Java a Tope:

Java2D

Cómo tratar con Java
figuras, imágenes y texto en dos dimensiones

Sergio Gálvez Rojas
Manuel Alcaide García
Miguel Ángel Mora Mata

UNIVERSIDAD DE MÁLAGA

JAVA A TOPE: JAVA2D (CÓMO TRATAR CON JAVA FIGURAS, IMÁGENES Y TEXTO EN DOS DIMENSIONES). EDICIÓN ELECTRÓNICA

AUTORES: SERGIO GÁLVEZ ROJAS
MANUEL ALCAIDE GARCIA
MIGUEL ÁNGEL MORA MATA

ILUSTRACIÓN DE PORTADA: HTTP://WWW.INTERPEQUES2.COM/

Depósito Legal: MA-0722-2007
ISBN: 978-84-690-5677-6

Java a tope:

Java2D

Cómo tratar con Java figuras, imágenes y texto en dos dimensiones

Sergio Gálvez Rojas
Doctor Ingeniero en Informática

Manuel Alcaide García
Ingeniero Técnico en Informática de Sistemas

Miguel Ángel Mora Mata
Ingeniero Técnico Superior en Informática

Dpto. de Lenguajes y Ciencias de la Computación
E.T.S. de Ingeniería Informática
Universidad de Málaga

Índice

Prólogo. v

***Capítulo 1*: Introducción**. 1
1.1 Descripción del capítulo. 1
1.2 Renderizado con Graphics2D. 1
1.2.1 Atributos de pincel. 3
1.2.2 Atributos de relleno. 4
1.2.3 La porción a dibujar. 4
1.2.4 Transformaciones. 4
1.2.5 Métodos de composición. 5
1.3 ¿Qué puede dibujarse?. 6
1.3.1 Figuras geométricas. 6
1.3.1.1 Conceptos. 7
1.3.2 Fuentes y diseño de texto. 8
1.3.3 Tratamiento de imágenes. 10
1.4 Tratamiento del color. 11

***Capítulo 2*: Renderizado de imágenes con Graphics2D**. 15
2.1 Introducción. 15
2.1.1 Ejemplo preliminar. 15
2.2 El contexto. 17
2.3 El sistema de coordenadas de Java2D. 18
2.3.1 Coordenadas de usuario. 18
2.3.2 Coordenadas de dispositivo. 18
2.4 Un paso adelante: el canal alfa. 20
2.5 Modificación del contexto de Graphics2D. 22
2.5.1 Preferencias, pinceles y colores. 22
2.5.1.1 Preferencias. 23
2.5.1.2 Especificar el estilo de línea. 24
2.5.1.3 Especificación de los atributos de relleno. 28
2.5.1.3.1 Gradiente de color 28
2.5.1.3.2 Relleno con texturas. 29
2.5.2 Establecimiento del *clipping path*. 31
2.5.3 Transformaciones de objetos. 32
2.5.4 Composición de objetos gráficos. 34

Capítulo 3: **Figuras geométricas en Java2D. 39**
3.1 Introducción. 39
3.2 Figuras básicas.. 39
3.2.1 Line2D.. 39
3.2.2 Rectangle2D.. 40
3.2.3 RoundRectangle2D. 41
3.2.4 Ellipse2D.. 42
3.2.5 Arc2D. 42
3.2.6 QuadCurve2D 43
3.2.7 CubicCurve2D 45
3.2.8 Point2D. 46
3.2.9 Resumen de clases relacionadas con figuras geométricas. 47
3.3 Un primer ejemplo compuesto: bañera. 48
3.4 La interfaz Shape.. 48
3.4.1 Otra manera de construir figuras: GeneralPath.. 50
3.4.2 Figuras geométricas a la medida: la bañera. 51
3.4.3 Áreas Geométricas Constructivas (CAG). 55
3.4.3.1 Sherezade en el crepúsculo. 57
3.5 JOGL (Java/OpenGL). 58

Capítulo 4: **Tratamiento de texto con Java2D. 61**
4.1 Introducción. 61
4.2 ¿Cómo encontrar las fuentes?.. 62
4.2.1 Crear y derivar fuentes.. 63
4.3 Dibujar texto. 63
4.3.1 Dibujar una única línea de texto.. 63
4.3.2 Dibujar un párrafo completo.. 66
4.3.3 Cómo dibujar texto sobre una imagen.. 70
4.3.4 Letra cursiva invertida mediante transformaciones.. 73
4.3.5 Rellenar un *clipping path* con distintos objetos.. 74
4.4 Manejo de la clase TextAttribute. 77

Capítulo 5: **Imágenes con Java2D.. . . . . 81**
5.1 Introducción. 81
5.2 Obtención de imágenes. 81
5.2.1 Cargar una imagen.. 81
5.2.2 Cargar una imagen como BufferedImage. 83
5.2.3 Crear imágenes. 85
5.3 Mostrar una imagen.. 87
5.4 La clase MediaTracker.. 88

5.5 Dibujar sobre imágenes. 89
5.5.1 *Double Buffering* (dibujo en memoria). 91
5.6 Procesamiento de imágenes. 93
5.6.1 Operaciones predefinidas. 93
5.6.1.1 ConvolveOp. 94
5.6.1.2 AffineTransformOp. 99
5.6.1.3 LookupOp. 100
5.6.1.4 RescaleOp. 107
5.6.1.5 ColorConvertOp. 112
5.6.2 Dentro de BufferedImage. 115

***Capítulo 6*: El color en Java2D. 117**
6.1 Introducción. 117
6.2 Cómo crear un color. 117
6.2.1 Un ejemplo con colores. 119
6.3 Espacios de Colores. 121
6.3.1 El espacio de colores CIEXYZ. 121
6.3.2 El espacio de colores sRGB 122
6.4 La clase ColorSpace. 122
6.5 Perfiles. 124
6.6 La verdad sobre el color. 125

Índice

Prólogo

El objetivo básico del texto que el lector tiene entre sus manos es realizar un estudio completo y didáctico sobre la tecnología Java2D que sirva como documentación para actuales y futuros programadores en esta tecnología. Se pretende ser lo más claro y conciso posible pero sin dejar de estudiar, aunque en algunos casos sea sólo someramente, todos y cada uno de los conceptos que abarca Java2D.

Como objetivo complementario este texto pretende ser, ante todo, práctico. Para ello se incluye una gran cantidad de ejemplos que muestran cada concepto explicado, teniendo en mente siempre su capacidad de ilustración y vistosidad de forma que la lectura resulte amena y distraída, pero sobre todo útil.

Merced a una elaborada estructuración del texto en apartados y subapartados, cada uno centrado en un concepto distinto, nuestros objetivos se completan con el deseo de satisfacer tanto a los lectores que quieran aprender a usar la tecnología Java2D en general como a aquéllos que sólo necesiten aclarar una duda puntual y sólo deseen consultar un apartado concreto.

Las posibilidades de Java2D son casi ilimitadas. Partiendo del control básico de figuras geoméricas simples y trozos de texto gestionado como imágenes el lector irá profundizando en las distintas opciones de configuración hasta poseer un control completo sobre los efectos visuales producidos. En concreto, la manipulación que puede ejercerse con Java2D sobre las imágenes en general es muy potente existe una gran cantidad de detalles que pueden controlarse con poco esfuerzo. Además de los efectos visuales producidos por la modificación de colores y pinceles, transparencias, tramas de relleno y demás, también resulta interesante el poder aplicar transformaciónes geométricas a las imágenes, con lo que éstas incluso pueden llegar a adoptar un curioso aspecto de tridimensionalidad.

Capítulo 1

Introducción

1.1 Descripción del capítulo

La API Java2D amplía muchas de las capacidades gráficas de la biblioteca AWT (*Abstract Window Toolkit* - Herramientas Abstractas de Ventanas), permitiendo la creación de mejores interfaces de usuario y de aplicaciones Java mucho más impactantes visualmente. El rango que abarcan todas estas mejoras es muy amplio, ya que comprende el renderizado[1], la definición de figuras geométricas, el uso de fuentes de letras, la manipulación de imágenes y el enriquecimiento en la definición del color. También permite la creación de bibliotecas personalizadas de gráficos avanzados o de efectos especiales de imagen e incluso puede ser usada para el desarrollo de animaciones u otras presentaciones multimedia al combinarla con otras APIs de Java, como puedan ser JMF (*Java Media Framework* - Entorno de Trabajo de Java para Medios Audiovisuales) o Java 3D.

En esta introducción estableceremos un marco general en el que quede bien definido y delimitado lo que esta API puede ofrecer en cuanto a posibilidades y prestaciones. Para ello describiremos todos sus aspectos y características, así como sus aplicaciones, empezando por el renderizado 2D.

1.2 Renderizado con Graphics2D

java.awt.Graphics2D es una clase que extiende a java.awt.Graphics proporcionándole un control más potente sobre la presentación de texto, imágenes o figuras geométricas. Un objeto Graphics (que es una clase abstracta) representa el lienzo abstracto y el contexto en el que puede dibujarse cualquier cosa; este lienzo puede estar enlazado con un área física de un monitor, o representar una imagen en memoria que sólo se desea manipular y no tiene representación directa durante este proceso.

El proceso de renderizado de Java2D está controlado por un objeto de esta

[1] Debido a la lentitud con que la Real Academia de la Lengua Española reacciona ante la necesidad de incorporar nuevos vocablos o ampliar acepciones en los ya existentes, los autores nos hemos visto obligados a admitir el verbo renderizar con el siguiente significado: «Generar una imagen a partir de un modelo»

clase Graphics2D y sus atributos o características contextuales que son, a su vez, objetos (un atributo contextual puede ser, por ejemplo, el tamaño del pincel con que se dibuja una línea recta); el conjunto de estos atributos es lo que conforma el contexto del objeto Graphics2D. Así por ejemplo, para dibujar una imagen sobre un Graphics2D es necesario modificar su contexto adecuadamente y llamar después a alguno de los métodos de renderizado.

Como caso general, el proceso de renderizado de un objeto java.awt.Shape se puede dividir en cuatro pasos. Para simplificar en estos preliminares, basta decir que Shape es una interfaz que sustenta la definición de clases con formas geométricas diversas. Los cuatro pasos comentados son:

1. En el momento de pintar al objeto Shape, se utiliza un objeto de la clase Stroke (que establece el pincel en el contexto del Graphics2D) para generar un nuevo Shape que representa el contorno del objeto a pintar. Stroke hace las veces de pincel o lápiz de contorno.
2. Las coordenadas en que dibujar al nuevo objeto Shape se transforman desde el *user space* al *device space* de acuerdo con el atributo de transformación del contexto del Graphics2D. El *user space* (coordenadas de usuario) es un sistema de coordenadas absoluto e independiente del sistema físico donde plasmar el dibujo, mientras que el *device space* (coordenadas de dispositivo) es un sistema de coordenadas que sí depende del sistema final sobre el que dibujar; por ejemplo, si se usa un sistema multipantalla se obtendrá un renderizado distinto al monopantalla. En los casos más usuales, ambos sistemas de coordenadas coinciden.
3. El contorno de la figura de tipo Shape se obtiene como una secuencia de segmentos que se dibujan en secuencia.
4. El contorno resultante se rellena usando los atributos de tipo Paint y Composite del contexto del Graphics2D.

Entre las clases e interfaces más importantes, incluyendo las utilizadas por el contexto, podemos citar las siguientes (contenidas en los paquetes java.awt y java.awt.geom):

- Interfaces:
    1. Composite: define métodos para realizar composiciones de dibujos, esto es, definir el comportamiento del sistema a la hora de dibujar por ejemplo una figura geométrica sobre un área de gráficos que ya contenga algo dibujado. Entre otras cosas permite definir transparencias.
    2. Paint: extiende a Transparency y define la forma en que se construyen las tramas de color durante las operaciones draw() y fill().
    3. Stroke: permite a un Graphics2D generar un objeto Shape que

representa el contorno de la figura que se quiere dibujar.

- Clases:
    1. AffineTransform: representa una transformación en 2 dimensiones: traslación, inversión, rotación, etc.
    2. AlphaComposite: implementa a Composite. Gestiona la composición alfa (transparencias) básica para las figuras, textos e imágenes.
    3. BasicStroke: implementa a Stroke. Define el estilo del pincel que se aplica al dibujar el contorno de una figura.
    4. Color: implementa a Paint. Define por ejemplo el color del relleno o del contorno al dibujar una figura.
    5. GradientPaint: implementa a Paint. Define un patrón de relleno en forma de gradiente lineal de color. En otras palabras, al rellenar una figura, no se utiliza un solo color, sino un degradado de color que parte de un color inicial hasta otro final pasando por toda la gama intermedia.
    6. Graphics2D: extiende a Graphics. Es la clase fundamental para el renderizado 2D.
    7. TexturePaint: define un patrón de relleno complejo al rellenar una figura. Este patrón -también llamado textura- está almacenado en un objeto de tipo BufferedImage.

Como ya se ha comentado anteriormente, para dibujar un objeto gráfico, antes debe establecerse el contexto en que se realizará el renderizado y después debe llamarse a uno de los métodos de renderizado de Graphics2D. Pues bien, los atributos de estado que conforman el contexto y que pueden modificarse son los siguientes:

1. Variar la anchura del pincel.
2. Definir colores o patrones de relleno para las figuras.
3. Delimitar un área concreta a renderizar (*clipping path*).
4. Trasladar, rotar, reducir o ampliar objetos cuando son renderizados.
5. Especificar la forma en que se componen las figuras superpuestas.

1.2.1 Atributos de pincel

Los atributos del pincel (que pertenecen a la interfaz Stroke) definen las características del trazo dibujado por el lápiz o pincel en la imagen. Con BasicStroke pueden definirse características tales como el ancho de línea, la forma en que acaba un trazo o el estilo con que se unen varios segmentos en un dibujo.

Los métodos de renderizado de Graphics2D que usa como atributo contextual un objeto de tipo Stroke son draw(), drawArc(), drawLine(), drawOval(), drawPolygon(), drawPolyline(), drawRect() y drawRoundRect(). Cuando se invoca a uno de estos métodos se renderiza el contorno correspondiente (según la

función llamada) y, acto seguido, el atributo Stroke define las características del trazo mientras que el atributo Paint define el color o el patrón de relleno de la marca dibujada por el pincel.

1.2.2 Atributos de relleno

Los atributos de relleno del contexto de un Graphics2D están representados por un objeto Paint. Cuando una figura o un *glyph* (un *glyph* es, tipográficamente hablando, el rasgo de un signo o una letra según una fuente de texto; en otras palabras, la forma de una letra concreta a partir de su tipo, tamaño y peculiaridades: negrita, cursiva, fileteada, etc. La impresión de una secuencia de rasgos tipográficos produce una cadena de texto, y cada uno de estos rasgos, como se verá más adelante, son tratados como un objeto Shape más) el objeto Paint se aplica a todos los píxeles que quedan dentro de la figura en sí, y que representa el contorno del objeto dibujado por el pincel. Al rellenar el interior de una figura, el objeto Paint se encarga de gestionar todos los píxeles de la figura, a excepción de los del contorno.

Rellenar una figura con un único color opaco se puede hacer fácilmente estableciendo dicho color en el contexto del objeto Graphics2D con el método setColor(). La clase Color es la implementación mas básica de la interfaz Paint. Para hacer rellenos más complejos, pueden usarse las clases GradientPaint y TexturePaint, que también heredan de Paint en Java2D.

1.2.3 La porción a dibujar

La porción a dibujar (en inglés *clipping path*) es el trozo de una figura o imagen que debe ser renderizada; este trozo puede tener cualquier forma, y no tiene porqué ser necesariamente un rectángulo sino que su forma puede ser establecida por programa. Cuando se define una porción a dibujar (*clipping path*) en el propio contexto de un Graphics2D, sólo las zonas delimitadas por dicha porción (las que caen en su interior) son las que serán renderizadas. Para cambiar esta porción del dibujo, puede usarse setClip() con objeto de crear un área nueva o también es posible invocar al método clip(Shape) para modificar uno ya existente mediante la intersección con otro que se pasa como parámetro.

1.2.4 Transformaciones

El contexto de un objeto Graphics2D contiene una transformación que se usa al reubicar objetos desde el espacio de usuario (*user space*) al espacio del dispositivo (*device space*) durante el renderizado. Graphics2D proporciona varios

métodos que permiten modificar la transformación por defecto en su contexto. Lo más sencillo es llamar a uno de los métodos de transformación de Graphics2D como rotate(), scale(), shear() o translate(): sólo es necesario especificar para cada una de ellos las características de la transformación requerida y Graphics2D automáticamente hará los cambios pertinentes en el momento de hacer el dibujo.

Es más, también es posible concatenar transformaciones mediante el uso de un objeto de la clase AffineTransform, el cual puede realizar transformaciones en secuencia, tales como una rotación seguida de un cambio de escala. Cuando una transformación se concatena con otra existente, la última que se especificó es la primera en ser aplicada. Graphics2D contiene también un método setTransform(), que sobrescribe las transformaciones a aplicar a los objetos que se vayan a dibujar pero que, ojo, no es posible usar para realizar concatenación de transformaciones.

1.2.5 Métodos de composición

Cuando dos objetos se superponen en un mismo dibujo (ya sean figuras, rasgos tipográficos o imágenes) es necesario determinar qué colores renderizar en los píxeles superpuestos: este proceso se denomina composición. Las interfaces básicas de composición de Java2D son Composite and CompositeContext. Por ejemplo, para especificar el estilo de composición que debe usarse puede establecerse un objeto de la clase AlphaComposite en el contexto de un Graphics2D llamando a su método setComposite(). Las instancias de la clase AlphaComposite establecen una regla de composición que describe la manera de mezclar un nuevo color con otro ya existente definiendo, por ejemplo, transparencias.

Con este objetivo, para manejar transparencias, se dispone de un valor alfa adicional al crear un objeto AlphaComposite. Este valor alfa (asociado al llamado «canal alfa» de una imagen), incrementa o decrementa el canal de transparencia del objeto según el valor que tome: si el valor de alfa es 1,0 el color será opaco, mientras que el color será totalmente transparente si vale 0,0. Por suspuesto, los valores intermedios especifican transparencias intermedias del color. Así pues, el canal alfa se usa en conjunto con un objeto de la clase Composite en el contexto de un Graphics2D para mezclar la imagen con los dibujos ya existentes. Un ejemplo de manejo de transparencias aparece en la figura 1.1.

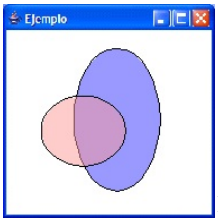


Figura 1.1. Ejemplo de transparencia

1.3 ¿Qué puede dibujarse?

Anteriormente hemos hecho mención a la clase Shape como ejemplo de objeto susceptible de ser dibujado en un Graphics2D. Sin embargo no es el único y en los siguientes apartados se introducen las distintas clases cuyas instancias pueden renderizarse siguiendo un proceso parecido al del epígrafe 1.2.

1.3.1 Figuras geométricas

Java2D provee varias clases, pertenecientes a los paquetes java.awt y java.awt.geom, que definen figuras geométricas simples, tales como puntos, líneas, curvas y rectángulos. Usando las clases geométricas, es posible definir y manipular virtualmente cualquier objeto bidimensional de una manera sencilla. Las clases e interfaces a utilizar para este fin son las siguientes:

- Interfaces:
  1. PathIterator: define métodos para iterar sobre los distintos segmentos o subtrazos que conforman el contorno de una figura o rasgo tipográfico.
  2. Shape: proporciona un conjunto básico de métodos para describir y generar contornos de objetos geométricos. Es implementada por GeneralPath y una multitud de clases geométricas.
- Clases:
  1. Area: representa una área geométrica (con la forma que sea) que soporta operaciones de intersección, unión, etc. para obtener nuevas áreas con formas diferentes.
  2. FlatteningPathIterator: se ha comentado que PathIterator proporciona los subtrazos de un contorno; estos subtrazos pueden ser segmentos o curvas de escasa complejidad. La clase FlatteningPathIterator es igual a PathIterator pero siempre devuelve segmentos, de ahí lo de *flattening* (aplanar en español).
  3. GeneralPath: implementa a Shape. Representa el trazo de un objeto geométrico construido a partir de líneas y curvas cuadráticas y cúbicas (curvas que pueden expresarse matemáticamente con escasa complejidad).
  4. RectángularShape: proporciona la base de un gran número de figuras (Shape) que se dibujan enmarcadas en un rectángulo.
  5. Figuras en sí:
     - Arc2D: representa un arco.
     - CubicCurve2D: representa un segmento curvo.
     - Ellipse2D: representa una elipse, y extiende a la clase RectangularShape
     - Line2D: representa un segmento de línea, e implementa a Shape.
     - Point2D: es un punto que representa un localización en un sistema

de coordenadas
- QuadCurve2D: representa un segmento de curva cuadrática, e implementa a Shape
- Rectangle2D: representa un rectángulo, y extiende a RectangularShape
- RoundRectangle2D: representa un rectángulo con los vértices redondeados, y extiende también a RectangularShape.

Las medidas de todas estas figuras pueden especificarse tanto en formato **double** como **float**; para ello sólo es necesario añadir la extensión .Double o .Float al nombre del constructor de la figura que se quiere crear, como por ejemplo Arc2D.Float(...).

1.3.1.1 Conceptos

Una figura geométrica es, entre otras cosas, una instancia de cualquier clase que implemente la interfaz Shape, como Line2D o Rectangle.Float. El contorno de una figura Shape se denomina *path* en la jerga Java2D. Cuando se dibujar una figura, el estilo del pincel definido por un objeto Stroke en el contexto de Graphics2D se aplica a cada uno de los trazos que componen el contorno. A su vez, el contorno de una figura también puede usarse como un *clipping path* (porción que delimita el área en la que se verá un dibujo o imagen posterior). La porción a dibujar determina los píxeles que se deben renderizar, que son sólo los que entran dentro del área definida por él, es decir, el *clipping path* define una zona concreta, un trozo de imagen que es la que se renderizará.

Con respecto a la clase Area, la definición Constructiva de Áreas Geométricas o CAG (*Constructive Area Geometry*) es el proceso de crear nuevos objetos geométricos mediante la ejecución de operaciones de conjuntos entre objetos geométricos ya existentes. Como ya se ha comentado, la clase Area es la encargada de esto en Java2D, siendo posible construir un objeto Area a partir de cualquier objeto Shape. La clase Area soporta las operaciones de conjuntos de unión, intersección, diferencia y OR exclusivo o XOR.

Para combinar objetos de tipo Area y crear otras nuevas, lo único que hay que hacer es crear las áreas de partida a combinar (usando objetos de tipo Shape en el constructor de Area) y, acto seguido, invocar al operador lógico adecuado, como add() (union) o intersect() (intersección).

Un recuadro delimitador (*bounding box* en inglés) es un rectángulo que engloba totalmente la geometría de una figura. Estos recuadros se utilizan para determinar si un objeto ha sido seleccionado por el usuario cuando hace clic sobre un panel: si las coordenadas del clic caen dentro del recuadro delimitador de una figura **x** entonces se asume que el usuario quiere seleccionar dicho objeto **x** con algún

propósito (posiblemente de edición). La interfaz Shape define dos métodos para obtener el recuadro delimitador de una figura: getBounds() y getBounds2D(). El segundo método es más eficiente, ya que retorna el recuadro delimitador en forma de objeto Rectangle2D, en contraposición a getBounds() que retorna un Rectangle que posee menor precisión.

Finalmente, para crear una figura (Shape) personalizada lo que hay que hacer es crear una clase nueva que la defina; para ello deben implementarse los métodos de la interfaz Shape: contains(), getBounds(), getBounds2D(), getPathIterator() e intersects(), cuyos propósitos pueden intuirse por el propio nombre de los métodos.

1.3.2 Fuentes y diseño de texto

Antes de entrar de lleno en cómo dibujar texto en un objeto Graphics2D es necesario conocer algunos conceptos básicos relativos a la fuentes tipográficas y al diseño que éstas adoptan para ser impresas.

Con respecto a las fuentes, la clase Font es la que soporta la especificación de los tipos de letra y permite el uso de complicados rasgos tipográficos. Un objeto de la clase Font representa una instancia de un tipo de fuente de entre todas las que hay instaladas en el ordenador. Una fuente tiene tres nombres:

- Su nombre lógico, el cual es un nombre que esta mapeado en una de las fuentes específicas de la plataforma y se usa en versiones antiguas del JDK,
- Su nombre de familia, que es el nombre de la familia a la que pertenece la fuente, con la que comparte rasgos tipográficos comunes, tales como la familia Times New Roman.
- El nombre real de la fuente, el cual se refiere a una fuente real instalada en el sistema.

Es posible acceder a toda la información acerca de un objeto Font a través de su método getAttributes().

Por otro lado, un objeto de la clase LineMetrics es el encargado de contener toda la información acerca de las medidas de una fuente. Esta información se utiliza para posicionar correctamente los caracteres a lo largo de la línea.

Con respecto a los conceptos de diseño, podemos decir que antes de que un trozo de texto se muestre, se le debe dar forma y posicionar en su sitio cada uno de los rasgos tipográficos utilizando el espacio interlínea y entre caracteres apropiado. Esto es lo que se llama el diseño del texto, que engloba los siguientes pasos:

1. Dar forma al texto mediante los rasgos tipográficos y los espaciados apropiados. Un rasgo tipográfico es la representación visual de uno o más caracteres según una fuente concreta. La forma, el tamaño y la

posición del rasgo dependen de su contexto. Para representar un mismo carácter se pueden usar muchos rasgos distintos e incluso combinaciones de ellos, dependiendo de la fuente y del estilo. En algunos contextos, dos rasgos pueden cambiar su forma y fusionarse en un único rasgo simple, lo que forma una ligadura: esto sucedía antiguamente, por ejemplo, en las máquinas de escribir en las que para representar una vocal acentuada eran necesarios dos rasgos: el acento por sí sólo y la vocal después.

2. Ordenar el texto. Hay dos tipos de orden, el orden lógico, que es aquél en el que las secuencias de caracteres se almacenan en memoria y el orden visual, que se corresponde con el orden en que los rasgos tipográficos se muestran para ser leídos por una persona. A este orden también se le llama orden de escritura (*script order* en inglés). Estas dos ordenaciones no tienen porqué coincidir. Por ejemplo, el orden de escritura de la fuente **Arial** es de izquierda a derecha, mientras que el de la **Arabic** es de derecha a izquierda (ya que el árabe se lee de derecha a izquierda). El orden visual debe mantenerse con sentido incluso cuando se mezclan idiomas en un mismo trozo de texto dando lugar a un texto multidireccional: cuando en un texto se mezclan textos que se leen en un sentido con textos que se leen en otro toma especial relevancia lo que se denomina la dirección base. La dirección base es el sentido de escritura del sistema de escritura predominante (como por ejemplo un texto en español con caracteres árabes).

3. Medir y posicionar el texto. A menos que se trabaje con una fuente de ancho constante, cada caracteres posee su propio ancho. Esto significa que todo el posicionamiento y medida del texto tiene que valorarse de manera que se mire qué caracteres se han usado, no cuántos. Para adecuar la posición en que se debe renderizar cada rasgo o letra, es necesario guardar información relativa a cada carácter individual, la fuente utilizada y el estilo en uso. Por suerte, la clase TextLayout se encarga automáticamente de todo ello.

Si solamente se desea mostrar un bloque de texto en una ventana o incluso editarlo, puede emplearse un objeto de tipo JTextComponent, que realizará el diseño del texto por nosotros. Si lo que se quiere es dibujar una cadena de caracteres, se debe invocar a al método drawString() de la clase Graphics2D y dejar que Java2D haga el diseño él solo. También puede usarse drawString() para renderizar cadenas personalizadas o que contengan texto bidireccional.

También es posible implementar nuestro propio gestor de rasgos tipográficos sobrescribiendo los métodos de la clase TextLayout que se usa como punto de partida. Finalmente, para obtener un control total sobre la manera en que el texto se forma y se posiciona, es posible personalizar nuestros propios rasgos tipográficos y

gestionarlos a través de la clase java.awt.font.GlyphVector y de la clase Font.

1.3.3 Tratamiento de imágenes

Las clases e interfaces para tratamiento de imágenes en Java2D proporcionan técnicas para manejar imágenes pixeladas (formadas por una matriz de puntos a cada uno de los cuales se denomina pixel) cuyos datos están almacenados en memoria. En general, Java2D permite acceder y trabajar con imágenes almacenadas según una amplia gama de formatos: PNG (*Portable Network Graphics*), JPEG (*Joint Photographic Experts Group*), GIF, (*Graphics Interchange Format*), etc. además de poder manipular los datos de una imagen a través de varios tipos de operaciones de filtro. El modelo de tratamiento de imágenes soporta imágenes almacenadas en memoria con resoluciones fijas, siendo posible obtener clones de una imagen a diferentes escalas.

La clase BufferedImage proporciona el tratamiento general de la imagen. Es posible crear un objeto BufferedImage directamente en memoria y luego usarlo para contener y manipular datos de una imagen obtenidos desde un fichero o una dirección URL (*Uniform Resource Locator*). Un objeto BufferedImage puede mostrarse utilizando como destino de visualización a un objeto Graphics2D, o puede renderizarse en algún otro destino usando apropiadamente el contexto de Graphics2D. La clase BufferedImage contiene básicamente dos objetos, uno de clase Raster y otro de clase ColorModel.

La clase Raster proporciona el manejo primitivo de los datos de la imagen: Representa las dimensiones rectangulares de la imagen, mantiene los datos de la imagen en memoria y nos dota de un mecanismo para crear varias subimágenes a partir de un sólo *buffer* de datos, además de proporcionar métodos de acceso a píxeles específicos de la imagen. Un objeto Raster contiene, a su vez, a otros dos objetos: un DataBuffer y un SampleModel. La clase DataBuffer contiene datos de píxeles en memoria, mientras que la clase SampleModel interpreta los datos del *buffer* y los devuelve o bien como píxeles individuales, o bien como trozos rectangulares de píxeles.

La clase ColorModel permite interpretar el color asociado a los píxeles proporcionados por el SampleModel de la imagen (SampleModel permite interpretar los datos asociados a cada pixel y convertirlos, por ejemplo, en un color concreto). Así, se denomina muestra a cada uno de los distintos datos que representan a un píxel en una imagen en el momento en que ésta se codifica computacionalmente. Por ejemplo, un píxel en un sistema de colores RGB está formado por tres muestras: una para el rojo, otra para el verde y otra para el azul.

El paquete java.awt.image proporciona otras muchas clases que definen, por ejemplo, operaciones de filtrado en un objeto BufferedImage o Raster. Cada

operación de procesado de imagen se encuentra inmersa en una clase que implementa la interfaz BufferedImageOp, la interfaz RasterOp o ambas a la vez, y posee un cierto grado de adaptación en el sentido de que puede afinarse su funcionamiento en base a parámetros.

Las operaciones soportadas son las siguientes:

1. Transformaciones personalizadas.
2. Cambios de escala.
3. Modificaciones independientes a cada banda de color. Una banda es el conjunto de todas las muestras de un mismo tipo en una imagen, como por ejemplo todas las muestras rojas o azules.
4. Conversión de colores
5. Modificaciones por convolución. La convolución es una operación que transforma un píxel en función del valor de los que lo rodean.

