Java™ 2D API

Enhanced Graphics and Imaging for Java

A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043 USA
415 960-1300 fax 415 969-9131

Contents

Java 2D API

Enhanced Graphics and Imaging

The Java 2D API (Application Programming Interface) provides a powerful, flexible framework for using device and resolution independent graphics in Java programs. The Java 2D API builds on the graphics and imaging classes defined by `java.awt`, extending the capabilities while maintaining compatibility for existing programs. The Java 2D API will enable developers to easily incorporate high-quality 2D graphics, text, and images in Java applications and applets.

The Java 2D API provides a two-dimensional imaging model for line art, text, and images that uniformly addresses color, spatial transformations, and compositing. With the Java 2D API, you use the same imaging model for both screen and print, which provides a highly WYSIWYG (What You See Is What You Get) experience for the user.

Sun Microsystems and Adobe Systems Incorporated are the primary authors of the Java 2D API specification.

This paper describes the Java 2D API and illustrates how the key classes are used. The first section provides an overview of the Java 2D API, using a simple example to introduce the drawing model and key rendering features. The following sections describe the primary elements of the Java 2D API: the graphics rendering pipeline, text and font support, color management, imaging, and graphics device support.

This paper is not intended as an exhaustive description of advanced 2D graphics and imaging for Java or a complete programmer's guide.

1.0 Java 2D API Fundamentals

The Java 2D API handles arbitrary shapes, text, and images and provides a uniform mechanism for performing transformations such as rotation and scaling. The Java 2D API also provides extensive font and color support.

The Java 2D API allows you to control how graphics primitives are rendered through a comprehensive set of attributes associated with the `Graphics2D` state. You can specify characteristics such as the stroke width, join types, and color and texture fills, as well as how the graphics are blended to the screen and whether or not they are antialiased.

Coordinate Spaces

The Java 2D API defines two coordinate spaces: the *User Coordinate Space* and the *Device Coordinate Space*. The origin of the Device Coordinate Space lies in the upper left-hand corner with x-coordinate values increasing to the right and y-coordinate values increasing downward.

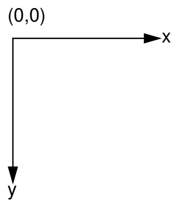

Figure 1-1 Device Coordinate Space and default User Coordinate Space

All graphics objects are described in the device-independent User Coordinate Space until they are rendered on a device such as screen or printer. The rendering state of a `Graphics2D` object associated with the target device includes a `Transform` object that converts the graphics object's User Space coordinates to Device Space coordinates. The default `Transform` results in a default User Coordinate Space that has the same orientation as the Device Coordinate Space

1.1 Drawing

The Java 2D API uses the drawing model defined by the `java.awt` package for drawing to the screen: each `Component` object implements a `paint` method that is invoked automatically whenever something needs to be drawn. When `paint` is invoked, it is passed a `Graphics` object that knows how to draw into the component.

1.1.1 Basic Drawing Process

Suppose you have a component whose job it is to draw a red rectangle. To draw the rectangle using `java.awt`, you implement `Component.paint`:

```
public void paint(Graphics g) {
   g.setColor(Color.red);
   g.fillRect(300, 300, 200, 100);
}
```

This example illustrates the basic drawing process for any component:

1. Specify the rendering attributes for the shape you want to draw by calling one of the `Graphics` attribute methods, such as `setColor`.
2. Define the shape that you want to draw, such as a rectangle.
3. Use the `Graphics` object to render the shape by calling one of the `Graphics` rendering methods, such as `fillRect`.

1.1.2 Drawing with the Java 2D API

The basic drawing process is the same when you use Java 2D API features. The Java 2D API simply provides additional features for specifying fancy paint styles, defining complex shapes, and controlling the rendering process.

`Component.paint` is overloaded to support the Java 2D API drawing features. To use these features, you can implement the version of `paint` that accepts a `Graphics2D` object as a parameter. `Graphics2D` extends `Graphics` to support advanced drawing operations.

Note: For backward compatibility, you can also implement the original paint method, which takes a `Graphics` object as a parameter. To use the new Java 2D API features, cast the `Graphics` parameter to a `Graphics2D`. The default implementation of `Component.paint(Graphics2D)` is to call `Component.paint(Graphics)`.

For example, you could use the new features of the Java 2D API to draw the red rectangle by implementing `Component.paint(Graphics2D)`:

```
public void paint(Graphics2D g2d) {
   // 1. Specify the rendering attributes
```

```
        g2d.setColor(Color.red);
        // 2. Define the shape. (Use Even-Odd rule.)
        BezierPath path = new BezierPath(BezierPath.EVEN_ODD);
        path.moveTo(300.0f, 400.0f); // lower left corner
        path.lineTo(500.0f, 400.0f); // lower right corner
        path.lineTo(500.0f, 300.0f); // upper right corner
        path.lineTo(300.0f, 300.0f); // upper left corner
        path.closePath(); // close the rectangle
        // 3. Render the shape
        g2d.fillPath(path);
    }
```

The process is the same, but the Java 2D API class `BezierPath` is used to define the rectangle. For drawing simple shapes such as the rectangle, it is slightly more complicated to use the Java 2D API; however, the Java 2D API enables you to manage complex drawing operations using the same process:

1. Specify the rendering attributes.

   With the Java 2D API classes you can fill a shape with a solid color, but the Java 2D API also supports more complex fills such as gradients and patterns. To specify complex fills, you use the `setPaint` method. (For more information, see Section 2.6, “Paint.”)

2. Define a shape, a text string, or an image.

   The Java 2D API treats paths, text, and images uniformly; they can all be rotated, scaled, skewed, and composited using the methods introduced in the following sections. In this example, a single rectangle is defined.

   The Java 2D API provides an implementation of the `Path` interface that can be used to define complex shapes. This class, `BezierPath`, allows you to describe a shape using a combination of lines and Bezier curves. (For additional information about the `Path` interface, see “Creating a New Type of Path” on page 16.)

   Using `BezierPath` to define a shape also allows you to control the location of the shape. (The shape can also be translated to a new position using the `Graphics2D` transformation attribute.)

   Winding Rules

   The `BezierPath` constructor takes a parameter that specifies the winding rule to be used for the object. The winding rule is used to determine whether or not a point lies inside the shape when path segments cross. Two different winding rules can be specified for a `BezierPath` object: the even-odd winding rule or the nonzero winding rule. The even-odd rule is specified in this example, but has no effect as the path segments in the rectangle do not cross.

3. Render the shape, text string, or image.

   To actually render the shape, text, or image you call one of the `Graphics2D` rendering methods. In this example, the rectangle is rendered using `fillPath`.

   Transformations

   In the Java 2D API, objects are processed by a `Transform` associated with the `Graphics2D` object before they are drawn. A `Transform` object takes a point or a path and transforms it to a new point or path. The default `Transform` object created when the `Graphics2D` object is constructed performs simple scaling to device coordinates. To get effects such as rotation, translation, or custom scaling, you create `Transform` objects and apply them to the `Graphics2D` object.

   The most commonly used `Transform` is the `AffineTransform`, which performs linear transformations such as translation, rotation, scaling, and skewing. (For more information about transformations, see Section 2.3, “Transformations”).

1.1.3 Managing Complex Drawing Operations

The power of the Java 2D API lies in its ability to manage complex drawing operations within the same framework used to draw the rectangle in the example in Section 1.1.2. Suppose that you want to draw a second rectangle, covering part of the first rectangle, rotated 45° counterclockwise. The new rectangle is filled with blue and rendered 50% transparent, so that the original rectangle is still visible underneath. With the Java 2D API, the second rectangle can easily be added to the previous example:

```
public void paint(Graphics2D g2d) {
   g2d.setColor(Color.red);
   BezierPath path = new BezierPath(BezierPath.EVEN_ODD);
   path.moveTo(0.0f, 0.0f); // lower left corner
   path.lineTo(200.0f, 0.0f); // lower right corner
   path.lineTo(200.0f, -100.0f); // upper right corner
   path.lineTo(0.0f, -100.0f); // upper left corner
   path.closePath(); // close the rectangle
   AffineTransform at = new AffineTransform();
   at.setToTranslation(300.0, 400.0);
   g2d.transform(at);
```

```
    g2d.fillPath(path);
    // Add a second rectangle
    g2d.setColor(Color.blue); // define the color
    AlphaComposite comp =
      AlphaComposite.getInstance(AlphaComposite.SRC_OVER, 0.5);
    g2d.setComposite(comp); //set the composite mode
    // Rotate about the origin -45 deg in radians
    at.setToRotation(-Math.PI/4.0));
    g2d.transform(at);
    g2d.fillPath(path);
}
```

The drawing process is the same for both rectangles:

- The rectangle is defined using a `BezierPath` object.
- The rendering attributes are set by calling `setColor`.
- Transformations are applied before the rectangle is rendered. (`Graphics2d.transform` is used to position both rectangles at `(300, 400) and` rotate the blue rectangle 45° counterclockwise.)
- The rectangle is rendered by calling `fillPath`. In addition, before the blue rectangle is rendered, the transfer mode is specified by creating an instance of `AlphaComposite`.

