convolution, and apply blurring to it again during the next frame. Unfortunately, ConvolveOps cannot be applied in place, and so a new destination image must be created each time.

Instead, our approach is to generate increasingly blurry ConvolveOps in each frame, and apply them to the original image via drawImage().

Increasingly blurry kernels are simply larger matrices which generate a destination pixel based on more neigbours. We begin with a 3 by 3 matrix, then a 5 by 5, and so on, increasing to 15 by 15. The matrices must have odd length dimensions so there is a center point. Also, the weights in the matrix must add up to 1 so, for instance, the 5 by 5 matrix will be filled with 1/25 in every cell.

The top-level method in ImagesTests is blurringImage():

```
private void blurringImage(Graphics2D g2d, BufferedImage im,
                                                int x, int y)
{
  int fadeSize = (counter%8)*2 + 1;    // gives 1,3,5,7,9,11,13,15
  if (fadeSize == 1)
    drawImage(g2d, im, x, y);    // start again with original image
  else
    imageSfx.drawBlurredImage(g2d, im, x, y, fadeSize);
}
```

drawBlurredImage() in ImageSFXs takes a fadeSize argument which becomes the row and column lengths of the kernel. The method is complicated by making sure that the kernel dimensions are odd, not too small, and not bigger than the image.

```
public void drawBlurredImage(Graphics2D g2d,
                        BufferedImage im, int x, int y, int size)
/* The size argument is used to specify a size*size blur kernel,
   filled with 1/(size*size) values. */
{
  if (im == null) {
    System.out.println("getBlurredImage: input image is null");
    return;
  }
```



```
    int imWidth = im.getWidth();
    int imHeight = im.getHeight();
    int maxSize = (imWidth > imHeight) ? imWidth : imHeight;

    if ((maxSize%2) == 0)  // if even
      maxSize--;  // make it odd

    if ((size%2) == 0) {  // if even
      size++;   // make it odd
      System.out.println(
        "Blur size must be odd; adding 1 to make size = " + size);
    }

    if (size < 3) {
      System.out.println("Minimum blur size is 3");
      size = 3;
    }
    else if (size > maxSize) {
      System.out.println("Maximum blur size is " + maxSize);
      size = maxSize;
    }

    // create the blur kernel
    int numCoords = size * size;
    float blurFactor = 1.0f / (float) numCoords;

    float[] blurKernel = new float[numCoords];
    for (int i=0; i < numCoords; i++)
      blurKernel[i] = blurFactor;

    ConvolveOp blurringOp = new ConvolveOp(
         new Kernel(size, size, blurKernel),
         ConvolveOp.EDGE_NO_OP, null);   // leaves edges unaffected
      // ConvolveOp.EDGE_ZERO_FILL, null); //edges filled with black

    g2d.drawImage(im, blurringOp, x, y);
  }  // end of drawBlurredImage() with size argument
```

A drawback with larger kernels is that more of the pixels at the edges of the source image will be affected by the edge behaviour constants. With EDGE_NO_OP, a increasingly thick band of pixels around the edges will be unaffected. With EDGE_ZERO_FILL, the band will be pitch black. There is a need for more edge behaviour options in future versions of the ConvolveOp class.

14.5. LookupOp Processing

At the heart of LookupOp is the representation of a pixel using the sRGB colour space, which stores the red, green, blue (and alpha) channels in 8 bits (1 byte) each, snugly fitting them all into a single 32-bit integer. This is shown in Figure 19.

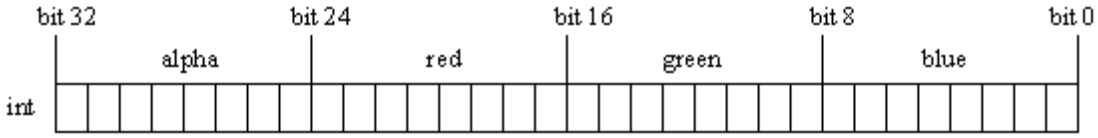

Figure 19. The sRGB Colour Space Format.

The red, green, blue and alpha components can each have 256 different values (2^8), with 255 being full on. For the alpha part, 0 means fully transparent, 255 fully opaque.

A LookupOp operation utilizes a lookup table (perhaps several lookup tables) with 256 entries. Each entry contains a colour value (i.e. an integer between 0 and 255), so that the table defines a mapping from the image's existing colour values to new values.

The simplest form of LookupOp is one that uses a single lookup table. The example below converts a colour component value i to (255-i), and is applied to all the channels in the image. For example, a red colour component of 0 (no red) is mapped to 255 (full on). Thus, the table will invert the colour scheme.

```
short[] invert = new short[256];
for (int i = 0; i < 256; i++)
  invert[i] = (short)(255 - i);

LookupTable table = new ShortLookupTable(0, invert);
LookupOp invertOp = new LookupOp(table, null);

g2d.drawImage(im, invertOp, x, y);   // draw the image
```

The ShortLookupTable class is mostly just an array, used to initialize the operation. There is also a ByteLookupTable that is built with an array of bytes.

A visual way of understanding the mapping defined by invert[] is shown in Figure 20.