Si solamente se desea mostrar y manipular imágenes, únicamente es necesario comprender el funcionamiento de la clase BufferedImage y de las clases básicas de filtrado, mientras que si lo que se quiere es escribir filtros o acceder directamente a los datos de la imagen deben comprenderse en profundidad las clases asociadas con BufferedImage.

1.4 Tratamiento del color

En Java2D, las clases principales relacionadas con el tratamiento del color son ColorSpace, Color y ColorModel. ColorSpace representa un sistema cualquiera para poder medir los colores; por norma general se usan tres valores o componentes numéricos distintos. Un objeto de tipo Color es un color fijo, definido en términos de sus componentes en base a un ColorSpace particular. Por último, la clase ColorModel describe un modo particular de representación interna de las muestras que describen cada píxel: mediante 4 *bytes*, con un entero de 32 bits, etc. A continuación, vamos a aclarar un poco estas clases y los conceptos que representan.

La clase ColorModel se usa para interpretar las muestras de un píxel de una imagen y convertirlas en un color. Esto incluye mapear cada muestra de cada banda de una imagen en componentes de un sistema particular de colores. Esta clase puede gestionar el valor de un píxel de dos formas diferentes: mediante una muestra por cada banda, o mediante un único entero de 32 bits; en este último caso debe encargarse de desempaquetar los componentes codificados por dicho entero.

Para determinar el color de un píxel particular en una imagen, es necesario saber qué información de color esta codificada en cada píxel. El objeto ColorModel asociado con una imagen encapsula los métodos necesarios para traducir un píxel hacia las muestras de color que lo componen y viceversa. La API Java2D proporciona

varios modelos de color predefinidos:

1. IndexColorModel. Trabaja con el espacio RGB con cada muestra representada mediante un entero de 8 bits. Opcionalmente puede usarse un valor para el canal alfa.
2. ComponentColorModel. Parecido al anterior pero puede trabajar con cualquier ColorSpace de forma que en lugar de usar cuatro muestras usa un *array* de muestras acorde con el ColorSpace que sea. Esta clase puede usarse para representar la mayoría de los modelos de colores propios de cualquier dispositivo gráfico.
3. PackedColorModel. Es una clase base abstracta para modelos que representan cada píxel mediante un único valor entero de 32 bits. Estos bits deben trocearse para extraer la información asociada a cada muestra, por lo que cualquier clase que herede de PackedColorModel debe establecer los algoritmos que describen cómo se extraen las componentes del color y del canal alfa.
4. DirectColorModel. Hereda de PackedColorModel y trabaja con el espacio RGB, por lo que es similar a IndexColorModel pero sin diferenciar directamente entre cada componente o muestra.

Un objeto ColorSpace representa un sistema de cuantificación de colores usando normalmente tres valores numéricos distintos: por ejemplo RGB (*Red-Green-Blue* en inglés) es el sistema de colores más extendido, pero no el único. Un objeto ColorSpace identifica el sistema de colores de un objeto Color o, a través de un objeto ColorModel, de una imagen entera representada por un objeto Image, BufferedImage o GraphicsConfiguration (ésta última es una clase que describe las características de un dispositivo capaz de procesar gráficos, como un monitor o una impresora. Un mismo dispositivo físico puede tener asociadas distintas configuraciones, y por ende distintos objetos GraphicsConfiguration, utilizadas para representar por ejemplo a diferentes resoluciones).

La clase Color proporciona la descripción de un color concreto mediante un sistema de colores particular. Una instancia de Color contiene el valor de las componentes de color, así como una referencia a un objeto ColorSpace. Así, cuando se crea un objeto de tipo Color puede pasarse como parámetro del constructor, además de las componentes de color, un objeto ColorSpace, lo que permite que la clase Color puede manejar cualquier color en cualquier sistema de colores.

Color tiene unos métodos que soportan un sistema de colores estándar llamado sRGB (estándar RGB) que, a su vez, es el sistema de colores por defecto en la API Java2D; la figura 1.2 representa este espacio de colores. A pesar de ello también puede crearse un color en un sistema distinto, para lo que se utiliza el constructor anteriormente mencionado que toma como parámetros un objeto ColorSpace y un *array* de reales que representan las componentes o muestras adecuadas a ese espacio: el objeto ColorSpace identificará el sistema de colores.

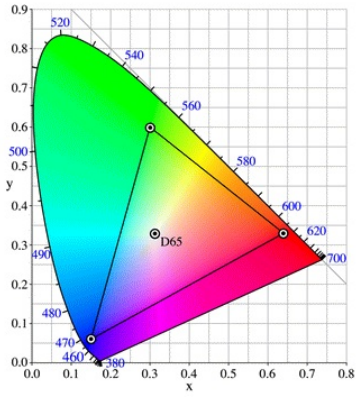


Figura 1.2. Gamut del espacio de colores sRGB en el que también se muestra la ubicación exacta de los componentes puros rojo, verde y azul, además del blanco. Se denomina gamut al subconjunto de los colores capaces de ser representados con precisión en un cierto dispositivo o en un espacio de colores.

Capítulo 2

Renderizado de imágenes con Graphics2D

2.1 Introducción

Java2D ofrece una amplia gama de posibilidades para tratamiento de imágenes y figuras geométricas en dos dimensiones. En este capítulo se van a tratar los conceptos relacionados con el renderizado de imágenes bidimensionales mediante la clase Graphics2D.

Como ya se ha comentado, el verbo renderizar es una adaptación al español del verbo inglés *render* y que define un proceso de cálculo más o menos complejo desarrollado por un ordenador destinado a producir una imagen o secuencia de imágenes. En otros contextos, se habla de renderizado para definir el proceso por el cual se pretende dibujar, en un lienzo de dos dimensiones, un entorno en tres dimensiones formado por estructuras poligonales tridimensionales, luces, texturas y materiales, simulando ambientes y estructuras físicas verosímiles. También se habla de renderización en el contexto de procesos 2D que requieren cálculos complejos como la edición de vídeo, la animación o el desarrollo de efectos visuales.

2.1.1 Ejemplo preliminar

Vamos a estudiar todas las posibles aplicaciones que se puedan crear mediante el uso de la API Java2D, empezando por un ejemplo muy simple como primera toma de contacto: el código siguiente lo único que hace es pintar el rectángulo de la figura 2.1:

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

public class Ventana extends JFrame {
   public Ventana() {
      super ("Prueba de Ventana");
      Container c = getContentPane();
      c.setLayout(new FlowLayout());
      this.setSize(250,250);
      this.setDefaultCloseOperation(EXIT_ON_CLOSE);
      this.setVisible(true);
```

Figura 2.1. Dibujo de un rectángulo simple

```
    }
    public static void main (String [] args) {
        Ventana v = new Ventana();
    }
    public void paint (Graphics g) {
            Rectangle2D r2 = new Rectangle2D.Float(75, 50, 100, 25);
            Graphics2D g2 = (Graphics2D)g;
            g2.draw(r2);
    }
}
```

En el ejemplo anterior se ha construido una ventana muy simple, creando en el main() una instancia de la clase Ventana para que se nos muestre la misma. Lo importante es el método que le sigue: paint(). Este método es el encargado de pintar en la ventana cualquier objeto de Graphics2D que queramos renderizar; por defecto, este método dibuja todos los componentes que se hayan añadido a la ventana, tales como botones, casillas de verificación, etc. En nuestro caso, y a modo de muestra, se ha creado un objeto Rectangle2D, que define un rectángulo según un sistema de coordenadas cartesiano. El método Float sirve para representar las coordenadas en coma flotante simple (también se puede usar el método Rectangle2D.Double). La creación de figuras geométricas se verá en profundidad en capítulos posteriores, pero por ahora lo que interesa ver es que se ha creado un rectángulo, se ha ubicado 50 píxeles hacia abajo y 75 píxeles hacia la derecha, y se le ha dado un tamaño de 100 píxeles de ancho y 25 de alto. Nótese, pues, como el origen de coordenadas (0, 0) se encuentra en la esquina superior izquierda, justo donde empieza la ventana incluído el título.

De esta manera, una de las sentencias más importantes del método paint() es:

```
Graphics2D g2 = (Graphics2D)g;
```

cuyo objetivo es permitir el uso de funcionalidades ampliadas. La API de Java2D extiende la funcionalidad básica de gestión de gráficos del paquete AWT (*Abstract Window Toolkit*) implementando nuevos métodos en las clases existentes, extendiendo de éstas y añadiendo nuevas clases e interfaces que no afectan a las anteriores. Un ejemplo de ello es la clase Graphics2D, la cual extiende a la clase Graphics manteniendo la retrocompatibilidad. De esta forma, el método paint() toma como parámetro un objeto Graphics; en el caso en que se realiza un renderizado por pantalla (que es lo normal) se usa realmente un objeto Graphics2D que hereda de Graphics, de manera que si se quieren utilizar las nuevas funcionalidades debe hacerse lo que se muestra en la sentencia anterior: el objeto Graphics que se pasa como argumento en el método paint() se convierte en un objeto de la clase Graphics2D (acción que se denomina *upcasting*), con lo cual ya pueden aprovecharse todas sus ventajas. Siempre que se vaya a utilizar Graphics2D, esto será lo primero que se haga.

Por último, se invoca al método draw(), el cual simplemente pintará la figura que se le pase como parámetro, que en este caso ha sido un rectángulo.

2.2 El contexto

Cuando Graphics2D va a realizar un operación de renderizado simple, tal como dibujar un rectángulo, un texto o una imagen, lo hace en base a un contexto determinado. El contexto es, sencillamente un conjunto de atributos como puede ser un estilo de letra, una transformación geométrica, un color para el trazo del lápiz, el grosor de éste, etc. En resumen, todas las características que nuestro objeto tendrá una vez que se renderice, estarán en base al contexto que tenga Graphics2D en el momento en que se realice el renderizado. Para renderizar texto, imágenes o figuras, primero debe establecerse el contexto como convenga y después aplicar uno de los métodos de renderizado de Graphics2D, tales como draw() o fill(). El contexto necesario para dibujar cada figura puede ser distinto, esto es, se puede dibujar un rectángulo en rojo estableciendo un contexto de color de trazo rojo y, a continuación, modificar el contexto para dibujar un óvalo amarillo sobre dicho rectángulo rojo.

Si el programador no modifica el contexto del objeto Graphics2D sobre el que se va a renderizar, éste utilizará un contexto por defecto: trazo negro de 1 punto de ancho en el caso de las figuras geométricas.

Siguiendo este criterio, si se desea dibujar el rectángulo de la figura 2.1 un poco más grueso, tan sólo hay que cambiar el atributo del contexto llamado *stroke* o pincel, para lo cual se suministra el método setStroke(). Una vez definido el pincel o lápiz, tan sólo hay que invocar a draw() como se ha hecho antes:

```
public void paint (Graphics g) {
      Graphics2D g2 = (Graphics2D)g;
      Rectangle2D r2 = new Rectangle2D.Float(75, 50, 100, 25);
      Stroke pincel = new BasicStroke(4.0f,   BasicStroke.CAP_ROUND,
                                              BasicStroke.JOIN_MITER);
      g2.setStroke(pincel);
      g2.draw(r2);
   }
```

Se ha modificado el comportamiento del pincel justo antes de dibujar el rectángulo con lo que ahora el rectángulo aparece de forma diferente: el contexto de Graphics2D ha sido alterado en parte, concretamente el lápiz, para que una figura se dibuje de forma diferente a como lo hace por defecto. El resultado puede verse en la figura 2.2.

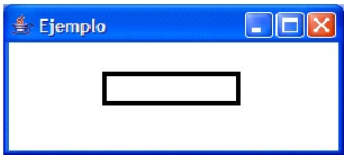


Figura 2.2. Dibujo de un rectángulo habiendo modificado previamente el contexto de dibujo del objeto **Graphics2D**

2.3 El sistema de coordenadas de Java2D

Java2D mantiene dos sistemas de coordenadas: las coordenadas de usuario (*user space*) y las coordenadas de dispositivo (*device space*):

Las coordenadas de usuario constituyen un sistema de coordenadas totalmente independiente del dispositivo final en el que se vaya a hacer el renderizado. Las aplicaciones usan exclusivamente este sistema de coordenadas, de forma que el tamaño y posición de todas las figuras geométricas se especifican mediante coordenadas de usuario.

Las coordenadas de dispositivo constituyen un sistema de coordenadas que sí depende del dispositivo en el cual va a renderizarse realmente. Por ejemplo, en un dispositivo multipantalla con un escritorio virtual donde una ventana puede ocupar más de una pantalla física, el sistema de coordenadas de dispositivo implica un número de pantalla y una posición física dentro de ella.

Java2D ejecuta automáticamente las conversiones entre los dos sistemas de coordenadas en el momento en que se realiza el renderizado real sobre un dispositivo. Un ejemplo de ello es que, aunque el sistema de coordenadas para un monitor es distinto del de la impresora, estas diferencias son invisibles para las aplicaciones Java2D.

2.3.1 Coordenadas de usuario

El origen de las coordenadas de usuario está situado en la esquina superior izquierda del objeto Graphics2D: este punto sería el (0,0). A medida que se incrementa la **x** (primer valor de la coordenada) habrá un desplazamiento hacia la derecha y a medida que se incrementa la **y** (segundo valor de la coordenada) habrá un desplazamiento hacia abajo.

2.3.2 Coordenadas de dispositivo

Java2D define tres bloques de información necesarios para soportar la conversión entre las coordenadas de usuario y las de dispositivo. Esta información se representa mediante tres objetos de las clases: GraphicsEnvironment, GraphicsDevice y GraphicsConfiguration.

Entre ellas representan toda la información necesaria para localizar un objeto que está siendo renderizado o para convertir coordenadas de un sistema a otro. GraphicsEnvironment describe la colección de dispositivos de renderizado visible en una aplicación Java dentro de una plataforma particular; entre estos dispositivos se incluyen pantallas, impresoras y *buffers* de imagen. También incluye una lista de todas las fuentes de texto disponibles en la plataforma.

GraphicsDevice describe un dispositivo concreto de renderizado que será

visible a la aplicación, como la pantalla o la impresora. Por ejemplo, un dispositivo de visualización SVGA puede funcionar en varios modos: 640x480x16 colores, 640x480x256 colores o 800x600x256 colores. Pues bien, la pantalla con SVGA se representa con un objeto GraphicsDevice y cada uno de esos modos de representa con un objeto GraphicsConfiguration.

Una situación común en la práctica es la programación en un sistema multipantalla. Java tiene tres configuraciones distintas de multipantalla, que consisten en:

- Dos o más pantallas independientes
- Dos o más pantallas en la que una es la principal y las otras visualizan copias de lo que aparece en la principal.
- Dos o más pantallas que forman un escritorio virtual, al que también se llama dispositivo virtual (*virtual device*).

En estas tres configuraciones, cada modo de pantalla se representa con un objeto GraphicsDevice y cada uno de ellos puede tener muchos objetos GraphicsConfiguration asociados. Estos objetos permiten crear un objeto Frame, JFrame, Window o JWindow según un modo de pantalla en el cual va a renderizarse.; por ejemplo, la clase JFrame posee un constructor que admite como parámetro un objeto GraphicsConfiguration que sirve para indicar qué configuración debe usarse en su renderizado.

Cuando se usan dos o más pantallas para formar un escritorio virtual, para representarlo se utiliza un sistema de coordenadas que existe fuera de las coordenadas de cada pantalla física. Los límites de este escritorio son relativos a este sistema de coordenadas virtual, por lo que podemos tener coordenadas negativas, ya que el origen de la pantalla principal (la física), el punto (0,0), puede estar situado en el centro del escritorio virtual.

Para saber si un entorno de pantalla es un entorno virtual en el que un objeto de tipo Window o Frame pueden albergar dos o más pantallas, puede invocarse al método getBounds() para ver si el origen es distinto al punto (0,0).

Si se dispone de un entorno de pantalla virtual entonces, debido a que el sistema de coordenadas es relativo al escritorio virtual, es necesario usar esas coordenadas virtuales al llamar al método setLocation() cuando se quiere colocar o recolocar un objeto de tipo Frame o Window en el escritorio. Un ejemplo de ello es el siguiente:

```
JFrame f = new JFrame(GraphicsConfiguration gc);
Rectangle limites = gc.getBounds(gc);
f.setLocation (25 + bounds.x, 25 + bounds.y);
```

En este ejemplo, si no se toman en cuenta los límites del objeto GraphicsConfiguration, el objeto de tipo JFrame se visualizará en la posición (25,25) de la pantalla principal, que puede ser una coordenada distinta de la pantalla

que se ha especificado en dicho objeto GraphicsConfiguration. Notar que el método getBounds() devuelve un objeto Rectangle en el sistema virtual de coordenadas.

En la tabla siguiente, a modo de resumen, se muestran las clases que intervienen en el sistema de coordenadas:

Clase	Descripción
GraphicsEnvironment	Define la colección de dispositivos visibles desde una plataforma Java.
GraphicsDevice	Define un dispositivo concreto de gráficos.
GraphicsConfiguration	Define un modo de ese dispositivo concreto.

2.4 Un paso adelante: el canal alfa

En este apartado se va a implementar un ejemplo algo más avanzado, aunque sigue siendo muy sencillo. El objetivo es ilustrar cómo es posible utilizar transparencias en el momento de dibujar figuras geométricas o cualquier otra cosa; en otras palabras, cuando se dibuja una figura sobre otra, la última dibujada no oculta a la de abajo sino que la deja entrever. El código es el siguiente:

```
public void paint (Graphics g) {
   Graphics2D g2 = (Graphics2D)g;
   g2.setColor(Color.red);
   Rectangle2D r1 = new Rectangle2D.Float(250.0f,50.0f,100.0f,100.0f);
   g2.fill(r1);
   AlphaComposite ac =
         AlphaComposite.getInstance(AlphaComposite.SRC_OVER, 0.5f);
   g2.setColor(Color.green);
   g2.setComposite(ac);
   Rectangle2D r2 = new Rectangle2D.Float(200.0f,100.0f,100.0f,100.0f);
   g2.fill(r2);
   ac = AlphaComposite.getInstance(AlphaComposite.SRC_OVER, 1.0f);
   g2.setColor(Color.magenta);
   g2.setComposite(ac);
   Rectangle2D r3 = new Rectangle2D.Float(150.0f,150.0f,100.0f,100.0f);
   g2.fill(r3);
   ac = AlphaComposite.getInstance(AlphaComposite.SRC_OVER, 0.8f);
   g2.setColor(Color.yellow);
   g2.setComposite(ac);
   Rectangle2D r4 = new Rectangle2D.Float(100.0f,200.0f,100.0f,100.0f);
   g2.fill(r4);
}
```

Notar que lo único que se ha modificado con respecto a ejemplos anteriores (ejemplo de la figura 1.1) es simplemente el método paint(). La clase Ventana y el main() permanecen intactos. El resultado del código anterior puede apreciarse en la figura 2.3.

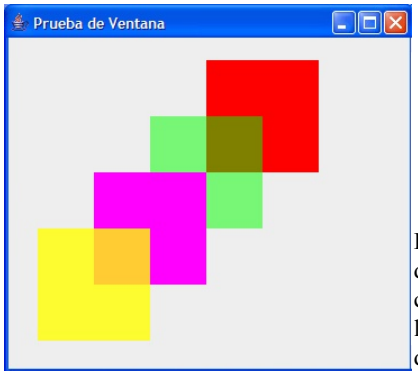


Figura 2.3. Dibujo de cuadrados rellenos de color y solapados entre sí. Cada cuadrado es translúcido en cierto grado, lo que hace que se vea lo que tiene por debajo.

Este ejemplo sirve para mostrar el tratamiento de transparencias en Java2D. Se han dibujado 4 rectángulos (aunque un rectángulo con la base igual a la altura es un cuadrado, Java2D no distingue entre rectángulos y cuadrados). El primer rectángulo se ha puesto de color rojo y sin definir ninguna transparencia, esto es totalmente opaco, por lo que se puede apreciar que la transparencia por defecto al renderizar la figura es nula, y no se ve nada de lo que hay por detrás (que es el fondo blanco). Acto seguido se ha dibujado otro rectángulo de color verde con una transparencia del 50 por ciento. La línea de código en la que se define la transparencia es:

```
AlphaComposite ac =
        AlphaComposite.getInstance(AlphaComposite.SRC_OVER, 0.5f);
```

En ella se crea un objeto AlphaComposite con un tratamiento de colores en el que el color renderizado predominante es el de la figura que se superpone (de ahí lo de **SRC_OVER**). La transparencia se define mediante el último parámetro, el cual puede tomar un valor entre 0.0 y 1.0. Ese parámetro es lo que se denomina el canal alfa, del que se hablará más adelante. Un valor del canal alfa de 1.0 indica que el objeto que se va a renderizar es totalmente opaco, mientras que un valor de 0.0 indica una transparencia total (ni siquiera se ve, ya que el color del fondo de la ventana lo cubre).

Puede verse en la figura 2.3 que el rectángulo magenta al ser opaco no deja que se vea la parte del rectángulo verde sobre la que se ha colocado, ya que la mezcla de colores, como se ha comentado antes, indica que el color predominante en el renderizado es el superpuesto. Por último, decir que para hacer el relleno de los rectángulos es necesario llamar al método fill() de Graphics2D, en lugar de draw(). Este tomará los valores que hemos modificado del contexto y los aplicará en el relleno (como el color o la transparencia).

Así pues, el canal alfa es la banda que mide la transparencia de un objeto. La imagen puede llevar información sobre su transparencia en cada pixel que contiene, de forma que el color de un píxel puede estar formado por cuatro bandas: rojo, verde, azul y alfa. El canal alfa se puede usar en conjunto con un objeto de tipo Composite

en el contexto de Graphics2D para mezclar la imagen con dibujos existentes previamente, teniendo en cuenta la transparencia que se indique; esto da lugar a la clase AlphaComposite del paquete java.awt.

Como ya se ha visto, el valor alfa se indica mediante un porcentaje normalizado, lo que le indica al sistema de renderizado qué cantidad de un color a renderizar debe mostrarse cuando éste y el ya existente debajo (como resultado de un dibujo previo) de él se superponen. Los colores opacos (con un valor de alfa de 1.0) no permiten mostrarse a ninguno de los colores que quedan debajo suya, mientras que los colores transparentes, con un valor alfa de 0.0 dejan pasar todo el color a través.

Cuando se construye un objeto de la clase AlphaComposite (con la función estática getInstance() ya que la clase AlphaComposite carece de constructor), puede especificarse un valor para el canal alfa. Cuando se incorpora este objeto AlphaComposite al contexto de Graphics2D, este valor alfa extra incrementa la transparencia de cualquier objeto gráfico que se renderice posteriormente. El valor que pueda tener cada objeto gráfico por sí mismo es multiplicado por el valor alfa del AlphaComposite.

En la tabla siguiente se pueden apreciar los dos métodos para definir transparencias de la clase AlphaComposite:

Métodos estáticos	Descripción
getInstance(int, float)	Define la transparencia en el segundo parámetro y en el primero el estilo de composición
getInstance(int)	Igual que el anterior, pero mantiene la transparencia que ya hubiese anteriormente.

2.5 Modificación del contexto de Graphics2D

Para configurar el contexto de Graphics2D a la hora de renderizar, deben usarse los métodos **set** de Graphics2D para especificar atributos tales como preferencias de renderizado (RenderingHints), pincel o lápiz (Stroke), gestión del color (Paint), la porción a dibujar (*clipping path*), superposición de objetos (Composite) y transformaciones geométrica (Transform). A continuación profundizaremos en el significado de cada uno de ellos.

2.5.1 Preferencias, pinceles y colores

En los apartados siguientes estudiaremos los elementos del contexto que con más frecuencia se suelen cambiar, y dejaremos para más adelante los más complejos. A modo de resumen introductorio, la siguiente tabla ilustra los métodos que intervienen en el cambio de contexto de Graphics2D con respecto a preferencias,

pinceles y colores:

Atributos a modificar	Forma de utilización	Descripción
Preferencias	RenderingHints rh = new RenderingHints(int,int); g2.setRenderingHints(rh);	Define ciertos aspectos del renderizado como por ejemplo la aplicación del *antialiaising*
Estilos de línea del pincel	Stroke s = new BasicStroke(float); g2.setStroke(s);	El constructor define el ancho de línea del pincel. El método setStroke() lo incorpora a Graphics2D
	Stroke s = new BasicStroke(float, int, int); g2.setStroke(s);	Además del ancho de línea, se definen el estilo de acabado de línea y los bordes de las líneas
	Stroke s = new BasicStroke(float, int,int,float,array,float); g2.setStroke(s);	Aquí se especifica un patrón de punteo o discontinuidad
Atributos de relleno	g2.setPaint(Color);	Se rellena una figura con un color simple
	g2.setPaint(GradientPaint gp);	Se rellena una figura con un patrón de colores difuminados
	g2.setPaint(TexturePaint tp);	Se rellena una figura con una textura definida por el usuario

2.5.1.1 Preferencias

Un objeto de la clase RenderingHints encapsula todas las preferencias no vinculantes sobre cómo se renderiza un objeto. Para modificar estas preferencias, solamente hay que crear un objeto de la clase RenderingHints convenientemente inicializado y pasárselo al contexto mediante el método setRenderingHints() de la clase Graphics2D.

RenderingHints soporta los siguientes tipos de preferencias:

Nombre	Valores posibles
Alpha interpolation (Cálculo de la transparencia)	El del sistema, calidad, o velocidad
Antialiasing	El del sistema, activado o desactivado
Color rendering	El del sistema, calidad, o velocidad
Dithering (Cálculo del color más aproximado)	El del sistema, activado o desactivado
Fractional metrics (Posicionamiento preciso de rasgos tipográficos)	El del sistema, activado o desactivado

Interpolation (Cálculo del color de un píxel en imágenes escaladas)	Vecino más cercano, bilinear, o bicúbico
Rendering	El del sistema, calidad, o velocidad
Text antialiasing	por defecto, activado o desactivado

En el siguiente ejemplo, se quita el antialiasing, (lo que puede provocar que las curvas se dibujen con los bordes dentados) y la calidad de renderizado se establece a favor de la velocidad:

```
public void paint(Graphics g) {
   RenderingHints rh = new RenderingHints(
                                    RenderingHints.KEY_ANTIALIASING,
                                    RenderingHints.VALUE_ANTIALIAS_OFF);
   rh.put(          RenderingHints.KEY_RENDERING,
                    RenderingHints.VALUE_RENDER_SPEED);
    Graphics2D g2 = (Graphics2D)g;
   g2.setRenderingHints(rh);
}
```

Es importante comentar que no todas las plataformas obedecen la modificación del modo de dibujo; en otras palabras, el especificar estas pistas o *hints* (que también podríamos traducir por recomendaciones o preferencias) de dibujo no garantiza que sean utilizados.

2.5.1.2 Especificar el estilo de línea

La clase BasicStroke define las características aplicadas al trazo con que se dibuja una figura geométrica, lo que incluye el ancho y la discontinuidad que deja al dibujarse (*dashing pattern*), la forma en que se conectan los segmentos que componen la figura, y el remate, si lo hay, del final de la línea. Para seleccionar el atributo *stroke* (pincel) de Graphics2D, se crea un objeto de la clase BasicStroke, para después pasárselo al método setStroke() y establecerlo en el contexto. Esta información es la que se usa después al renderizar las figuras.

En el ejemplo siguiente puede verse cómo se modifica la anchura de línea del rectángulo de la figura 2.1, de manera parecida a como se hizo en la figura 2.2.

```
public void paint (Graphics g) {
   Graphics2D g2 = (Graphics2D)g;
   Rectangle2D r2 = new Rectangle2D.Float(75,50,100,25);
   Stroke pincel = new BasicStroke(10.0f);
   g2.setStroke(pincel);
   g2.draw(r2);
}
```

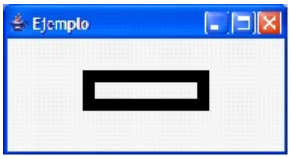


Figura 2.4. Rectángulo de trazo muy grueso (10 píxeles)

El resultado obtenido es un aumento considerable en la anchura del trazo, como se aprecia en la figura 2.4.

La anchura del trazo es la longitud de la línea medida perpendicularmente a su trayectoria. La anchura debe pasarse como un valor **float** en el sistema de coordenadas del usuario, que es el equivalente a 1/72 pulgadas cuando se usa la transformación por defecto existente.

Por otro lado, también puede modificarse el estilo de la unión de líneas, es decir, la forma en que se dibuja la conyuntura en que dos segmentos de una figura se encuentran. BasicStroke tiene los tres tipos de unión que se aprecian en la figura 2.5.

Figura 2.5. Tipos de unión entre segmentos de una misma figura

Además, se puede también modificar el estilo del remate en el extremo de un segmento, que también admite tres tipos diferentes, como puede apreciarse en la figura 2.6.

Figura 2.6. Tipo de remate en los extremos de un segmento

El siguiente trozo de código modifica las 3 características indicadas anteriormente, a saber, el grosor del trazo, las uniones entre segmentos y el remate final de cada segmento. Con respecto al ejemplo de la figura 2.4 lo único que se necesita modificar es la llamada al constructor de BasicStroke, al que en vez de un

parámetro se le pasan tres (evidentemente dicho constructor está sobrecargado):

```
public void paint (Graphics g) {
   Graphics2D g2 = (Graphics2D)g;
   Rectangle2D r2 = new Rectangle2D.Float(75,125,100,25);
   Stroke pincel =
    new BasicStroke(4.0f, BasicStroke.CAP_SQUARE, BasicStroke.JOIN_ROUND);
   g2.setStroke(pincel);
   g2.draw(r2);
}
```

Así, se obtiene el rectángulo de la figura 2.7, con un estilo distinto al de las anteriores.

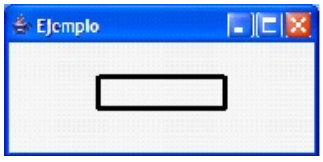


Figura 2.7. Rectángulo de esquinas redondeadas

Por otro lado, también es posible hacer que el trazo con que se dibuja una figura no sea contínuo, sino discontínuo. La definición de un patrón de línea discontínua es bastante sencilla gracias a un nuevo constructor de la clase BasicStroke: al crear un objeto BasicStroke es posible especificar un tipo de línea discontínua mediante un *array*. Las posiciones pares del *array* representan la longitud de los trozos visibles, mientras que las impares representan la longitud de los espacios o huecos entre trazos visibles; por ejemplo, la posición 0 del *array* representa el primer trazo visible y la posición 1 el primer espacio que le sigue.

En el siguiente código se dibujan dos rectángulos con distintospatrones de discontinuidad. El resultado puede verse en la figura 2.8.