1. Defining the Color

   To paint the second rectangle with a 50% transparent blue, you first set the color to be painted.

   You also need to indicate how the new color blends with existing colors. To do this, you create an `AlphaComposite` object. An `AlphaComposite` object defines a *transfer mode* that specifies how colors are blended. In this case, you want to create an `AlphaComposite` object that sets the transparency for rendering to 50% and blends the new color over the existing color. To do this, you specify the `SRC_OVER` transfer mode and an alpha value of 0.5 when you create the `AlphaComposite` object. You then call `Graphics2D.setComposite` to use the new `Composite` object.

2. Defining the Rotation

   The rotation is performed by creating a new `AffineTransform` and calling `setToRotation` to specify the counterclockwise rotation of 45 degrees. The transform is then composed with the previous transform of the `Graphics2D` object (the translation to `(300,400)`) by calling `transform`.

The effects of consecutive calls to `transform` are cumulative; from this point forward, anything you draw is translated to `(300,400)` and rotated 45° counterclockwise, as shown in Figure 1-2.

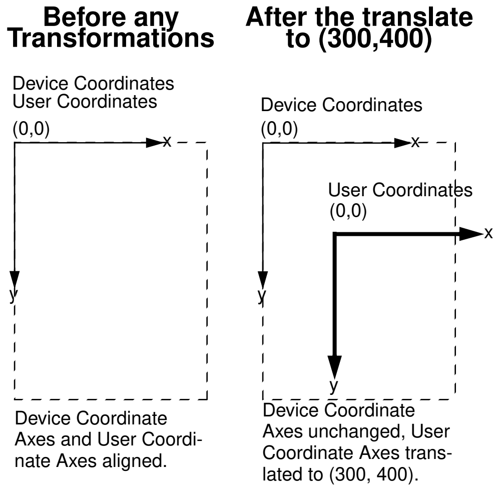

Figure 1-2 Transformation effects

3. Rendering the Blue Rectangle

   Just like the first rectangle, the blue rectangle is rendered by calling `fill-Path`.

   The `Graphics2D` object transforms the path, using the specified `Transform` object. The result is a rotated rectangle. It uses the specified color, blue, to determine what color the output should be. This color is then blended with the colors of the underlying image, based on the state of the `Composite` object and the `Color` object.

Figure 1-3 shows the results of invoking the complete method.

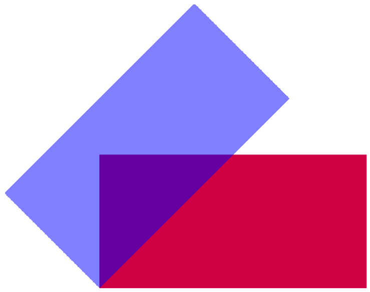

Figure 1-3 Results of invoking example paint implementation

1.1.4 Performing Hit Detection

Detecting where the user clicks the mouse on a graphic can be complicated for complex transformed graphics. The Java 2D API simplifies this task by providing a `Graphics2D` method called `hitPath`. This method takes a `Rectangle` object and a `Path` object as parameters and returns a `boolean` value that indicates whether any point in the rectangle would be painted by the path. The `hitPath` method also takes a `boolean` value that indicates whether or not the path's fill or stroke attributes should be taken into account.

1.2 Text

The Java 2D API provides text handling support that ranges from the simple use of fonts to professional-quality management of character layout and font features.

The Java 2D API enhanced `Font` class provides greater control over fonts than the existing `java.awt.Font` class. It also allows you to retrieve more information about a font, such as the Bezier paths of individual character glyphs. The Java 2D API `Font` class will supersede `java.awt.Font`.

1.2.1 Drawing Text

To draw text, you use the same process that you use for paths. Instead of using a `BezierPath` object to define a shape, you create a `Font` object and render the text, by calling `Graphics2D.drawString`.

For example, to draw a large letter 'J', rotated 45° counterclockwise, on top of the rectangles from the previous example, you add the following code to the body of the `paint` method:

```
// get a 200 point version of Helvetica-BoldOblique
Font myFont = new Font("Helvetica-BoldOblique",
                       Font.Plain, 200);
// display the character 'J' in green
// the rotation and translation have already been done
g2d.setColor(Color.green);
g2d.drawString("J", 0, 20);
```

Because the Java 2D API Font class provides a `getGlyphOutline` method (see Section 3.1.1 on page 20 for more information) that returns the character path, you can use a text string as a clipping path. For example, you could draw only those parts of the rectangles that would show through the rotated letter J by using the character path, scaled appropriately, as a clipping path:

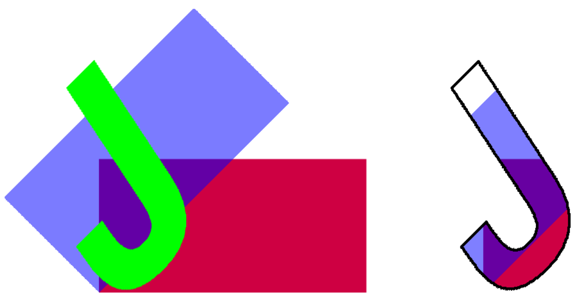

Figure 1-4 Using text as a clipping path

To do this, you get the character's shape by calling `getGlyphOutline`, which returns an instance of `Path`. You then supply the `Path` object as an argument to `setClip`, a method defined by `Graphics2D`. In Figure 1-4, the text outline is also stroked in black.

As illustrated by these examples, the Java 2D API treats text as a first-class citizen. It can be drawn, transformed, used as a clipping path, and composited just like any other graphic element. You can even perform hit detection on text with the `Graphics2D.hitString` method.

1.3 Images

The Java 2D API provides a full range of features for handling images by supplementing the image-handling classes in `java.awt` and `java.awt.image` with several new classes, including: `BufferedImage`, `Tile`, `Channel`, `ComponentColorModel`, and `ColorSpace`.

These classes give advanced Java programmers greater control over images. Using the Java 2D API imaging classes, you can create images in color spaces other than RGB and characterize colors for accurate reproduction. The Java 2D API imaging classes also allow you to specify exactly how pixels are laid out in memory.

Like all other graphic elements, images are transformed by the `Transform` object associated with the `Graphics2D` object when they are drawn. This means that images can be scaled, rotated, skewed, or otherwise transformed just like text and paths. However, images maintain their own color information[1] rather than using the current color.

Displaying an image is straightforward. Having acquired an image (perhaps from a URL), you specify the desired transformation and call `Graphics2D.drawImage`:

```
Image image = applet.getImage(url);
AffineTransform at = new AffineTransform();
at.rotate(Math.PI/4.0);
g2d.transform(at);
g2d.drawImage(image, 0, 0, this);
```

1.3.1 Transparency and Images

Images can carry transparency information for each pixel in the image. This information, called an *alpha channel*, is used in conjunction with the current `Composite` object to blend the image with an existing drawing.

Figure 1-5 contains three images with different transparency information. In each case, the image is displayed over a blue rectangle. This example assumes that an `AlphaComposite` object is installed that uses `SRC_OVER` as its transfer mode for compositing.

1. Images have an embedded color model to interpret pixel data as color.

Figure 1-5 Transparency and images

In the first image, all pixels are fully opaque (the dog's body) or fully transparent (the background). You often see this effect used on web pages. The second image is rendered with uniform, non-opaque transparency for the dog's body. The third image has opaque values around the dog's face and increasingly transparent values as the distance from the dog's face increases.

1.4 Summary

The Java 2D API extends AWT to provide a standard, cross-platform interface for handling complex shapes, text, and images. With the Java 2D API classes, you can incorporate high-quality 2D graphics, text, and images in your applications and applets. The Java 2D API:

- Enables high-quality device and resolution independent graphics
- Enhances font and text handling support
- Provides a single, comprehensive rendering model

The key Java 2D API classes introduced in this section are summarized in Table 1-1.

Class Name	**Description**
`Graphics2D`	A subclass of `Graphics` that encapsulates information about where to draw, drawing parameters such as the current font, and the actual drawing methods.
`Paint`	An interface used to specify the colors used to fill a shape, such as a gradient fill or pattern fill. Paints are alternatives to colors.

Class Name	Description
`Path`	An interface used to maintain a collection of points that describe the outline of a shape.
`BezierPath`	A path that is built using lines and Bezier curves.
`Stroke`	An interface that specifies how to turn a path to be rendered by `drawPath` into an outline of the stroked path. The stroked path can be filled to achieve the same results.
`Transform`	An interface that specifies how to transform a point or path into another point or path.
`AffineTransform`	A transformation that supports rotation, scaling, skewing, and other common linear transformations.
`Composite`	An interface that specifies how to blend two colors to form a third. Used to implement transparency and similar effects.
`Font`	An enhanced `Font` class that provides control over font characteristics and access to detailed information.
`BufferedImage`	A class that supports fine-grain control over an image by allowing you to specify the image's `ColorModel`, image data, and data layout (Tile and `Channel`).
`ColorSpace`	A class that identifies a color space and supports the conversion of color components in a particular color space to and from standard color conversion spaces.