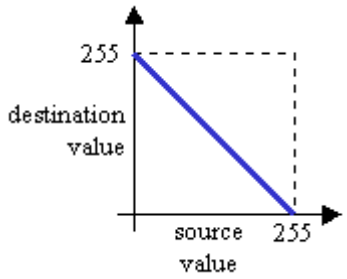

Figure 20. The invert[] Lookup Table.

The table defines a straight line in this case, but a table can hold any mapping from source colour component values to destination values.

In general, it is more common to utilize several lookup tables, different ones for each channel. Also, no mapping is usually applied to an alpha channel of a transparent or translucent image.

reddenImage() in ImagesTests draws its source image with increasing amounts of red over a period of 20 frames, and then starts again (e.g. see the 'house' image). The original image is unaffected, since the LookupOp writes directly to the screen via drawImage(). To increase the effect, as the redness increases, the amount of green and blue decreases, so necessitating different lookup tables for red and green/blue. Also, any alpha component in the image is left unaffected.

```
private void reddenImage(Graphics2D g2d, BufferedImage im,
                                                  int x, int y)
{
  float brightness = 1.0f + (((float) counter%21)/10.0f);
           // gives values in the range 1.0-3.0, in steps of 0.1
  if (brightness == 1.0f)
    drawImage(g2d, im, x, y);   // start again with original image
  else
    imageSfx.drawRedderImage(g2d, im, x, y, (float) brightness);
}
```

drawRedderImage() in ImageSFXs does the lookup-based colour changes, based on a brightness value that ranges from 1.0 to 3.0.

A minor hassle, illustrated by drawRedderImage(), is dealing with opaque versus non-opaque images. An opaque image requires two unique lookup tables (one for red, one used for green and blue), while a non-opaque image requires a third lookup table for the alpha channel. This separation occurs in all LookupOp (and RescaleOp and BandCombineOp) methods which may be passed both types of image.

```
public void drawRedderImage(Graphics2D g2d, BufferedImage im,
                                  int x, int y, float brightness)
/* Draw the image with its redness is increased, and its greenness
   and blueness decreased. Any alpha channel is left unchanged.
*/
{ if (im == null) {
```



```
      System.out.println("drawRedderImage: input image is null");
      return;
    }

    if (brightness < 0.0f) {
      System.out.println("Brightness must be >= 0.0f;set to 0.0f");
      brightness = 0.0f;
    }
    // brightness may be less than 1.0 to make the image less red

    short[] brighten = new short[256];    // for red channel
    short[] lessen = new short[256];   // for green and blue channels
    short[] noChange = new short[256];    // for the alpha channel

    for(int i=0; i < 256; i++) {
      float brightVal = 64.0f + (brightness * i);
      if (brightVal > 255.0f)
        brightVal = 255.0f;
      brighten[i] = (short) brightVal;
      lessen[i] = (short) ((float)i / brightness);
      noChange[i] = (short) i;
   }

    short[][] brightenRed;
    if (hasAlpha(im)) {
      brightenRed = new short[4][];
      brightenRed[0] = brighten;  // for the red channel
      brightenRed[1] = lessen;    // for the green channel
      brightenRed[2] = lessen;    // for the blue channel
      brightenRed[3] = noChange;  // for the alpha channel
        // without this the LookupOp fails; a bug (?)
    }
    else {  // not transparent
      brightenRed = new short[3][];
      brightenRed[0] = brighten;   // red
      brightenRed[1] = lessen;     // green
      brightenRed[2] = lessen;     // blue
    }
    LookupTable table = new ShortLookupTable(0, brightenRed);
    LookupOp brightenRedOp = new LookupOp(table, null);

    g2d.drawImage(im, brightenRedOp, x, y);
  }  // end of drawRedderImage()
```


The three lookup tables, brighten[], lessen[], and noChange[] are shown in Figure 21 when brightness has the value 2.0. As the value increases, more of the red colour components will be mapped to full on and the blue and green colour values will be lowered further.

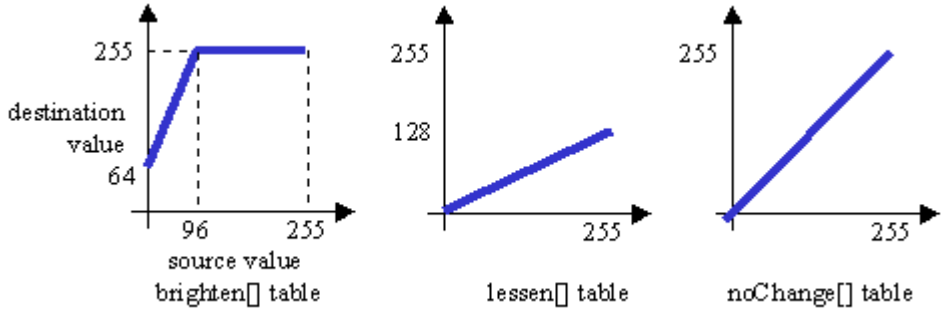

Figure 21. Lookup Tables Used in drawReddenImage().