```
public void paint (Graphics g) {
   Graphics2D g2 = (Graphics2D)g;
   Rectangle2D r2 = new Rectangle2D.Float(200,150,100,25);
   float punteo1[] = {10.0f, 4.0f};
   BasicStroke pincel1 = new BasicStroke(   4.0f,
                                            BasicStroke.CAP_ROUND,
                                            BasicStroke.JOIN_MITER,
                                            3.0f,
                                            punteo1,
                                            10.0f);
   g2.setStroke(pincel1);
   g2.draw(r2);
   Rectangle2D r3 = new Rectangle2D.Float(200,300,100,25);
   float punteo2[] = {10.0f, 3.0f, 10.0f, 4.0f, 2.0f, 8.0f};
   BasicStroke pincel2 = new BasicStroke(   4.0f,
```

```
                                    BasicStroke.CAP_ROUND,
                                    BasicStroke.JOIN_MITER,
                                    3.0f,
                                    punteo2,
                                    10.0f);
g2.setStroke(pincel2);
g2.draw(r3);
}
```

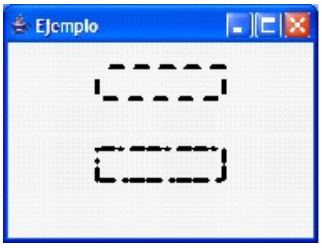


Figura 2.8. Rectángulos de línea discontínua

En la figura 2.8 se observa que en el rectángulo superior la distancia entre trazo y trazo es constante. Eso es debido a la definición del array para este rectángulo ya que, como se puede apreciar en el de abajo, los valores del array para espacios en blanco y tramos del trazo son distintos cada vez.

Por otro lado, también resulta interesante observar cómo el constructor de BasicStroke posee dos parámetros adicionales que no se han comentado aún: son los parámetros de las posiciones 4 y 6. El de la posición 6 sirve para indicar que la primera línea en que comienza la figura no tiene porqué ser el primer trazo del *array* de tramos, sino un *offset* o desplazamiento sobre dicho *array*. El parámetro 4 es más difícil de explicar, y está relacionado con el tipo de conexión entre segmentos JOIN_MITER; si los segmentos que se unen forman un ángulo muy agudo, entonces el pico generado por este tipo de unión puede ser excesivamente largo, de tal manera que este 4º parámetro permite especificar un límite máximo en la longitud del pico que se forma. En la figura 2.9 pueden verse dos uniones de segmentos; ambas tienen la misma altura, pero el pico formado en la segunda debido al extremo ángulo agudo que forman sus segmentos hace que aparente ser mucho más alta que la primera.

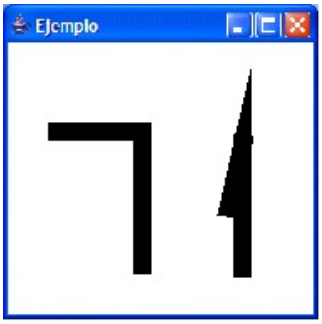


Figura 2.9. Ejemplo de JOIN_MITTER en ángulos muy agudos.

2.5.1.3 Especificación de los atributos de relleno

Para modificar la apariencia del relleno cuando se renderiza una figura geométrica cerrada, debe modificarse el objeto de tipo Paint del contexto, el cual determina el color o el patrón de relleno que se usa cuando una figura (que hereda de Shape) se renderiza.

Java2D permite varias formas de relleno, entre ellas el relleno con un gradiente de color y con una textura (imagen repetida en forma de mosaico).

2.5.1.3.1 Gradiente de color

El gradiente de color produce un efecto en el que el interior de la figura va cambiando de color progresivamente, desde un color inicial a uno final a lo largo de una línea recta imaginaria. Para un rellenado de color con gradiente se dispone de la clase GradientPaint, que se encarga de realizar automáticamente el relleno. Cuando se crea un GradientPaint se especifica una posición inicial y su color, y una posición final con su color. El color de relleno va cambiando progresivamente a medida que se desplaza de un color a otro sobre la línea imaginaria que los conecta. La figura 2.10 muestra dos ejemplos de degradado que van del azul al negro.

Para rellenar una figura con un gradiente de color hay que seguir los pasos siguientes:

Figura 2.10. Ejemplo de relleno de dos rectángulos, el primero con un degradado de color vertical, y el segundo en diagonal

1. Crear un objeto GradientPaint, con los colores a degradar. En este momento hay que especificar también los dos puntos que definen el segmento sobre el que se desplaza el degradado. Por supuesto, estos dos puntos están referidos al objeto Graphics2D en general, y se aplicarán a cuales quiera figuras que se dibujen posteriormente.
2. Llamar al método setPaint() de la clase Graphics2D.

3. Crear la figura (que hereda de la clase Shape)
4. Llamar al método fill(Shape) de la clase Graphics2D.

El código que produce el resultado de la figura 2.10 es el que sigue:

```
public void paint (Graphics g) {
        super.paint(g);
        Graphics2D g2 = (Graphics2D)g;
        GradientPaint gp1 =
             new GradientPaint(50.0f,25.0f,Color.blue, 50.0f, 225.0f, Color.black);
        g2.setPaint(gp1);
        Rectangle2D r1 = new Rectangle2D.Float(25, 25, 200, 200);
        g2.fill(r1);
        GradientPaint gp2 =
             new GradientPaint(250.0f,25.0f,Color.blue, 450.0f, 225.0f, Color.black);
        g2.setPaint(gp2);
        Rectangle2D r2 = new Rectangle2D.Float(250, 25, 200, 200);
        g2.fill(r2);
}
```

Nótese el uso de la sentencia super.paint(g) necesaria en JDK 6.0 para refrescar el fondo de la ventana (si no aparecería como fondo la "basura" que en ese momento haya en el escritorio del sistema operativo).

En cada llamada al constructor de GradientPaint del código anterior, los dos primeros parámetros indican la posición en la que está el color de inicio. El tercer parámetro es el color de inicio en sí. Los tres últimos parámetros tienen la misma función con respecto al color final y su posición.

2.5.1.3.2 Relleno con texturas

Para aplicar una textura en el relleno de una figura, se utiliza la clase TexturePaint. Ésta rellena una figura geométrica con una imagen (denominada patrón) que se repite yuxtaponiéndose de forma horizontal y vertical hasta completar todo el área de la figura. Cuando se crea un objeto del tipo TexturePaint, a su constructor se le pasa como parámetro un objeto BufferedImage para usarlo como patrón. BufferedImage es una clase que contiene una imagen almacenada en un *buffer* de datos accesible, esto es, en memoria; cada objeto de esta clase dispone, a su vez, de un objeto Graphics2D en el que puede dibujarse e incluso colocar una imagen almacenada en un fichero. La clase BufferedImage es sumamente importante, ya que proporciona muchas posibilidades para el tratamiento de imágenes en general, por lo que se verá en profundidad en capítulos posteriores. Por último, al constructor de TexturePaint también se le debe pasar un rectángulo para definir la escala a que se debe reducir o amplia la figura para formar la textura final. Un ejemplo de esta combinación aparece en la figura 2.11.

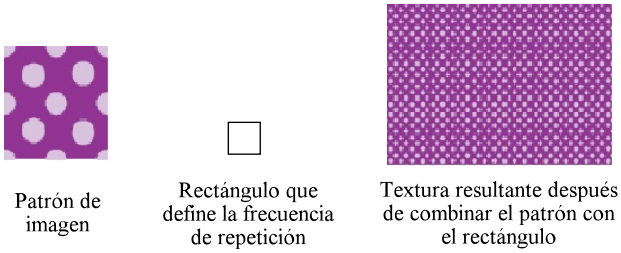


Figura 2.11. Combinación de una imagen y un rectángulo para formar una textura

Concretando, para rellenar una figura con una textura hay que seguir los siguientes pasos:

1. Crear un objeto de la clase TexturePaint utilizando para ello una BufferedImage como patrón.
2. Llamar al método setPaint() de la clase Graphics2D.
3. Crear la figura (que hereda de Shape).
4. Llamar al método fill(Shape) de la clase Graphics2D.

En el siguiente ejemplo se va a rellenar un rectángulo con una textura formado por un patrón que va a ser, a su vez, una elipse roja dentro de un rectángulo naranja. Este patrón no es ninguna imagen sacada de un fichero sino que, a su vez, se ha dibujado en un objeto BufferedImage en memoria utilizando el método fill() ya conocidos. Esto demuestra cómo es posible dibujar sobre un Graphics2D visible o sobre un invisible almacenado en memoria. El código para obtener el rectángulo rellenado con una textura simple es:

```
public void paint (Graphics g) {
   Graphics2D g2 = (Graphics2D)g;
   BufferedImage bi = new BufferedImage (5,5,BufferedImage.TYPE_INT_RGB);
   Graphics2D bigr = bi.createGraphics();

   bigr.setColor(Color.orange);
   Rectangle2D r2 = new Rectangle2D.Float(0.0f,0.0f,5.0f,5.0f);
   bigr.fill(r2);
   bigr.setColor(Color.red);
   Ellipse2D e2 = new Ellipse2D.Float(0.0f,0.0f,5.0f,5.0f);
   bigr.fill(e2);

   Rectangle2D r3 = new Rectangle2D.Double(0.0,0.0,5.0,5.0);
   TexturePaint tp = new TexturePaint(bi, r3);
```

```
    g2.setPaint(tp);

    Rectangle2D rt = new Rectangle2D.Float(0.0f, 0.0f, 250.0f, 250.0f);
    g2.fill(rt);
}
```

Las dos figuras con las que se ha rellenado la imagen **bi** a través de su contenido de tipo Graphics2D al que se ha llamado **bigr** son las que conforman el patrón de la textura, en concreto una elipse roja sobre un rectángulo naranja. Después de crear la textura, se construye el objeto TexturePaint a partir de la BufferedImage **bi** para, acto seguido, añadir la textura creada al contexto con el método setPaint(). Por último, se crea la figura que albergará el relleno con textura, y se dibuja.

2.5.2 Establecimiento del *clipping path*

El *clipping path* define la porción de un objeto Graphics2D que necesita ser renderizada. La porción a dibujar viene a ser algo así como un delimitador espacial; es una figura virtual que define un marco de manera que los trozos de figuras, imágenes y textos que caigan en su interior sí se dibujarán y los trozos que caigan fura no. Cuando se establece un *clipping path* en el contexto de un objeto Graphics2D, sólo la parte que entra dentro de la zona delimitada por el *clipping path* es renderizada.

Para establecer una porción a dibujar en el contexto, tan sólo se tiene que invocar al método setClip(). Cualquier figura puede usarse para definir el área del clipping path. Para cambiarlo, puede usarse otra vez setClip para especificar una nueva zona de delimitación o invocar al método clip para cambiar el clipping path por la intersección entre el antiguo y una nueva figura creada.

A continuación se ilustra todo esto con un ejemplo que se divide en dos partes para mayor claridad, véase la figura 2.12. El objetivo es rellenar todo el fondo

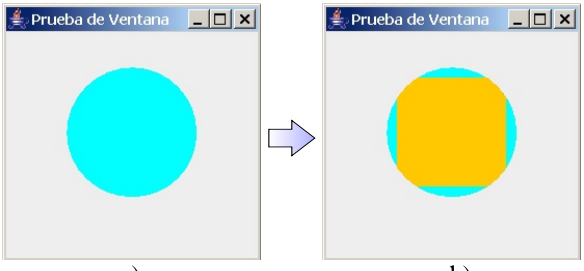


Figura 2.12. Establecimiento de la porción a dibujar (*clipping path*) en función del tamaño de la ventana

de la ventana de color celeste; sin embargo, previamente se establece como porción a dibujar una elipse (cuyo tamaño es relativo a la dimensión que el usuario quiera dar a la ventana, redimensionándola con el ratón). Por ello, lo único que se visualizará será el interior de dicha elipse como se aprecia en la parte **a)** de la figura 2.12. Para que esto quede más patente, en la parte **b)** del ejemplo se dibuja, además, un rectángulo naranja cuyas esquinas caen fuera de una nueva porción a dibujar y no son visibles.

El código queda como sigue:

```
public void paint (Graphics g) {
    Graphics2D g2 = (Graphics2D)g;
    // Primera parte del ejemplo
    int ancho = this.getSize().width;
    int largo = this.getSize().height;
    Ellipse2D e = new Ellipse2D.Float (     ancho/4.0f, largo/4.0f,
                                            ancho/2.0f, largo/2.0f);
    g2.setClip(e);
    g2.setColor(Color.cyan);
    g2.fill(new Rectangle2D.Float(0.0f,0.0f,ancho,largo));

    //Segunda parte del ejemplo
    Rectangle2D r2 = new Rectangle2D.Float(  ancho/4.0f+10.0f, largo/4.0f+10.0f,
                                             ancho/2.0f-20.0f, largo/2.0f-20.0f);
    g2.clip(r2);
    g2.setColor(Color.white);
    g2.fill(new Rectangle2D.Float(0.0f,0.0f,ancho,largo));
}
```

En resumen, con este código se ha creado una figura, concretamente una elipse adaptada al ancho y largo de la pantalla, y acto seguido se ha delimitado la porción de dibujo a esa área en concreto, por lo que la zona que se renderiza después es la que queda dentro de ella. En la segunda fase del ejemplo puede apreciarse cómo se crea un nuevo rectángulo se añade al *clipping path* ya existente, resultando una porción a dibujar que es la intersección de los dos *clipping paths*. Si el lector se fija, las esquinas del rectángulo no existen, ya que en las zonas en las que no está establecida la elipse, no aparece renderizado el rectángulo, ya que sale de sus límites.

2.5.3 Transformaciones de objetos

Las transformaciones que se pueden realizar con Java2D son las definidas por los métodos rotate(), scale(), translate() o shear(); por tanto, las transformaciones que se pueden hacer con estos métodos son rotar, escalar y trasladar, mientras que el método shear() realiza una combinación de ellas.

Para poder realizar transformaciones se necesita crear un objeto de la clase

AffineTransform. Sobre este objeto se aplicará cualquier método de transformación y será también el que se pase al contexto de Graphics2D a través del método transform().

En los siguientes ejemplos se verá qué ofrecen estos métodos y la manera de invocarlos. En el primero se traslada la coordenada (0,0) para que coincida con el centro de la ventana y luego se dibuja el texto "Java2D" varias veces con una rotación progresiva de 45º; nótese cómo cada rotación de 45º se hace con respecto a la anterior, esto es, no se hacen traslaciones de 45º, 90º, 135º, etc. sino siempre de 45º con respectoa a la hecha previamente, lo que produce el efecto deseado. El resultado puede verse en la figura 2.13.a).

```
public void paint (Graphics g) {
   Graphics2D g2 = (Graphics2D)g;
   int ancho = this.getSize().width;
   int largo = this.getSize().height;
   AffineTransform aT = g2.getTransform();
   aT.translate(ancho/2, largo/2);
   g2.transform(aT);
   aT.setToRotation(Math.PI/4.0);

   for (int i=0;i<8;i++) {
      g2.drawString("Java2D",0.0f,0.0f);
      g2.transform(aT);
   }
```

En el ejemplo de la figura 2.13.b) se puede ver cómo la misma cadena de texto es ampliada en dos ocasiones a una escala 2,5 veces mayor invocando al método setToScale(). Una curiosidad de este método es que si se pasa valores negativos no se reduce la escala, sino que aumenta, pero en este caso la cadena se invierte como si se aplicara un espejo por el eje **x** o **y**. Para reducir la escala, se debe poner valores entre 0 y 1.

```
public void paint (Graphics g) {
   Graphics2D g2 = (Graphics2D)g;
   AffineTransform aT = g2.getTransform();
   aT.translate(10, 170);
   g2.transform(aT);
   aT.setToScale(2.5f, 2.5f);
   for (int i=0;i<3;i++) {
      g2.drawString("Java2D", 0.0f, 0.0f);
      g2.transform(aT);
   }
}
```

En el código siguiente se invoca al método shear() para realiazar una transformación en general, ya que este método realiza una composición de las anteriores transformaciones. El resultado puede verse en la figura figura 2.13.c).

```
public void paint (Graphics g) {
   Graphics2D g2 = (Graphics2D)g;
   AffineTransform aT = g2.getTransform();
   aT.translate(100, 100);
   g2.transform(aT);
   aT.shear(20.0, -3.0);
   for (int i=0;i<2;i++) {
      g2.drawString("Java2D", 0.0f, 0.0f);
      g2.transform(aT);
   }
}
```

a) b) c)

Figura 2.13. Ejemplos de rotaciones, escalas y transformación general (*shear*)

Como ya se ha comentado, en los tres ejemplos anteriores se han aplicado también la transformación que se encarga de trasladar las figuras puesto que, en caso contrario, la palabra "Java2D" no se renderizaría en el centro de la ventana.

En la tabla siguiente se muestran los métodos de transformación que ofrece Java2D:

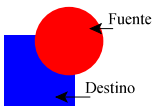
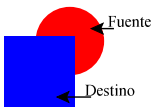
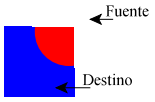
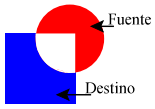
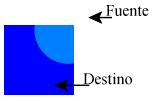
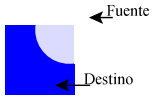
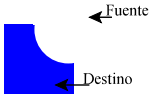
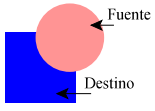
Método	Descripción
setToRotation(float ang)	Rota el objeto el ángulo indicado por el parámetro ang (en radianes)
translate(double x, double y)	Traslada el origen de coordenadas (0, 0) al punto (x, y)
shear(double x, double y)	Desplaza el objeto sobre cada uno de los ejes de coordenadas por separado
setToScale(float a, float l)	Amplía o reduce el tamaño del objeto a lo largo y a lo ancho

2.5.4 Composición de objetos gráficos

Cuando dos objetos gráficos se superponen, es necesario conocer qué colores se deben renderizar en los píxeles superpuestos. Este proceso se conoce como composición.

Para especificar el estilo de composición que debe usarse, hay que añadir un objeto AlphaComposite al contexto de Graphics2D invocando a setComposite(). AlphaComposite, que es una implementación de la interfaz Composite, soporta una variada gama de opciones de composición, de manera que las instancias de esta clase engloban una regla de composición que describe como mezclar un color existente con otro nuevo.

Las reglas de composición aparecen en la siguiente tabla:

Regla	Ejemplo	Regla	Ejemplo
SRC_OVER (Fuente encima)		DST_OVER (Destino encima)	
SRC_IN (Fuente dentro)		SRC_OUT (Fuente fuera)	
DST_IN (Destino dentro) (Ej. con α=0.75)		DST_OUT (Destino fuera) (Ej. con α=0.75)	
CLEAR (Borrado)		SRC (Sólo fuente) (Ej. con α=0.75)	

La explicación de cada regla (que es una constante de la clase AlphaComposite) es:

- SRC_OVER: si los píxeles de la figura que está siendo renderizada actualmente (*source* o fuente) se superponen a los de una figura renderizada anteriormente (*destination* o destino), los píxeles de la fuente se renderizan encima de los del destino.
- DST_OVER: si los píxeles entre la fuente y el destino se superponen, se renderizará solamente la parte de la fuente que quede fuera del área superpuesta, además de renderizar el área del destino al completo.
- SRC_IN: si los píxeles de la fuente y del destino se superponen, solo se renderiza el área de la fuente superpuesta; el destino no se renderiza.
- SRC_OUT: si los píxeles entre la fuente y el destino se solapan, tan sólo se renderiza el área de la fuente que está fuera de la superposición entre las dos; el destino se renderiza en función de su canal alfa.
- DST_IN: si los píxeles de la fuente y del destino se superponen, solo se renderiza el área del destino superpuesta; el fuente no se renderiza.

- DST_OUT: si los píxeles entre la fuente y el destino se solapan, tan sólo se renderiza el área del destino que está fuera de la superposición entre las dos el fuente se renderiza en función de su canal alfa.
- CLEAR: el área que se solape entre la fuente y el destino no se renderiza
- SRC: el fuente se dibuja siempre encima, y si tiene canal alfa éste sólo sirve para disminuir su intensidad, pero no para vislumbrar lo que hay debajo.

El estilo de composición que más se usa es el SRC_OVER y es el que se utiliza por defecto. Por ejemplo, fijándose en el ejemplo de los cuatro cuadrados en el que se trataban las transparencias (figura 2.3), ése es el estilo de composición que se usó.

Además de la composición, como ya se ha visto antes, un objeto AlphaComposite permite manejar transparencias; de hecho, el comportamiento de algnas composiciones depende de las transparencias. Para ello tan sólo se tiene que introducir el valor alfa en la construcción del objeto con getInstance(). Es necesario recordar que el valor alfa toma valores reales entre 0.0 y 1.0 y que, según el valor se acerque más a 0.0, el objeto tendrá un grado de transparencia mayor.

En resumen, para usar una composición, lo que hay que hacer es:

1. Crear un objeto AlphaComposite invocando al método getInstance() y especificando la regla de composición:
   ```
   AlphaComposite al =
           AlphaComposite.getInstance (AlphaComposite.CLEAR);
   ```
   Apreciar que en el constructor no se ha definido ningún canal alfa, pero si se añade un segundo argumento en el constructor especificando el valor, se manejarán las transparencias también.
2. Invocar a setComposite() para añadir el objeto AlphaComposite al contexto de Graphics2D:
   ```
   g2.setComposite(al);
   ```

El siguiente código usa estilos de composición distintos al SRC_OVER para ilustrar visualmente su comportamiento, así como el uso del canal alfa en la composición DST_IN (la figura 2.14 muestra el resultado):

```
public void paint (Graphics g) {
   Graphics2D g2 = (Graphics2D)g;
   Dimension d = getSize();
   int w = d.width;
   int h = d.height;
   // Crea una BufferedImage.
   BufferedImage buffImg = new BufferedImage(w, h,
                                             BufferedImage.TYPE_INT_ARGB);
   Graphics2D gbi = buffImg.createGraphics();

   // Dibujo de un cuadrado rojo
      gbi.setColor(Color.red);
```

```
        Rectangle2D r1 = new Rectangle2D.Float(150.0f, 50.0f, 100.0f, 100.0f);
        gbi.fill(r1);

    AlphaComposite ac;
    // Dibujo de un rectángulo verde encima
        ac = AlphaComposite.getInstance(AlphaComposite.SRC_OVER);
        gbi.setColor(Color.green);
        gbi.setComposite(ac);
        Rectangle2D r2 = new Rectangle2D.Float(150.0f, 100.0f, 150.0f, 100.0f);
        gbi.fill(r2);

    // Dibujo de un cuadrado magenta con preferencia de lo que hay debajo
        ac = AlphaComposite.getInstance(AlphaComposite.DST_OVER);
        gbi.setColor(Color.magenta);
        gbi.setComposite(ac);
        Rectangle2D r3 = new Rectangle2D.Float(100.0f, 100.0f, 100.0f, 100.0f);
        gbi.fill(r3);

    // Dibujo de un cuadrado amarillo que modifica sólo sobre lo que se superpone
        ac = AlphaComposite.getInstance(AlphaComposite.DST_IN, 0.15f);
        gbi.setColor(Color.yellow);
        gbi.setComposite(ac);
        Rectangle2D r4 = new Rectangle2D.Float(150.0f, 150.0f, 100.0f, 100.0f);
        gbi.fill(r4);

    g2.drawImage(buffImg, null, 0, 0);
}
```

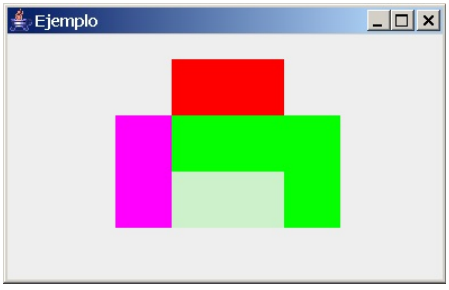


Figura 2.14. Ejemplo de utilización de diversas reglas de composición

Para finalizar, resulta importante añadir un último comentario sobre las reglas de composición. Hay momentos en los que la composición puede producir resultados extraños, ya que a la hora de realizar una composición de dos figuras, la que se superponga puede salir totalmente en negro o no respetar la reglas especificada. Cuando esto ocurre es debido a que las dos figuras se superponen, a su vez, a la superficie de dibujo, la cual es totalmente opaca (valor alfa de 1.0) si la imagen no

esta en *offscreen* (véase el capítulo 5 para más información). La solución a este conflicto es crear la superposición dentro de una imagen en *offscreen* (almacenada en memoria pero no visible), ya que ahí la superficie de dibujo es totalmente transparente (alfa 0.0) y puede superponerse cualquier figura sobre el lienzo base.

Para concluir este capítulo, la siguiente tabla muestra los paquetes, clases e interfaces necesarios para el renderizado con Graphics2D y sus atributos de contexto:

Clases e Interfaces de Java2D para el renderizado de gráficos	Descripción
java.awt	Paquete para generar interfaces de usuario y renderizar gráficos e imágenes
java.awt.GraphicsEnvironment	Clase que describe el conjunto de dispositivos de renderizado visible
java.awt.GraphicsDevice	Clase que describe un dispositivo concreto de renderizado
java.awt.GraphicsConfiguration	Clase que define la configuración de un dispositivo concreto de renderizado
java.awt.Graphics2D	Clase fundamental para el renderizado 2D. Extiende a la original java.awt.Graphics
java.awt.Shape	Clase para definir los contornos de figuras
java.awt.Stroke	Interfaz que rodea el contorno de la Shape que va a ser renderizada
java.awt.BasicStroke	Clase para crear un pincel
java.awt.Paint	Interfaz que define colores para una operación draw() o fill()
java.awt.GradientPaint	Clase que define un patrón gradiente lineal de color para una figura cerrada
java.awt.TexturePaint	Define una textura o un patrón de relleno para una figura
java.awt.Color	Clase que define un color para un trazo o un relleno
java.awt.Composite	Interfaz que define métodos para la composición
java.awt.AlphaComposite	Clase para definir composiciones básicas

Capítulo 3

Figuras geométricas en Java2D

3.1 Introducción

Java2D proporciona varias clases que definen objetos geométricos simples tales como puntos, líneas, curvas y rectángulos. Estos objetos pueden construirse con las clases proporcionadas por el paquete java.awt.geom, por lo que es necesario importar este paquete en nuestros programas antes de hacer uso de cualquiera de sus clases. Entre otras, se proporcionan las clases Rectangle2D, Line2D, Point2D y Ellipse2D.

Para soportar la retrocompatibilidad, las clases geométricas existentes en las versiones previas de JDK, como Rectangle, Point y Polygon, permanecen en el paquete java.awt y se pueden seguir usando.

3.2 Figuras básicas

En este apartado se van a mostrar todas y cada una de las figuras geométricas que vienen predefinidas en Java2D directamente, es decir, no vamos a crear ninguna figura geométrica nueva sino sólo usar las ya existentes. En este sentido, las figuras existentes en Java2D son las líneas rectas, las curvas cuadráticas, las curvas cúbicas, los rectángulos, las elipses u óvalos, los rectángulos con las esquinas redondeadas y los arcos. Estas figuras ocuparán los epígrafes siguientes. La mayoría de las clases que permiten trabajar con las figuras geométricas heredan de la interfaz Shape que, entre otras cosas, permite averiguar si dos figuras se intersecan, si una contiene a otra por completo, etc. Esta interfaz se verá con mayor detenimiento en el epígrafe 3.4.

Al igual que en ejemplos anteriores sólo nos preocuparemos de escribir la función paint() ya que el resto del programa permanece inalterable.

3.2.1 Line2D

Dibujar una línea recta es una de las cosas más sencillas que pueden hacerse con Java2D. Ello se consigue a través de la clase Line2D, cuyo constructor acepta cuatro parámetros, a saber, las coordenadas del punto de inicio y de final respectivamente. La figura 3.1 muestra el resultado de ejecutar el código siguiente:

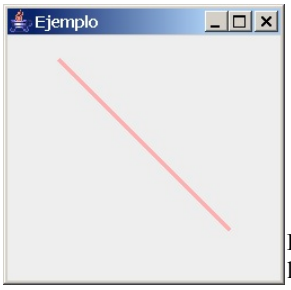


Figura 3.1. Ejemplo de línea recta de color rosa

```
public void paint (Graphics g) {
   super.paint(g);
   Graphics2D g2 = (Graphics2D)g;
   // Dibujo dela línea
      g2.setColor(Color.pink);
      g2.setStroke(new BasicStroke(3.0f));
      Line2D l = new Line2D.Float(50.0f, 50.0f, 200.0f, 200.0f);
      g2.draw(l);
}
```

La clase Line2D también posee otro constructor que, en lugar de aceptar cuatro parámetros, acepta sólo dos de tipo Point2D. La clase Point2D es una utilidad para facilitar el trabajo en general con la Java2D.

3.2.2 Rectangle2D

La clase que se usa para dibujar rectángulos y cuadrados es la Rectangle2D. El constructor especifica en los dos primeros parámetros la posición de la esquina superior izquierda con respecto al sistema de coordenadas de la ventana, y en elos dos siguientes el ancho y largo respectivamente. Estos cuatro parámetros pueden especificarse mediante valores **float** o **double**, usando para ello los constructores Rectangle2D.Float() y Rectangle2D.Double() respectivamente. Esta posibilidad de construir las figuras geométricas mediante coordenadas en **float** o en **double** es una constante que se repite en todas la figuras geométricas.

La figura 3.2 muestra el resultado producido por el siguiente código:

```
public void paint (Graphics g) {
   super.paint(g);
   Graphics2D g2 = (Graphics2D)g;
   // Creación del Rectangle2D
   g2.setColor(Color.red);
   g2.setStroke(new BasicStroke(3.0f));
   Rectangle2D r = new Rectangle2D.Float(100.0f, 75.0f, 50.0f, 100.0f);
```

```
    g2.draw(r);
}
```

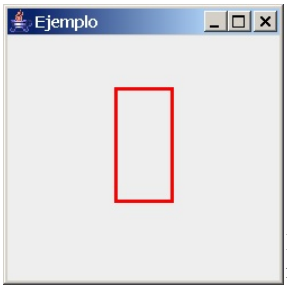


Figura 3.2. Ejemplo de rectángulo simple

3.2.3 RoundRectangle2D

La clase RoundRectangle2D permite dibujar un rectángulo con las esquinas redondeadas. La construcción del rectángulo es idéntica a como se hacía con la clase Rectangle2D, lo único que cambia en este caso son dos parámetros adicionales al final del constructor, que indican el ancho y largo de la curva que define cada esquina. Cuanto más grandes sean estos valores, mayor nivel de redondeo tendrán las esquinas.

La ventana que produce el código siguiente puede apreciarse en la figura 3.3.