Table 1-1 Basic Java 2D API classes

2.0 Rendering

In the Java 2D API, the rendering of graphics objects is controlled through the `Graphics2D` state attributes. With the `Graphics2D` state attributes, you can set a clipping path to limit the area that is rendered, vary the stroke width, change how strokes are joined together, and compose graphics objects in different ways. These attributes are applied during the rendering process.

2.1 Rendering Pipeline

The rendering process can be broken down into four stages. (Note that this process might actually be compressed to optimize rendering performance.)

1. The graphics object being rendered is converted to graphics primitives and

transformed into the Device Space using the `Transform` from the `Graphics2D` object associated with the target device. This determines where the graphics object should be rendered. How this is done depends on the type of graphics object being rendered:

- When a *path* is rendered, it is converted to a `BezierPath` object. If the path is to be stroked, the `Stroke` object in the `Graphics2D` is used to convert the path to a stroked path. This `BezierPath` is transformed into device coordinates using the `Transform` object in the `Graphics2D`.
- When *text* is rendered, the layout of the glyphs is determined using the information in the fonts used by the string. The glyphs are then converted to outlines that are described by `BezierPath` objects. These `BezierPath` objects are transformed into device coordinates using the `Transform` object in the `Graphics2D`.
- When an *image* is rendered, its bounding box (in user coordinates) is transformed into device coordinates using the `Transform` object in the `Graphics2D`.

2. The current clip is used to constrain the rendering operation. The clip can be any shape that can be described by a `Path` object. The clip is transformed into the Device Space using the `Transform` in effect when `setClip` was called.
3. The color to be rendered is determined. For image operations, the color is taken from the data of the image. For all other operations, the current `Paint` or `Color` object in the `Graphics2D` is queried for the color.
4. The color is applied to the rendering target using the current `Composite` object.

2.2 Controlling the Rendering Quality

When graphics primitives are rendered on graphics display devices, their edges can appear jaggy due to aliasing. *Antialiasing* is a technique used to render objects with smoother appearing edges. This technique requires additional computing resources and can impact the rendering speed. The Java 2D API lets you indicate whether you want objects to be rendered as quickly as possible, or whether you prefer that the rendering quality is as high as possible. Your preference is specified as a hint because not all platforms support modification of the rendering mode.

You specify your rendering preferences to a `Graphics2D` object by calling `setRenderingHints`. There are two types of hints. The first indicates whether or not objects should be antialiased when they are rendered. The second indicates a preference in the trade-off between speed and quality. For example, this hint could affect how precisely stroke joins are rendered or whether better dithering or interpolation should be performed.

2.3 Transformations

The Java 2D API transformations are based on the `Transform` interface. The `AffineTransform` class implements `Transform` to support operations such as scaling, rotation, and skewing.

2.3.1 Using Affine Transformations

An affine transformation performs a linear transformation on a set of graphics primitives. It always transforms straight lines into straight lines and parallel lines into parallel lines, but it might alter the distance between points and the angles between non-parallel lines. An affine transformation is based on a two-dimensional matrix of the following form:

$$\begin{bmatrix} a & b & t_x \\ c & d & t_y \end{bmatrix} \quad \text{where} \quad x' = ax + by + t_x \quad \text{and} \quad y' = cx + dy + t_y$$

To use the `AffineTransform` class, you do not need to interact directly with transformation matrices. You simply invoke the appropriate sequence of rotations, translations, and other transformations to get the effect you want.

The `Transform` associated with the `Graphics2D` object transforms all of the paths, text, and images you draw from User Space to Device Space; this is the `Transform` that a program interacts with most. `Graphics2D` implements a version of `drawImage` that takes an instance of `Transform` as a parameter. When you use this version of `drawImage`, the image is drawn as if you had appended the transform to the `Graphics2D` object. This allows you to perform additional transformations on an image object when it is drawn (this can be thought of as a transform from image space to User Space). Similarly, you can apply an instance of `AffineTransform` to a `Font` object to create a new `Font` object for drawing text, as discussed in “Text Handling” on page 20.

2.3.2 Creating Custom Transformations

Advanced clients can create classes that implement the `Transform` interface to provide new types of transformations. New transform classes must support the

`Transform.createInverse` method, which inverts a transformation and constructs a new `Transform` object that is the inverse of the current one. In other words, all Java 2D API transforms must produce an inverse if it exists. All Java 2D API transforms must also be able to operate on instances of `BezierPath`.

For example, you could define a new, non-linear transform to draw objects in perspective so that they appear to shrink away in the distance. (This effect cannot be accomplished with the linear `AffineTransform`.) Your new class, `PerspectiveTransform`, can transform points and paths using any method needed.[2] It is not constrained to transformations that can be described by a linear transform matrix.

2.3.3 Transformation Pipeline

Clients can use different implementations of `Transform` as needed. For example, you could apply the `PerspectiveTransform` to a drawing and then apply an `AffineTransform` to rotate the drawing, as shown in Figure 2-1.

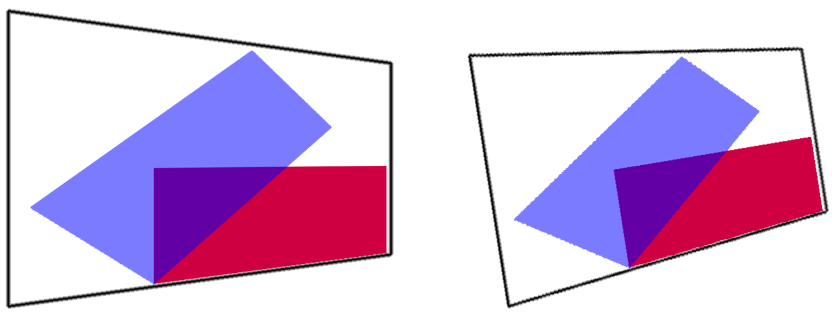

Figure 2-1 Perspective and affine transformations

When you append a `Transform` object to a `Graphics2D` object, it becomes part of a pipeline of transformations that are applied to the drawing. The `Graphics2D Transform` object might actually be a `TransformChain`. `TransformChain` is an implementation of the `Transform` interface that executes multiple transforms in a predefined order. It is useful for representing sequences of transformations that cannot be combined into a single simple type of `Transform` object.

---

2. Transform operations that require the use of rational curves to represent transformed Bezier curves (like the perspective example) must be able to produce approximations using non-rational Bezier curves.

2.4 Creating a New Type of Path

To create a new path, you can implement the `Path` interface. It doesn't matter how the path is represented internally, as long as the `Path` interface methods can be implemented. For example, a simple implementation of `Path` could be created to represent polygons as arrays of points. This class, `PolygonPath`, needs to define just one new method, `addPoint`. To build a `PolygonPath` object, a client would repeatedly call `addPoint`. Once the path is built, it could be used in a call to `drawPath`, `setClip` or any other method that expects a `Path` object as an argument.

The `PolygonPath` class must implement the `Path` interface methods, including `createTransformedPath` and `getAsBezierPath`. The `createTransformedPath` method creates a new `Path` object that represents the current path as transformed by the `Transform` parameter. The `getAsBezierPath` method constructs a new `BezierPath` object that contains a representation of the path.

Both `Transform` and `Path` define a `createTransformedPath` method. When the Java 2D API needs to transform a path before painting it, it first calls `Path.createTransformedPath`. If the path contains internal information that is necessary for its points to be transformed correctly (using the `Transform` object's `transform` method), or if the path contains special knowledge of `Transform` objects that should be applied, this processing is done in `Path.createTransformedPath`. If the `Path` object does not need to perform special processing in `createTransformedPath`, it can simply call the `createTransformedPath` method of the supplied `Transform` object.

When `Transform.createTransformedPath` is called, the `Transform` object is responsible for transforming an arbitrary path with an arbitrary internal representation. If the `Transform` does not recognize the particular type of path, it can call the path's `getAsBezierPath` method to get the path in a canonical form. All `Transform` objects must be able to operate on `BezierPath` objects.

Implementing `PolygonPath.getAsBezierPath` for the `PolygonPath` class is quite simple. First, a `BezierPath` object is constructed and then `moveTo` is called, followed by a sequence of calls to `lineTo`. For other types of custom paths, this conversion might be more complicated and could result in a loss of information.

2.5 Stroke

When a `Path` object is drawn, it is first converted to an equivalent `BezierPath`. In the Java 2D API, all `Graphics2D` objects know how to stroke and fill a `BezierPath`. Stroking a `BezierPath` object is equivalent to running a logical pen along the segments of the `BezierPath`. The `Stroke` object encapsulates the characteristics of the mark drawn by the pen. The Java 2D API provides a `BasicStroke` class that contains characteristics such as the line width, end-cap style, segment join-style, and the dashing pattern. The end-cap styles are chopped, round, and squared. The join styles are bevel, miter, and round. The miter join can be limited to a certain length. The first image in Figure 2-2 uses the miter join-style; the second image uses a round join-style, a round end-cap style, and a dashing pattern.

Figure 2-2 Stroke styles

2.6 Paint

Section 1.0, "Java 2D API Fundamentals" demonstrates how to specify a simple fill color for a path. With the Java 2D API, you can also fill a shape with more complex paint styles, such as gradients and textures. To facilitate the use of complex fills, the Java 2D API defines a new class called `Paint` and a `Graphics2D` method called `setPaint`. These features eliminate the time-consuming task of creating complex fills using simple solid-color paints.