A two-dimensional array, brightenRed[][] is declared and filled with 3 or 4 tables depending on if the image is opaque (i.e. only has RGB components) or also has an alpha channel. This array is used to create a LookupOp table called table, and then the operation.

A LookupOp operation will raise an exception if the source image has an alpha channel and the operation only contains three tables. Therefore it is essential to check for the presence of an alpha band in the image, which is achieved with hasAlpha()

```
public boolean hasAlpha(BufferedImage im)
// does im have an alpha channel?
{
  if (im == null)
    return false;

  int transparency = im.getColorModel().getTransparency();

  if ((transparency == Transparency.BITMASK) ||
      (transparency == Transparency.TRANSLUCENT))
    return true;
  else
    return false;
}
```

A colour model may use BITMASK transparency (as in GIFs), TRANSULENT (as in translucent PNGs), or OPAQUE (as in JPEGs).

14.6. RescaleOp Processing

The rescaling operation is a specialized form of LookupOp. As with a lookup, a pixel is considered to be in sRGB form: the red, green, blue (and alpha) channels are each stored in 8 bits (1 byte), allowing the color components to range between 0 and 255.

Instead of specifying a table mapping, the new colour component is defined as a linear equation involving a scale factor applied to the existing colour value, plus an optional offset:

$$colour_{dest} = scaleFactor * colour_{source} + offset$$

The destination colour is bounded to be between 0 and 255.

This means that any Lookup table that can be defined by a straight line can be rephrased as a RescaleOp operation. Conversely, any RescaleOp can be written as a LookupOp. LookupOp is more general since the table mapping permits non-linear relationships to be defined between the source and destination colour components.

Since LookupOp is functionally a superset of RescaleOp, and probably more efficient to execute, it is somewhat unclear why Java 2D offers RescaleOp at all.

drawReddenImage(), which was defined as a LookupOp using three (or four) tables, can be rephrased as a RescaleOp consisting of three (or four) rescaling equations. Each equation has two parts: a scale factor and an offset.

```
RescaleOp brigherOp;
if (hasAlpha(im)) {
  float[] scaleFactors =
     {brightness, 1.0f/brightness, 1.0f/brightness, 1.0f};
      // don't change alpha
      // without the 1.0f the RescaleOp fails; a bug (?)
  float[] offsets = {64.0f, 0.0f, 0.0f, 0.0f};
  brigherOp = new RescaleOp(scaleFactors, offsets, null);
}
else {  // not transparent
  float[] scaleFactors =
      {brightness, 1.0f/brightness, 1.0f/brightness};
  float[] offsets = {64.0f, 0.0f, 0.0f};
  brigherOp = new RescaleOp(scaleFactors, offsets, null);
}
g2d.drawImage(im, brigherOp, x, y);
```

The RescaleOp constructor takes an array of scale factors, an array of offsets, and optional rendering hints as its arguments.

The three equations employed in this code fragment are:

$$red\_colour_{dest} = brightness * red\_colour_{source} + 64$$

$$green/blue\_colour_{dest} = (1/brightness) * green/blue\_colour_{source} + 0$$

$$alpha\_colour_{dest} = 1 * alpha\_colour_{source} + 0$$

Note that the new red colour component is clamped to 255, even if the equation returns a larger value. The green/blue_colour equation is used for both the green and blue channels.

These equations are the same as the LookupOp tables used in the first version of drawReddenImage().

As with LookupOp, it is essential that the right number of scale factors and offsets are supplied according to the number of channels in the image. For instance, if only three equations are defined for an image with an alpha channel, then an exception will be raised at run time when the operation is applied.

14.6.1. Brightening the Image

ImagesTests' brigheningImage() increases the brightness of its image over a period of 9 frames, then start again with the original colours (see 'scooter' for an example). The original image is unaffected, since the operation writes to the screen. The brightness only affects the RGB channels, the alpha component is unchanged.

```
private void brighteningImage(Graphics2D g2d, BufferedImage im,
                                                int x, int y)
{ int brightness = counter%9;    // gives 0-8
  if (brightness == 0)
    drawImage(g2d, im, x, y);   // start again with original image
  else
    imageSfx.drawBrighterImage(g2d, im, x, y, (float) brightness);
}
```

The ImageSFXs method, drawBrighterImage(), uses a RescaleOp based around the equations:

$$RGB\_colour_{dest} = brightness * RGB\_colour_{source} + 0$$

$$alpha\_colour_{dest} = 1 * alpha\_colour_{source} + 0$$

The RBG_colour equation is used for the red, green, and blue channels. When the source image has no alpha, we can utilize a RescaleOp constructor that takes a single scale factor and offset. It will automatically apply the equation to all the RGB channels:

```
public void drawBrighterImage(Graphics2D g2d, BufferedImage im,
                                    int x, int y, float brightness)
{ if (im == null) {
    System.out.println("drawBrighterImage: input image is null");
    return;
  }
  if (brightness < 0.0f) {
    System.out.println("Brightness must be >= 0.0f; set to 0.5f");
    brightness = 0.5f;
  }
  RescaleOp brigherOp;
  if (hasAlpha(im)) {
     float[] scaleFactors =
       {brightness, brightness, brightness, 1.0f};
     float[] offsets = {0.0f, 0.0f, 0.0f, 0.0f};
     brigherOp = new RescaleOp(scaleFactors, offsets, null);
  }
  else   // not transparent
    brigherOp = new RescaleOp(brightness, 0, null);
```



```
    g2d.drawImage(im, brigherOp, x, y);
  }  // end of drawBrighterImage()
```