```
public void paint (Graphics g) {
        super.paint(g);
        Graphics2D g2 = (Graphics2D)g;
        // Dibujo delrectángulo redondeado
        g2.setColor(Color.pink);
        g2.setStroke(new BasicStroke(3.0f));
        RoundRectangle2D q = new RoundRectangle2D.Float(
                                50.0f, 50.0f, 100.0f, 150.0f, 25.0f, 25.0f);
        g2.draw(q);
}
```

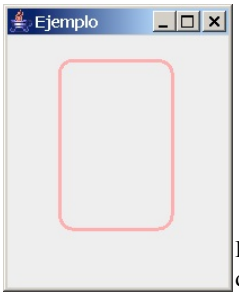


Figura 3.3. Ejemplo de utilización dela clase RoundRectangle2D

3.2.4 Ellipse2D

Las elipses u óvalos se dibujan mediante la clase Ellipse2D. Para ello, debe definirse un rectángulo que será el que encierre entre sus límites a la elipse. No existe el concepto de círculo, por lo que éste debe obtenerse mediante una elipse encerrada en un cuadrado. Un ejemplo de dibujo de elipse aparece en la figura 3.4, siendo el código el siguiente:

```
public void paint (Graphics g) {
   super.paint(g);
   Graphics2D g2 = (Graphics2D)g;
   g2.setColor(Color.orange);
   g2.setStroke(new BasicStroke(3.0f));
   Ellipse2D e = new Ellipse2D.Float(100.0f,75.0f,50.0f,100.0f);
   g2.draw(e);
}
```

Figura 3.4. Ejemplo de elipse

Como ya se ha comentado, y queda reflejado en el código, el constructor de la elipse no define los bordes de la elipse, sino que define los límites de un rectángulo imaginario en el cual se inscribe la elipse. Por tanto, los dos primeros parámetros del constructor indican el desplazamiento conforme al origen de coordenadas y los dos últimos especifican respectivamente el ancho y el alto de tal rectángulo.

3.2.5 Arc2D

La clase Arc2D permite dibujar un arco de una elipse o de un círculo. Para ello, primero se define el rectángulo que contiene la elipse cuyo arco se desea dibujar; a continuación, suponiendo que el centro de dicho rectángulo establece el punto (0, 0) de un eje de coordenadas cartesianas, se especifican los ángulos de inicio y de final del arco en grados sexagesimales. Por último, se indica si se desea cerrar el arco uniendo sus extremos o no. Este cierre puede ser de tres tipos:

- Arc2D.OPEN: el arco queda abierto.

- Arc2D.CHORD: los extremos del arco se unen con un segmento de cuerda.
- Arc2D.PIE: Cada extremo del arco de une mediante un segmento con el punto (0, 0) del eje de coordenadas.

El siguiente código dibuja un rectángulo de referencia. A continuación, y dentro de un rectángulo igual al de referencia, se dibuja un arco que comienza en el ángulo 0º y finaliza en el 135º. Por último el arco se cierra como si fuera un trozo de un diagrama de tarta (Arc2D.PIE). El resultado puede verse en la figura 3.5.

```
public void paint (Graphics g) {
   super.paint(g);
   Graphics2D g2 = (Graphics2D)g;
   // Dibujo del rectángulo de referencia
      g2.setColor(Color.blue);
      g2.setStroke(new BasicStroke(1.0f));
      Rectangle2D r = new Rectangle2D.Float(100.0f, 75.0f, 50.0f, 100.0f);
      g2.draw(r);
   // Dibujo del arco
      g2.setColor(Color.red);
      g2.setStroke(new BasicStroke(3.0f));
      Arc2D a =
         new Arc2D.Float(100.0f, 75.0f, 50.0f, 100.0f, 0.0f, 135.0f, Arc2D.PIE);
   g2.draw(a);
}
```

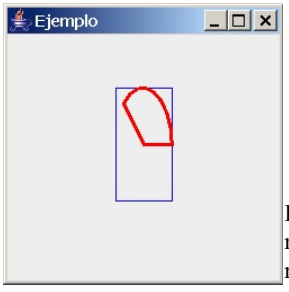


Figura 3.5. Ejemplo de dibujo de un arco. Para una mejor comprensión, en azul también se ha dibujado el rectángulo contenedor de referencia

3.2.6 QuadCurve2D

La clase QuadCurve2D permite construir un segmento curvo basado en ecuaciones matemáticas. La curva generada también recibe el nombre de curva cuadrática de Bézier (a pesar de haber sido desarrolladas por Paul de Casteljau), y se basa en una idea muy sencilla que consiste en establecer dos puntos que definen los extremos de un segmento curvo, así como un tercer punto, llamado punto de control que permite "estirar" más o menos la curvatura de dicho segmento. La figura 3.6 muestra un ejemplo de esto.

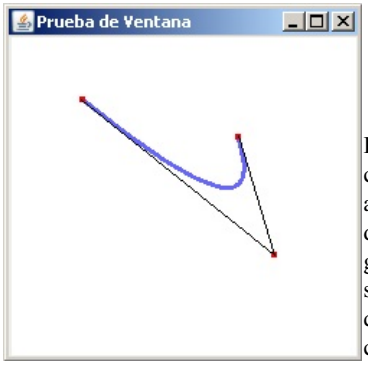


Figura 3.6. Ejemplo de curva cuadrática en la que pueden apreciarse los extremos y el punto de control que "estira" la curva generada. Por motivos ilustrativos se muestran en negro las tangentes que unen los extremos y el punto de control

Supuesto que los puntos de inicio y de fin son $p_1=(x_1, y_1)$ y $p_3=(x_3, y_3)$ y el punto de control es $p_2=(x_2, y_2)$, la fórmula que calcula todos los puntos por lo que va pasando la curva es:

$$B(t) = (1-t)^2 p_1 + 2t(1-t)p_2 + t^2 p_3$$, donc

Nótese que para t=0 se cumple que $B(t)=p_1$ y cuando t=1 entonces $B(t)=p_3$, que son los extremos del segmento.

Pues bien, la clase QuadCurve2D permite hacer esto mismo, tomando como parámetros los tres puntos p_1 p_2 y p_3. Un ejemplo de utilización aparece en el siguiente código, en el que también se han resaltado los tres puntos de referencia. El resultado puede observarse en la figura 3.7.

```
public void paint (Graphics g) {
   super.paint(g);
   Graphics2D g2 = (Graphics2D)g;
   g2.setColor(Color.blue);
   g2.setStroke(new BasicStroke(3.0f));
   QuadCurve2D q = new QuadCurve2D.Float(
                        40.0f, 70.0f, 40.0f, 170.0f, 190.0f, 220.0f);
   g2.draw(q);
```

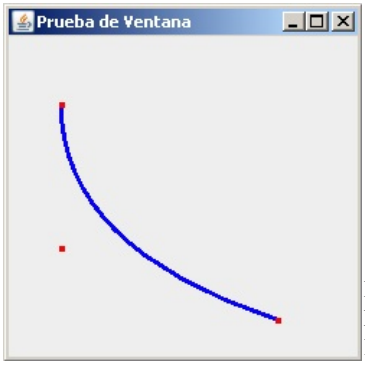


Figura 3.7. Curva cuadrática de Bézier en la que pueden apreciarse los extremos y el punto de control

```
            g2.setColor(Color.red);
        g2.draw(new Rectangle2D.Float(40.0f, 70.0f, 1.0f, 1.0f));
        g2.draw(new Rectangle2D.Float(40.0f, 170.0f, 1.0f, 1.0f));
        g2.draw(new Rectangle2D.Float(190.0f, 220.0f, 1.0f, 1.0f));
    }
```

3.2.7 CubicCurve2D

Las curvas de Bézier pueden ser generalizadas a cualquier grado. No obstante, las más utilizadas son las de grado dos (vistas en el epígrafe anterior) y las de grado tres modeladas mediante la clase CubicCurve2D. Supuesto que los puntos de inicio y de fin son $p_1=(x_1, y_1)$ y $p_4=(x_4, y_4)$, ahora hay dos puntos de control, a saber $p_2=(x_2, y_2)$ y $p_3=(x_3, y_3)$, de manera que la fórmula que calcula todos los puntos por lo que va pasando la curva es:

$$B(t) = (1-t)^3 p_1 + 3t(1-t)^2 p_2 + 3t^2(1-t) p_3 + t^3 p_4$$

Nótese que para t=0 se cumple que $B(t)=p_1$ y cuando t=1 entonces $B(t)=p_4$, que son los extremos del segmento curvo. La figura 3.8 muestra una figura de prueba.

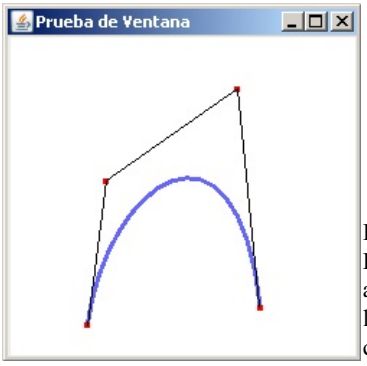


Figura 3.8. Ejemplo de curva de Bézier cúbica en la que también aparecen los segmentos que unen los extremos con los puntos de control y éstos entre sí

Un ejemplo de utilización aparece en el siguiente código, en el que también se han resaltado los cuatro puntos de referencia. El resultado puede observarse en la figura 3.9.

```
public void paint (Graphics g) {
    super.paint(g);
    Graphics2D g2 = (Graphics2D)g;
    g2.setColor(Color.blue);
    g2.setStroke(new BasicStroke(3.0f));
    CubicCurve2D c=new CubicCurve2D.Float(
                40.0f, 60.0f, 60.0f, 120.0f, 140.0f, 130.0f, 150.0f, 210.0f);
    g2.draw(c);
```

```
        g2.setColor(Color.red);
        g2.draw(new Rectangle2D.Float(40.0f, 60.0f, 1.0f, 1.0f));
        g2.draw(new Rectangle2D.Float(60.0f, 120.0f, 1.0f, 1.0f));
        g2.draw(new Rectangle2D.Float(140.0f, 130.0f, 1.0f, 1.0f));
        g2.draw(new Rectangle2D.Float(150.0f, 210.0f, 1.0f, 1.0f));
    }
```

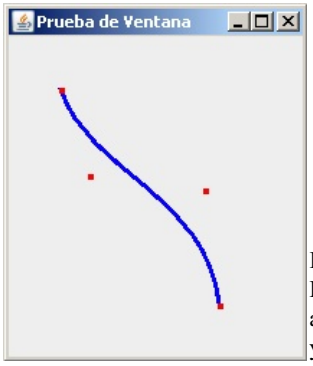


Figura 3.9. Curva cúbica de Bézier en la que pueden apreciarse los puntos extremos y de control

3.2.8 Point2D

La última de las clases que definen una figura geométrica es la clase Point2D. Esta clase no dibuja nada, sino que es la representación de los puntos en Java2D. Ahora bien, sí se pueden dibujar figuras a partir de puntos, y todos los constructores de figuras que hemos visto en epígrafes anteriores están sobrecargados para poder construir figuras tanto a partir de coordenadas sueltas como a partir de objetos Point2D. Un ejemplo de ello aparece en el siguiente código:

```
public void paint (Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        g2.setStroke(new BasicStroke(3.0f));
        Point2D p1 = new Point2D.Float(23.5f, 48.9f);
        Point2D p2 = new Point2D.Float(158.0f, 173.0f);
        Line2D l = new Line2D.Float(p1, p2);
        g2.draw(l);
        }
```

cuyos resultados se aprecian en la figura 3.10.

Por último, sólo nos queda decir que los constructores de todas las clases anteriores están preparados para que los valores que definen sus coordenadas sean de tipo **Double**. Esto se hace sencillamente llamando al constructor Double de la clase. Por ejemplo, la línea recta de la figura 3.10 Se podría haber hecho de manera similar

(aunque admitiendo una mayor precisión) invocando al constructor Double; el resultado visual sería el mismo:

```
public void paint (Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        g2.setStroke(new BasicStroke(3.0f));
        Point2D p1 = new Point2D.Double(23.5, 48.9);
        Point2D p2 = new Point2D.Double(158.0, 173.0);
        Line2D l = new Line2D.Double(p1, p2);
        g2.draw(l);
}
```

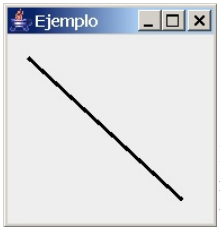


Figura 3.10. Ejemplo de segmento recto construido mediante puntos del tipo Point2D

3.2.9 Resumen de clases relacionadas con figuras geométricas

La siguiente tabla muestra un resumen de las clases vistas hasta ahora y de la relación de herencia que se establece entre ellas.

Clase o interfaz	¿Es una Shape?	¿Es una RectangularShape?	Descripción
Shape (interfaz)			Conjunto de métodos de manipulación y descripción de figuras geométricas.
RectangularShape (interfaz)	✔		Define rutinas de manipulación comunes a todas las figuras que tienen bordes rectangulares.
Point2D			Representa una localización en el sistema de coordenadas.
Line2D	✔		Define una línea recta.
Rectangle2D	✔	✔	Representa un rectángulo.
RoundRectangle2D	✔	✔	Representa un rectángulo con los bordes redondeados.
Ellipse2D	✔	✔	Define una elipse.
Arc2D	✔	✔	Define un arco.
QuadCurve2D	✔		Describe una curva Bézier cuadrática.

CubicCurve2D	✔		Define una curva Bézier cúbica.

3.3 Un primer ejemplo compuesto: bañera

A continuación se va a realizar un primer ejemplo práctico que consiste en componer varias figuras para representar una bañera. Un dibujo tal podría usarse, por ejemplo, en la construcción de planos en planta de edificios residenciales. La bañera es muy simple, y tan sólo esta compuesta por dos rectángulos y un círculo. El ejemplo es el siguiente:

```
public void paint (Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        Rectangle2D r = new Rectangle2D.Float(50.0f, 50.0f,110.0f, 200.0f);
        RoundRectangle2D rr = new RoundRectangle2D.Float(
                                60.0f, 60.0f, 90.0f, 180.0f, 25.0f, 25.0f);
        Ellipse2D e = new Ellipse2D.Float(100.0f, 80.0f, 10.0f, 10.0f);
        g2.draw(r);
        g2.draw(rr);
        g2.draw(e);
}
```

El resultado se muestra en la figura 3.11.

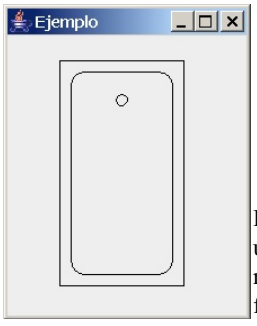


Figura 3.11. Dibujo de una bañera simple mediante secuencias de figuras

Este ejemplo se usará en futuros epígrafes de manera que la secuencia de figuras que ahora mismo conforma la bañera se encapsulará en un sólo componente. De esta manera, la bañera podrá dibujarse de una sola tacada.

3.4 La interfaz Shape

Como ya se vio en la tabla del epígrafe 3.2.9, la mayoría de las figuras geométricas heredan de la interfaz Shape, como por ejemplo Rectangle2D.Float. También otras clases que veremos más adelante, como Area o GeneralPath, heredan de Shape lo que da una idea de la importancia de esta interfaz.

El concepto que subyace en la clase Shape es el de algo dibujable, una figura en general. Como cualquier figura, un objeto que hereda de Shape posee un contorno (denominado *path* en inglés), también llamado trazo. La interfaz Shape proporciona un mecanismo estándar para describir e interpretar el trazo de un objeto geométrico. Como ya hemos comentado, el trazo de un objeto Shape se puede utilizar para pintarlo, rellenarlo o como porción de dibujo (*clipping path*), por lo que existe un amplio abanico de funcionalidades que proporcionan los trazos. Otro de los usos que tiene esta interfaz es que posibilita heredar de ella y crear así Shapes personalizadas. O sea, sólo es necesario crear una clase que implemente la interfaz Shape para definir un nuevo tipo de figura geométrica. No importa cómo se represente la figura internamente, siempre y cuando se implementen los métodos de la interfaz Shape. Lo más importante es que la clase así creada debe ser capaz de generar un trazo que especifique su contorno.

Siguiendo este esquema, se puede crear una clase que represente un triángulo, y para ello es necesario implementar métodos como contains(), getBounds(), getBounds2D(), getPathIterator(), intersects() y otros de la clase Shape. En resumen, los métodos a que fuerza la interfaz Shape son los que aparecen en la siguiente tabla:

Método	Devuelve un	Descripción
contains(double x, double y)	boolean	Determina si las coordenadas caen dentro de la Shape
contains(Point2D p)	boolean	Igual al anterior pero con un Point2D
contains(double x, double y, double ancho, double largo)	boolean	Determina si el área rectangular entra dentro de la Shape
contains(Rectangle2D r)	boolean	Igual que el anterior pero con un Rectangle2D
getBounds()	Rectangle	Devuelve el rectángulo mínimo que recubre la Shape
getBounds2D()	Rectangle2D	Devuelve un rectángulo más optimizado que en el anterior caso
getPathIterator(AffineTransform at)	PathIterator	Devuelve un Iterator que itera sobre los sub-trazos de la figura. Si queremos transformar las iteraciones, usamos la transformada, sino sencillamente pasamos null en el método
getPathIterator(AffineTransform at, double aplanamiento)	PathIterator	Igual que el anterior, pero el segundo parámetro otorga un comportamiento plano a las secciones iteradas de la figura que sean curvas
intersects(double x, double y, double ancho, double largo)	boolean	Testea si el interior de la Shape interseca con el interior del área rectangular pasada como parámetro

intersects(Rectangle2D r)	boolean	Igual que el anterior pero con un Rectangle2D.

3.4.1 Otra manera de construir figuras: GeneralPath

Otra forma de construir figuras es usando la clase GeneralPath, que también hereda de Shape pero no es una interfaz, sino una clase. Esta clase permite definir una figura mediante una secuencia de trazos que pueden establecerse a través de las funciones de la tabla siguiente:

Método	Descripción
GeneralPath (constructor)	Representa una figura geométrica construida a partir de líneas rectas y de curvas cuadráticas y cúbicas
moveTo	Desplaza la punta del lápiz hacia un punto concreto
curveTo	Describe una curva cúbica desde un punto inicial a uno final
lineTo	Define una línea recta desde un punto inicial a uno final
quadTo	Representa una curva cuadrática desde un punto inicial a uno final
closePath	Cierra el trazado de la figura uniendo mediante una línea recta la posición actual del lápiz con la del inicio del dibujo

El siguiente código ilustra alguna de estas operaciones, produciendo una figura imposible de dibujar con las clases vistas en los epígrafes anteriores. El resultado puede apreciarse en la figura 3.12.

```
public void paint (Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        // Creación del GeneralPath
        g2.setColor(Color.green);
        GeneralPath gp = new GeneralPath();
        gp.moveTo(50.0f,50.0f);
        gp.lineTo(100.0f,50.0f);
        gp.curveTo(120.0f,30.0f,120.0f,100.0f,180.0f,125.0f);
        gp.lineTo(50.0f,150.0f);
        gp.closePath();
        // Relleno y contorno
        g2.fill(gp);
        g2.setColor(Color.blue);
        g2.draw(gp);
}
```

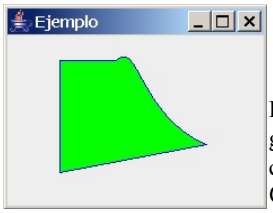


Figura 3.12. Figura geométrica de prueba creada mediante la clase GeneralPath

Cuando se crea un objeto GeneralPath realmente se crea un lápiz imaginario con el que se comenzará a dibujar y que se coloca inicialmente en la posición (0,0).

En primer lugar se ha trasladado el lápiz hacia en punto (50,50) con el mensaje moveTo(). Luego se traza una linea recta con lineTo() y a continuación una curva cúbica con el método curveTo() que parte de la posición actual del lápiz, llega hasta el punto (180, 125) y utiliza los puntos (120, 30) y (120, 100) como controles. Acto seguido se traza otra línea recta hacia el punto (50, 150) y, por último, se cierra la figura con el método closePath(), es decir, el punto donde se había quedado el lápiz imaginario se ha unido con el punto inicial de dibujo, el (50, 50). Por último, añadir que se ha rellenado de verde la figura con el método fill(), y también se ha mostrado el contorno en azul con draw().

3.4.2 Figuras geométricas a la medida: la bañera

En el apartado 3.3 se estudió un ejemplo en el que se creaba una bañera dibujando sucesivamente un rectángulo que delimitaba sus dimensiones, un rectángulo de esquinas redondeadas para representar el hueco de la bañera y un círculo para representar el sumidero. Supongamos por un momento que, por motivos de nuestro desarrollo, nos viéramos obligados a dibujar multitud de estas bañeras (por ejemplo en un plano de viviendas residenciales). Con lo visto hasta ahora nos veríamos obligados a crear las tres figuras integrantes de una bañera una y otra vez para reubicarlas en los diferentes puntos del plano. Sin embargo, parece bastante claro que tal solución es impropia de la reutilización de la que se vanagloria la programación orientada a objetos.

Para evitar esta repetición de código, lo más interesante resulta crear nuestra propia figura geométrica, por ejemplo la clase Bañera (sí, con eñe y todo, ya que Java permite usar la eñe y los acentos en los identificadores de usuario) que hereda de Shape. Claro está, Shape es una interfaz y es necesario implementar en nuestra clase Bañera todos sus métodos, que aparecen en la tabla del epígrafe 3.4. Vayamos por pasos.

Antes de nada, nuestra clase Bañera va a contener las tres figuras que van a soportar su dibujo (estará compuesta por tres campos) :

- Un rectángulo exterior (campo exterior) de tipo Rectangle2D que delimita sus dimensiones.
- Un hueco formado por un objeto (interior) de la clase RoundRectangle2D.
- Un sumidero representado por una Ellipse2D guardada en el campo sumidero.

La mayoría de las funciones que debe implementar una Shape y que aparecen en la tabla del epígrafe 3.4 están referidas a los límites de la figura que, en

nuestro caso, coinciden con los del campo exterior. Por ejemplo, por ser una Shape nuestra Bañera debe implementar la función getBounds() que devuelve sus límites; dado que los límites de la bañera coinciden con los de su campo exterior, nuestra implementación de getBounds() será:

```
public Rectangle getBounds() {
        return exterior.getBounds();
}
```

de tal manera que la mayoría de funciones que debe implementar nuestra Bañera quedan relegadas a sus homónimas del campo exterior. Así el primer bloque de código que conforma nuestra clase Bañera es:

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

public class Bañera implements Shape {

   private Rectangle2D exterior;
   private RoundRectangle2D interior;
   private Ellipse2D sumidero;

   public Bañera(double x, double y, double w, double h){
      exterior = new Rectangle2D.Double(x, y, w, h);
      interior = new RoundRectangle2D.Double(x+10, y+10, w-20, h-20, 25.0, 25.0);
      sumidero = new Ellipse2D.Double(x+w/2-5, y+15, 10.0, 10.0);
   }
   public boolean contains(Point2D p) {
      return exterior.contains(p);
   }
   public boolean contains(Rectangle2D r) {
      return exterior.contains(r);
   }
   public boolean contains(double x, double y) {
      return exterior.contains(x, y);
   }
   public boolean contains(double x, double y, double w, double h) {
      return exterior.contains(x, y, w, h);
   }
   public Rectangle getBounds() {
      return exterior.getBounds();
   }
   public Rectangle2D getBounds2D() {
      return exterior.getBounds2D();
   }
   public boolean intersects(Rectangle2D r) {
      return exterior.intersects(r);
   }
   public boolean intersects(double x, double y, double w, double h) {
      return exterior.intersects(x, y, w, h);
```

```
    }
...
```

Ahora quedan por implementar las dos funciones más peliagudas, que son getPathIterator() en sus dos versiones (una con un parámetro y otra con dos). El objetivo de esta función es devolver un objeto de tipo PathIterator; un objeto tal es, sencillamente, un iterador que el entorno de dibujo Java va invocando para ir recuperando en secuencia los segmentos que van formando el dibujo final. Conceptualmente, esto tiene mucha relación con la clase GeneralPath; en esta clase nosotros, como programadores, añadíamos uno a uno los segmentos que conformaban el dibujo (operaciones moveTo(), curveTo(), lineTo(), quadTo() y closePath()) y son precisamente tales segmentos los que el PathIterator retorna al sistema cuando éste va realizando progresivamente el dibujo de la figura.

En resumen, nuestra Bañera tiene dentro tres objetos Shape que, por tanto, también incorporan sus correspondientes operaciones getPathIterator(). Así pues, para poder implementar las operaciones getPathIterator() en nuestra clase Bañera lo que vamos a hacer es:

- Iterar primero sobre los segmentos del campo exterior
- Iterar después sobre los segmentos del campo interior
- Por último iterar sobre los del campo sumidero

Para poder hacer esto nos crearemos una clase interna BañeraPathIterator que hereda de PathIterator. Esta BañeraPathIterator irá delegando funciones en los PathIterators de los campos interno, externo y sumidero, en ese orden, para lo cual hará uso de un campo interno actualIterator que apuntará en cada momento al PathIterator de objeto Shape que se esté dibujando en ese momento. Todas las funciones de BañeraPathIterator se delegan a las homónimas del campo actualIterator: cuando se acaba de iterar sobre la figura externa, comienza a iterarse sobre interna y, después, sobre sumidero. Siguiendo este criterio, el resto del código que nos queda por definir es (teniendo en cuenta las dos modalidades con que el sistema de dibujo Java puede querer obtener un PathIterator, esto es, con un AffineTransform sólo, o con un AffineTransform y un **double**):

```
...
   public PathIterator getPathIterator(AffineTransform at) {
      return new BañeraPathIterator(at);
   }
   public PathIterator getPathIterator(AffineTransform at, double flatness) {
      return new BañeraPathIterator(at, flatness);
   }

   private enum Parte {EXTERIOR, INTERIOR, SUMIDERO}
   private class BañeraPathIterator implements PathIterator {
      private PathIterator actualIterator;
      private Parte parteActual;
```

```
    private boolean hayFlatness;
    private AffineTransform at;
    private double flatness;
    private BañeraPathIterator(){
        parteActual = Parte.EXTERIOR;
    }
    public BañeraPathIterator(AffineTransform at){
        this();
        hayFlatness = false;
        this.at = at;
        actualIterator=exterior.getPathIterator(at);
    }
    public BañeraPathIterator(AffineTransform at, double flatness){
        this();
        hayFlatness = true;
        this.flatness = flatness;
        this.at = at;
        actualIterator=exterior.getPathIterator(at, flatness);
    }
    public int currentSegment(float[] coords) {
        return actualIterator.currentSegment(coords);
    }
    public int currentSegment(double[] coords) {
        return actualIterator.currentSegment(coords);
    }
    public int getWindingRule() {
        return actualIterator.getWindingRule();
    }
    public boolean isDone() {
        if (actualIterator.isDone()){
            switch(parteActual){
                case EXTERIOR : parteActual = Parte.INTERIOR;
                                if (hayFlatness)
                                    actualIterator = interior.getPathIterator(at, flatness);
                                else
                                    actualIterator = interior.getPathIterator(at);
                                break;
                case INTERIOR : parteActual = Parte.SUMIDERO;
                                if (hayFlatness)
                                    actualIterator = sumidero.getPathIterator(at, flatness);
                                else
                                    actualIterator = sumidero.getPathIterator(at);
                                break;
            }
        }
        return actualIterator.isDone();
    }
    public void next() {
```

```
            actualIterator.next();
        }
    }
}
```

Para obtener el fichero **Bañera.java** basta con yuxtaponer estos dos grandes trozos de código.

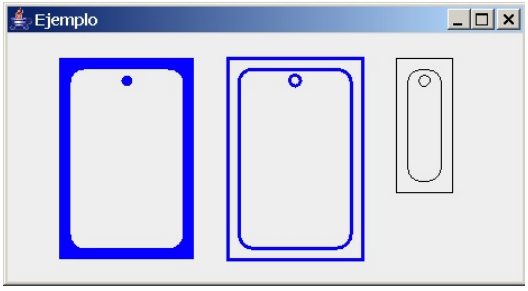


Figura 3.13. Ahora las bañeras pueden dibujarse como cualquier otro objeto Shape

La figura 3.13 muestra el resultado de dibujar diferentes bañeras mediante el código:

```
public void paint (Graphics g) {
        super.paint(g);
        Graphics2D g2 = (Graphics2D)g;
        g2.setColor(Color.blue);
        g2.setStroke(new BasicStroke(3.0f));
        Bañera b1 = new Bañera(50, 50, 120, 180);
        g2.fill(b1);
        Bañera b2 = new Bañera(200, 50, 120, 180);
        g2.draw(b2);
        g2.setColor(Color.black);
        g2.setStroke(new BasicStroke(1.0f));
        Bañera b3 = new Bañera(350, 50, 50, 120);
        g2.draw(b3);
}
```

3.4.3 Áreas Geométricas Constructivas (CAG)

Las áreas geométricas constructivas (*Constructive Area Geometry*, CAG en inglés) son el resultado de crear nuevas figuras geométricas realizando operaciones do conjuntos con figuras ya existentes. En la API de Java2D, una clase heredera de Shape llamada Area es capaz de realizar estas operaciones. El mecanismo de funcionamiento consiste en convertir en un objeto Area cada objeto Shape que deba

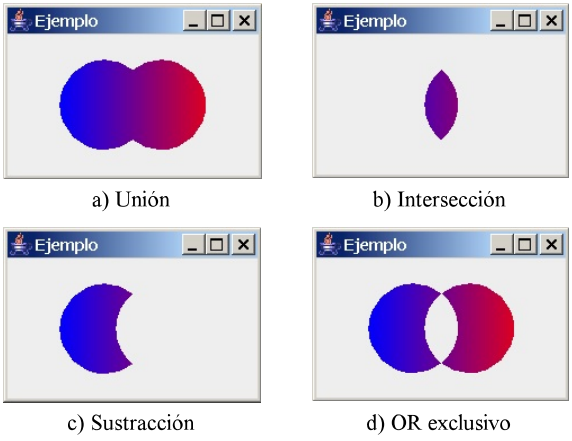
intervenir en la operación de mezcla: esto se hace gracias a un constructor de Area que toma a un objeto Shape como parámetro; a continuación se mezclan tales Areas según la operación u operaciones de conjuntos que se quieran realizar y, por último, dibujar el Area resultante de tales operaciones en el lienzo gráfico, esto es el objeto Graphics2D.

Las operaciones de conjuntos que soportan los objetos de tipo Area son las siguientes:

- Union
- Intersección
- Sustracción
- Unión exclusiva (XOR)

En el siguiente código se va a realizar una unión de 2 círculos, ilustrándose el resultado de aplicar todas estas operaciones. El resultado puede verse en la figura 3.14:

```
public void paint (Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        GradientPaint gp = new GradientPaint(50.0f, 50.0f, Color.blue,
                                             200.0f, 50.0f, Color.red);
        g2.setPaint(gp);
        Ellipse2D e1 = new Ellipse2D.Double(50.0, 50.0, 80.0, 80.0);
        Ellipse2D e2 = new Ellipse2D.Double(100.0, 50.0, 80.0, 80.0);
        Area a1 = new Area(e1);
        Area a2 = new Area(e2);
        // a1.add(a2);
        // a1.intersect(a2);
```



a) Unión b) Intersección

c) Sustracción d) OR exclusivo

Figura 3.14. Ejemplos de operaciones de conjuntos entre objetos de tipos Area

```
                // a1.subtract(a2);
        a1.exclusiveOr(a2);
        g2.fill(a1);
}
```

El resultado visual cambia tan sólo modificando la operación lógica con la que se mezclan los objetos de tipo Area, y para eso sólo hay que llamar al método correspondiente. La única diferencia entre las figuras 3.14.a), 3.14.b), 3.14.c) y 3.14.d) es que en la 3.14.a) se ha utilizado la sentencia a1.add(a2); mientras que en las demás se han usado, respectivamente, a1.intersect(a2);, a1.sustract(a2); y a1.exclusiveOr(a2);.