Figure 2-3 Complex paint styles

Conceptually, all drawing is done with a `Paint` object. A `Color`[3] object can be thought of as a very simple type of `Paint` object, and the `setColor` method as a special case of `setPaint`. In effect, `setColor` installs a `Paint` object for you that paints with a single color. (You can actually pass a `Color` to the `setPaint` method, because `Color` implements the `Paint` interface and is just another type of `Paint` object.)

3. The enhanced Java 2D API `Color` class will supersede `java.awt.Color`.

Once you call `setPaint`, everything you draw (such as text and paths) is painted using the specified `Paint` object.

A `Paint` object must ultimately specify what color to paint each pixel in a shape. Conceptually, the Java 2D API determines what pixels comprise a shape and asks the `Paint` object for the color of each. It then converts that color to an appropriate pixel value for the output device and writes the pixel to that device. This is a tedious process that provides few opportunities for optimization.

To streamline this process, the Java 2D API processes pixels in batches. A batch can be either a contiguous set of pixels on a given scanline or a block of pixels. This batch processing is done in two steps:

1. A `PaintContext` object is created from the `Paint` object. The `PaintContext` object stores the contextual information about the current rendering operation and the information necessary to generate the colors. The `createContext` method takes as parameters the bounding boxes of the graphics object being filled in User Space and in Device Space, the `ColorModel` in which the colors should be generated, and the transform used to map User Space into Device Space. The `ColorModel` is treated as a hint because not all `Paint` objects can support an arbitrary `ColorModel`. (For more information about `ColorModels`, see Section 4.0, “Color Management.”)
2. The `PaintContext` is asked for the `ColorModel` of the generated paint color and the `Tile` that contains the actual color data for a given batch. The `getColorModel` method is only called once, but `getTile` is called repeatedly as the area being rendered is processed in batches by the rendering pipeline. This information is then passed to the next stage in the rendering pipeline, which draws the generated color using the current `Composite` object.

2.7 Composite

In Section 1.1 we discussed basic compositing and introduced the `AlphaComposite class`, an implementation of the `Composite` interface. This class supports a number of different composition styles and is intended to meet the needs of most clients. Instances of this class embody a composition rule that describes how to blend a new color with an existing one. The alpha values for rendering are derived from a `Color`, `Paint`, or `Image` object, combined with pixel coverage information from a rasterized path (when antialiased rendering is being performed).

2.7.1 Managing Transparency

One of the most commonly used compositing rules in the `AlphaComposite` class is `SRC_OVER`. When `AlphaComposite.SRC_OVER` is applied, it indicates that the new color (the source color) should be blended over the existing color (the destination color). The alpha value indicates, as a percentage, how much of the existing color should show through the new color. It is a measure of the transparency of the new color. Opaque colors don't allow any existing color show through, while transparent colors let all of the existing color show through.

You can also use an `AlphaComposite` object to add an additional level of transparency to everything drawn. To do this, you create an `AlphaComposite` object with an alpha value that increases the transparency of every object drawn:

```
comp = AlphaComposite.getInstance(AlphaComposite.SRC_OVER, 0.5);
```

The specified alpha value, 0.5, is combined with the alpha values of a `Color`, `Paint`, or `Image` prior to rendering. This reduces the alpha of everything drawn by 50%, making everything 50% more transparent.

In this example, which extends the example in Section 1.1.3, the text is drawn before the `Composite` object is created so that it is totally opaque. Then the `AlphaComposite.SRC_OVER` object is created to set the transparency to 50% and the two overlapping rectangles are drawn, as shown in Figure 2-4:

Figure 2-4 Compositing

The red rectangle that was completely opaque is now partially transparent and the blue rectangle is even more transparent than it was originally. This additional layer of transparency provided by the `AlphaComposite` class can be useful in a number of circumstances.

2.7.2 Defining Custom Composition Rules

You can create an entirely new type of compositing operation by implementing the `Composite` and `CompositeContext` interfaces. A `Composite` object provides a `CompositeContext` object that actually holds the state and performs the compositing work. Multiple `CompositeContext` objects can be created from one `Composite` object to maintain the separate states in a multi-threaded environment.

3.0 Text and Fonts

You can use the Java 2D API transformation and drawing mechanisms with text strings. The Java 2D API also adds new text related classes that support sophisticated text layout and fine-grain font control.

3.1 Text Handling

The Java 2D API provides an enhanced `Font` class that provides greater control over fonts than the previous `java.awt.Font`. This enhanced `Font` class supports:

- Specification of detailed font information
- Access to information about a font and its glyphs

3.1.1 Specifying and Obtaining Font Information

There is a rich body of information that can be used to describe a font, including its name, the type technology it uses, its version, and its style parameters. Such information is represented by the `FontDescriptor` class, which can be used to locate a `Font` for a specific purpose. A `FontDescriptor` consists of a set of key/value pairs.

Table 3-1 lists possible `FontDescriptor` keys. You can use a `FontDescriptor` object to obtain a list of host system fonts that share particular characteristics.

Name	Semantics
`Name`	If present, supplies the name of the requested font, such as `Helvetica-BoldOblique`.
`Family`	If present, supplies the family of the requested font, such as `Helvetica`.
`Style`	If present, supplies the style parameters of the requested font. Possible values include `Plain`, `Italic`, and `Bold`.
`Technology`	If present, identifies the type technology of the requested font. Possible values represent technologies such as `Type1` and `TrueType`.

Name	Semantics
`Version`	If present, identifies the version of the requested font.

Table 3-1 Font descriptor keys

Every `Font` object contains attributes for font name, size, and transform and an array of `FontFeatures` that describe the particular `Font`. The `Font` class defines several convenience methods that allow you to access this data directly.

The `Font` class also provides access to font metrics. Every font object contains detailed metrics for the font. `Font` allows you to access metric and outline information through the methods `getDesignMetrics`, `getGlyphMetrics`, and `getGlyphOutline`.[4] The path returned by `getGlyphOutline` is scaled using the `Font` size and transform, but does not reflect the `Transform` associated with the `Graphics2D` object.

3.1.2 Accessing Text Paths

You can use the `Font.getGlyphOutline` method to access the path of any glyph in a font, as illustrated in Section 1.2. The `StyledString` class also provides a `getStringPath` method that simplifies the conversion of an entire block of text to a path. This method returns an instance of `Path` that describes the character shapes of the laid-out text. The returned `Path` object reflects any transformations applied to the `Font` object associated with the string, but not the `Transform` associated with the `Graphics2D` object.

3.1.3 Transforming Text

Using the `Font.deriveFont` methods, you can create a new `Font` object with different attributes from an existing `Font` object. For example, to scale a font to a custom size, you could create an instance of `Font` with a unitary size and use `Font.deriveFont` to apply a `Transform` and create a new scaled `Font` object.

Similarly, you could apply a `Transform` to the `Font` to skew the text, as shown in the second part of Figure 3-1. In the first image in Figure 3-1, the string “Java” is rotated several times around a center point. In the second image, a `Transform` object is used to generate a skewed version of the font before the string is drawn:

---

4. Note that both getGlyphMetrics and getGlyphOutline take a glyph identifier, not a character.

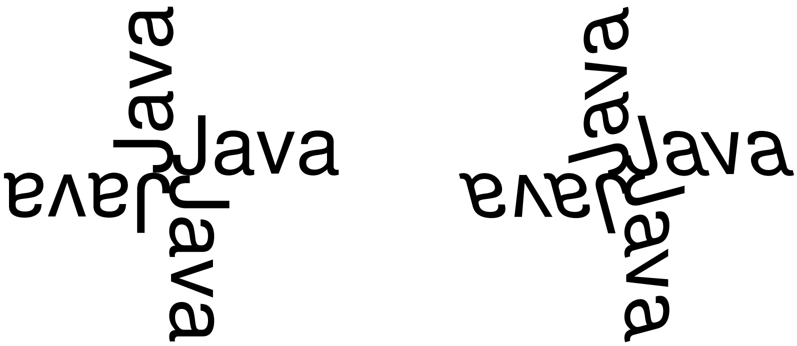

Figure 3-1 Transforming text

The following code uses the `deriveFont` method to implement this effect:

```
// Create a transformation for the font.
AffineTransform fontAT = new AffineTransform();
fontAT.setToScale(72.0, 72.0);
// Describe the font you want to use and then instantiate it
Font theFont = new Font("Helvetica", Font.PLAIN, 1);
Font theDerivedFont = theFont.deriveFont(fontAT);
// Define the rendering transform
AffineTransform at = new AffineTransform();
at.setToTranslation(400.0, 400.0);
g2d.transform(at);
at.setToRotation(Math.PI / 2.0);
// Create a StyledString object, specifying the text and
// transformed font.
StyledString ss = new StyledString("Java", theDerivedFont);
// Draw four copies of the string at 90 degree angles
for (int i = 0; i < 4; i++) {
    g2d.drawString(ss, 0.0f, 0.0f);
    g2d.transform(at);
}
// Create a skewed version of the font
fontAT.append(new AffineTransform(1.0, 0.0, -1.2, 1.0,
              0.0, 0.0));
theDerivedFont = theFont.deriveFont(fontAT);
// Create a StyledString object, specifying the text and the
// skewed font.
ss = new StyledString("Java", theDerivedFont);
```

```
// Translate to a new location
at.setToTranslation(400.0, 0.0);
g2d.transform(at);
at.setToRotation(Math.PI / 2.0);
// Draw four more copies of the string at 90 degree angles
// with the skewed font.
for (int i = 0; i < 4; i++) {
    g2d.drawString(ss, 0.0f, 0.0f);
    g2d.transform(at);
}
```

3.2 Advanced Layout

Before a piece of text can be displayed, it is necessary to determine exactly where each character should be placed. Most clients leave this layout process up to the system, which supplies a set of algorithms that compute the layout based on information contained in the font (such as the font metrics) and provided by the client (such as the text itself and the requested point size).