14.6.2. Negating the Image

ImagesTests' negatingImage() keeps switching between the original image and its negative depending on the counter value (see the 'owl' image for an example). A colour component value i is converted to 255-i in the RGB channels, but the alpha is untouched.

```
private void negatingImage(Graphics2D g2d, BufferedImage im,
                                                int x, int y)
{ if (counter%10 < 5)   // show the negative
    imageSfx.drawNegatedImage(g2d, im, x, y);
  else  // show the original
    drawImage(g2d, im, x, y);
}
```

When the ImageSFXs object is first created, the negative rescaling operations, negOp and negOpTrans, are predefined. negOpTrans is used when the image has an alpha channel, and contains the equations:

$$RGB\_colour_{dest} = -1 * RGB\_colour_{source} + 255$$

$$alpha\_colour_{dest} = 1 * alpha\_colour_{source} + 0$$

The RGB_colour equation is applied to the red, green, and blue channels.

negOp is for opaque images, so only requires the RGB equation.

```
// global rescaling ops for image negation
private RescaleOp negOp, negOpTrans;
  :

private void initEffects()
{
  // image negative.
  // Multiply each colour value by -1.0 and add 255
  negOp = new RescaleOp(-1.0f, 255f, null);

  // image negative for images with transparency
  float[] negFactors = {-1.0f, -1.0f, -1.0f, 1.0f};
                          // don't change the alpha
  float[] offsets = {255f, 255f, 255f, 0.0f};
  negOpTrans = new RescaleOp(negFactors, offsets, null);

  ...
}


public void drawNegatedImage(Graphics2D g2d, BufferedImage im,
                                                  int x, int y)
{ if (im == null) {
    System.out.println("drawNegatedImage: input image is null");
    return;
```



```
    }
    if (hasAlpha(im))
      g2d.drawImage(im, negOpTrans, x, y);  // predefined RescaleOp
    else
      g2d.drawImage(im, negOp, x, y);
  }  // end of drawNegatedImage()
```

14.7. BandCombineOp Processing

Both LookupOp and RescaleOp specify transformations that take a single colour component in a pixel (e.g. the red colour) and map it to a new value. A BandCombineOp generalizes this idea to allow a new colour component to be potentially defined in terms of a combination of *all* the colour components in the source pixel.

To be precise, a BandCombineOp does not manipulate the colour components directly, but rather the *samples* in a pixel. However, the intention is to cause the colours in the destination image to change.

The destination pixel {redN, greenN, blueN, alphaN} is created from some combination of the source pixel {red, green, blue, alpha}, where the combination is defined using matrix multiplication as in Figure 22.

$$\begin{pmatrix} m11 & m12 & m13 & m14 \\ m21 & m22 & m23 & m24 \\ m31 & m32 & m33 & m34 \\ m41 & m42 & m43 & m44 \end{pmatrix} * \begin{pmatrix} red_{sample} \\ green_{sample} \\ blue_{sample} \\ alpha_{sample} \end{pmatrix} = \begin{pmatrix} redN_{sample} \\ greenN_{sample} \\ blueN_{sample} \\ alphaN_{sample} \end{pmatrix}$$

source pixel — destination pixel

Figure 22. BandCombineOp as a Matrix Operation.

For example,

$$redN_{sample} = m11 * red_{sample} + m12 * green_{sample} + m13 * blue_{sample} + m14 * alpha_{sample}$$

If the source image has no alpha channel, then a 3 by 3 matrix is used.

BandCombineOp is different from the other operations we have discussed since it implements the RasterOp interface, not BufferedImageOp. This means that a little extra work is required to access the Raster object inside the source BufferedImage, and that the resulting changed Raster must be built up into a destination BufferedImage.

ImagesTests' mixedImage() draws an image with its green and blue bands modified in random ways, while keeping the red band and any alpha band unchanged. See the 'balls' and 'basn6a08' images for examples.

```
  private void mixedImage(Graphics2D g2d, BufferedImage im,
```



```
                                                    int x, int y)
{ if (counter%10 < 5)   // mix it up
    imageSfx.drawMixedColouredImage(g2d, im, x, y);
  else  // show the original
    drawImage(g2d, im, x, y);
}
```