3.4.3.1 Sherezade en el crepúsculo

La clase Area puede utilizarse para construir fácilmente figuras complejas a partir de otras más simples, como cuadrados o círculos. Como ya se ha visto, para crear una nueva Shape mediante este método, tan sólo hay que seguir dos pasos:

1. Construir las áreas a combinar usando Shapes
2. Llamar a cualquiera de los métodos explicados en el apartado anterior para combinarlas.

Como ejemplo romántico, el código siguiente consigue dibujar una luna menguante en un cielo crepuscular, con tan sólo dos círculos superpuestos formando una nueva figura gracias a una operación de sustracción. El crepúsculo se consigue con un degradado de fondo que va del azul oscuro al negro. El resultado puede verse en la figura 3.15, siendo el código:

```
public void paint (Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
                // Dibujo del cielo. Este es el fondo.
                // Se ha usado un gradiente que va desde el azul al negro
        GradientPaint gp1 = new GradientPaint(10.0f, 125.0f, Color.blue,
                                                250.0f, 125.0f, Color.black);
        Rectangle2D r = new Rectangle2D.Double(0.0, 0.0, 250.0, 250.0);
        g2.setPaint(gp1);
        g2.fill(r);
        // Dibujo de la luna sobre el fondo
        GradientPaint gp2 = new GradientPaint (50.0f, 50.0f, Color.yellow,
                                                300.0f, 50.0f, Color.darkGray);
        g2.setPaint(gp2);
        Ellipse2D e1 = new Ellipse2D.Double(70.0, 100.0, 80.0, 80.0);
        Ellipse2D e2 = new Ellipse2D.Double(100.0, 100.0, 80.0, 80.0);
        Area a1 = new Area(e1);
        Area a2 = new Area(e2);
        a1.subtract(a2);
        g2.fill(a1);
```

```
}
```

Figura 3.15. Vistosa composición obtenida gracias a las clases Area y GradientPaint

Siguiendo el código, el Area a1 era en principio un círculo, pero después de la operación de sustracción ha resultado ser el casquete lunar amarillo tal como se aprecia en la figura 3.15, mientras que a2 permanece tal y como se construyó, ya que la operación de sustracción se ha realizado sobre a1, y es a1 quien se ha dibujado.

Como se puede apreciar, el ejemplo es muy simple -sencillamente son 2 círculos sustraídos- pero lo que se pretende es dar una idea acerca de la potencia que esto tiene: simplemente con imaginación se pueden obtener resultados muy buenos.

3.5 JOGL (Java/OpenGL)

Independientemente de todos los paquetes y clases proporcionados por la API Java2D, e incluso por la Java3D, existe una aproximación diferente que permite dibujar imágenes y figuras en dos y tres dimensiones aprovechando el hardware de aceleración que posea la plataforma en que se ejecuta la aplicación Java. Esta aproximación viene dada por la API JOGL (Java/OpenGL) que encapsula dentro de unas pocas clases Java toda la potencia de la biblioteca OpenGL. De hecho, la API JOGL es básicamente una envoltura JNI (*Java Native Interface*) alrededor de la biblioteca nativa escrita en C para cada plataforma gráfica. Por ello, su utilización no implica utilizar únicamente algunos ficheros **.jar** sino que también es necesario hacer uso de bibliotecas de enlace dinámico propias de cada plataforma (ficheros **.dll** en el caso de MS Windows).

Uno de los problemas que plantea el uso de esta API es que no es realmente orientada a objetos, pues la biblioteca OpenGL no ha sido remodelada y adaptada a Java, sino sólo envuelta por una capa JNI, lo que obliga al programador experto en Java a retroceder en el tiempo y volver a la época de la programación modular o estructurada. Además, JOGL utiliza como base la API AWT en lugar de la Swing, ya que la AWT no es 100% Java y recae en parte sobre la plataforma concreta en que se realizan los renderizados. Este hecho hace difícil realizar aplicaciones que posean una dosis importante de interfaz de usuario, ya que es la API Swing la que más puede

aportar en este aspecto y es de sobra conocida la incompatibilidad entre componentes AWT (*heavyweight*-pesados) y componentes Swing (*lightweight*-ligeros) en una misma interfaz gráfica.

En cualquier caso, resulta importante que el lector conozca la existencia de esta API y la posibilidad de hacer uso de OpenGL directamente desde Java, aprovechando de esta manera toda la potencia aceleradora de las tarjetas gráficas de hoy día. Sin embargo no nos extenderemos más sobre ella puesto que implicaría profundizar en las interioridades de OpenGL y no es ése nuestro objetivo en estas páginas.

Capítulo 4

Tratamiento de texto con Java2D

4.1 Introducción

En este capítulo estudiaremos como gestiona Java2D el texto desde un punto de vista estrictamente gráfico. En otras palabras, para Java2D el texto es sencillamente un tipo especial de figura que puede ser representado y con el que se puede trabajar de diferentes maneras, más o menos parecidas a las disponibles con la interfaz Shape; desde este punto de vista carecen de sentido operaciones como: buscar cadenas de texto, reemplazar textos, etc. Java2D se centra en una infinidad de posibilidades y funcionalidades relativas a las fuentes de texto y, en general, a la escritura sobre una interfaz gráfica.

En primer lugar es necesario aclarar qué es exactamente una fuente de texto, ya que es el concepto principal con el que Java2D maneja las cadenas de caracteres. Pues bien, una fuente de texto es un conjunto completo de caracteres con ciertas características gráficas distintivas, con un tamaño determinado y un cierto tipo de trazado. Por ejemplo, todos los caracteres y símbolos españoles en la fuente *Times New Roman* con un tamaño de 10 puntos y un trazado tipo negrita (*bold* en inglés) conforman una fuente. La fuente define la vista, tamaño y estilo característico (este último puede ser **bold** (negrita), **italic** (cursiva) o **plain** (normal)) con el que se puede dibujar una cadena de texto.

¿Cómo una fuente define su apariencia? La respuesta es que una fuente se crea a partir de rasgos tipográficos (*glyphs* en inglés), y un rasgo puede ser o un mapa de bits o una secuencia de trazos que definen la apariencia de cada uno de los caracteres o símbolos de la fuente. Todas las fuentes dentro de una misma familia de fuentes tienen una apariencia similar porque todas están hechas del mismo conjunto

Figura 4.1. Partes más comunes de un rasgo tipográfico: http://www.fotonostra.com

de rasgos: lo que cambia es el tamaño, el estilo, etc. De esta forma, un mismo carácter, cuando es referenciado por otra familia de fuentes, usa un rasgo diferente que le da su toque distintivo. Los rasgos propios de cada fuente proporcionan una apariencia común y homogénea, esto es, en una fuente *Sans Serif* (Palo Seco en español) ningún rasgo poseerá remates en los extremos de las astas. La figura 4.1 muestra algunas de las características que pueden modificar la apariencia de un rasgo, permitiendo crear familias de fuentes completas.

4.2 ¿Cómo encontrar las fuentes?

Para usar una fuente en Java2D hay que utilizar un objeto de la clase Font, y para ello es necesario saber qué fuentes están disponibles en el sistema operativo y sus nombres. Las fuentes tienen un nombre lógico, un nombre que hace referencia al estilo general al que pertenece (nombre de familia) y el nombre de la fuente en sí:

- El nombre lógico es el que se utiliza para crear la fuente mediante el constructor apropiado. No debe confundirse con el concepto de fuente lógica; una fuente lógica es un concepto que aporta Java2D y que le permite clasificar cualquier fuente concreta en una lógica; las cinco fuentes lógicas de Java2D son: **Serif**, **SansSerif**, **Monospaced**, **Dialog**, and **DialogInput**. Para averiguar el nombre lógico lo único que se debe realizar es una llamada a la función java.awt.Font.getName().
- El nombre de familia proporciona el nombre del diseño tipográfico general, tal como la familia Arial, Helvetica o Times New Roman. Este nombre se obtiene invocando a la función java.awt.Font.getFamily().
- El nombre en sí de la fuente representa una fuente específica dentro de una familia, tal como **Helvetica Negrita**. Casi siempre suele coincidir con el nombre lógico. Para obtener este nombre hay que llamar a la función java.awt.Font.getFontName().

Por último, resulta de indudable utilidad saber todas las fuentes disponibles en un sistema, lo que se consigue invocando a la función getAllFonts() de la clase java.awt.GraphicsEnvironment. Dado que este método no es estático, para poderlo usar debe obtenerse previamente un objeto GraphicsEnvironment para lo que suele usarse la sentencia:

```
GraphicsEnvironment.getLocalGraphicsEnvironment();
```

El bucle:

```
for (Font f: GraphicsEnvironment.getLocalGraphicsEnvironment().getAllFonts())
        System.out.println(f.getName()+" - "+f.getFontName()+" - "+f.getFamily());
```

permite conocer toda la información acerca de las fuentes instaladas en un ordenador.

A modo de resumen, la siguiente tabla muestra los métodos para recuperar

nombres de fuentes ya creadas, así como las fuentes disponibles (todos son métodos de la clase Font excepto el último):

Método	Descripción
getName() getFamily() getFontName()	Devuelve el nombre lógico de la fuente. Devuelve el nombre de la familia de la fuente. Devuelve el nombre en sí de la fuente.
getAllFonts()	Devuelve un array con todas las fuentes existentes en el sistema. Pertenece a la clase GraphicsEnvironment

4.2.1 Crear y derivar fuentes

El camino más sencillo para crear una fuente es utilizar el constructor de la clase Font que toma como parámetros el nombre de la fuente en sí (también vale un nombre de familia), el tamaño en puntos y el estilo. El estilo es una de las constantes Font.PLAIN, Font.BOLD o Font.ITALIC. Una vez creado el objeto de tipo Font es posible derivar de él otros nuevos objetos Font similares llamando al método deriveFont() y especificando un nuevo tamaño, estilo, transformada o mapa con atributos (este último caso no lo veremos en nuestro estudio, pero en el apartado 4.4 introduciremos el concepto de texto con atributos, lo que nos dará una idea de lo que puede hacerse mediante el referido mapa de atributos).

Así pues, una fuente puede crearse con una sentencia tan sencilla como:

```
Font f = new Font("Arial", Font.ITALIC, 24);
```

4.3 Dibujar texto

Dibujar texto dentro de un objeto Graphics2D puede ser una tarea muy sencilla o tener cierta complejidad, en función de las necesidades que tengamos y del control que se quiera ejercitar sobre el renderizado. En concreto, podemos diferenciar entre dibujar una sola línea de texto o un párrafo completo.

4.3.1 Dibujar una única línea de texto

Una vez creada la fuente, la forma más fácil de dibujar una línea de texto en un Graphics2D es utilizar su método drawString() habiendo establecido previamente la fuente actual en el contexto. Un ejemplo sencillo podría ser:

```
public void paint (Graphics g) {
    super.paint(g);
    Graphics2D g2 = (Graphics2D)g;
    g2.setColor(Color.blue);
    g2.setFont(new Font("Arial", Font.ITALIC, 24));
```

```
        g2.drawString("Lex Flavia Malacitana", 20, 60);
}
```

cuyos resultados se observan el la figura 4.2.

Figura 4.2. Ejemplo sencillo de cómo dibujar una línea de texto

Un mecanismo mucho más potente para dibujar texto es el que proporciona la clase TextLayout. Esta clase no sólo permite dibujar un texto de manera parecida a como se ha hecho antes sino que, además, suministra una serie de métodos que permiten conocer características concretas de lo que se va a dibujar. Por ejemplo, con el código anterior nos hemos limitado a escribir un texto mediante una fuente concreta y ya está, pero imaginemos una situación en la que el texto debe salir lo más grande posible pero sin exceder una longitud en puntos determinada. En otras palabras, antes de dibujar el texto queremos saber cuánto espacio va a ocupar, si ocupa más de la cuenta disminuimos el tamaño de la fuente y volvemos a probar, y así hasta que el texto quepa en el hueco apropiado. Esta situación se soluciona fácilmente gracia a la función getBounds() propia de la clase TextLayout.

Así pues, el mismo ejemplo de antes con la clase TextLayout quedaría:

```
public void paint (Graphics g) {
    super.paint(g);
    Graphics2D g2 = (Graphics2D)g;
                FontRenderContext frc = g2.getFontRenderContext();
                Font f = new Font("Arial", Font.ITALIC, 24);
                TextLayout tl = new TextLayout("Lex Flavia Malacitana", f, frc);
                g2.setColor(Color.red);
                tl.draw(g2, 20, 60);
}
```

Nótese cómo el constructor de un objeto TextLayout toma tres parámetros:

- El texto a visualizar.
- La fuente que utilizar.
- Un objeto de tipo FontRenderContext. Esta clase contiene información sobre las transformaciones (rotación, escalado, etc) a realizar sobre el texto y permite al objeto TextLayout construido conocer, entre otras cosas, las dimensiones reales del texto a dibujar. La mejor forma de obtener este objeto es enviando el mensaje getFontRenderContext() al objeto Graphics2D sobre el que se pretende dibujar.

Por último, nótese cómo el dibujo en sí se realiza invocando al método draw() de la clase TextLayout y no al método draw() o drawString() de la clase Graphics2D como veníamos haciendo hasta ahora.

Siguiendo este mismo esquema, el siguiente código muestra una situación más compleja gracias a la clase TextLayout y cuyos resultados pueden observarse en la figura 4.3:

```
public void paint(Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        FontRenderContext frc = g2.getFontRenderContext();
        Dimension tamaño = getSize();
        TextLayout tl;
        Font f;

        f = new Font("Arial", Font.ITALIC, 24);
        tl = new TextLayout("Tamaño 24 Arial Cursiva", f, frc);
        g2.setColor(Color.pink);
        tl.draw(g2, (float)(tamaño.width - tl.getBounds().getWidth())/2,
                tamaño.height/2.0f);

        f = f.deriveFont(Font.BOLD, 16);
        tl = new TextLayout("Tamaño 16 Arial Negrita", f, frc);
        g2.setColor(Color.red);
        tl.draw(g2, (float)(tamaño.width - tl.getBounds().getWidth())/2,
                tamaño.height/3.5f);

        f = new Font("Times New Roman", Font.PLAIN, 20);
        tl = new TextLayout("Tamaño 20 Times New Roman Normal", f, frc);
        g2.setColor(Color.orange);
        tl.draw(g2, (float)(tamaño.width - tl.getBounds().getWidth())/2,
                tamaño.height/1.5f);
}
```

Figura 4.3. Dibujo de texto mediante la clase TextLayout. Esta clase permite conocer las dimensiones del texto a dibujar y, por tanto, centrarlo adecuadamente en los límites de la ventana

En este ejemplo puede observarse cómo se dibuja en tres tipos distintos de fuente. La primera fuente se define con la familia **Arial**, tamaño 24 y estilo cursiva. La segunda fuente se define gracias al método deriveFont() de forma que el objeto Font obtenido de esta manera pertenece a la misma familia que el objeto Font de partida: lo único que es posible cambiar es el tamaño de punto y el estilo de la letra,

como se ha hecho en el código de ejemplo. Finalmente, la tercera fuente creada pertenece a la familia Times New Roman, con un tamaño 20 y un estilo normal.

Las líneas de código que se encargan de crear el texto a pintar son:

```
FontRenderContext frc = g2.getFontRenderContext();
...
f = new Font("Arial", Font.ITALIC, 24);
tl = new TextLayout("Tamaño 24 Arial Cursiva", f, frc);
```

con las que consigue crear un objeto TextLayout, el cual ha sido el causante de que la línea se dibujara gracias a su método draw(). Para poder crear el TextLayout, es necesario modelar primero el contexto de renderizado, el aspecto característico de la fuente y la cadena que se quiere dibujar, para después introducirlos en el constructor. Como ya se ha comentado anteriormente, el objeto FontRenderContext contiene la información necesaria para posicionar y tratar el texto correctamente cuando se renderiza.

El método draw() de TextLayout permite dibujar la cadena de texto sobre la pantalla. Para llamarlo, necesitamos del objeto de tipo Graphics2D sobre el que renderizar y de las coordenadas donde se desea dibujar.

En la siguiente tabla se muestran los métodos implicados en este apartado:

Método	Clase	Descripción
getFontRenderContext()	Graphics2D	Devuelve información acerca del contexto de renderizado.
TextLayout(String s, Font f, FontRenderContext frc)	constructor	Crea un objeto TextLayout a partir de una cadena, una fuente y el contexto de renderizado de texto.
Font(String nombreFuente, int estilo, int tamaño)	constructor	Crea una fuente a partir de un nombre, un estilo de trazado y un tamaño.
draw(Graphics2D, float x, float y)	TextLayout	Dibuja texto sobre Graphics2D. x e y indican el desplazamiento con respecto al eje de coordenadas.

4.3.2 Dibujar un párrafo completo

Para poder dibujar un párrafo completo no se puede usar únicamente la clase TextLayout, ya que ésta sólo escribe una línea de texto. Para un párrafo es necesaria otra clase más, la clase LineBreakMeasurer, con la que se crean y se dibujan líneas de texto (objetos TextLayout) que se ajustan a la anchura que se desee.

En el siguiente código se dibujan varias líneas de texto sobre una ventana, y el resultado puede apreciarse en la figura 4.4:

```
import javax.swing.*;
```

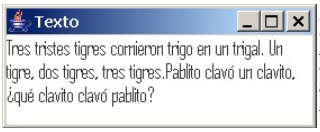


Figura 4.4. Dibujo de un párrafo de texto con una anchura ajustada a la de la ventana

```
import java.awt.*;
import java.util.*;
import java.text.*;
import java.awt.font.*;

public class Texto extends JFrame {
    private static final Hashtable mapa = new Hashtable();

    private static AttributedString trabalenguas;
    static {
        mapa.put(TextAttribute.SIZE, new Float(18.0));
        trabalenguas = new AttributedString(
            "Tres tristes tigres comieron trigo "
            + "en un trigal. Un tigre, dos tigres, tres tigres. "
            + "Pablito clavó un clavito, ¿qué clavito clavó Pablito?",
            mapa);
    }

    public Texto() {
        super("Texto");

        super.setBackground(Color.white);

        PanelIteradorSobreLineas panel = new PanelIteradorSobreLineas();
        add(panel, BorderLayout.CENTER);
    }

    class PanelIteradorSobreLineas extends JPanel {
        private LineBreakMeasurer lineMeasurer;
        private int comienzoParrafo;
        private int finParrafo;

        public PanelIteradorSobreLineas() {
            AttributedCharacterIterator parrafo = trabalenguas.getIterator();
            comienzoParrafo = parrafo.getBeginIndex();
            finParrafo = parrafo.getEndIndex();
            lineMeasurer = new LineBreakMeasurer(parrafo,
                    new FontRenderContext(null, false, false));
        }

        public void paint(Graphics g) {
            Graphics2D graphics2D = (Graphics2D) g;
```

```
                Dimension tamaño = getSize();
                float anchodelinea = (float) tamaño.width;
                float pintaPosY = 0;
                float pintaPosX;
                lineMeasurer.setPosition(comienzoParrafo);
                while (lineMeasurer.getPosition() < finParrafo) {
                    TextLayout layout = lineMeasurer.nextLayout(anchodelinea);
                    pintaPosY += layout.getAscent();
                    if (layout.isLeftToRight())
                        pintaPosX = 0;
                    else
                        pintaPosX = anchodelinea - layout.getAdvance();
                    layout.draw(graphics2D, pintaPosX, pintaPosY);
                    pintaPosY += layout.getDescent() + layout.getLeading();
                }
            }
        }

        public static void main(String[] args) {
            Texto v = new Texto();
            v.setSize(new Dimension(400, 250));
            v.setDefaultCloseOperation(EXIT_ON_CLOSE);
            v.setVisible(true);
        }
}
```

Vamos a analizar el brevemente el programa recién creado. En primer lugar, puede apreciarse perfectamente que el código es totalmente distinto al de los ejemplos en que se dibujaba una única línea y, por ende, más complejo. La variable mapa, es la que permite asignarle los atributos de diseño de fuente al objeto de tipo AttributedString (esta clase se tratará más adelante en el epígrafe 4.4; por ahora basta saber que contiene una cadena de caracteres en la que cada carácter puede tener sus propios atributos de color, fuente, etc.). A través de esta variable es posible cambiar cualquier aspecto de la fuente tipográfica. Un ejemplo de ello es que si, por ejemplo, se sustituye la sentencia:

```
mapa.put(TextAttribute.SIZE, new Float(18.0));
```

por esta otra:

```
mapa.put(TextAttribute.FONT,
        Font.getFont("Fuente", new Font("Vivaldi", Font.PLAIN, 40)));
```

se obtendrá el resultado de la figura 4.5. El resultado queda bien patente: se ha modificado la fuente, el estilo y el tamaño de la misma con tan solo cambiar una línea de código.

En la clase PanelIteradorSobreLineas se ajusta el texto para que después pueda trabajarse sobre él: se crea un iterador que se utiliza para definir los límites del

Figura 4.5. Dibujo de un párrafo de texto con una fuente distinta a la de por defecto. El texto se ajusta a los márgenes de la ventana

texto y para crear un manejador de líneas, lineMeasurer de la clase LineBreakMeasurer, que será quien se encargue de coger las líneas del párrafo, una por una y pintarlas en el objeto Graphics2D.

Finalmente, en el método paint() se realiza el dibujo del texto, teniendo en cuenta las medidas de la ventana y de las líneas, tanto horizontal como verticalmente. Para poder entender el trozo de código completamente es necesario entender ciertos conceptos acerca de las métricas de las fuentes: la distancia llamada *ascent* es la distancia desde la base del carácter a la línea ascendente, es decir, la distancia entre la base del carácter a la altura máxima que pueda tener cualquier símbolo de la misma fuente (por ejemplo la "b" alcanza la altura máxima). La distancia *descent* es la distancia desde la base del carácter hasta la línea descendente, es decir, hasta la profundidad mínima a la que puede descender cualquier símbolo de la fuente (un ejemplo de ello es la letra "g"). Finalmente, la distancia de interlineado (*leading*) es la distancia recomendada desde el fondo de la línea descendente hasta la ascendente de la línea inferior. La figura 4.6 ilustra visualmente estos conceptos.

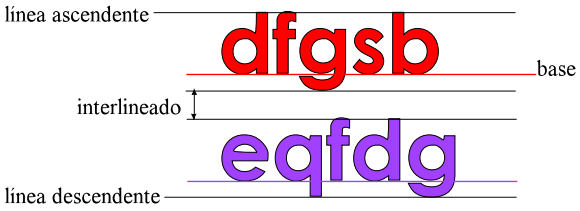


Figura 4.6. Elementos tipográficos a considerar en el renderizado de un párrafo de texto

Haciendo uso de estos componentes tipográficos, se puede observar como la variable lineMeasurer es la encargada de obtener cada una de las líneas, mientras que pintaPosY y pintaPosX son las que se encargan de posicionar la línea que se está gestionando en cada momento: pintaPosY sitúa correctamente la línea verticalmente y pintaPosX se encarga de situar la línea conforme al ancho de línea marcado.

En la siguiente tabla se muestran los métodos implicados en este apartado:

Método	Descripción
AttributedString(String, Map) constructor de clase	Crea una cadena con atributos a partir de un String y un mapa de atributos
TextLayout.getAscent() TextLayout.getDescent()	Devuelve la línea ascendente del texto Devuelve la línea descendente del texto
TextLayout.getLeading() TextLayout.getAdvance()	Devuelve el espacio entre la línea descendente y la línea siguiente Avanza dentro de una línea del párrafo
LineBreakMeasurer(AttributedString, FontRenderContext) constructor de clase	Crea un manejador de texto de más de una línea
LineBreakMeasurer. getPosition()	Obtiene la posición actual del objeto LineBreakMeasurer
LineBreakMeasurer. setPosition(Float x)	Coloca el objeto LineBreakMeasurer en la posición **x** del texto
LineBreakMeasurer. nextLayout(Float x)	Avanza una línea del párrafo, marcada por **x**

4.3.3 Cómo dibujar texto sobre una imagen

Con el siguiente código vamos a estudiar cómo se dibuja texto sobre una imagen cualquiera, obteniendo resultados como el de la figura 4.7.

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

public class Texto extends JFrame {

    public Texto() {
        super("Es el barato");
    }

    public void paint(Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        int w = getSize().width;
        int h = getSize().height;
        // Se dibuja la imagen
        Image im = (new ImageIcon("Maserati.jpg")).getImage();
        g2.drawImage(im,0,0,this);
        // Se prepara el texto
        FontRenderContext frc = g2.getFontRenderContext();
        Font f = new Font("Times",Font.BOLD,w/25);
```

Figura 4.7. Renderización de imágenes y texto. El texto aparece tanto sólido como hueco (sólo el contorno)

```
        String s = new String ("Os presento mi auto");
        TextLayout tl = new TextLayout(s,f,frc);
        float sw = (float)tl.getBounds().getWidth();
        AffineTransform transform = new AffineTransform();
        transform.setToTranslation(w/2-sw/2,h*3/8);
        g2.setColor(Color.yellow);
        // Se dibuja sólo el contorno
        Shape shape = tl.getOutline(transform);
        g2.draw(shape);
        // Se dibuja el texto completo
        tl.draw(g2,w/2-sw/2,h*5/8);
    }

    public static void main(String[] args) {
            Texto v = new Texto();
            v.setSize(445,335);
            v.setDefaultCloseOperation(EXIT_ON_CLOSE);
            v.setVisible(true);
    }
}
```

En este código se puede observar que se dibuja el texto de dos maneras: la primera es la más sencilla, que consiste en la llamada al método draw() de TextLayout tal y como se ha hizo en el epígrafe anterior. En la segunda forma se ha querido obtener un efecto curioso y se ha creado un contorno (objeto Shape) con los caracteres de la cadena de texto a escribir.

En los apartados siguientes se van a seguir mostrando otros posibles efectos y aplicaciones que se pueden implementar con el texto, su contorno, imágenes, simular texto dentro de texto, etc. Como anticipo de ello, el siguiente código hace uso

Figura 4.8. Uso del contorno de un texto para dibujar una imagen en su interior

del contorno de la palabra "Auto" para dibujar una imagen en su interior (figura 4.8); basta sustituir el paint() anterior por el siguiente (que es muy similar):

```
public void paint(Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        int w = getSize().width;
        int h = getSize().height;
        // Se dibuja la imagen
        Image im = (new ImageIcon("Maserati.jpg")).getImage();
        // Se prepara el texto
        FontRenderContext frc = g2.getFontRenderContext();
        Font f = new Font("Times",Font.BOLD,180);
        TextLayout tl = new TextLayout("Auto",f,frc);
        float sw = (float)tl.getBounds().getWidth();
        AffineTransform transform = new AffineTransform();
        transform.setToTranslation(w/2-sw/2,h*5/8);
        Shape shape = tl.getOutline(transform);
        // Se dibuja la imagen en el interior del contorno
        g2.setClip(shape);
        g2.drawImage(im,0,0,this);
        // Se dibuja sólo el contorno
        g2.setColor(Color.YELLOW);
        g2.draw(shape);
}
```

En este código se puede apreciar como se ha construido un *clipping path* con el contorno del área de texto de la palabra "Auto" ubicada en medio de la ventana. En la zona dibujada se puede ver como la imagen queda detrás de la cadena pintada. Como se ve, el efecto es muy vistoso y sencillo de realizar.

4.3.4 Letra cursiva invertida mediante transformaciones

Otra aplicación de Java2D consiste en la posibilidad de crear multitud de estilos de letra aplicando transformaciones y después llamando al método deriveFont() ya comentado anteriormente en el epígrafe 4.2.1. El siguiente código utiliza una transformación para simular la escritura itálica o cursiva invertida (ver figura 4.9):

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

public class Texto extends JFrame {
    public Texto() {
        super("Cursiva");
    }

    public void paint(Graphics g) {
        Graphics2D g2 = (Graphics2D) g;
        int w = getSize().width;
        AffineTransform at = new AffineTransform();
        at.setToTranslation(30, 60);
        AffineTransform inclinarIzquierda = new AffineTransform();
        inclinarIzquierda.shear(0.2, 0.0);
        FontRenderContext frc = g2.getFontRenderContext();
        Font fuente = new Font("Times", Font.BOLD, w / 15);
        String s = new String("Texto pintado normal");
        TextLayout tstring = new TextLayout(s, fuente, frc);
        Font fuenteDerivada = fuente.deriveFont(inclinarIzquierda);
        String str = new String("Texto pintado en cursiva");
        TextLayout tstring2 = new TextLayout(str, fuenteDerivada, frc);
        g2.setColor(Color.BLUE);
        g2.transform(at);
        tstring.draw(g2, 0.0f, 0.0f);
        g2.setColor(Color.RED);
        g2.transform(at);
        tstring2.draw(g2, 0.0f, 0.0f);
    }

    public static void main(String[] args) {
        Texto v = new Texto();
        v.setSize(481, 150);
        v.setDefaultCloseOperation(EXIT_ON_CLOSE);
        v.setVisible(true);
    }
}
```

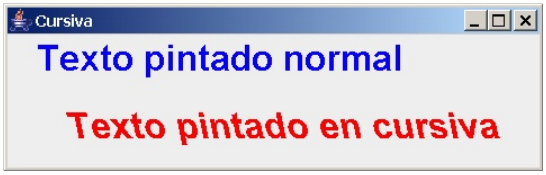


Figura 4.9. Escritura normal y cursiva invertida (en rojo) mediante transformaciones

Como se ha dicho antes, pueden conseguirse multitud de efectos cambiando únicamente la transformación aplicada, así que la única dificultad que esto entraña es la de echar imaginación al estilo; el resto, es coser y cantar.