The Java 2D API provides text layout facilities that handle most common cases, including text strings with mixed fonts, mixed languages, and bidirectional text.

Advanced clients might want to compute the text layout themselves so that they can exercise detailed control over what glyphs are used and where they are placed. Using information such as glyph sizes, kerning tables, and ligature information, advanced clients can use their own algorithms to compute the text layout, bypassing the system's layout mechanism.

The `GlyphSet` class provides a way of displaying the results of custom layout mechanisms. A `GlyphSet` object can be thought of as the output of an algorithm that takes a string and computes exactly how the string should be displayed. The system has a built-in algorithm and the Java 2D API lets advanced clients define their own algorithms. Normally, when you construct a `StyledString` object, you pass in the text you want to be displayed. The system then processes this text and builds a `GlyphSet` object for you, based on its layout algorithm.

A `GlyphSet` object is basically an array of glyphs and glyph locations. Glyphs are used instead of characters to provide total control over layout characteristics such as kerning and ligatures. For example, when displaying the string "`final`", you might want to replace the leading "`fi`" substring with the ligature "ﬁ". (In professional publishing, certain combinations of two or more characters are commonly replaced by a single glyph, known as a ligature.) In this case, the `GlyphSet` object will have fewer glyphs than the number of characters in the original string.

Figure 3-2 and Figure 3-3 illustrate how `GlyphSet` objects are used with the default layout mechanism and with custom layout mechanisms. In Figure 3-2, the client builds a `StyledString` object and passes it to the `drawString` method. The built-in layout algorithm determines which glyphs to use and where each of them should be placed. This information is stored internally using instances of `GlyphSet`. These `GlyphSet` objects are then passed to a glyph rendering routine that does the actual drawing.

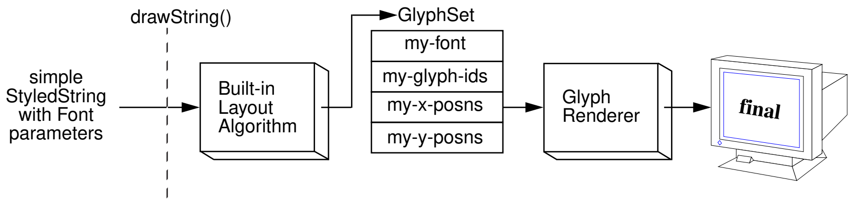

Figure 3-2 Using the built-in layout algorithm

In Figure 3-3, the client assembles the information necessary to lay out the text. A custom layout algorithm is used to determine which glyphs to use and where they should be placed. In this example, the `"fi"` substring is replaced with the “ﬁ” ligature. This layout information is then stored in a `GlyphSet` object. The `GlyphSet` object is then passed to the `drawString` method, which passes it directly to the glyph renderer.

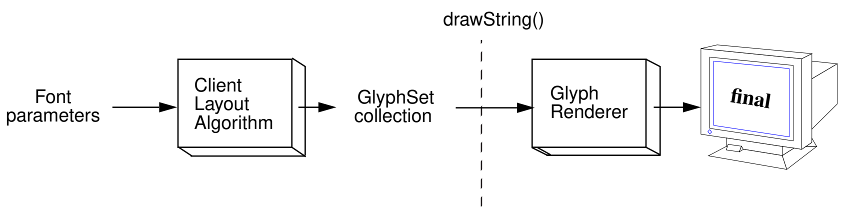

Figure 3-3 Using custom layout algorithms

4.0 Color Management

Color imaging is one of the fundamental components of any graphics system, and it is often a source of great complexity in the imaging model. The Java 2D API provides support for high-quality color output that is both easy to use and allows advanced clients to make sophisticated use of color.

4.1 Specifying Colors

To display a rectangle of a certain color, such as the process color cyan, you need a way to describe this color to Java. There are a number of different ways to describe a color; for example, a color could be described as a set of red, green, and blue (RGB) components, or a set of cyan, magenta, yellow, and black (CMYK) components. These different techniques for specifying colors are called *color spaces*.

As you probably know, colors on a computer screen are generated by blending different amounts of red, green, and blue light. Therefore, using an RGB color space is standard for imaging on computer monitors. Similarly, four-color process printing uses cyan, magenta, yellow, and black ink to produce color on a printed page; the printed colors are specified as percentages in a CMYK color space.

Due to the prevalence of computer monitors and color printing, RGB and CMYK color spaces are both commonly used to describe colors. However, both types of color spaces have a fundamental drawback—they are device-dependent. The cyan ink used on one printer might not exactly match the cyan ink used on another. Similarly, a color described as an RGB color might look blue on one monitor and purplish on another.

The Java 2D API refers to RGB and CMYK as color space types. A particular model of monitor with its particular phosphors defines its own RGB color space. Similarly, a particular model of printer has its own CMYK color space. Different RGB or CMYK color spaces can be related to each other through a device-independent color space.

Standards for the device-independent specification of color have been defined by the International Commission on Illumination (CIE). The most commonly used device-independent color space is the three-component XYZ color space developed by CIE. When you specify a color using `CIEXYZ`, you are insulated from device dependencies.

Unfortunately, it's not always practical to describe colors in the `CIEXYZ` color space—there are valid reasons for representing colors in other color spaces. To obtain consistent results when a color is represented using a device-dependent color space such as a particular RGB space, it is necessary to show how that RGB space relates to a device-independent space like `CIEXYZ`.

One way to map between color spaces is to attach information to the spaces that describes how the device-dependent space relates to the device-independent space. This additional information is called a *profile*. A commonly used type of color profile is the ICC Color Profile, as defined by the International Color Consortium. For details, see the ICC Profile Format Specification, version 3.3 available at `http://www.color.org`.

Figure 4-1 illustrates how a solid color and a scanned image are passed to the Java 2D API, and how they are displayed by various output devices. As you can see in Figure 4-1, both the input color and the image have profiles attached.

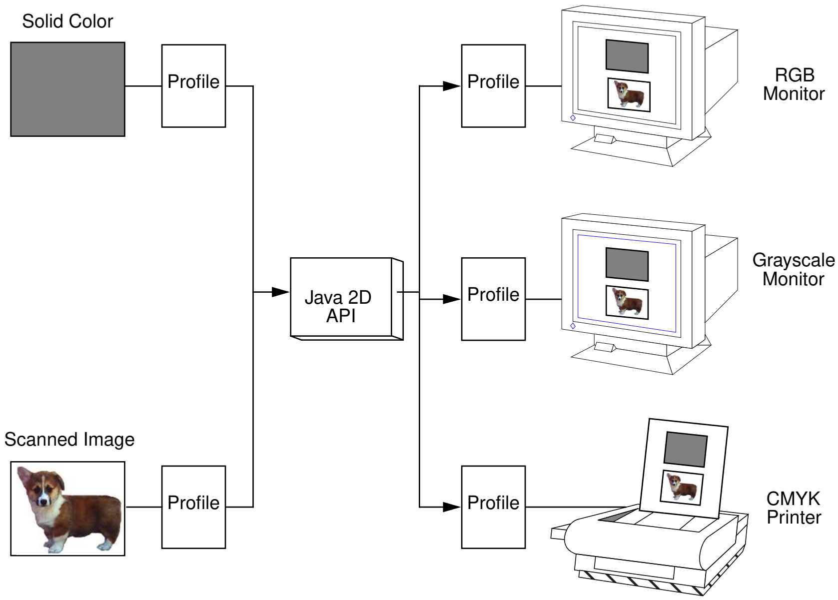

Figure 4-1 Using profiles to map between color spaces

Once the API has an accurately specified color, it must reproduce that color on an output device, such as a monitor or printer. These devices have imaging characteristics of their own that must be taken into account to make sure that they produce the correct results. Another profile is associated with each output device to describe how the colors need to be transformed to produce accurate results.

Achieving consistent and accurate color requires that both input colors and output devices be profiled against a standard color space. For example, an input color could be mapped from its original color space into a standard device-independent space, and then mapped from that space to the output device's color space. In many respects, the transformation of colors mimics the transformation of graphical objects in an (x, y) coordinate space. In both cases, a transformation is used to specify coordinates in a "standard" space and then map those coordinates to a device-specific space for output.