drawMixedColouredImage() distinguishes whether the source has an alpha channel, and creates a 4 by 4 or 3 by 3 matrix accordingly. The source Raster is accessed, the operation applied using filter(), and the result packaged up as a new BufferedImage which is drawn.

```
public void drawMixedColouredImage(Graphics2D g2d, B
                              ufferedImage im, int x, int y)
// Mix up the colours in the green and blue bands
{ if (im == null) {
    System.out.println("drawMixedColouredImage: input is null");
    return;
  }
  BandCombineOp changeColoursOp;
  Random r = new Random();
  if (hasAlpha(im)) {
    float[][] colourMatrix = {   // 4 by 4
       { 1.0f, 0.0f, 0.0f, 0.0f },    // new red band, unchanged
       { r.nextFloat(), r.nextFloat(), r.nextFloat(), 0.0f },
                                      // new green band
       { r.nextFloat(), r.nextFloat(), r.nextFloat(), 0.0f },
                                      // new blue band
       { 0.0f, 0.0f, 0.0f, 1.0f} };   // unchanged alpha

    changeColoursOp = new BandCombineOp(colourMatrix, null);
  }
  else {    // not transparent
    float[][] colourMatrix = {  // 3 by 3
       { 1.0f, 0.0f, 0.0f },           // new red band, unchanged
       { r.nextFloat(), r.nextFloat(), r.nextFloat() },
                                       // new green band
       { r.nextFloat(), r.nextFloat(), r.nextFloat() }};
                                       // new blue band

    changeColoursOp = new BandCombineOp(colourMatrix, null);
  }

  Raster sourceRaster = im.getRaster();  // access source Raster
  WritableRaster destRaster =
              changeColoursOp.filter(sourceRaster, null);

  // make the destination Raster into a BufferedImage
  BufferedImage newIm = new BufferedImage(im.getColorModel(),
                                      destRaster, false, null);

  g2d.drawImage(newIm, x, y, null);   // draw it
}  // end of drawMixedColouredImage()
```

The matrices are filled with random numbers in the rows applied to the green and blue components of the source pixel.

The matrix row for the red component is {1, 0, 0, 0}, which will send the red source unchanged into the destination pixel. Similarly, the alpha component is {0, 0, 0, 1} which leaves the alpha part unchanged..

It is possible to treat a pixel as containing an additional unit element, which allows the BandCombineOp matrix to contain an extra column. This permits a wider range of equations to be defined. Figure 23 shows the resulting multiplication using a 4 by **5** matrix.

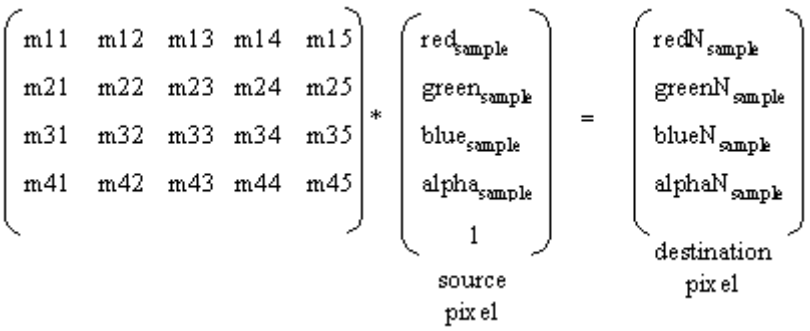

Figure 23. BandCombineOp with an Additional Pixel Element.

For example,

$$redN_{sample} = m11 * red_{sample} + m12 * green_{sample} + m13 * blue_{sample} + m14 * alpha_{sample} + \mathbf{m15}$$

The change is the additional m15 element, which can be used to define equations with offsets.

If the source image has no alpha channel, then a 3 by **4** matrix is used.

14.8. Pixel Effects

The great advantage of BufferedImage is the ease with which its elements can be accessed: pixel data, sample model, colour space, etc. However, a lot can be done using only the BufferedImage methods getRGB() and setRGB() to manipulate a given pixel (or array of pixels).

The single pixel versions are:

```
int getRGB(int x, int y);
void setRGB(int x, int y, int newValue);
```

The getRGB() method returns an integer representing the pixel at location (x,y), formatted using sRGB. The red, green, blue (and alpha) channels use 8 bits (1 byte) each, so they can fit into the 32-bit integer result. The sRGB format is shown in Figure 19.

The colour components can be extracted from the integer using bit manipulation.

```
BufferedImage im = ...;
int pixel = im.getRGB(x,y);

int alphaVal = (pixel >> 24) & 255;
int redVal = (pixel >> 16) & 255;
int greenVal = (pixel >> 8) & 255;
int blueVal = pixel & 255;
```

alphaVal, redVal, greenVal, and blueVal will have values between 0 and 255.

The setRGB() method takes an integer argument, newValue, constructed using similar bit manipulation in reverse.

```
int newValue = blueVal | (greenVal << 8) |
               (redVal << 16) | (alphaVal << 24);
im.setRGB(x, y, newVal);
```

Care should be taken that alphaVal, redVal, greenVal, and blueVal have values between 0 and 255 or the resulting integer will be incorrect.