4.3.5 Rellenar un *clipping path* con distintos objetos

Otro efecto más que se va a implementar es el de conseguir efectos en el texto con figuras geométricas o caracteres. En el código siguiente se rellena el interior de un texto (convirtiendo su contorno en la porción a dibujar) con un color rosa, para luego dibujar unas líneas amarillas sobre el mismo separadas entre sí (ver figura 4.10); el fondo se ha puesto azul para resaltar el rayado del texto:

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

public class Texto extends JFrame {
    public Texto() {
        super("Efecto");
        super.setBackground(Color.BLUE);
    }
    public void paint(Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        int w = getSize().width;
        int h = getSize().height;
        FontRenderContext frc = g2.getFontRenderContext();
        Font f = new Font("Times",Font.BOLD,w/15);
        String s = new String ("Programando con Java2D");
        TextLayout tl = new TextLayout(s,f,frc);
        float sw = (float)tl.getBounds().getWidth();
        AffineTransform transformada = new AffineTransform();
        transformada.setToTranslation(w/2-sw/2,h/1.5);
        Shape shape = tl.getOutline(transformada);
        // Se establece la porción a dibujar en el contorno del texto
        g2.setClip(shape);
        // La porción a dibujar se rellena de rosa
```

Figura 4.10. Contorno de texto usado como porción a dibujar y rellenado del mismo con un fondo rosa y líneas amarillas alternadas

```
        g2.setColor(Color.PINK);
        g2.fill(shape.getBounds2D());
        // Se dibujan líneas en amarillo separadas entre sí
        g2.setColor(Color.YELLOW);
        g2.setStroke(new BasicStroke(2.0f));
        for (    double j = shape.getBounds().getY();
                 j<shape.getBounds2D().getY()+shape.getBounds2D().getHeight();
                 j=j+4) {
            Line2D linea = new Line2D.Double(0.0,j,w,j);
            g2.draw(linea);
        }

    }
    public static void main(String[] args) {
            Texto v = new Texto();
        v.setSize(481,150);
            v.setDefaultCloseOperation(EXIT_ON_CLOSE);
            v.setVisible(true);
    }
}
```

Otro curioso efecto más es el que se muestra en el código que viene a continuación, en el que se ha dibujado repetidamente el carácter "*" en el interior de una cadena de texto cualquiera; la figura 4.11 muestra el resultado aunque hay que fijarse con cuidado para poder distinguir dicha secuencia de asteriscos. En este caso sólo se va a mostrar el método paint(), la clase es la misma que en el caso anterior (sólo que el fondo se ha puesto negro para resaltar las letras):

```
public void paint(Graphics g) {
    Graphics2D g2 = (Graphics2D) g;
    int w = getSize().width;
    int h = getSize().height;

    FontRenderContext frc = g2.getFontRenderContext();
    Font f = new Font("Times", Font.BOLD, w / 15);
    String s = new String("Programando con Java2D");
    TextLayout tl = new TextLayout(s, f, frc);
```

```
        float sw = (float) tl.getBounds().getWidth();

        AffineTransform transformada = new AffineTransform();
        transformada.setToTranslation(w / 2 - sw / 2, h / 1.5);
        Shape shape = tl.getOutline(transformada);

        g2.setClip(shape);
        g2.setColor(Color.blue);
        g2.fill(shape.getBounds2D());
        g2.setColor(Color.white);

        Font f2 = new Font("Times", Font.BOLD, w / 12);
        String s2 = new String("*");
        tl = new TextLayout(s2, f2, frc);
        sw = (float) tl.getBounds().getWidth();

        Rectangle2D r = shape.getBounds2D();
        double x = r.getX();
        double y = r.getY();

        while (y < (r.getY() + r.getHeight() + tl.getAscent())) {
            tl.draw(g2, (float) x, (float) y);
            if ((x += sw) > (r.getX() + r.getWidth())) {
                        x = r.getX();
                        y += tl.getAscent();
            }
        }
}
```

Figura 4.11. Utilización del contorno de un texto como porción a dibujar más texto

Es otro efecto de los muchos más que pueden conseguirse. Con estas demostraciones lo que se pretende es apelar a la creatividad del usuario. Nótese la facilidad y rapidez con la que pueden obtenerse vistosos efectos, sencillamente cambiando pocas líneas de código. Java2D ofrece muchas posibilidades en este sentido y podemos aprovecharlas con imaginación dedicando un tiempo a pensar y reflexionar sobre qué se quiere conseguir.

4.4 Manejo de la clase TextAttribute

En este apartado abordaremos la clase TextAttribute. La clase TextAttribute sirve para definir atributos de texto que luego se pueden aplicar a toda una cadena del tipo AttributedString, o bien a sólo una porción de ella. La clase AttributedString ya se introdujo de manera preliminar en el apartado 4.3.2 cuando dibujamos un párrafo completo. En el siguiente código se realizan algunos efectos gracias a la modificación de varios atributos de esta clase (ver figura 4.12):

```
import javax.swing.*;
import java.awt.*;
import java.text.*;
import java.awt.font.*;
public class Texto extends JFrame {
    public Texto() {
        super("Ejemplo de AttributedString");
        super.setBackground(Color.white);
    }
    public void paint(Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                RenderingHints.VALUE_ANTIALIAS_ON);
        g2.setRenderingHint(RenderingHints.KEY_RENDERING,
                RenderingHints.VALUE_RENDER_QUALITY);
        g2.setColor(Color.blue);
        String texto = "El último mohicano";
        // Crea el AttributedString
        AttributedString ast = new AttributedString (texto);
        // Le pone un tipo de letra determinado
        // (excepto a los dos últimos caracteres que usan la fuente por defecto)
        Font f = new Font("Arial", Font.ITALIC, 15);
        ast.addAttribute(TextAttribute.FONT,f,0,texto.length()-2);
        // Sustituye dos caracteres por dibujos de una taza
        Image im = (new ImageIcon("taza.jpg")).getImage();
        ImageGraphicAttribute iga = new ImageGraphicAttribute (im,
                                    (int)CENTER_ALIGNMENT);
        ast.addAttribute(TextAttribute.CHAR_REPLACEMENT,iga,4,5);
        ast.addAttribute(TextAttribute.CHAR_REPLACEMENT,iga,11,12);
        // Cambia algunos caracteres por otro tipo de letra
        f = new Font("Times", Font.ITALIC, 24);
        ast.addAttribute(TextAttribute.FONT,f,3,9);

        AttributedCharacterIterator it = ast.getIterator();
        FontRenderContext frc = g2.getFontRenderContext();
        TextLayout tl = new TextLayout(it,frc);
        tl.draw(g2, 80, 70);
```

```
    }
    public static void main(String[] args) {
        Texto v = new Texto();
        v.setSize(400,150);
        v.setDefaultCloseOperation(EXIT_ON_CLOSE);
        v.setVisible(true);
  }
}
```

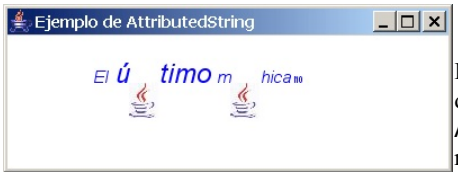


Figura 4.12. Modificación de la apariencia de una cadena de caracteres (objeto AttributedString) mediante la aplicación reiterada de objetos TextAttribute

En el código anterior puede observarse que con el método addAttribute() se pueden realizar varios efectos. En concreto se ha establecido una fuente **Arial cursiva** al texto completo excepto a los dos últimos caracteres (que quedan con la fuente por defecto); se ha realizado una sustitución del cuarto y undécimo caracteres por un icono. Por último, también puede apreciarse una modificación en la fuente del texto para la palabra "último".

En la siguiente tabla se muestran todos los atributos susceptibles de modificación:

Atributo	Descripción del atributo
TextAttribute.BACKGROUND	Adornar el fondo del objeto Paint
TextAttribute.BIDI_EMBEDDING	Ejecución bidireccional de caracteres
TextAttribute.CHAR_REPLACEMENT	Mostrar un rasgo tipográfico definido por el usuario en lugar de un carácter
TextAttribute.FAMILY	Localizar el nombre de la familia de la fuente
TextAttribute.FONT	Fuente con la que se renderizará el texto
TextAttribute.FOREGROUND	Adornar el primer plano del objeto Paint
TextAttribute.INPUT_METHOD_HIGHLIGHT	Métodos de entrada de los estilos de enfatizado
TextAttribute.INPUT_METHOD_UNDERLINE	Métodos de entrada de los adornos de subrayado
TextAttribute.JUSTIFICATION	Justificación de un párrafo
TextAttribute.NUMERIC_SHAPING	Convertir números decimales en ASCII a otros rangos decimales
TextAttribute.POSTURE	Definir la postura de una fuente
TextAttribute.RUN_DIRECTION	Definir el sentido de la escritura
TextAttribute.SIZE	Determinar el tamaño de la fuente
TextAttribute.STRIKETHROUGH	Adornar la sobreimpresión
TextAttribute.SUPERSCRIPT	Definir subíndices y superíndices

TextAttribute.SWAP_COLORS	Intercambiar colores de fondo y primer plano
TextAttribute.TRANSFORM	Transformaciones de una fuente
TextAttribute.UNDERLINE	Subrayar de la línea
TextAttribute.WEIGHT	Definir el peso de una fuente
TextAttribute.WIDTH	Definir el ancho de una fuente

Capítulo 5

Imágenes con Java2D

5.1 Introducción

Una imagen es, básicamente, un *array* bidimensional de puntos de colores a cada uno de los cuales se le llama píxel. Aunque la superficie sobre la que se dibuja la imagen también está compuesta por píxeles, cada píxel de la imagen no tiene porque corresponderse directamente con un píxel de la superficie de dibujo, ya que puede existir un cambio de escala con el que la imagen original se dibuje en pantalla más grande o más pequeña de lo que es realmente; por tanto son conceptos distintos. Una imagen tiene un ancho y un largo expresados en píxeles y, por el motivo anterior, un sistema de coordenadas independiente de cualquier superficie de dibujo.

En los siguientes apartado veremos las posibilidades básicas que ofrece Java2D para trabajar con imágenes, incluyendo cambios de brillo, traslaciones, giros, tratamiento de las componentes de color, difuminaciones, suavizados, filtros, etc.

5.2 Obtención de imágenes

Antes de introducirse en el tratamiento de imágenes con Java2D es necesario recapitular un poco y analizar el tratamiento en general de las imágenes en Java: realizar un pequeño repaso a través de todos los pasos necesarios para entender bien lo que está entre manos en este capítulo.

Una imagen se puede cargar desde una URL o desde un fichero, además de poder ser creada de la nada. A continuación se verán estas operaciones.

5.2.1 Cargar una imagen

El método usado para cargar una imagen desde una URL o desde un fichero es el método `getImage()` del paquete `java.awt.Toolkit`. Para poder usar esta función hay que obtener antes el objeto `Toolkit` por defecto invocando a la función estática `Toolkit.getDefaultToolkit()`.

El siguiente código muestra cómo se carga una imagen desde un archivo ubicado en el mismo directorio que la clase Java (el resultado puede verse en la figura 5.1):

```
import java.awt.*;
import javax.swing.*;
public class Imagenes extends JFrame {
      public Imagenes() {
            super("Abrir Imagen");
      }
      public void paint(Graphics g) {
            Graphics2D g2 = (Graphics2D)g;
            Image im = Toolkit.getDefaultToolkit().getImage("lenguajes.jpg");
            g2.drawImage(im, 0, 20, this);
      }
      public static void main (String args[]) {
            Imagenes v = new Imagenes();
            v.setDefaultCloseOperation(EXIT_ON_CLOSE);
            v.setSize(405,450);
            v.setVisible(true);
      }
}
```

Figura 5.1. Carga de una imagen ubicada en el sistema de ficheros local

Lo más interesante de este código son las sentencias:

```
Image im = Toolkit.getDefaultToolkit().getImage("lenguajes.jpg");
g2.drawImage(im, 0, 20, this);
```

Como ya se ha dejado entrever, la primera sentencia es para cargar imágenes desde un archivo; para ello se llama al método getImage() de la clase Toolkit, que es una clase abstracta. Por este motivo, para obtener una instancia de ella se invoca al método getDefaultToolkit() para obtener la implementación por defecto de Toolkit apropiada a la plataforma de ejecución Java que se está utilizando. Realmente la sentencia getImage() no carga inmediatamente la imagen sino que la mete en una cola asociada al Toolkit por defecto; cuando el Toolkit finaliza la carga vuelve a invocar al

repintado del panel. En otras palabras, la función paint() es invocada varias veces hasta que la imagen se acaba de cargar.

La segunda sentencia es la encargada de renderizar la imagen en sí y se explicará en apartados posteriores de este tema.

5.2.2 Cargar una imagen como BufferedImage

Para trabajar con imágenes en formato JPEG Java2D incluye el paquete sun.com.image.codec.jpeg que permite un tratamiento exhaustivo de la imágenes en este formato. Gracias a sus clases es posible crear una imagen a partir de un flujo de datos de entrada en formato JPEG (decodificar) o escribir los datos de una imagen en una imagen JPEG (codificar). El conjunto de las dos operaciones de codificar/descodificar es lo que se denomina un códec.

Aunque las imágenes en JPEG pueden cargarse como cualesquiera otras produciendo un objeto de tipo Image, hay ciertas operaciones de transformación que requieren un objeto de tipo BufferedImage y, para obtenerlo, la imagen JPEG se puede cargar mediante el mencionado códec. Para cargar una imagen es necesario un decodificador, mientras que para guardarla es necesario un codificador. La clase JPEGCodec contiene los métodos que realizan estas operaciones:

- public static JPEGImageDecoder
  createJPEGDecoder(InputStream fuente)
  Este método devuelve un objeto JPEGImageDecoder construido a partir de los datos leídos desde el flujo de entrada definido por la variable fuente y que tienen que conformar una imagen en formato JPEG.
- public static JPEGImageEncoder
  createJPEGEncoder(OutputStream dest)
  Este método devuelve un objeto JPEGImageEncoder que puede escribir datos de imagen en formato JPEG al flujo de salida definido por la variable dest.

El que nosotros más vamos a usar es el método createJPEGDecoder(). Una vez que, mediante esta función, se obtiene un objeto JPEGImageDecoder, se puede leer la imagen invocando a su método:

public BufferedImage decodeAsBufferedImage()
throws IOException, ImageFormatException

que producirá el deseado objeto BufferedImage (que hereda de Image). Si la imagen no tiene el formato correcto, se producirá una ImageFormatException, mientras que si ocurre cualquier otro problema de E/S, se lanzará una IOException.

El siguiente código muestra cómo abrir una imagen JPEG mediante este método (el resultado se muestra en la figura 5.2); este código es muy similar al del epígrafe anterior:

```
import java.awt.*;
import java.awt.image.*;
import java.io.*;
import javax.swing.*;
import com.sun.image.codec.jpeg.*;
public class Imagenes extends JFrame {
    public Imagenes() {
        super("Abrir Imagen");
    }
    public void paint(Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        try {
            InputStream in =
                getClass().getResourceAsStream("../neuschwanstein.jpg");
            JPEGImageDecoder decoder = JPEGCodec.createJPEGDecoder(in);
            BufferedImage image = decoder.decodeAsBufferedImage();
            g2.drawImage(image, 0, 20,this);
            in.close();
        }
        catch (IOException e) {e.printStackTrace();}
        catch (ImageFormatException e) {e.printStackTrace();}
    }
    public static void main (String args[]) {
        Imagenes v = new Imagenes();
        v.setDefaultCloseOperation(EXIT_ON_CLOSE);
        v.setSize(375,300);
        v.setVisible(true);
    }
}
```

Figura 5.2. Imagen JPEG abierta como BufferedImage. Se trata del castillo de Neuschwanstein en Baviera, Alemania

En cualquier caso, un objeto de tipo Image cualquiera puede convertirse en una imagen mediante el trozo de código:

```
// Convierte un objeto Image denominado im en BufferedImage
BufferedImagebufferedImage = new BufferedImage(
```

```
        im.getWidth(this), im.getHeight(this),BufferedImage.TYPE_INT_ARGB);
    Graphics2D bufImageGraphics = bufferedImage.createGraphics();
    bufImageGraphics.drawImage(im, 0, 0, null);
```

Antes de ejecutar este bloque de código hay que asegurarse de que el objeto Image verdaderamente existe, esto es, si la imagen se ha cargado mediante el Toolkit, entonces éste lo habrá encolado y tardará un tiempo en realizarse la carga completa. Esta verificación puede hacerse de diferentes formas, aunque la más usual es utilizar un objeto de tipo ImageObserver que esté pendiente de la evolución de la carga.

En cualquier caso, la forma más fácil de cargar una imagen como objeto BufferedImage en lugar de Image consiste en hacer uso de los métodos:

public static BufferedImage read (File input) throws IOException
public static BufferedImage read (URL input) throws IOException

que pertenecen a la clase javax.imageio.ImageIO y que, a diferencia de la carga con un Toolkit, no finalizan hasta que la carga de la imagen se ha completa o se produce una excepción. Haciendo uso de esta clase, la función paint() del ejemplo anterior quedaría:

```
public void paint(Graphics g) {
    Graphics2D g2 = (Graphics2D)g;
    try {
        BufferedImage image = ImageIO.read(new File("neuschwanstein.jpg"));
        g2.drawImage(image, 0, 20, this);
    } catch (IOException e){ e.printStackTrace(); }
}
```

Para finalizar, cuando se va a construir una aplicación Java y ésta se va a comprimir dentro de un fichero **.jar** (*Java Archive*), no debe hacerse referencia directa a los ficheros contenidos en el **.jar**, sino que hay que referenciarlos mediante la función getResource() de la clase Class.

5.2.3 Crear imágenes

Hay varias formas para crear imágenes, pero el camino más fácil es llamando al método createImage() de la clase Toolkit. Este método posee dos parámetros de tipo entero que indican, respectivamente, el ancho y el largo de la imagen vacía que se quiere crear. En la API Java2D ese objeto devuelto puede convertirse sin ningún problema en un objeto BufferedImage con un simple *casting* (forzado de tipo).

En el apartado anterior ya se ha introducido la clase BufferedImage, pero ¿qué es exactamente una BufferedImage? Esta clase es una extensión de java.awt.Image que se encuentra en el paquete java.awt.image, por lo que permite acceso directo a todos los métodos de un objeto Image. Por otro lado, una BufferedImage siempre tiene los datos de la imagen accesibles por lo que nunca será necesario tratar con la clase ImageObserver, cosa que no ocurre con la Image. En resumen, la clase BufferedImage de Java2D ofrece un manejo mucho más simple y

eficiente en el tratamiento de imágenes, y añade métodos para realizar operaciones diversas sobre las imágenes. Esta clase se estudiará más a fondo en apartados posteriores; ahora continuaremos con el aprendizaje de otros conceptos.

En el siguiente código se puede observar cómo se crea una imagen a partir del método createImage(). A primera vista, parece exactamente lo mismo que cargar una imagen con getImage():

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

public class Imagenes extends JFrame {
    private Image im;
    public Imagenes() {
        super("Abrir Imagen");
    }
    public void paint(Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        if (im == null)
            im = Toolkit.getDefaultToolkit().createImage("lenguajes.jpg");
        g2.drawImage(im,0,20,this);
        System.out.println("Hola"+im.getHeight(this));
    }
    public static void main (String args[]) {
        Imagenes v = new Imagenes();
        v.setDefaultCloseOperation(EXIT_ON_CLOSE);
        v.setSize(405,450);
        v.setVisible(true);
    }
}
```

El resultado es elmismo que el de la figura 5.1. La diferencia fundamental entre getImage() y createImage() consiste en que getImage() mantiene un *buffer* en el que almacena todas las imágenes que se le manda cargar. Por ello, si se hace varias veces un getImage("lenguajes.jpg"), el sistema no intentará cargar una y otra vez la misma imagen sino que, al reconocer que se encuentra en proceso de carga, o bien ya cargada en el *buffer*, continúa el proceso de carga o bien la retorna respectivamente. Sin embargo, createImage() intenta cargar la imagen cada vez que es invocada, y de ahí la sentencia:

```
if (im == null)
        im = Toolkit.getDefaultToolkit().createImage("lenguajes.jpg");
```

ya que si simplemente cambiásemos el getImage() del epígrafe 5.2.1 por un createImage(), el sistema no pararía nunca al intentar cargar una y otra vez la imagen sin que le dé tiempo entre una y otra llamada al sistema de repintado.

En la tabla siguiente se muestran los métodos para crear y cargar imágenes en una aplicación Java. El método createImage() es de la clase Component,

mientras que dos métodos CreateCompatibleImage() son pertenecientes a la clase java.awt.GraphicsConfiguration. Como ya se ha comentado, esta última clase representa una colección de propiedades de alguno de los dispositivos de salida.

Método	Descripción
public Image createImage(int ancho, int largo)	Crea una imagen con un ancho y un largo definidos
public abstract BufferedImage createCompatibleImage (int ancho, int largo)	Crea una imagen que es compatible con el GraphicsConfiguration actual. Una imagen compatible es aquélla que tiene la misma estructura de datos y color que el dispositivo sobre el que actúa, por lo que será renderizada de manera eficiente
public abstract BufferedImage createCompatibleImage (int ancho, int largo, int transparencia)	Igual que el método anterior, pero con la diferencia de que se introduce un grado de transparencia (canal alfa)

5.3 Mostrar una imagen

Antes de aprender como mostrar imágenes en Java2D, es necesario saber el concepto que implica un objeto ImageObserver. Un objeto de esta clase es necesario ya que Java considera que puede tomar un tiempo grande el hecho de cargar una imagen desde la red o incluso desde el sistema de ficheros local. Pensando en esto, los métodos drawImage() de las clases Graphics y Graphics2D soportan el concepto del ImageObserver, que no es más que una interfaz para monitorizar el progreso de la carga de datos de una imagen mediante su función imageUpdate(). Cuando se invoca al método drawImage() para renderizar una imagen, se dibujan todos los datos de la imagen que están disponibles en ese preciso instante y se retorna inmediatamente. Si no existen datos disponibles de la imagen, el método retorna inmediatamente, aunque en un hilo (*thread*) distinto se sigue cargando la imagen. El método drawImage() toma como parámetro un objeto ImageObserver, al cual se le va informando a medida que los datos de la imagen van siendo cargados. Dado que un JFrame hereda de ImageObserver e invocamos a drawImage() con **this** como parámetro, la funciónn imageUpdate() que implementa el JFrame llama al método repaint(), el cual volverá a llamar a drawImage(), y así sucesivamente hasta que la imagen se cargue del todo y sea visualizada definitivamente.

La siguiente tabla muestra los métodos drawImage() de Graphics2D (y por ende lo de Graphics, de quien hereda).

Método	Descripción
public boolean drawImage(Image im, int x, int y, ImageObserver iob)	Renderiza la imagen proporcionada en las coordenadas (**x**, **y**). La variable **iob** es notificadaa medida que la imagen se carga. Si alguna parte de la

	imagen es transparente, el color del fondo de la superficie sobre la que se pinta es el que se verá a través
public boolean drawImage(Image im, int x, int y, Color colorgb, ImageObserver iob)	Este método es el mismo que el anterior, pero se diferencia en que se especifica el color del fondo que se verá en las porciones transparentes de la imagen
public boolean drawImage(Image im, int x, int y, int ancho, int largo, ImageObserver iob)	Renderiza la imagen en escala para que pueda introducirse en el rectángulo definido por **x**, **y**, **ancho** y **largo**. El algoritmo que se aplica para hacer la escala es el de la interpolación bilinear o el del vecino más cercano, que se explicarán más adelante
public boolean drawImage(Image im, int x, int y, int ancho, int largo, Color colorgb, ImageObserver iob)	Este método es el mismo que el anterior, pero definiendo el color de fondo para las zonas transparentes
public boolean drawImage(Image img, int dx1, int dy1, int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, ImageObserver iob)	Renderiza una porción de la imagen dentro de un rectángulo, escalándola si es necesario. Las variables **sx1**, **sy1**, **sx2** y **sy2** definen la esquina superior derecha y la esquina inferior izquierda de la porción escogida en el sistema de coordenadas propio de la imagen, mientras que las variables **dx1**, **dy1**, **dx2**, **dy2** definen, en el sistema de coordenadas del espacio de usuario, las esquinas superior-izquierda e inferior-derecha del rectángulo donde la porción escogida se renderizará
public boolean drawImage(Image img, int dx1, int dy1, int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, Color colorgb, ImageObserver iob)	Este método es el mismo que el anterior, pero definiendo el color de fondo para las partes transparentes.
public void drawImage(Image im, AffineTransform at, ImageObserver iob)	Renderiza una imagen después de aplicarle una transformación
public void drawImage(BufferedImage img, BufferedImageOp op, int x, int y)	Renderiza una versión procesada (mediante el objeto BufferedImageOp) de una BufferedImage. La imagen original no cambia

5.4 La clase MediaTracker

La clase java.awt.MediaTracker es una clase útil, ya que realiza parte del

trabajo sucio a que oblliga el concepto de ImageObserver. Puede monitorizar el progreso de cualquier número de imágenes, de manera que se puede esperar a que todas las imágenes se carguen o a que se carguen sólo algunas específicas. Además, se pueden comprobar errores. El constructor es el siguiente:

```
public MediaTracker(Component comp)
```

Este constructor crea un nuevo MediaTracker para las imágenes que serán mostradas en la variable comp, aunque realmente este parámetro no resulta especialmente importante.

Para cargar imágenes, un objeto MediaTracker dispone del método siguiente:

```
public void addImage(Image image, int id)
```

Este método añade la imagen image a la lista de imágenes por cargar del objeto MediaTracker que recibe el mensaje. La variable id sirve como número de identificación de la imagen a la hora de usar otros métodos concernientes a esta clase. Nótese que puede asignarse un mismo número de identificación a más de una imagen. Para eliminar una imagen de la lista de imágenes del MediaTracker se usa el método:

```
public void removeImage(Image image)
```

Resulta muy interesante el método:

```
public void waitForID(int id) throws InterruptedException
```

puesto que hace que se espere a que se cargue totalmente la imagen o imágenes seleccionadas con el número de identificación id . Si algo va mal y existe un error, se puede consultar al siguiente método, que devuelve **true** si se ha producido un error en la carga de l imagen asociada con el número id:

```
public boolean isErrorID(int id)
```

5.5 Dibujar sobre imágenes

Hasta ahora se ha tratado la imagen como algo que se renderiza sobre una superficie de dibujo, pero la propia imagen también se puede tratar como una superficie de dibujo en sí misma. Esto puede servir para dibujar sobre ellas y crear así imágenes nuevas. La clase Image provee métodos para este propósito, que son:

- public abstract Graphics getGraphics(): este método devuelve la instancia de Graphics que renderiza a la imagen que lo llama. Este método solo es soportado por imágenes creadas de fondo. Este método está obsoleto, ya que devuelve una instancia de Graphics (aunque se le puede hacer un *casting* a Graphics2D).
- public Graphics2D createGraphics(): este método sirve para representar una BufferedImage como una superficie de dibujo.

  En el siguiente código se utiliza una BufferedImage a la que se le crea un

Graphics2D con createGraphics(); este Graphics2D se usa como superficie de dibujo sobre la que renderiza una imagen y luego se pintan unas líneas de color verde. Con esto se muestra que todo lo que se ha dibujado en capítulos anteriores es posible dibujarlo sobre una imagen cualquiera. El quid de la cuestión estriba en obtener el objeto Graphics2D asociado a un BufferedImage. El resultadopuede verse en la figura 5.3.

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

public class Imagenes extends JFrame {
    public Imagenes() {
        super("Imagen de fondo");
    }

    public static void main(String[] args) {
        Imagenes v = new Imagenes();
        v.setDefaultCloseOperation(EXIT_ON_CLOSE);
        v.setSize(375, 300);
        v.setVisible(true);
    }

    public void paint(Graphics g) {
        BufferedImage mImagen;
        Graphics2D g2 = (Graphics2D)g;
        Dimension d = getSize();
        int a = d.width, l = d.height;
        mImagen = new BufferedImage(a, l, BufferedImage.TYPE_INT_RGB);
        Graphics2D gOculta = mImagen.createGraphics();
        gOculta.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                RenderingHints.VALUE_ANTIALIAS_ON);
        Image im = Toolkit.getDefaultToolkit().getImage("neuschwanstein.jpg");
        gOculta.drawImage(im,0,20,this);
        gOculta.setStroke(new BasicStroke(1.5f));
        Color[] colors = { Color.red, Color.blue, Color.green };
        gOculta.setPaint(Color.green);
        for (int i = -32; i < 40; i += 8) {
            gOculta.drawLine(i, i, a - i * 2, l - i * 2);
            gOculta.rotate(0.05f);
        }
        g2.drawImage(mImagen, 0, 0, this);
    }
}
```

Como se puede apreciar en este código, los pasos a seguir son:

- Crear un objeto BufferedImage nuevo.

- Llamar a su método createGraphics().
- Establecer su configuración gráfica.
- Dibujar sobre el objeto Graphics2D lo que se desee.
- Pintar la BufferedImage en el panel del JFrame.

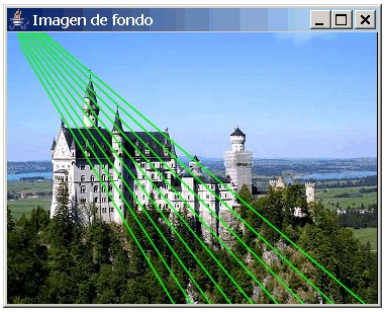


Figura 5.3. Creación de una nueva imagen (BufferedImage) mediante el renderizado de imágenes y figuras geométrica sobre ella

5.5.1 *Double Buffering* (dibujo en memoria)

Una aplicación muy importante de dibujar sobre una imagen es la técnica del *double buffering*, que permite eliminar el parpadeo en pantalla cuando se suceden rápidamente muchas imágenes (ligeramente diferentes) para dar la sensación de animación. En otras palabras, esta técnica elimina el parpadeo en una animación gráfica. Para ello, en el *double buffering* existen dos áreas gráficas, una activa (en *on*, la que se ve e pantalla) y otra pasiva (en *off*, que sólo existe en memoria). Como la animación es el proceso de mostrar una imagen detrás de otra (fotograma), pues si no se emplea ninguna técnica especial sino que, sencillamente, se limpia la superficie de dibujo y se muestra después otra imagen, se produce un parpadeo cuando esto se hace repetidamente. El *double buffering* elimina el parpadeo dibujando el siguiente fotograma en memoria, esto es, en la imagen que está en *off* y, una vez dibujada al completo, se transfiere en un solo paso a la imagen que está activa. Dado que esta transferencia es sumamente veloz se elimina cualquier efecto visual indeseado entre la transición de un fotograma a otro.

En el siguiente código se muestra una aplicación de la técnica del *double buffering*. En él se puede arrastrar un rectángulo rojo mediante el puntero del ratón. Gracias a que se pintan dos imágenes, una delante y otra detrás, se podrá observar que no se produce parpadeo alguno al dibujar el cuadrado. Cada vez que se mueve el ratón se invoca al método repaint(), el cual volverá a crear el nuevo rectángulo. La figura 5.4 muestra el aspecto visual del programa.

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

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

public class Imagenes extends JFrame
implements MouseMotionListener {
    private int mX, mY;
    public static void main(String[] args) {
        Imagenes v = new Imagenes();
        v.setDefaultCloseOperation(EXIT_ON_CLOSE);
        v.setSize(405,450);
        v.setVisible(true);
    }
    public Imagenes() {
        super("Suavizado con doble Buffering");
        addMouseMotionListener(this);
    }
    public void mouseDragged(MouseEvent me) {
        mX = (int)me.getPoint().getX();
        mY = (int)me.getPoint().getY();
        repaint();
    }
    public void mouseMoved(MouseEvent me) {}
    public void pintarEnOffscreen(Graphics2D g2) {
       int s = 100;
       g2.setColor(Color.red);
       Rectangle2D r2 = new Rectangle2D.Float ((float)mX - s / 2, (float)mY - s/2,
                                               (float)s, (float)s);
       g2.fill(r2);
    }
    public void paint(Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        Dimension d = getSize();
                Image mImagen = createImage(d.width, d.height);
                Graphics2D offG = (Graphics2D)mImagen.getGraphics();
        offG.setColor(Color.blue);
```

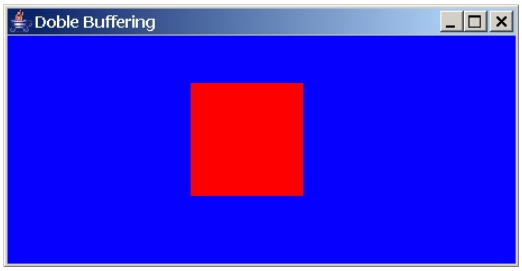


Figura 5.4. Técnica de *double buffering*. El cuadrado rojo se puede arrastrar y soltar mediante el uso del ratón

```
            offG.fill(new Rectangle2D.Double(0, 0, d.width, d.height));
            pintarEnOffscreen(offG);
            g2.drawImage(mImagen, 0, 20, null);
        }
}
```

5.6 Procesamiento de imágenes

En este epígrafe vamos a describir cómo las imágenes pueden manipularse matemáticamente adentrándonos en las distintas operaciones que Java2D pone a nuestra disposición. El procesamiento de imágenes también puede denominarse filtrado de imágenes, en honor al uso de los filtros que antiguamente se usaban en fotografía. Una muestra de ello es que si se observa a la interfaz BufferedImageOp, su principal método es filter().