4.2 Color Classes

The key color management classes in the Java 2D API are `Color`, `ColorModel`, and `ColorSpace`. The `Color` class describes a color in terms of its constituent components in its particular color space. The `ColorModel` class provides information necessary to convert the components of a pixel in an `Image` or `BufferedImage` into color components in a particular color space. The `ColorSpace` class has methods for converting color components in a particular color space to and from a well-defined `CIEXYZ` color conversion space as well as a standard RGB color space.

4.2.1 Color

The `Color` class provides a description of a color in a particular color space. An instance of `Color` contains the value of the color components and a `ColorSpace` object. Because a `ColorSpace` object can be specified in addition to the color components when a new instance of `Color` is created, the `Color` class can handle colors in any color space.

The `Color` class has a number of methods that support a proposed standard RGB color space called `sRGB` (see `http://www.w3.org/pub/WWW/Graphics/Color/sRGB.html`). `sRGB` is the default color space for the Java 2D API. Several constructors defined by the Color class omit the `ColorSpace` parameter. These constructors assume that the color's RGB values are defined in `sRGB`, and use a default instance of `ColorSpace` to represent that space.

Java uses `sRGB` as a convenience to application programmers, not as a reference color space for color conversion. Many applications are primarily concerned with RGB images and monitors, and defining a standard RGB color space makes writing such applications easier. The `ColorSpace` class defines the methods `toRGB` and `fromRGB` so that developers can easily retrieve colors in this standard space. These methods are not intended to be used for highly accurate color correction or conversions. See Section 4.2.3, “ColorSpace” for more information.

To create a color in a color space other than `sRGB`, you use the `Color` constructor that takes a `ColorSpace` object and an array of floats that represent the color components appropriate to that space. The `ColorSpace` object identifies the color space.

4.2.2 ColorModel

The `ColorModel` class contains data which is used to interpret pixel data in an image. This includes mapping components in the data channels of an image to components of a particular color space. It might also involve extracting pixel components from packed pixel data, retrieving multiple components from a single data channel using masks, and converting pixel data through a lookup table. See Section 5.2, "Using Offscreen Buffers" for more information on the `ColorModel` class.

4.2.3 ColorSpace

A `ColorSpace` object represents a color space, such as a particular RGB or CMYK space. A `ColorSpace` object serves as a colorspace tag that identifies the specific color space of a `Color` object or, through a `ColorModel` object, of an `Image`, `BufferedImage`, or `GraphicsConfiguration`. `ColorSpace` provides methods that transform `Colors` in a specific color space to and from `sRGB` and to and from a well-defined `CIEXYZ` color space.

All `ColorSpace` objects must be able to map a color from the represented color space into `sRGB` and transform an `sRGB` color into the represented color space. Since every color contains a `ColorSpace` object, set explicitly or by default, every color can also be converted to `sRGB`. Similarly, since every `GraphicsConfiguration` is also associated with a `ColorSpace` object, any `sRGB` color can be displayed on any output device. It follows that a color specified in any color space can be displayed by any device by mapping it through `sRGB` as an intermediate color space.

The methods used for this process are `toRGB` and `fromRGB`:

- `toRGB` transforms a color in the represented color space, such as a CMYK space, to a color in `sRGB`.
- `fromRGB` takes a color in `sRGB` and transforms it into the represented color space.

Though mapping through `sRGB` always works, it's not always the best solution. For one thing, `sRGB` cannot represent every color in the full gamut of `CIEXYZ` colors. If a color is specified in some space that has a different gamut (spectrum of representable colors) than `sRGB`, then using `sRGB` as an intermediate space results in a loss of information. To address this problem, the `ColorSpace` class can map colors to and from another color space, the "conversion space" `CIEXYZ`.

The methods `toCIEXYZ` and `fromCIEXYZ` map color values from the represented color space to the conversion space. These methods support conversions between any two color spaces at a reasonably high degree of accuracy. However, it is expected that built-in `ColorSpace` implementations (such as `ICC_ColorSpace`) will support high-performance conversion based on underlying platform color-management systems.

Figure 4-2 and Figure 4-3 illustrate the process of translating a color specified in a CMYK color for display on an RGB color monitor. Figure 4-2 shows a mapping through `sRGB`. As this figure illustrates, the translation of the CMYK color to an RGB color is not exact because of a gamut mismatch. [5]

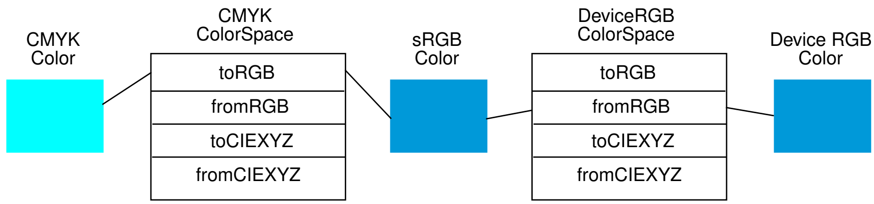

Figure 4-2 Mapping through sRGB

Figure 4-3 shows the same process using `CIEXYZ` as the conversion space. When `CIEXYZ` is used, the color is passed through accurately.

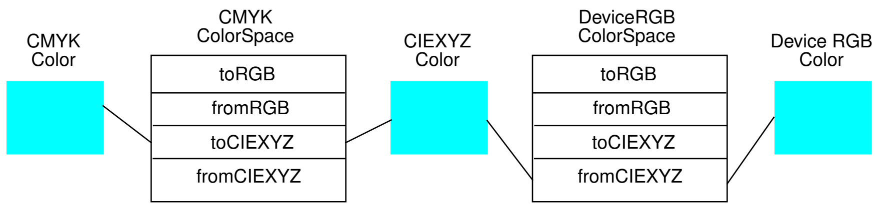

Figure 4-3 Mapping through CIEXYZ

4.2.4 ICC_Profile and ICC_ColorSpace

`ColorSpace` is actually an abstract class. The Java 2D API provides one implementation, `ICC_ColorSpace`, which is based on ICC Profile data as represented by the `ICC_Profile` class. You can define your own subclasses to represent arbitrary color spaces, as long as the methods discussed above are implemented. However, most developers can simply use the default `sRGB ColorSpace` or color

5. Of course, the colors used in these diagrams are illustrative, not accurate. The point is that colors might not be mapped accurately between color spaces unless an appropriate conversion space is used.

spaces that are represented by commonly available ICC Profiles, such as profiles for monitors and printers, or profiles embedded in image data.

Section 4.2.3 describes how `ColorSpace` objects represent a color space and how colors in the represented space can be mapped to and from a conversion space. Color management systems are often used to handle the mapping between color spaces. A typical color management system (CMS) manages ICC profiles, which are similar to `ColorSpace` objects; ICC profiles describe an input space and a connection space, and define how to map between them. Color management systems are very good at figuring out how to map a color tagged with one profile into the color space of another profile.

The Java 2D API defines a class called `ICC_Profile` that holds data for an arbitrary ICC Profile. `ICC_ColorSpace` is an implementation of the abstract `ColorSpace` class. `ICC_ColorSpace` objects can be constructed from `ICC_Profiles`. (There are some limitations—not all ICC Profiles are appropriate for defining an `ICC_ColorSpace`).

`ICC_Profile` has several subclasses that correspond to specific color space types, such as `ICC_ProfileRGB` and `ICC_ProfileGray`. Each subclass of `ICC_Profile` has a well-defined input space (such as an RGB space) and a well-defined connection space (like `CIEXYZ`). The Java 2D API can use a platform's CMS to access color profiles for various devices such as scanners, printers, and monitors. It can also use the CMS to find the best mapping between profiles.

5.0 Imaging

Image processing involves the manipulation of raster images, often to improve visual appearance or bring out subtle shapes and patterns that might otherwise escape visual detection. Any of the image processing effects provided in popular photo-editing programs can be produced using the Java 2D API image-processing classes, or by extending those classes.

5.1 Image Processing and Enhancement

The Java 2D API provides a set of classes that define operations on `BufferedImage` and `Tile` objects. These image processing classes share a common architecture. Each image processing operation is embodied in a class. Each class defines a `filter` method that performs the actual image manipulation. This method might operate on a source and destination `BufferedImage`, or a source and destination `Tile`. Figure 5-1 illustrates the basic model for Java 2D API image processing:

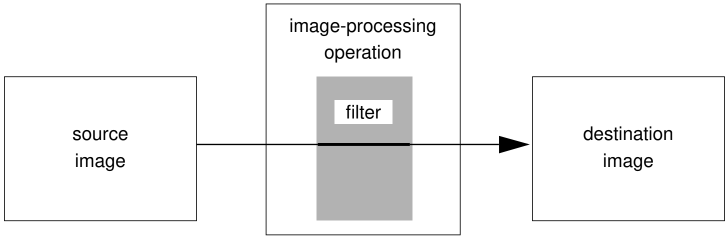

Figure 5-1 Image processing model

The operations supported include amplitude scaling, lookup-table modification, linear combinations of channels, color conversion, and convolutions. The classes that implement these operations include `AffineTransformOp`, `ChannelCombineOp`, `ColorConvertOp`, `ConvolveOp`, `LookupOp`, `RescaleOp`, and `ThresholdOp`. These classes can be used to blur, sharpen, enhance contrast, threshold, and color correct images.