Of more use are the versions of getRGB() and setRGB() that work with an array of pixels. getRGB() is general enough to extract an arbitrary rectangle of data from the image, returning it as a one-dimensional array. However, its most common use is to extract all the pixel data. Then a loop can be employed to traverse over the data, as below:

```
int imWidth = im.getWidth();
int imHeight = im.getHeight();

// make an array to hold the data
int[] pixels = new int[imWidth * imHeight];

// extract the data from the image into pixels[]
im.getRGB(0, 0, imWidth, imHeight, pixels, 0, imWidth);
```



```
  for(int i=0; i < pixels.length; i++) {
    // do something to pixels[i]
  }
  // update the image with pixels[]
  im.setRGB(0, 0, imWidth, imHeight, pixels, 0, imWidth);
```

At the end of the loop, the updated pixels[] array can be placed back inside the BufferedImage via a call to setRGB().

The prototypes for the array versions of getRGB() and setRGB() are:

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

void setRGB(int startX, int startY, int w, int h,
                    int[] RGBArray, int offset, int scansize);
```

The extraction rectangle is defined by startX, startY, w, and h. offset states where in the pixels array the extracted data should start being written. scansize specifies the number of elements in a row of the returned data; normally it is the width of the image.

14.8.1. Teleporting an Image

The 'teleport' effect causes the image to disappear, multiple pixels at a time, spread over the course of 7 frames (after which the effect repeats). Individual pixels are assigned the value 0, which results in their becoming transparency. The 'bee' image has this effect applied to it.

This pixilated visual should be compared with the smoother fading offered by fadingImage(), described in section 14.2 on alpha compositing.

The changes are applied to a copy of the image (stored in the global teleImage). The copy is assigned an alpha channel if the original does not have one, to ensure that the image becomes transparent (rather than black).

A global is used so that pixel erasing can be repeatedly applied to the same image, and so be cumulative.

The ImageSFXs method used is eraseImageParts(). Its second argument specifies that the affected pixels are located in the image's pixel array at positions which are multiple of the supplied number.

```
  private BufferedImage teleportImage(Graphics2D g2d,
              BufferedImage im, BufferedImage teleIm, int x, int y)
  {
    if (teleIm == null) {    // start the effect
      if (imageSfx.hasAlpha(im))
        teleIm = imageSfx.copyImage(im);
      else   // no alpha channel
        teleIm = imageSfx.makeTransImage(im);
             // give the copy an alpha channel
    }

    int eraseSteps = counter%7;    // range is 0 to 6
    switch(eraseSteps) {
```



```
      case 0:      // restart the effect
        if (imageSfx.hasAlpha(im))
          teleIm = imageSfx.copyImage(im);
        else  // not transparent
          teleIm = imageSfx.makeTransImage(im);
        break;
      case 1:
        imageSfx.eraseImageParts(teleIm, 11); //every 11th pixel goes
        break;
      case 2:
        imageSfx.eraseImageParts(teleIm, 7);  // every 7th pixel
        break;
      case 3:
        imageSfx.eraseImageParts(teleIm, 5);  // 5th
        break;
      case 4:
        imageSfx.eraseImageParts(teleIm, 3);  // 3rd
        break;
      case 5:
        imageSfx.eraseImageParts(teleIm, 2);  // every 2nd pixel
        break;
      case 6:
        imageSfx.eraseImageParts(teleIm, 1);
        break;             // every pixel goes, i.e. fully erased
      default:
        System.out.println("Unknown count for teleport");
        break;
    } // end switch

    drawImage(g2d, teleIm, x, y);
    return teleIm;
  }  // end of teleportImage()
```

The ImageSFXs support methods, copyImage() and makeTransImage(), both make copies of a BufferedImage, and are quite similar.

copyImage() utilizes GraphicsConfiguration's createCompatibleImage() to make a BufferedImage object, and then the source image is drawn into it.

makeTransImage() creates a new BufferedImage object of type TYPE_INT_ARGB to ensure that it has an alpha channel. Then the source image is drawn into it.

```
  public BufferedImage makeTransImage(BufferedImage src)
  {
    if (src == null) {
      System.out.println("makeTransImage: input image is null");
      return null;
    }
    BufferedImage dest = new BufferedImage(
                  src.getWidth(), src.getHeight(),
                  BufferedImage.TYPE_INT_ARGB);  // alpha channel
    Graphics2D g2d = dest.createGraphics();

    // copy image
    g2d.drawImage(src, 0, 0, null);
    g2d.dispose();
    return dest;
  }
```


ImageSFXs' eraseImageParts() has the same structure as the array-based getRGB() and setRGB() code outlined above.

```
public void eraseImageParts(BufferedImage im, int spacing)
{
  if (im == null) {
    System.out.println("eraseImageParts: input image is null");
    return;
  }
  int imWidth = im.getWidth();
  int imHeight = im.getHeight();
  int [] pixels = new int[imWidth * imHeight];
  im.getRGB(0, 0, imWidth, imHeight, pixels, 0, imWidth);

  int i = 0;
  while (i < pixels.length) {
    pixels[i] = 0;    // make transparent (or black if no alpha)
    i = i + spacing;
  }
  im.setRGB(0, 0, imWidth, imHeight, pixels, 0, imWidth);
}
```

The loop jumps over the array, setting every i^{th} pixel to have the value 0. This causes the red, green, blue and alpha channels to be completely filled with 0 bits. Due to the alpha channel, this causes the pixel to become transparent. If there was no alpha, then the 0 bits signify that red, green, and blue are switched *off*, and the pixel would be drawn in black.

14.8.2. Zapping an Image

Zapping means the gradual changing of the image's visible parts to a random mix of red and yellow pixels. The number of changed pixels increases over the course of the effect (11 frames). See 'pumpkin' for an example of the effect in action.

The changes are applied to a copy of the image (stored in the global zapImage). After 11 frames, the image is restored, and the effect begins again.

As with the teleportation effect, a global is used so that the colour changes can be repeatedly applied to the same image, and so be cumulative.

The amount of 'zapping' is controlled by the likelihood value which increases from 0 to 1.

The method used in ImageSFXs is zapImageParts()