Para hacer uso de estos filtros no puede emplearse el método Toolkit.getDefaultToolkit.getImage() ya que éste devuelve un objeto de tipo Image y no un BufferedImage, que es el requerido por las operaciones de filtrado. Podríamos utilizar ImageIO.read(), pero seguiremos tratando con el códec JPEG para resaltar su importancia.

Java2D posee un modelo de procesamiento de imágenes muy sencillo. En principio se tiene una instancia de un objeto BufferedImage, que se denomina imagen fuente, sobre el que se aplica uno de los métodos existentes en la interfaz BufferedImageOp, dando lugar a otro objeto BufferedImage al que se llama imagen destino. Para procesar la imagen, tan sólo hace falta instanciar una de las implementaciones de la interfaz BufferedImageOp y llamar al método filter() cuya cabecera es:

```
public BufferedImage filter(BufferedImage fuente, BufferedImage destino)
```

Este método procesa el objeto fuente; si destino es **null**, se creará una nueva BufferedImage con el resultado de la operación realizada sobre fuente. Si, por el contrario, destino no es **null** entonces el resultado se guardará en destino. En ambos casos la función devuelve el resultado.

Hay casos en los que la imagen fuente y la imagen destino pueden ser el mismo objeto, lo que quiere decir que el operador realiza el procesado de la imagen y la deposita sobre la misma imagen. Este mecanismo se llama procesado *in situ* y no todos los operadores sobre imágenes lo soportan ya que requiere un *buffer* interno.

5.6.1 Operaciones predefinidas

Java2D contiene un nutrido grupo de operadores sobre imágenes, que son implementaciones de la interfaz BufferedImageOp para el procesamiento de una

imagen. En la siguiente tabla se resumen las clases que contienen los operadores sobre imágenes que definidas en el paquete java.awt.image. La columna de la tabla que indica las clases soportadas, especifica los parámetros admitidos por el constructor de la clase que implementa la operación. La tabla es la siguiente:

Nombre de la clase	Clases soportadas	Efecto que produce	Procesa *in situ*
ConvolveOp	Kernel	Definir borrosidad, perfilado, detección de bordes, etc.	✘
AffineTransformOp	AffineTransform	Transformaciones geométricas	✘
LookupOp	LookupTable, ByteLookupTable, ShortLookupTable	Reducir el número de colores, crear negativos, etc.	✔
RescaleOp		Abrillantar y oscurecer	✔
ColorConvertOp	ColorSpace	Convertir entre espacios de colores	✔

A continuación procederemos a explicar cada una de estas clases por separado.

5.6.1.1 ConvolveOp

ConvolveOp es uno de los operadores sobre imágenes más versátiles que existen en la API Java2D. Representa una convolución espacial. Éste es un término matemático que, aplicado al tratamiento de imágenes que nos ocupa, quiere decir que el color de cada píxel de la imagen destino vendrá determinado por una combinación de colores del píxel fuente y de sus píxeles vecinos. Un operador de convolución puede usarse para suavizar o perfilar una imagen, entre otras cosas.

El operador de convolución usa un objeto Kernel para definir la manera en que un píxel se combina con sus vecinos. Un Kernel es simplemente una matriz en la que el valor central representa al píxel fuente y los valores restantes representan a los píxeles vecinos. Cada banda de color destino se calcula multiplicando cada banda de color de píxel fuente por su correspondiente coeficiente en el Kernel y sumando los resultados de todos y cada uno de ellos.

Ejemplos de *kernels* pueden ser los siguientes:

$$\begin{bmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{bmatrix} \quad \begin{bmatrix} 0.5 & 0.7 & 0.8 \\ 0.4 & 0.1 & 0.8 \\ 0.9 & 1.0 & 1.0 \end{bmatrix} \quad \begin{bmatrix} -0.2 & 0.2 & 0.2 \\ 0.2 & 0.2 & -0.2 \\ 0.2 & -0.2 & 0.2 \end{bmatrix}$$

El primer *kernel* es el que no hace absolutamente nada, es el *kernel* identidad. El segundo *kernel* aumenta muchísimo el brillo de la imagen, mientras que el tercero la

oscurece bastante.

Como se ha comentado, los *kernels* están representados por instancias de la clase java.awt.image.Kernel y se pueden definir mediante un array unidimensional, tal y como hace el constructor:

```
public Kernel(int nfilas, int ncolumnas, float[] datos).
```

que construye un kernel indicando el número de filas de la matriz en el primer parámetro que se pasa como argumento, el número de columnas en el segundo parámetro y, finalmente, los valores concretos de la matriz. Un ejemplo de uso de este constructor puede ser:

```
float [] valores = {
        0.3f, 0.4f, 0.3f,
        0.4f, 0.5f, 0.4f,
        0.5f, 0.6f, 0.5f
};
Kernel ker = new Kernel(3, 3, valores);
```

Una vez construido el Kernel, se puede proceder a crear el operador de convolución utilizando dico Kernel como parámetro.

Si se analiza la manera de actuar la convolución, se presenta una pregunta acerca de cómo funcionará el operador de convolución en los bordes de una imagen. Pues bien, el constructor de la clase ConvolveOp, el cual se encargará de la operación que estamos tratando en este apartado, provee dos constantes que representan dos posibles comportamientos del operador de convolución en los márgenes de la imagen. Las dos constantes son EDGE_ZERO_FILL y EDGE_NO_OP. La primera rellena de ceros los márgenes de la imagen, mientras que la segunda constante no modifica los píxeles de los bordes.

Los constructores de la clase ConvolveOp son los siguientes:

- public ConvolveOp(Kernel kernel)
- public ConvolveOp(Kernel kernel, int comportamientoBordes)

El primer constructor crea un objeto ConvolveOp sin especificar ningún comportamiento en los bordes de la imagen, mientras que el segundo sí lo especifica. Como se ha dicho antes, la constante que se le puede pasar como argumento puede tomar los valores EDGE_ZERO_FILL o EDGE_NO_OP. En el primer constructor se toma por defecto el comportamiento EDGE_ZERO_FILL.

Ahora se mostrarán algunos ejemplos de convolución. A la hora de hacer esta operación, se debe tener en cuenta que la suma de todos los valores del *kernel* debe ser igual a 1 siempre y cuando se quiera conservar el brillo de la imagen. Si lo que se desea es crear una imagen más clara que la original, hay que asegurarse de que la suma de los valores del *kernel* supera el 1, mientras que para hacer la imagen destino más oscura, estos valores deben sumar menos de 1.

Concretamente, hay tres ejemplos de kernels que son de gran utilidad a la

hora de realizar este procesamiento de imagen, aunque el usuario perfectamente puede definir el que le plazca. Son los kernels que actúan en los tres ejemplos que se explican a continuación.

En el primero se muestra la imagen de la figura 5.2, pero a la que se le ha aplicado una operación de convolución de suavizado. El código es el siguiente:

```
import java.awt.*;
import java.awt.image.*;
import java.io.*;
import javax.swing.*;
import com.sun.image.codec.jpeg.*;
public class Imagenes extends JFrame {
    public Imagenes() {
        super("Procesamiento de imágenes");
    }
    public void paint(Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        float  noveno = 1/9f;
        float [] valores = {    noveno, noveno, noveno,
                                noveno, noveno, noveno,
                                noveno, noveno, noveno,};
        try {
            InputStream in =
                getClass().getResourceAsStream("../neuschwanstein.jpg");
            JPEGImageDecoder decoder = JPEGCodec.createJPEGDecoder(in);
            BufferedImage image = decoder.decodeAsBufferedImage();
            Kernel ker = new Kernel(3,3,valores);
            ConvolveOp cop = new ConvolveOp(ker);
            BufferedImageOp operacion = cop;
            BufferedImage destino = operacion.filter(image, null);
            g2.drawImage(destino,0,20,this);
            in.close();
        }
        catch (IOException e) {e.printStackTrace();}
                catch (ImageFormatException e) {e.printStackTrace();}
    }
    public static void main (String args[]) {
        Imagenes v = new Imagenes();
        v.setDefaultCloseOperation(EXIT_ON_CLOSE);
        v.setSize(375,300);
        v.setVisible(true);
    }
}
```

Como se puede apreciar en la figura 5.5, la imagen se ve más borrosa que con respecto a los anteriores ejemplos. El efecto producido es un suavizado o

difuminado de la foto. Este aumento de la borrosidad se debe a la actuación del *kernel* en cuestión. El *kernel* aplicado en la figura 5.5 es una matriz totalmente compuesta por los valores 1/9. Se puede ver que la suma de todos los valores es exactamente igual a 1 por lo que, como se dijo antes, se conserva el brillo de la imagen fuente.

Figura 5.5. Aplicación de un *kernel* de difuminado o suavizado

La utilidad de este difuminado de la imagen reside en su capacidad para filtrar imágenes con ruido, es decir, imágenes en las cuales se puedan apreciar pequeños errores, ya sea por formato o por fallos producidos durante su transmisión. Este efecto ayuda a disminuir la apreciación de esos errores.

Basta con cambiar el vector de convolución para obtener resultados muy diversos. Por ejemplo, si en el código anterior el vector `valores` se crea de la forma:

```
float [] valores = {0.0f,-1.0f ,0.0f ,-1.0f , 4.0f, -1.0f, 0.0f, -1.0f, 0.0f};
```

se obtendrá el resultado de la figura 5.6 en la que, como puede observarse, han sido detectados los bordes de la imagen (típico de los filtros pasa-alta).

En la figura 5.6 se muestra lo que se denomina un detector de bordes. El *kernel* de este ejemplo hace que no se aprecie ningún color. Lo único que se muestran son los cambios muy abruptos de color, lo que se traduce en mostrar los contornos de las figuras que componen la imagen.

Figura 5.6. Aplicación de un *kernel* de tipo filtro pasa-alta

Como se ha comentado, para obtener esta imagen lo único que ha hecho falta es modificar la matriz que compone el *kernel*; por tanto, en el siguiente ejemplo tan solo se va a mostrar el vector que se usa y su resultado. La matriz es la siguiente:

```
float [] valores = {0.0f,-1.0f ,0.0f ,-1.0f , 5.0f, -1.0f, 0.0f, -1.0f, 0.0f};
```

y el resultado puede verse en la figura 5.7.

Figura 5.7. Aplicación de un *kernel* que realza los perfiles de la imagen

Este efecto hace énfasis en los cambios de color profundos que hay en la imagen a la vez que mantiene el brillo y la mayoría de los datos originales. Realiza un afilado o perfilado (en inglés *sharpening*) de la imagen.

Desde aquí se incita al lector a que pruebe efectos distintos modificando los valores del *kernel*. En cualquier caso, a veces es un tanto difícil observar un *kernel* y deducir el efecto que producirá sobre una imagen después de realizar la convolución.

Para finalizar este apartado, la siguiente tabla resume las características de estas clases:

Clase o Método	Descripción
java.awt.image.ConvolveOp	Clase para realizar una convolución espacial
java.awt.image.Kernel	Clase para crear una matriz que después sirva a la clase ConvolveOp en la aplicación de la operación de convolución
public Kernel(int nfilas, int ncolumnas, float[] datos)	Constructor de la clase Kernel que crea una matriz de nfilas y ncolumnas y la rellena con los valores del *array* datos.
public ConvolveOp(Kernel kernel)	Constructor de la clase ConvolveOp que crea el operador de convolución deseado. Para trabajar con los píxeles del borde de la imagen toma por defecto el valor EDGE_ZERO_FILL.
public ConvolveOp(Kernel kernel, int comportamientoBordes)	Constructor de la clase ConvolveOp, pero especificando el comportamiento en los bordes de la imagen.

5.6.1.2 AffineTransformOp

Las imágenes pueden transformarse geométricamente usando la clase java.awt.image.AffineTransformOp. Esta clase se vale de una instancia de AffineTransform (clase ya estudiada con detenimiento en el apartado 2.5.3) para crear una imagen destino que es una versión de la imagen fuente con la transformación geométrica aplicada. Esta operación es distinta de aplicar una transformación a un objeto Graphics2D y después renderizar una imagen, ya que aquí se está creando una imagen nueva que es una versión transformada de la imagen fuente, y que puede ser posteriormente dibujada sobre cuantos Graphics2D se desee.

Para realizar transformaciones, ya sea rotación, escala, etc., la clase AffineTransformOp aplica la transformación a todos los píxeles y crea la nueva imagen. El trabajo duro consiste en averiguar el color para cada uno de los píxeles de la imagen destino, ya que, a no ser que se haga una transformación en la que cuadren todos los píxeles, los píxeles de la imagen destino son una aproximación de la imagen fuente (piénsese por ejemplo en un giro de 45º).

Existen tres tipos de algoritmos de aproximación en la clase AffineTransformOp, que son el vecino más cercano (en inglés *nearest neighbour*) y las interpolación bilinear y bicúbica. El primero es más rápido, mientras que los otros dos ofrecen imágenes de mayor calidad.

Para realizar una transformación geométrica sobre una imagen se llama al constructor de esta clase:

```
public AffineTransformOp ( AffineTransform transformacion,
                           RenderingHints contexto):
```

que crea un nuevo objeto AffineTransformOp para procesar una imagen según la transformación especificada como primer parámetro. El segundo parámetro indica el contexto de renderizado. Si no es null y contiene un valor específico en el campo RenderingHints.KEY_INTERPOLATION, entonces ese valor determinará el algoritmo de aproximación en la transformación.

En el código que viene a continuación se muestra cómo aplicar una rotación a una imagen (sólo se muestra el método paint() ya que el resto es igual que en ejemplos anteriores); el resultado puede verse en la figura 5.8:

```
public void paint(Graphics g) {
    Graphics2D g2 = (Graphics2D)g;
    try {
        InputStream in =
        getClass().getResourceAsStream("../neuschwanstein.jpg");
            JPEGImageDecoder decoder = JPEGCodec.createJPEGDecoder(in);
        BufferedImage image = decoder.decodeAsBufferedImage();
        // Aqui se realiza la transformación
            AffineTransform at = AffineTransform.getRotateInstance(Math.PI/8);
```

```
                RenderingHints rh =
                    new RenderingHints(RenderingHints.KEY_INTERPOLATION,
                            RenderingHints.VALUE_INTERPOLATION_BILINEAR);
                AffineTransformOp atop = new AffineTransformOp(at,rh);
                BufferedImage destino = atop.filter(image, null);
                // Aquí termina el proceso
                g2.drawImage(destino,0,20,this);
                in.close();
        }
        catch (IOException e) {e.printStackTrace();}
        catch (ImageFormatException e) {e.printStackTrace();}
}
```

Figura 5.8. Renderizado de una imagen previamente rotada π/8 radianes en el sentido de las agujas del reloj

La tabla siguiente resume este apartado dedicado a las transformaciones:

Clase o Método	Descripción
AffineTransformOp public AffineTransformOp(AffineTransform transf, RenderingHints contex)	Clase que representa al operador que realiza transformaciones geométricas sobre una imagen. Constructor de AffineTransformOp que crea una instancia de la misma a partir de la transformación transf. El contexto determina el algoritmo de aproximación del color mediante el valor que tome el campo RenderingHints.KEY_INTERPOLATION: - VALUE_INTERPOLATION_NEAREST_NEIGHBOR - VALUE_INTERPOLATION_BILINEAR - VALUE_INTERPOLATION_BICUBIC

5.6.1.3 LookupOp

La clase LookupOp permite realizar transformaciones de color sobre píxeles pero, aunque el operador de convolución también podía conseguir efectos parecidos, la clase LookupOp proporciona un control más directo sobre dichas

transformaciones.

Este operador parte de una tabla de sustitución (en inglés *lookup table*), que básicamente es un *array* que contiene los valores de color de los píxeles destino. Los valores de color de los píxeles fuente sirven como índices para esta tabla. La tabla se construye para cada una de las tres bandas de color en el sistema RGB -rojo, verde y azul- de manera que se puede construir una única tabla de sustitución para los tres o una por banda. La tabla de sustitución se representa por una instancia de la clase LookupTable, que contiene subclases para representar *arrays* de tipo **short** y **byte** llamadas ShortLookupTable y ByteLookupTable. Cada una de estas clases contiene dos constructores, que son los siguientes:

- public ShortLookupTable (int desplazamiento, short[] datos)

  Con este constructor se crea una tabla de sustitución basada en el *array* datos. Cada componente de color de un píxel fuente se usa como índice en este *array*, siendo el valor de la componente de color del píxel destino el almacenado en la casilla del array dada por ese índice. El valor de la variable desplazamiento se resta a los datos de entrada antes de realizar el direccionamiento al array, es decir, que si los valores que se esperan tener en los datos de entrada están, por ejemplo, en el rango de 100 a 200, pues el *array* datos deberá contener 101 elementos el primero de los cuales se corresponderá con el valor de color 100 de los píxeles de entrada. Así nos aseguramos de que siempre se direcciona a un índice dentro de los límites del *array* (si se hace referencia a una posición fuera del rango del *array* se producirá una excepción). Como normalmente, el rango de las componentes del color van desde 0 a 255 (sistema RGB), pues el *array* en cuestión tendrá 256 elementos con un desplazamiento de 0.

- public ShortLookupTable (int desplazamiento, short[][] datos)

  Este constructor es exactamente igual al anterior, sólo que aquí se pasa un *array* bidimensional en la que se tendrá una tabla por cada banda o componente de color, de manera que si se está en el sistema RGB se tendrán tres tablas o *arrays* de datos, mientras que en los sistemas CYMK o ARGB (RGB con canal alfa) se pasará un *array* de cuatro tablas. El desplazamiento funciona de la misma manera que en el constructor anterior y cada una de las tablas tendrá el mismo desplazamiento.

- public ByteLookupTable (int desplazamiento, byte[] datos)

  Este constructor es exactamente igual al primero, pero funciona con valores de tamaño **byte**.

- public ByteLookupTable (int desplazamiento, byte[][] datos)

  Este constructor es exactamente igual al segundo, pero funcionando con valores de tamaño **byte**.

Una vez creada la tabla de sustitución sólo queda definir la operación en cuestión creando un objeto de la clase LookupOp, para lo cual se llama a su constructor:

public LookupOp (LookupTable lt, RenderingHints contexto)

Ahora se van a demostrar algunas de las utilidades de la tabla de sustitución merced a los próximos ejemplos. Siguiendo con la tónica que caracteriza el código que venimos presentando, los ejemplos son muy sencillos de programar: en tan solo unas pocas líneas se pueden implementar efectos más que notables. Una prueba de ello es el siguiente código que realiza un negativo de la imagen (el resultado puede apreciarse en la figura 5.9):

```
import java.awt.*;
import java.awt.image.*;
import java.io.*;
import javax.swing.*;
import com.sun.image.codec.jpeg.*;
public class Imagenes extends JFrame {

    static final short [] coloresInvertidos = new short[256];
    static {
        for (int i=0;i<256;i++)
            coloresInvertidos[i] = (short)(255-i);
    }
    public Imagenes() {
        super("Procesamiento de imágenes");
    }
    public void paint(Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        try {
            InputStream in =
                getClass().getResourceAsStream("../neuschwanstein.jpg");
            JPEGImageDecoder decoder = JPEGCodec.createJPEGDecoder(in);
            BufferedImage imagen = decoder.decodeAsBufferedImage();
            // Aqui se realiza la transformación
            LookupTable lt = new ShortLookupTable (0,coloresInvertidos);
            LookupOp lop = new LookupOp(lt,null);
            BufferedImage destino = lop.filter(imagen,null);
            // Aquí termina el proceso
            g2.drawImage(destino,0,20,this);
            in.close();
        }
        catch (IOException e) {e.printStackTrace();}
        catch (ImageFormatException e) {e.printStackTrace();}
    }
    public static void main (String args[]) {
        Imagenes v = new Imagenes();
```

```
            v.setDefaultCloseOperation(EXIT_ON_CLOSE);
            v.setSize(375,300);
            v.setVisible(true);
        }
}
```

Figura 5.9. Ejemplo de imagen en la que cada banda de color se ha sustituido por su negativo

En la figura 5.9 se muestra lo que parece el negativo de la fotografía correspondiente a los ejemplos anteriores. Lo que se ha hecho es, sencillamente, crear un *array* con los valores del sistema de colores RGB invertido, de manera que el valor del color del píxel fuente se invierta con respecto al destino: el color i se sistituye por 255-i. Con ese *array* se construye un objeto LookupTable para, acto seguido, crear el operador y llamar a su método filter(), el cual crea la nueva imagen que, por fin, se renderiza por pantalla.

En este ejemplo todas las bandas de color se gestionan con el mismo *array*. En los ejemplos siguientes se muestra cómo manejar cada componente de color por separado. En el primero se pretende invertir las componentes verde y roja (el resultado puede verse en la figura 5.10):

```
import java.awt.*;
import java.awt.image.*;
import java.io.*;
import javax.swing.*;
import com.sun.image.codec.jpeg.*;
public class Imagenes extends JFrame {

    static final short [] coloresInvertidos = new short[256];
    static final short [] coloresSinInvertir = new short[256];
    static short [][] inversionVerdeYRojo = {
            coloresInvertidos, coloresInvertidos, coloresSinInvertir};
    static {
        for (int i=0;i<256;i++){
            coloresInvertidos[i] = (short)(255-i);
            coloresSinInvertir[i] = (short)(i);
```

```
            }
        }
        public Imagenes() {
            super("Procesamiento de imágenes");
        }
        public void paint(Graphics g) {
            Graphics2D g2 = (Graphics2D)g;
            try {
                InputStream in =
                    getClass().getResourceAsStream("../neuschwanstein.jpg");
                JPEGImageDecoder decoder = JPEGCodec.createJPEGDecoder(in);
                BufferedImage imagen = decoder.decodeAsBufferedImage();
                // Aqui se realiza la transformación
                LookupTable lt = new ShortLookupTable (0,inversionVerdeYRojo);
                LookupOp lop = new LookupOp(lt,null);
                BufferedImage destino = lop.filter(imagen,null);
                // Aquí termina el proceso
                g2.drawImage(destino,0,20,this);
                in.close();
            }
            catch (IOException e) {e.printStackTrace();}
            catch (ImageFormatException e) {e.printStackTrace();}
        }
        public static void main (String args[]) {
            Imagenes v = new Imagenes();
            v.setDefaultCloseOperation(EXIT_ON_CLOSE);
            v.setSize(375,300);
            v.setVisible(true);
        }
}
```

Figura 5.10. Ejemplo de imagen en la que las bandas de color verde y roja han sido sustituidas por sus respectivos negativos

Como resultado de la inversión de las bandas roja y verde, se ha transformado totalmente el color de la imagen, predominando solamente el azul y el amarillo (ya que la imagen original casi carece de rojo y el posee mucho verde, cuyos

componentes primarios son precisamente el azul y el amarillo). En esta ocasión se han construido dos *arrays*: uno conteniendo los valores del sistema de colores RGB invertido y otro sin invertir. Para crear el objeto LookupTable esta vez se ha llamado al segundo constructor que comentamos anteriormente, al que se le debe proporcionar un *array* de dos dimensiones representando los tres componentes del sistema RGB, que es el que se está usando. Para ello se ha definido el *array* como sigue:

```
short [][] inversionVerdeYRojo = {
                    coloresInvertidos,
                    coloresInvertidos,
                    coloresSinInvertir};
```

Como el sistema de colores es RGB (recordemos que viene del inglés *Red, Green and Blue*) el primer componente de color es el rojo, el segundo el verde y el tercero el azul. Como a la componente roja y verde que están en la casilla primera y segunda del *array* inversionVerdeYRojo se le han introducido los valores invertidos, serán esos colores los que se invertirán. El resto del proceso es análogo al del ejemplo anterior, por lo que no reviste complicación alguna.

El siguiente y último ejemplo hace otra modificación distinta, que es la de eliminar el color rojo de la imagen, poniendo a cero todos los valores del *array* para dicho color. El resultado, que se muestra en la figura 5.11, se corresponde con el siguiente código:

```
import java.awt.*;
import java.awt.image.*;
import java.io.*;
import javax.swing.*;
import com.sun.image.codec.jpeg.*;
public class Imagenes extends JFrame {
    static final short [] coloresSinInvertir = new short[256];
    static final short [] colorACero = new short[256];
    static short [][] inversionRojoACero = {
        colorACero ,coloresSinInvertir, coloresSinInvertir };
    static {
        for (int i=0;i<256;i++){
            coloresInvertidos[i] = (short)(255-i);
            coloresSinInvertir[i] = (short)(i);
            colorACero[i] = 0;
        }
    }
    public Imagenes() {
        super("Procesamiento de imágenes");
    }
    public void paint(Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        try {
            InputStream in =
```

```
                getClass().getResourceAsStream("../neuschwanstein.jpg");
            JPEGImageDecoder decoder = JPEGCodec.createJPEGDecoder(in);
            BufferedImage imagen = decoder.decodeAsBufferedImage();
            // Aqui se realiza la transformación
            LookupTable lt = new ShortLookupTable (0, inversionRojoACero);
            LookupOp lop = new LookupOp(lt,null);
            BufferedImage destino = lop.filter(imagen,null);
            // Aquí termina el proceso
            g2.drawImage(destino,0,20,this);
            in.close();
        }
        catch (IOException e) {e.printStackTrace();}
        catch (ImageFormatException e) {e.printStackTrace();}
    }
    public static void main (String args[]) {
        Imagenes v = new Imagenes();
        v.setDefaultCloseOperation(EXIT_ON_CLOSE);
        v.setSize(375,300);
        v.setVisible(true);
    }
}
```

Figura 5.11. Ejemplo de imagen en la que se ha eliminado el color rojo

Resumiendo, la tabla siguiente concentra el contenido de este apartado:

Clase o Método	Descripción
java.awt.image.LookupOp	Clase que define el operador de sustitución de colores. Tiene varias utilidades, entre ellas la de realizar una convolución
java.awt.image.LookupTable	Clase que define la tabla de sustitución en sí. Es un *array* de valores naturales que asocia a cada color fuente un color destino: el color fuente se usa como índice del *array* y el destino es el contenido de la casilla de dicho índice. Posee dos subclases, ShortLookupTable y ByteLookupTable que definen *arrays* de tipo **short** o **byte** respectivamente

public ShortLookupTable (int desplazamiento, short[] datos)	Constructor que define una tabla de sustitución formada por un *array* de datos de tipo **short**. Se usa un único *array* para todas las componentes del espacio de colores. El desplazamiento se puede aplicar en suma al valor de color fuente
public ShortLookupTable (int desplazamiento, short[][] datos)	Constructor igual que el anterior, pero con la diferencia de que se trata cada banda de color del espacio de colores de manera individual. Por ello, se pasa un *array* de dos dimensiones cuya primera dimensión es el número de bandas del espacio de colores concreto que se esté usando
public ByteLookupTable (int desplazamiento, short[] datos)	Constructor igual que el anterior pero manejando un *array* de tipo **byte**
public ByteLookupTable (int desplazamiento, short[][] datos)	Constructor igual al de antes pero cada *array* se compone de valores de tipo **byte**
public LookupOp (LookupTable lt, RenderingHints contex)	Constructor de la clase LookupOp que se encarga de construir el operador de la tabla de sustitución

5.6.1.4 RescaleOp

La clase RescaleOp se usa para aumentar o disminuir el brillo de una imagen, aunque su nombre puede dar lugar a confusión pues parece que lo que hace es cambiar el tamaño de la imagen. El cambio de brillo no es ni mas ni menos que multiplicar cada componente de color de cada píxel por un factor de escala. Es decir, si multiplicamos por el factor de escala 1,0 la imagen quedará inmutada.

El cambio de brillo está encapsulado en la clase java.awt.image.RescaleOp, cuyo constructor admite un desplazamiento además del factor de escala. El desplazamiento se le añade a los valores justo después de haber sido multiplicados por el factor de escala. Los resultados de este cálculo se truncan de manera que, si la nueva componente de color supera el máximo permitido, su valor se ajusta al máximo.

Posee dos constructores, que son:

- public RescaleOp (
  float factorDeEscala, float desplazamiento,
  RenderingHints contexto)

  Este constructor crea un nuevo objeto RescaleOp usando el factor de escala y el desplazamiento dados. El contexto se podrá usar cuando las imágenes se filtren con este operador.

- public RescaleOp(
  float[] factoresDeEscalas, float[] desplazamientos,

RenderingHints contexto)

Este constructor actúa de la misma manera que el anterior, pero acepta un factor de escala y un desplazamiento por cada una de las bandas de color del sistema por separado, de la misma forma que se ha visto en el epígrafe anterior. Si por ejemplo se desea procesar una imagen en formato ARGB, se proporcionarán cuatro factores de escalas y cuatro desplazamientos.

Para incrementar el brillo de una imagen en un 50% por ejemplo, tan sólo se tiene que multiplicar por 1.5:

RescaleOp aumentaBrillo = new RescaleOp(1.5f, 0, null)

Por el contrario, para disminuir el brillo de una imagen en un 50%, se multiplica por 0.5:

RescaleOp aumentaBrillo = new RescaleOp(0.5f, 0, null)

En el siguiente código se implementa una muestra de lo que se acaba de explicar (ver figura 5.12):

```
import java.awt.*;
import java.awt.image.*;
import java.io.*;
import javax.swing.*;
import com.sun.image.codec.jpeg.*;
public class Imagenes extends JFrame {

    static final float[] componentes = {1.5f, 1.5f, 1.5f};
    static final float[] desplazamientos = {0.0f,0.0f,0.0f};
    public Imagenes() {
        super("Procesamiento de imágenes");
    }
    public void paint(Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        try {
            InputStream in =
                getClass().getResourceAsStream("../neuschwanstein.jpg");
            JPEGImageDecoder decoder = JPEGCodec.createJPEGDecoder(in);
            BufferedImage imagen = decoder.decodeAsBufferedImage();
            // Aqui se realiza la transformación
            RescaleOp rop = new RescaleOp (componentes, desplazamientos, null);
            BufferedImage destino = rop.filter(imagen,null);
            // Aquí termina el proceso
            g2.drawImage(destino,0,20,this);
            in.close();
        }
        catch (IOException e) {e.printStackTrace();}
        catch (ImageFormatException e) {e.printStackTrace();}
    }
```

```
    public static void main (String args[]) {
        Imagenes v = new Imagenes();
        v.setDefaultCloseOperation(EXIT_ON_CLOSE);
        v.setSize(375,300);
        v.setVisible(true);
    }
}
```

Figura 5.12. Incremento del brillo en un 50% en las tres componentes de una imagen en sistema RGB

Como se puede observar en la figura 5.12, se ha aumentado el brillo multiplicando cada componente de color por el factor 1.5f. Sin embargo, esto también se podría haber hecho directamente, sin necesidad de llamar al constructor que acepta las componentes por separado, sino que se podría haber llamado al que acepta un solo valor. Se ha hecho de esta manera para que el lector se maneje con varios puntos de vista.