Figure 5-2 illustrates edge detection and enhancement, an operation that searches for sharp changes in intensity within an image and emphasizes them. Edge detection is commonly used in medical imaging and mapping applications. Edge detection is used to increase the contrast between adjacent structures in an image, allowing the viewer to discriminate greater detail.

Figure 5-2 Edge detection and enhancement

Figure 5-3 demonstrates lookup table manipulation via rescaling and thresholding. Rescaling can increase or decrease the intensity of all points. Rescaling can be used to increase the dynamic range of an otherwise neutral image, bringing out detail in a region that appears neutral or flat. Thresholding can clip ranges of intensities to a specified level.

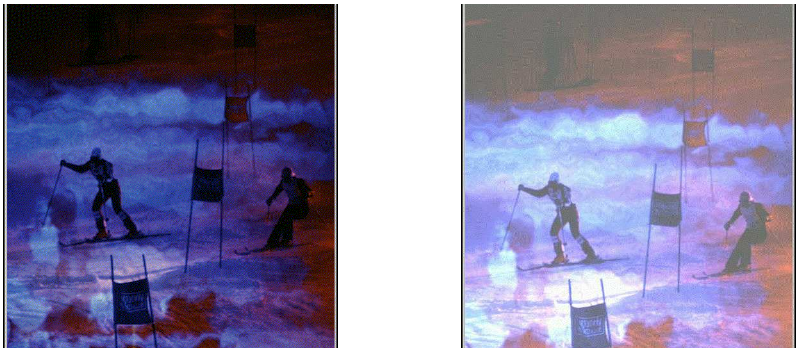

Figure 5-3 Lookup-table manipulation

The image processing classes provided by the Java 2D API are summarized in Table 5-1:

Class Name	Operates on	Description
`AffineTransformOp`	`BufferedImage, Tile`	Performs an affine transformation on the image or tile.
`ChannelCombineOp`	`BufferedImage, Tile`	Performs arbitrary linear combinations on channels.
`ColorConvertOp`	`BufferedImage, Tile`	Performs color conversion.
`ConvolveOp`	`BufferedImage, Tile`	Performs spatial filtering.
`LookupOp`	`BufferedImage, Tile`	Uses a lookup table to remap pixel data from one intensity to another.
`RescaleOp`	`BufferedImage, Tile`	Rescales the data by multiplying each pixel by a scale factor and adding an offset.
`ThresholdOp`	`BufferedImage, Tile`	Sets pixel intensities in a given range to a constant.

Table 5-1 Image processing classes

5.1.1 Processing an Image

The following code fragment illustrates how to use one of the image processing classes, `ConvolveOp`. Convolution is the process that underlies most spatial filtering algorithms. Convolution is the process of weighting or averaging the value of each pixel in an image with the values of neighboring pixels. In this example, each pixel in the source image is averaged equally with the eight pixels that surround it.

```
float weight = 1.0f/9.0f;
float[] elements = new float[9]; // create 2D array
// fill the array with nine equal elements
for (i = 0; i < 9; i++) {
   elements[i] = weight;
}
// use array of elements as argument to create a Kernel
private Kernel myKernel = new Kernel(3, 3, 1, 1, elements);
public ConvolveOp simpleBlur = new ConvolveOp(myKernel);
// sourceImage and destImage are instances of BufferedImage
simpleBlur.filter(sourceImage, destImage) // blur the image
```

The variable `simpleBlur` contains a new instance of `ConvolveOp` that implements a blur operation on a `BufferedImage` or a `Tile`. Suppose that `sourceImage` and `destImage` are two instances of `BufferedImage`. When you call `filter`, the core method of the `ConvolveOp` class, it sets the value of each pixel in the destination image by averaging the corresponding pixel in the source image with the eight pixels that surround it. The convolution kernel in this example could be represented by the following matrix, with elements specified to four significant figures:

$$\mathbf{K} = \begin{bmatrix} 0.1111 & 0.1111 & 0.1111 \\ 0.1111 & 0.1111 & 0.1111 \\ 0.1111 & 0.1111 & 0.1111 \end{bmatrix}$$

When an image is convolved, the value of each pixel in the destination image is calculated by using the kernel as a set of weights to average the pixel's value with the values of surrounding pixels. This operation is performed on each channel of the image.

The following formula shows how the weights in the kernel are associated with the pixels in the source image when the convolution is performed. Each value in the kernel is tied to a spatial position in the image.

$$\mathbf{K} = \begin{bmatrix} i-1, j-1 & i-1, j & i-1, j+1 \\ i, j-1 & i, j & i, j+1 \\ i+1, j-1 & i+1, j & i+1, j+1 \end{bmatrix}$$

The value of a destination pixel is the sum of the products of the weights in the kernel multiplied by the value of the corresponding source pixel. For many simple operations, the kernel is a matrix that is square and symmetric, and the sum of its weights adds up to one.[6]

The convolution kernel in this example is relatively simple. It weights each pixel from the source image equally. By choosing a kernel that weights the source image at a higher or lower level, a program can increase or decrease the intensity of the destination image. The `Kernel` object, which is set in the `ConvolveOp` constructor, determines the type of filtering that is performed. By setting other values, you can perform other types of convolutions, including blurring (such as Gaussian blur, radial blur, and motion blur), sharpening, and smoothing operations.

5.2 Using Offscreen Buffers

`BufferedImage` is derived from `java.awt.Image`. This class supports fine-grain control over an image by allowing you to specify the image's `ColorModel`, image data, and data layout (`Tile` and `Channel)`. You also can supply storage space for the image data or access the existing storage data, allowing you to manipulate the contents of an image directly.

An image's `ColorModel` specifies the color space of the data in the `Tile` and how the data is mapped to color and alpha components. A `Tile` is comprised of an array of `Channel` objects. A `Channel` is a collection of data and data layout parameters for one or more bands of an image. With the information encapsulated by the `Tile` and `ColorSpace` objects, you can directly access and manipulate the pixels in an image. If you don't want to manipulate pixels directly, you can use the `getData` and `putData`, methods defined by `Tile`.

The `Channel`, `ColorModel`, and `ColorSpace` classes are important whenever you need to know about pixel layouts; for example, when implementing new `Composite` objects or using `BufferedImage` objects.

6. If the sum of the weights in the matrix is one, the intensity of the destination image is unchanged from the source.

5.2.1 Color Models

Images are two dimensional arrays of pixel values, not `Color` objects. To determine the color value of a particular pixel, you need to know how color information is encoded in each pixel. The `ColorModel` associated with an image encapsulates the data and methods necessary for translating a pixel value to and from its constituent color components.

The Java 2D API provides three types of color models:

- `An IndexColorModel` contains a lookup table that maps an index to a color. It can be associated with a `Tile` that has either one or two `Channel` objects (the second channel is an alpha channel).
- `A ComponentColorModel` is associated with a `Tile` that has the same number of `Channel` objects as color and alpha components in the `ComponentColorModel`. The placement of color component names in the color space of the color model determines the mapping between color components and channels in the channel array. For example, a color model with an RGB color space would map red to the channel at index 0, green to the channel at index 1, and blue to the channel at index 2. If there is an alpha channel, it would be the last channel in the channel array, the channel at index 3.
- `A PackedColorModel` is associated with a `Tile` that has one `DiscreteChannel` object. The `DirectColorModel` in JDK1.1 is a `PackedColorModel`. The packing information that describes how color and alpha components are extracted from the channel is stored in the `PackedColorModel`.

5.2.2 Tiles

If a `Tile` is not embedded in a `BufferedImage`, it can have many more channels than color components. This is useful for image processing applications. For example, a Landsat satellite image would have seven channels of data in a `Tile`, corresponding to the different sensors onboard the satellite. To display the image, you would typically create a subtile of the `Tile` to include only channels 3, 1, and 0, and associate it with a RGB color model so that channel 3 maps to red, channel 1 maps to green, and channel 0 maps to blue. However, you could use a different combination of channels because the channel data is false color; the color doesn't necessarily correspond to a color that a person would see when viewing the area.

5.2.3 Channels

The `Channel` class describes exactly how pixels are encoded and how they are laid out in memory. It can handle a wide variety of data layouts such as a 5/6/5 layout in a RGB image, a band-sequential layout where the data in each channel of the image is contiguous, and the layout supported by the current `java.awt.Image` class, where 4 bytes of alpha, red, green and blue are packed into an integer.

There are two types of `Channel` classes: `DiscreteChannel` and `PackedChannel`. `DiscreteChannel` objects have one channel element per data array element. For example, each channel element in a `ByteDiscreteChannel` would fit in one byte. A `PackedChannel` can have multiple channel elements in a data array element. For example, a binary image might use a `BytePackedChannel` with 8 channel elements packed into a byte.

In a `BufferedImage`, the number and type of `Channel` objects in the `Tile` must match the number of color and alpha components and type of `ColorModel`.