```
private BufferedImage zapImage(Graphics2D g2d, BufferedImage im,
                               BufferedImage zapIm, int x, int y)
{  if ((zapIm == null) || (counter%11 == 0))
    zapIm = imageSfx.copyImage(im);    // restart the effect
  else {
    double likelihood = (counter%11)/10.0;
                              // produces range 0 to 1
    imageSfx.zapImageParts(zapIm, likelihood);
  }
  drawImage(g2d, zapIm, x, y);
  return zapIm;
```



```
}  // end of zapImage()
```

zapImageParts() uses the same approach as previously: the pixel array is extracted, modified in a loop, then written back into the BufferedImage object.

```
public void zapImageParts(BufferedImage im, double likelihood)
{
  if (im == null) {
    System.out.println("zapImageParts: input image is null");
    return;
  }
  if ((likelihood < 0) || (likelihood > 1)) {
    System.out.println("likelihood must be in the range 0 to 1");
    likelihood = 0.5;
  }

  int redCol = 0xf90000;      // nearly full-on red
  int yellowCol = 0xf9fd00;   // a mix of red and green

  int imWidth = im.getWidth();
  int imHeight = im.getHeight();
  int [] pixels = new int[imWidth * imHeight];
  im.getRGB(0, 0, imWidth, imHeight, pixels, 0, imWidth);

  double rnd;
  for(int i=0; i < pixels.length; i++) {
    rnd = Math.random();
    if (rnd <= likelihood) {
       if (rnd <=  15*likelihood/16 )    // red more likely
         pixels[i] = pixels[i] | redCol;
       else
         pixels[i] = pixels[i] | yellowCol;
    }
  }

  im.setRGB(0, 0, imWidth, imHeight, pixels, 0, imWidth);
}  // end of eraseImageParts()
```

The random effect of changing pixels to red or yellow is achieved by the use of Math.random().

The red colour (redCol) is defined as the octal `0xf90000`, and yellow (yellowCol) as `0xf9fd00`. To understand these, it helps to remember that the sRGB format stores colour components in the order alpha, red, green, and blue, each in 8 bits. Eight bits can be represented by the octals 0x00 to 0xFF, as in Figure 24.

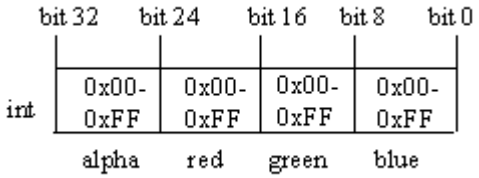

Figure 24. The sRGB Format in Octal.

Consequently, the red field in the sRGB format will be the 5th and 6th octal digits from the right, the green field will be the 3rd and 4th.

The octals are bitwise-or'ed with a pixel, which causes the relevant colour components to be overwritten. redCol overwrites the red colour component only, while yellowCol replaces the red and yellow parts, which is a more drastic change. This is balanced in the code, by having the red change done more often.

15. Packaging ImagesTests as a JAR

We're converting the ImagesTests application into a JAR so that all the resources (images in this case) are packaged with the code in a single file. This makes the application easier to transport, and we get the additional benefit of compression.

The JAR file is configured so that ImagesTests automatically starts when the user double-clicks on its icon.

We will not consider how to use applets and JAR together, or advanced topics like signing, and manipulating JARs from inside Java code. The Java tutorial (trail) on JARs should be consulted on these matters.

Before JARing begins, it is important to organize the resources in relation to the application. The ImagesTests code is located in the directory ImagesTests/ (see Figure 25), which acts as the top-level directory for the JAR. The images are placed in an Images/ subdirectory within ImagesTests/. This makes their inclusion into the JAR particularly easy.

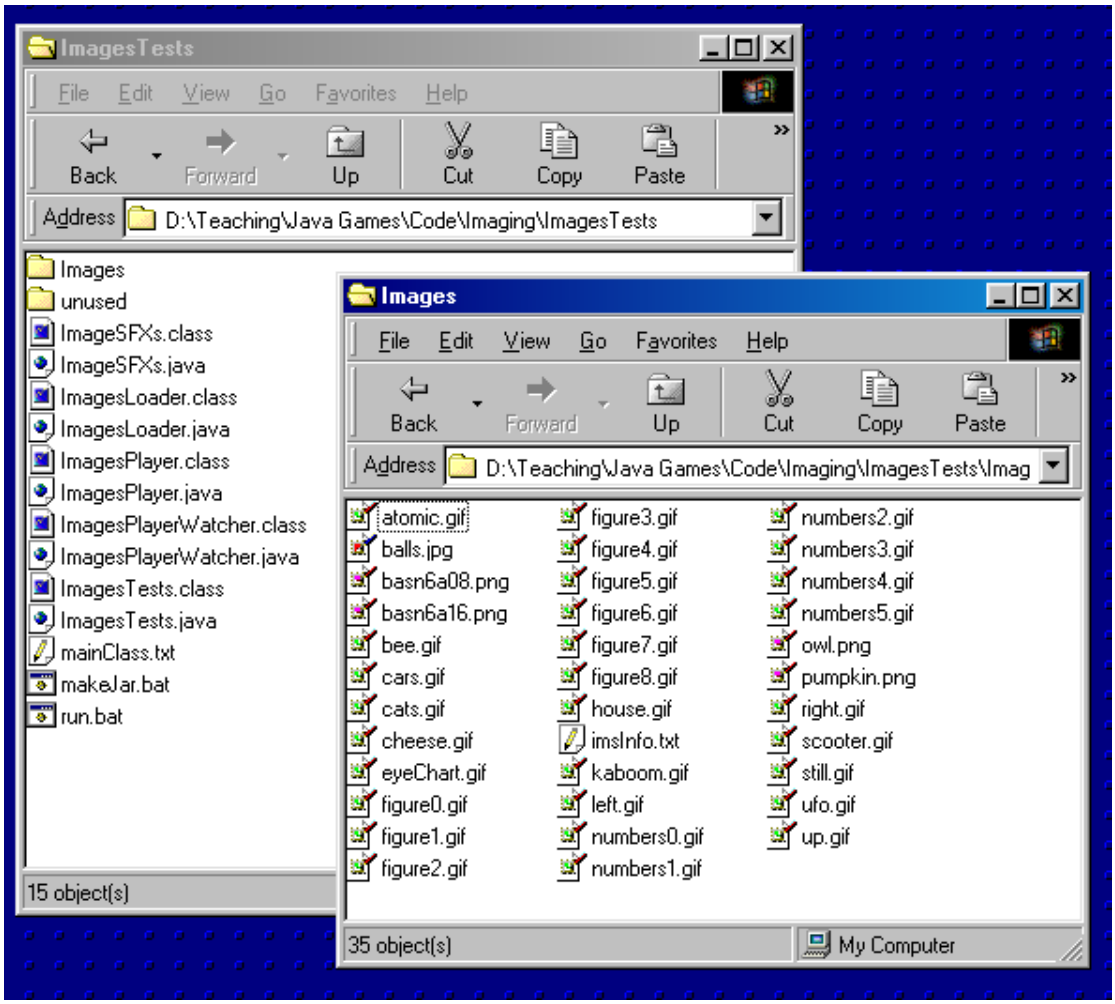

Figure 25. The ImagesTests/ Directory and Images/ Subdirectory.

One issue with using MS Windows is that it displays filenames in a 'user-friendly' lowercase format. Unfortunately, Java is less forgiving, and will be unable to find a

file such as BASN6A08.PNG if told to load basn6a08.png. The application developer should open a DOS window and check the filenames in Images/ directly.

The next step is to create a text file, which will become the basis of the *manifest* inside the JAR file. The manifest holds a range of meta-information about the JAR, related to matters like authentication, extensions, and sealing. However, we'll only add the name of the top-level class, ImagesTests, which contains the main() method. This permits the application to be started by double-clicking.

The text file, mainClass.txt (any name will do), contains a single line:

```
Main-Class: ImagesTests
```

The file should be stored in the same directory as the application.

The JAR file can now be made, using the command:

```
> jar cvmf mainClass.txt ImagesTests.jar *.class Images
```

This command should be executed in the application directory. It has the format:

```
jar <options> <manifest info file> <name of JAR file>
                      <list of input files/directories>
```

The options, cvmf, specify:

- c: create a JAR file;
- v: verbose output goes to stdout during the creation process, including a list of everything added to the JAR;
- m: a manifest information file is included on the command line, and its information should be incorporated into the JAR's manifest;
- f: a filename for the resulting JAR is given on the command line.

The list of input files can use the wildcard symbol, *, as here. All the .class files in the current directory are added to the JAR (these are ImageSFXs.class, ImagesLoader.class, ImagesPlayer.class, ImagesPlayerWatcher.class, and ImagesTests.class, as shown in Figure 25). Also, the subdirectory Images/ is added, together with all its contents.

The ImagesTests.jar file will appear in ImagesTests/, and can be started by double-clicking upon its icon. It is about 130K in size, compressed by 13% from the original collection of files.

The application can be started from the command line as well, by typing:

```
> java –jar ImagesTests.jar
```

The advantage of this approach is that the output from the application will appear in the DOS window, whereas it is lost if the program is started via its icon.

A simple way of checking the contents of the JAR file is to open it with a zip utility, such as WinZip (http://www.winzip.com). Alternatively, type:

```
> jar tf ImagesTests.jar
```