5.2.4 Using a BufferedImage as an Offscreen Cache

Preparing a graphic element offscreen and then copying to the screen can be useful, particularly if the graphic is complex or is used repeatedly. For example, if you want to display a complicated shape several times, you could draw it once into an offscreen buffer and then copy it to different locations in the window. By drawing the graphic once and copying it, you can display the graphics more quickly. Using offscreen buffers can also improve performance and reduce flickering in animation effects.

The `java.awt` package facilitates the use of offscreen buffers by letting you draw to an `Image` object the same way that you draw to a window. All of the Java 2D API rendering features can be used when drawing to an offscreen images. To copy an offscreen drawing to the screen, you simply call the `drawImage` method.[7]

The new `BufferedImage` class allows you to directly manipulate the pixels in an image. It also provides more flexibility in the data layout and in the color model associated with the image. `BufferedImage` can be used to create an image that can be efficiently blitted to a graphics device.

Offscreen buffers are often used for animation. For example, you could use an offscreen buffer to draw an object once and then move it around in a window. Simi-

7. Actually, you can draw it to another offscreen buffer just as easily. Some applications use two (or more) offscreen buffers to compose a complete drawing before copying it to a window.

larly, you could use an offscreen buffer to provide feedback as a user moves a graphic using the mouse. Instead of redrawing the graphic at every mouse location, you could draw the graphic once to an offscreen buffer, and then copy it to the mouse location as the user drags the mouse.[8]

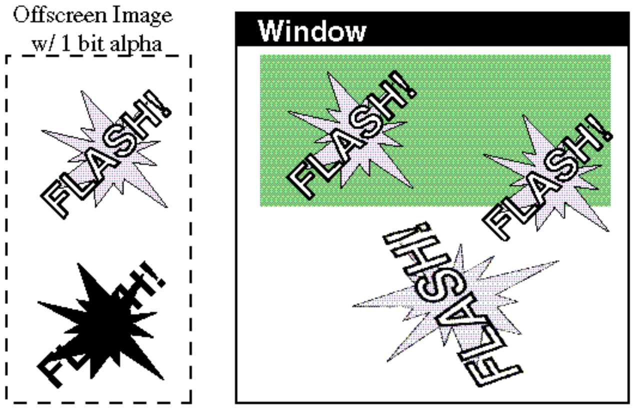

Figure 5-4 Using an offscreen buffer

Figure 5-4 demonstrates how a program can draw to an offscreen image and then copy that image into a window multiple times. The last time the image is copied, it is transformed. Note that transforming the image instead of redrawing the graphic with the transformation might produce unsatisfactory results.

A `BufferedImage` object can contain an alpha channel, just like any other image. In situations like the one illustrated in Figure 5-4, a 1-bit deep alpha channel is sufficient to distinguish the painted areas from the unpainted areas. The mask allows the image to blend with graphics that have already been painted (in this case, a green rectangle). In other situations, you might want a deeper alpha channel so you can manipulate the level of transparency in the image. You can control the alpha characteristics of a `BufferedImage` object by selecting an alpha channel of the appropriate depth, manipulating the data in the alpha channel, and changing the composite mode in the `Graphics2D` object used to draw the `BufferedImage`.

Often, it is important to create an instance of `BufferedImage` whose color space, depth, and pixel layout exactly match the window into which you are drawing. This allows `drawImage` to do its job quickly. `GraphicsConfiguration` provides

8. It is up to the programmer to "erase" the previous version of the image before making a new copy at a new location. This can be done by redrawing the background or copying the background from another offscreen buffer.

convenience methods that automatically create buffered images in the right format. You can also query the graphics configuration associated with the graphics device on which the window resides to get the information you need to construct a compatible `BufferedImage` object. (See the section “GraphicsConfiguration” on page 39).

The same mechanism can be extended to a variety of output devices, including printers. For example, you could create a `BufferedImage` that is compatible with a specific printer, resulting in an image whose color space, depth and pixel layout allow it to be drawn efficiently to that printer.

6.0 Graphics Devices

All graphics devices, such as a monitors and printers, are represented by a `GraphicsDevice` object that encapsulates a device’s capabilities and attributes. The Java 2D API allows you to query the attributes of the environment in which your application is running through the `GraphicsEnvironment` class and these `GraphicsDevice` objects and their associated `GraphicsConfiguration` objects.

6.1 Graphics Environment

You can access information about the overall operating environment through the `GraphicsEnvironment` class, which:

- `P`rovides a list of `GraphicsDevice` objects that represents all attached output devices
- Encapsulates the text capabilities of a system

You can examine the list of graphics devices and query the associated graphics configurations to determine the capabilities and properties of attached devices. Normally, you don’t need to access this information.

`GraphicsEnvironment` also allows you to search for fonts based on properties such as the font’s name, family, and style. You can also use it to list all available fonts. This might be useful, particularly if you are implementing a font panel or font menu. Font searching and enumeration capabilities are based on the `FontDescriptor` class, which is discussed in “Text Handling” on page 20.

6.2 GraphicsDevice

The Java 2D API uses the `GraphicsDevice` class to represent an actual output device. Typically, you do not need to access this class. `GraphicsDevice` objects are bound to actual output devices, such as printers or windows.

Each `GraphicsDevice` has a list of `GraphicsConfiguration` objects associated with it. In the X-windows environment, different windows on the same monitor can have different pixel layouts—a single monitor can provide multiple visuals with different pixel configurations. For example, one might be 8-bit pseudo-color and another might be 24-bit direct color. When you list the `GraphicsConfiguration` objects for a graphics device, there is one for each supported visual. In the MacOS and Windows environments, every pixel in every window on a monitor has the same configuration.

You can get a reference to a `GraphicsDevice` object from:

- `GraphicsEnvironment`, which maintains a list of `GraphicsDevice` objects that corresponds to every device that the Java 2D API knows about (one for every monitor or printer available on the system).
- `GraphicsConfiguration`, which defines a method that returns the `GraphicsDevice` associated with the configuration. You can get the `GraphicsConfiguration` associated with a `Graphics2D` object by calling `getDeviceConfiguration`.

6.3 GraphicsConfiguration

To get a list of the `GraphicsConfiguration` objects associated with a particular `GraphicsDevice`, you can call `GraphicsDevice.getConfigurations`. You can query a `GraphicsConfiguration` to determine the capabilities of a `GraphicsDevice` and create `BufferedImage` objects that are optimized for use with that configuration of the device.

You can also retrieve the `ColorModel` object and the default and normalizing `Transforms` for the configuration. The `ColorModel` maintains a `ColorSpace` object that characterizes the color output capabilities of the device and is used to match colors during the rendering process; see “Color Classes” on page 27. The `Transform` objects define how to transform from two convenient user coordinate spaces into the device coordinate space.

Figure 6-1 illustrates how a `GraphicsDevice` object is related to other key 2D objects.

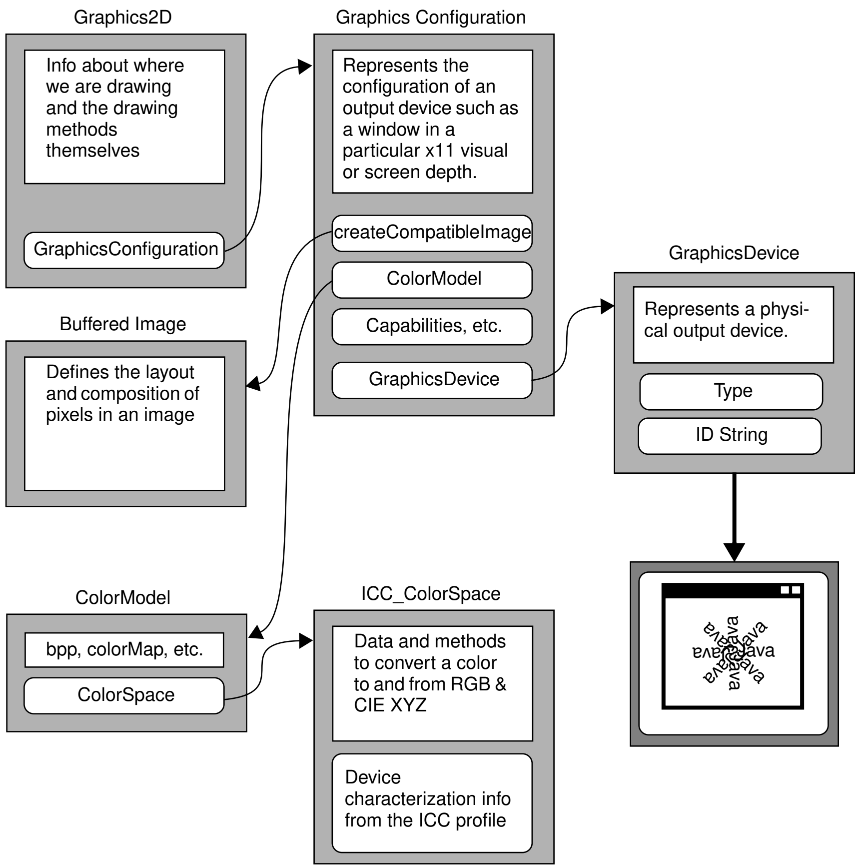

Figure 6-1 Graphics configuration relationships