Asimismo, es posible aumentar el brillo de uno de los colores solamente, o disminuir uno y aumentar otro, etc., lo que abre un amplio abanico de posibilidades. Por ejemplo, si cambiamos la línea de código:

```
static final float[] componentes = {1.5f, 1.5f, 1.5f};
```

por la línea

```
static final float[] componentes = {0.5f, 0.6f, 0.5f};
```

el resultado se muestra en la figura 5.13:

Figura 5.13. Decremento del brillo en diferentes factores para cada componente de una imagen RGB

Se ha oscurecido la imagen como consecuencia de los factores de escala; el efecto deseado se ha conseguido con tan sólo una línea de código.

Otra forma de aumentar el brillo sin necesidad de usar un RescaleOp consiste en utilizar una tabla de sustitución (*lookup table*); se trata de un mecanismo más preciso pero, a la vez, más tedioso de realizar. Hay varias maneras de realizar esta operación; una de ellas es mapear el rango de valores de entrada en uno más pequeño, por ejemplo, pasar los valores de 0 a 255 al rango de 128 a 255 o al rango de 64 a 127, etc. En el código siguiente se muestra como realizar un mapeo 0-255 a 128-255 mediante la clase LookupOp (ver figura 5.14):

```
import java.awt.*;
import java.awt.image.*;
import java.io.*;
import javax.swing.*;
import com.sun.image.codec.jpeg.*;
public class Imagenes extends JFrame {

    static final short [] brillo = new short[256];
    static{
        for (int i=0;i<256;i++)
            brillo[i] = (short)(128+i/2);
    }
    public Imagenes() {
        super("Procesamiento de imágenes");
    }
    public void paint(Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        try {
            InputStream in =
                getClass().getResourceAsStream("../neuschwanstein.jpg");
            JPEGImageDecoder decoder = JPEGCodec.createJPEGDecoder(in);
            BufferedImage imagen = decoder.decodeAsBufferedImage();
            // Aqui se realiza la transformación
            LookupTable lt = new ShortLookupTable (0,brillo);
            LookupOp lop = new LookupOp(lt,null);
            BufferedImage destino = lop.filter(imagen,null);
            // Aquí termina el proceso
            g2.drawImage(destino,0,20,this);
            in.close();
        }
        catch (IOException e) {e.printStackTrace();}
        catch (ImageFormatException e) {e.printStackTrace();}
    }
    public static void main (String args[]) {
        Imagenes v = new Imagenes();
        v.setDefaultCloseOperation(EXIT_ON_CLOSE);
```

```
            v.setSize(375,300);
            v.setVisible(true);
        }
}
```

Figura 5.14. Incremento del aclarado mediante una tabla de sustitución que disminuye el número de colores

Como se puede apreciar en la figura 5.14, se ha aumentado el brillo de la imagen (más bien se ha aclarado), pero el efecto que consigue la clase RescaleOp es más pulcro, se implementa de una manera más intuitiva y, además, con la tabla de sustitución la imagen aparece descolorida.

Otra forma de abrillantar una imagen usando la tabla de sustitución consiste en usar la función raíz cuadrada en los píxeles de origen normalizados (el rango de naturales 0-255 se pasa al de reales 0,0-1,0) con lo que los valores más oscuros son los que más se abrillantan. Para ello sólo es necesario cambiar el contenido del *array* brillo:

```
static final short [] brillo = new short[256];
static{
      for (int i=0;i<256;i++)
            brillo[i] = (short)(Math.sqrt((float)i/255.0f)*255.0f);
}
```

Figura 5.15. Incremento del brillo en una imagen mediante una tabla de sustitución que potencia el aclarado de los valores más oscuros

El resultado se muestra en la figura 5.15.

El efecto que se obtiene con la función raíz cuadrada está mucho mejor conseguido que el que produce una función lineal en los valores del *array*. Realmente, lo que se pretende con esta función es potenciar todos los valores de color inferiores e intermedios de entre los valores de entrada, mientras que los de los extremos superior no se potencian tanto:

```
brilloCuadrada[i] = (short)(Math.sqrt((float)i/255.0f)*128.0f);
```

La siguiente tabla resume el contenido de este apartado:

Clase o Método	Descripción
java.awt.image.RescaleOp	Clase que representa al operador para abrillantar u oscurecer una imagen. Este operador multiplica los valores de color de todos los píxeles de una imagen por un factor de escala, el cual oscurecerá si el factor es menor que 1 y abrillantará si es mayor
public RescaleOp (float factorDeEscala, float desplazamiento, RenderingHints contex)	Constructor que acepta un factor de escala y un desplazamiento que incrementa los valores resultantes después de la operación de multiplicación
public RescaleOp(float[] factoresDeEscalas, float[] desplazamientos, RenderingHints contex)	Constructor con el mismo fin que el anterior pero con la diferencia de que se trata cada componente del espacio de colores por separado, de manera que si el espacio de colores tiene, por ejemplo, tres bandas, entonces el *array* tendrá tres casillas.

5.6.1.5 ColorConvertOp

El último de los operadores que está incluido en la API Java2D se usa para convertir los colores de una imagen de un sistema de colores a otro. Esto puede servir para cuando se desee crear un espacio de colores optimizado para un dispositivo de salida concreto (véase el epígrafe 1.4). La clase que realiza esta conversión es java.awt.image.ColorConvertOp. Tiene cuatro constructores, que son:

- public ColorConvertOp(RenderingHints contexto)

  Este constructor crea una nueva instancia de la clase ColorConvertOp que transforma desde el espacio de colores de la imagen fuente al espacio de colores de la imagen destino cuando se invoque a filter(). Cuando se llame a este método, no se puede poner **null** en el parámetro reservado a la imagen destino pues, en tal caso, el operador no sabrá a que espacio de colores convertir.

- public ColorConvertOp(
  ColorSpace espColDestino, RenderingHints contexto)

  Aquí se crea una nueva instancia que convierte desde el espacio de

colores de la imagen al espacio de colores espColDestino. Aquí el método filter() sí puede contener un null en el campo de la imagen destino.

- public ColorConvertOp(
  ColorSpace espFuente,ColorSpace espDestino,
  RenderingHints cont)

  Este constructor convierte desde el espacio de colores fuente dado hasta el espacio destino dado en el método. Este constructor crea una instancia de ColorConvertOp que se usa para filtrar *rasters*. Un *raster* es la parte de la clase BufferedImage que contiene los datos de la imagen en sí.

- public ColorConvertOp(
  ICC_Profile[] perfiles, RenderingHints contexto)

  En esta ocasión se convierte desde el espacio de colores de la imagen a través de todos los espacios de colores dados, los cuales están representados cada uno por un perfil distinto. Un objeto ICC_Profile representa un perfil de color basado en la especificación del Consorcio Internacional del Color (ICC-*International Color Consortium*).

El siguiente código muestra un ejemplo de conversión entre espacios de colores. Se ha elegido el convertir a escala de grises ya que es el que denota una mayor diferencia entre la imagen fuente y la destino (el resultado puede verse en la figura 5.16):

```
import java.awt.*;
import java.awt.color.*;
import java.awt.image.*;
import java.io.*;
import javax.swing.*;
import com.sun.image.codec.jpeg.*;
public class Imagenes extends JFrame {
    public Imagenes() {
        super("Procesamiento de imágenes");
    }
    public void paint(Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        try {
            InputStream in =
                getClass().getResourceAsStream("../neuschwanstein.jpg");
            JPEGImageDecoder decoder = JPEGCodec.createJPEGDecoder(in);
            BufferedImage imagen = decoder.decodeAsBufferedImage();
            // Aqui se realiza la transformación
            ColorConvertOp ccop = new ColorConvertOp(ColorSpace.getInstance
                                        (ColorSpace.CS_GRAY), null);
            BufferedImage destino = ccop.filter(imagen,null);
            // Aquí termina el proceso
            g2.drawImage(destino,0,20,this);
```

```
                in.close();
            }
            catch (IOException e) {e.printStackTrace();}
            catch (ImageFormatException e) {e.printStackTrace();}
        }
        public static void main (String args[]) {
            Imagenes v = new Imagenes();
            v.setDefaultCloseOperation(EXIT_ON_CLOSE);
            v.setSize(375,300);
            v.setVisible(true);
        }
}
```

Figura 5.16. Conversión a espacio de colores monocromático (blanco y negro)

Como puede apreciarse en el código, el proceso ha resultado muy sencillo: an sólo se ha tenido que crear el objeto y llamar al método filter().

A modo de resumen, en la tabla siguiente se aúnan los contenidos de este apartado:

Clase o Método	Descripción
java.awt.image.ColorConvertOp	Clase que define un operador para convertir entre espacios de colores
public ColorConvertOp(RenderingHints contex)	Constructor que convierte del espacio de colores de la imagen fuente al espacio de colores de la imagen destino. Para ello el método filter() no puede contener un valor **null** como parámetro de imagen destino
public ColorConvertOp(ColorSpace espColDestino, RenderingHints contex)	Constructor de clase que convierte el espacio de colores de la imagen fuente al espacio de colores dado por espColDestino
public ColorConvertOp(ColorSpace espFuente, ColorSpace espDestino, RenderingHints contex)	Constructor que convierte desde el espacio de colores espFuente hasta el espacio de colores destino espDestino
public ColorConvertOp(	Constructor de clase que hace pasar el espacio

ICC_Profile[] perfiles, RenderingHints contex)	de colores fuente por cada uno de los espacios de colores asociado a cada perfil ICC

5.6.2 Dentro de BufferedImage

java.awt.BufferedImage es la piedra angular de todas las nuevas capacidades en el tratamiento de imágenes de Java2D. La gran ventaja de esta clase es que se pueden acceder a los datos de la imagen directamente, operación que se presenta bastante difícil de otra forma. En este apartado tan sólo echaremos un rápido vistazo a la utilidad que proporciona esta clase.

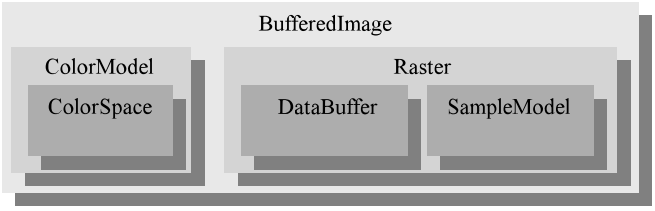


Figura 5.17. Conceptualización abstracta de los componentes que integran un objeto BufferedImage

En la figura 5.17 se puede ver un esquema de lo que compone interiormente a un objeto de la clase BufferedImage, y que proporciona una visión simple de lo que en realidad es esta clase.

La explicación es la siguiente: toda BufferedImage contiene un *raster* (clase Raster) y un modelo de color (clase ColorModel) que se corresponde con un espacio de colores (clase ColorSpace). Los datos de la imagen en sí están guardados en el *raster*, mientras que el modelo de color interpreta esos datos como colores. El concepto de *raster* implica que los datos no se hallan comprimidos de ninguna manera y que integran la imagen bidimensional al completo. Adoptaremos esta palabra como española de ahora en adelante.

El raster contiene los datos que determinan el color de cada píxel, de manera que cada píxel tiene un valor por cada banda de color; estos valores se denominan muestras pues proceden de una captura digital resultante del muestreo de una imagen analógica (potencialmente con infinitos colores). Por ejemplo, una imagen en escala de grises tiene una muestra por cada píxel, mientras que una imagen RGB tiene tres. Un raster es la secuencia de todas las muestras de todos los píxeles de la imagen. Dentro de un raster, un *buffer* de datos contiene las muestras de los píxeles, normalmente almacenados en *arrays* de tipos primitivos como **int** o **byte** según la precisión de color que se quiera. La otra parte de un objeto Raster, integrada por un objeto de la clase SampleModel, es la encargada de interpretar los datos del *buffer*

y extraer las muestras que componen cada píxel concreto.

El trabajo de la clase ColorModel es el de interpretar las muestras de un píxel como un color. En una imagen en escala de grises, que tiene una sola muestra por píxel, ColorModel la interpreta como un color entre blanco y negro. En una imagen RGB, ColorModel usa las tres muestras de cada píxel como las componentes rojo, verde y azul de este sistema. Por supuesto, el número de muestras por píxel debe coincidir con el número de bandas del modelo de color, es decir, deben ser compatibles.

La tarea de encontrar el color de un pixel concreto en un objeto BufferedImage utiliza todos los componentes de la figura 5.17. Suponer, por ejemplo, que se pide a una BufferedImage el color del píxel de una posición cualquiera, por ejemplo la (0, 0):

1. La BufferedImage pide a su Raster las muestras del pixel.
2. El Raster pide a su SampleModel encontrar las muestras del píxel.
3. El SampleModel extrae las muestras del pixel del *buffer* de datos del Raster. Las muestras se devuelven al Raster quien, a su vez las pasa a la BufferedImage.
4. La BufferedImage utiliza su ColorModel (basado en un ColorSpace) para interpretar las muestras del pixel como un color.

Fijar el color de un píxel es básicamente el mismo proceso, pero al revés:

1. La BufferedImage pregunta a su ColorModel qué muestras corresponden a un color particular.
2. La BufferedImage dice a su Raster que fije las muestras para el píxel.
3. El Raster traslada la petición a su SampleModel.
4. El SampleModel almacena las muestras en el *buffer* del Raster.

Capítulo 6

El color en Java2D

6.1 Introducción

El tratamiento del color es un asunto verdaderamente delicado. Por ejemplo, si se pinta un barco naranja en un papel y acto seguido se digitaliza con un escáner resulta que el naranja que se obtiene en el ordenador es ligeramente distinto al del papel. Es más, si la imagen se visualiza en otro monitor, o bien se imprime en una impresora de determinado fabricante, o le sacamos una foto digital al barco, tendremos que, seguramente, todas las imágenes presentarán un color naranja diferente.

Java2D está diseñado para cumplir estrictos requisitos acerca del color. Para ello proporcionar un tratamiento del color de gran calidad mediante el uso de espacios de colores y de perfiles estándares de colores.

6.2 Cómo crear un color

La clase java.awt.Color representa un color en el espacio de colores RGB por defecto. Para crear un objeto Color se dispone de tres constructores:

- public Color (int r, int g, int b)
- public Color (float r, float g, float b)
- public Color (int rgb)

El primer constructores requiere tres parámetros cada uno de los cuales especifica el valor de un color en el sistema RGB; los valores que pueden tomar los parámetros van desde 0 a 255. El segundo constructor necesita también los 3 colores del espacio RGB como parámetros, pero éstos deben estar normalizados, estos es, sólo pueden tomar valores del 0,0 a 1,0. Por último, al tercer constructor sólo se le pasa un parámetro, que es un entero en Java que ocupa 32 bits. Los 8 primeros bits (0-7) indican el tono de azul, del bit 8 al 15 se determina el nivel de verde y, finalmente, del bit 16 al 23 se define el nivel de rojo; los últimos 8 bits pertenecen a la transparencia que se le quiera aplicar al color, es decir, indican el canal alfa del cual se ha hablado anteriormente. Como consecuencia de esto, los 3 métodos anteriores también tienen su correspondiente versión incluyendo el canal alfa:

- public Color (int r, int g, int b, int alfa)
- public Color (float r, float g, float b, float alfa)

- public Color (int rgbAlfa, boolean hayAlfa)

En el primer constructor el canal alfa se especifica como otro entero más con valores desde 0 a 255, siendo 255 el valor para la opacidad completa y 0 para una absoluta transparencia. En el constructor con números reales alfa también tiene valores entre 0,0 (opaco) y 1,0 (transparente). En el tercer constructor los últimos 8 bits del parámetro rgbalfa son para la transparencia, pero además se incluye el parámetro hayAlfa que, si es falso, sencillamente hará que se ignoren los 8 últimos bits de transparencia con lo que el color se tratará como opaco.

Java2D permite obtener los componentes de un color determinado. Para ello existen los métodos getRed(), getGreen(), getBlue() y getAlpha(), que devuelven cada componente de un color. El método getRGB() devuelve un entero de 32 bits con las cuatro componentes del objeto Color, tal y como se especificaba en el tercer constructor anterior.

En la tabla siguiente se pueden observar los métodos de creación y obtención del color:

Método	Descripción
public Color(int r, int g, int b, int alfa)	Crea un objeto Color especificando cada componente con un valor entre 0 y 255. El canal alfa es opcional
public Color(float r, float g, float b, float alfa)	Crea un objeto Color especificando cada componente con un valor entre 0,0 y 1,0. El canal alfa es opcional
public Color(int rgbAlfa, boolean hayAlfa)	Crea un objeto Color especificando cada componente dentro de un entero Java. El parámetro hayAlfa indica si se quiere o no introducir la transparencia en el color
public int getRed()	Obtiene la componente roja del objeto Color
public int getGreen()	Obtiene la componente verde del objeto Color
public int getBlue()	Obtiene la componente azul del objeto Color
public int getAlpha()	Obtiene el canal alfa del objeto Color
public int getRGB()	Obtiene un entero Java con todas las componentes del objeto Color en cuestión

En la tabla que viene a continuación aparecen los colores predefinidos de forma estática en la clase Color, así como el valor de sus componentes RGB.

Color predefinido	Valor del color
Color.blue	new Color(0,0,255)
Color.green	new Color(0,255,0)
Color.red	new Color(255,0,0)
Color.orange	new Color(255,200,0)
Color.yellow	new Color(255,255,0)
Color.black	new Color(0,0,0)
Color.white	new Color(255,255,255)
Color.grey	new Color(128,128,128)

Color.cyan	new Color(0,255,255)
Color.magenta	new Color(255,0,255)
Color.pink	new Color(255,175,175)
Color.darkGray	new Color(64,64,64)
Color.lightGray	new Color(192,192,192)

Es posible enviar a un objeto Color el mensaje darker() o brighter() para obtener un nuevo Color más oscuro o más claro, respectivamente, que el original.

6.2.1 Un ejemplo con colores

El siguiente programa demuestra las posibilidades de la API Java2D con respecto al color. El ejemplo es muy similar al que puede encontrarse en el libro «Java2D Graphics» de Jonathan Knudsen, un gran experto en la materia. El resultado puede verse en la figura 6.1 y el código es el siguiente:

```
import java.awt.*;
import javax.swing.*;
import java.awt.geom.Rectangle2D;

public class PanelDeColores {
    public static void main(String[] args) {
        JFrame f = new JFrame("Bloques de colores") {
            public void paint(Graphics g) {
                Graphics2D g2 = (Graphics2D) g;
                Dimension d = getSize();
                g2.translate(d.width / 2, d.height / 2);
                Color[] colors = {
                        Color.WHITE, Color.LIGHT_GRAY, Color.GRAY,
                        Color.DARK_GRAY, Color.BLACK, Color.RED,
                        Color.PINK, Color.ORANGE, Color.YELLOW,
                        Color.GREEN, Color.MAGENTA,
                        Color.CYAN, Color.BLUE };
                int limit = colors.length;
                float s = 20;
                float x = -s * limit / 2;
                float y = -s * 3 / 2;
                // Se muestran los colores predefinidos
                for (int i = 0; i < limit; i++) {
                    Rectangle2D r = new Rectangle2D.Float(x + s * i, y, s, s);
                    g2.setColor(colors[i]);
                    g2.fill(r);
                }
                // Se muestra un gradiente lineal
                y += s;
                Color c1 = Color.YELLOW;
```

```
                Color c2 = Color.BLUE;
                for (int i = 0; i < limit; i++) {
                    float ratio = (float) i / (float) limit;
                    int red = (int) (c2.getRed() * ratio + c1.getRed() * (1 - ratio));
                    int green = (int) (c2.getGreen() * ratio + c1.getGreen() * (1 - ratio));
                    int blue = (int) (c2.getBlue() * ratio + c1.getBlue() * (1 - ratio));
                    Color c = new Color(red, green, blue);
                    Rectangle2D r = new Rectangle2D.Float(x + s * i, y, s, s);
                    g2.setColor(c);
                    g2.fill(r);
                }
                // Se muestra un gradiente alfa
                y += s;
                c1 = Color.RED;
                for (int i = 0; i < limit; i++) {
                    int alpha = (int) (255 * (float) i / (float) limit);
                    Color c=new Color(c1.getRed(), c1.getGreen(), c1.getBlue(), alpha);
                    Rectangle2D r = new Rectangle2D.Float(x + s * i, y, s, s);
                    g2.setColor(c);
                    g2.fill(r);
                }
                // Dibujamos un marco alrededor de los cuadrados
                g2.setColor(Color.BLACK);
                g2.draw(new Rectangle2D.Float(x, y-s*2, s * limit, s * 3));
            }
        };
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setSize(300, 200);
        f.setVisible(true);
    }
}
```

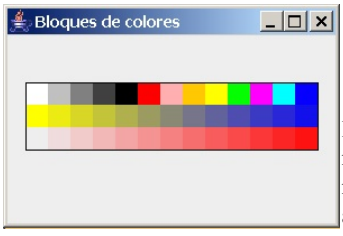


Figura 6.1. Bloques de colores. La fila 1 muestra algunos colores predefinidos, la 2 muestra un gradiente de color y la 3 la aplicación de transparencias progresivas

En la figura 6.1 se ha pintado un rectángulo dividido en tres filas de pequeños bloques de colores. La primera fila sirve para mostrar algunos colores predefinidos por la clase Color. La segunda fila toma un gradiente de color que va progresivamente desde el amarillo hacia el azul, y la tercera fila muestra el tratamiento de la transparencia con el color rojo. Como se puede ver, en el ejemplo

se hace un uso provechoso de los métodos para poder obtener las componentes de un color, como por ejemplo en el caso del gradiente de color, que ha sido codificado manualmente tratando las componentes por separado: nada tiene que ver con la clase GradientPaint ya vista anteriormente.

6.3 Espacios de Colores

Un espacio de colores es un conjunto de colores que puede ser mostrado por un aparato de visualización particular. Un monitor en concreto, por ejemplo, usa un espacio de colores formado por el rojo, el verde y el azul y sus intensidades respectivas, es decir, el espacio de colores RGB.

Ahora bien, si usamos el espacio de colores RGB en otro monitor, éste presentará otro espacio RGB distinto, debido a que las variaciones en la intensidad de los colores son distintas, hay diferencias entre la electrónica de un monitor u otro o incluso debido a especificaciones de un fabricante y otro (véase el concepto de gamut en la figura 1.2). Aunque los 2 monitores usan el mismo tipo de espacio de colores, los dos tienen un espacio de colores que depende del dispositivo. Para solucionar este problema, es necesario representar los colores en términos absolutos, de manera que desde uno a otro dispositivo se puedan trasladar los colores sin que el resultado varíe.

6.3.1 El espacio de colores CIEXYZ

Este espacio de colores ha sido propuesto por la Comisión Internacional de la Luminosidad (en francés *Comission Internationale de l'Eclairage*) y está compuesto por tres componentes básicos X, Y y Z que se corresponden sólo aproximadamente al rojo, verde y azul. Este espacio de colores presenta dos cualidades importantes:

- El componente Y se denomina luminancia y siempre está normalizado al valor 100.
- Los componentes X y Z definen la tonalidad (cualidad por la que un color se distingue de otro) y el cromatismo (grado de gris que tiene un color).
- Los valores de los componentes X, Y y Z son siempre positivos.

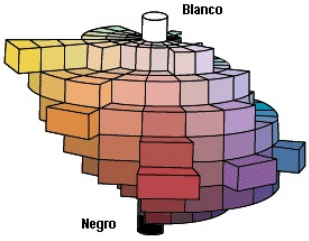


Figura 6.2. Representación tridimensional del espacio de colores CIEXYZ definido por el actor Henry Munsell entre 1898 y 1905

La figura 6.2 muestra una descripción simbólica del significado de cada componente: el eje principal es la luminancia, el ángulo proporciona la tonalidad y la distancia al eje el cromatismo. No suele ser conveniente usar este espacio de colores ya que, por ejemplo, en un monitor RGB, los colores que aparecen son una mezcla de rojo, verde y azul, lo cual implica que una representación en un espacio de colores RGB es más natural. No obstante, ante determinadas circunstancias, el sistema RGB necesita usar valores negativos en alguno de sus componentes para representar algunos colores lo que hace que éstos no puedan visualizarse en dispositivos que sólo aceptan valores positivos.

6.3.2 El espacio de colores sRGB

sRGB son las siglas de estándar RGB. Es una representación RGB del color, pero definida de manera absoluta. Es más sencillo de usar que CIEXYZ pero a la vez menos potente, ya que no se pueden especificar todos los colores en este espacio de colores. sRGB es el espacio de colores por defecto en Java2D. Cuando se crea un Color sin especificar su espacio de colores, éste es el que se usa.

6.4 La clase ColorSpace

La clase java.awt.color.ColorSpace encapsula un espacio de colores específico. Una instancia de esta clase puede representar un espacio de colores dependiente del dispositivo, tal como un espacio RGB de un monitor en concreto, pero también puede representar un espacio de colores independiente del dispositivo como puede ser CIEXYZ. ColorSpace contiene constantes estáticas que representan tipos genéricos de espacios de colores, siendo los más comunes:

- public static final int TYPE_RGB: esta constante representa al tipo de espacio de colores que se obtiene al mezclar rojo, verde y azul.
- public static final int TYPE_GRAY: representa al tipo espacio de colores en escala de grises.
- public static final int TYPE_HSV: constante que representa un tipo de espacio de colores definido por el tono (hue), la saturación (saturation) y el valor (value) de sus componentes. Es un tipo de espacio de colores útil ya que fue diseñado pensando en como la gente aprecia el color, a diferencia de RGB, que fue diseñado pensando en la manera en que trabajaban los monitores y los televisores.
- public static final int TYPE_XYZ: representa un tipo de espacio de colores con coordenadas X, Y y Z, como CIEXYZ.
- public static final int TYPE_CMY: representa un tipo de espacio de colores gracias a la mezcla entre cian (*cyan*), magenta (*magenta*) y amarillo (*yellow*). El negro se representa con la máxima mezcla de los 3 colores.

- public static final int TYPE_CMYK: es idéntico al espacio CMY, pero incorpora el color negro (*black*).

Los espacios de colores absolutos son los que se obtienen mediante el método estático getInstance() que, a partir de una de las constantes anteriores como parámetro, devuelve un objeto de tipo ColorSpace. Como parámetro también pueden usarse espacios de colores particulares (no genéricos), como son:

- public static final int CS_sRGB: representa el espacio de colores sRGB.
- public static final int CS_CIEXYZ: representa el espacio de colores CIEXYZ.
- public static final int CS_GRAY: representa un espacio de colores en escala de gris.
- public static final int CS_PYCC: representa el espacio de colores usado en las Kodak Photo CDs
- public static final int CS_LINEAR_RGB: representa un espacio RGB especializado.

Cada espacio de colores usa un número determinado de componentes para definir un color. Así, el tipo RGB necesita 3 componentes, que son el rojo, el verde y el azul, mientras que el tipo CMYK necesita 4; el cian, el magenta, el amarillo y el negro. Para poder obtener el número de componentes que un objeto ColorSpace necesita para crear un color concreto puede usarse el método getNumComponents().

Para poder visualizar los componentes de un color concreto es necesario ejecutar una llamada al método getColorComponents(float[] arrayDeComp) de la clase Color, el cual devuelve los componentes como un *array* de valores reales. Si el *array* que se introduce como parámetro no es nulo, las componentes se devuelven en él, mientras que si lo es, se crea un nuevo *array* con los componentes y luego se lo retorna. Aclarar que no se devuelven valores alfa con este método.

Para poder mostrar los componentes de un objeto Color concreto dentro de un ColorSpace diferente a aquél en que fue creado, se puede invocar al método getColorComponents(ColorSpace cs, float[] arrayDeComp), que actúa de manera muy similar al método anterior.

La clase ColorSpace puede convertir directamente un color de un espacio de colores a otro. Estos métodos sólo existen para los espacios RGB y CIEXYZ, y son:

- toCIEXYZ(float[] vcolores): convierte un color definido en el espacio de colores que recibe el mensaje al espacio de colores CIEXYZ. El array que se pasa como parámetro debe tener el número de componentes adecuado. Se devuelve un array de valores float con tres elementos, correspondientes a los componentes X, Y y Z de CIEXYZ.
- fromCIEXYZ(float[] CIEvalores): convierte el espacio CIEXYZ al espacio de colores actual. El array que se pasa como parámetro debe contener tres elementos, correspondientes a X, Y y Z. El array defloat devuelto contiene

el número de componentes de este espacio de colores.
- toRGB(float vcolores[]): este método es análogo al toCIEXYZ().
- fromRGB(float rgbvalores[]): este método es análogo al fromCIEXYZ().

Recordar que sRGB no puede representar todo el espectro de colores, por tanto, a la hora de hacer conversiones, es necesario ser conscientes de que puede haber cierta pérdida de información del color.

Como ya se ha comentado, la clase Color crea por defecto los colores en el espacio sRGB, pero también dispone de un constructor que permite crear colores en otros espacios de colores, Color(ColorSpace cs, float[] componentes, float alfa), que crea un color en el espacio de colores indicado por el parámetro cs. El *array* componentes debe ser de longitud igual al número de componentes del espacio de colores, es decir cs.getNumComponents() y componentes.length deben ser iguales; por último, alfa es un valor entre 0,0 y 1,0. Para poder obtener el espacio de colores de un objeto Color, tan sólo es necesario invocar a su método getColorSpace().

6.5 Perfiles

Una manera de compensar las dependencias de los dispositivos es usando un perfil. Un perfil es un conjunto de información que indica como mapear desde un espacio de colores estándar, tal como CIEXYZ o sRGB, hacia el espacio de colores propio del dispositivo. Por ejemplo, teniendo un monitor que funciona con el espacio de colores RGB, un color puede ser mapeado desde el espacio de colores sRGB de nuestro sistema hacia el espacio de colores del monitor. Si los perfiles son correctos, el color debe parecer el mismo en los dos sistemas.

Los fabricantes de dispositivos de entrada o salida suelen proporcionar perfiles para sus dispositivos, pero estos no son totalmente fiables ya que no tienen en cuenta las variaciones de cada dispositivo. Seguramente sea necesario realizar algún tipo de procedimiento de calibración para obtener una visualización óptima en un dispositivo específico, lo que puede llegar a consumir tiempo. Para facilitar y agilizar

Figura 6.3. Dispositivo que se coloca en la pantalla del monitor y realiza una calibración automática del color, creando un perfil particularmente adaptado para visualizar de forma adecuada la gama estándar de colores.

este proceso existen dispositivos especializados que realizan la calibración de manera automática, como el SpyderTV de ColorVision, que puede verse en la figura 6.3.

En Java, los perfiles se representan como instancias de la clase java.awt.color.ICC_Profile. Existen 3 métodos para instanciar esta clase, que son:

- getInstance(byte[] datos): devuelve un objeto de tipo ICC_Profile con un perfil construido a partir del *array* datos.
- getInstance(InputStream s): crea un perfil a partir de un objeto InputStream.
- getInstance(String nombreFichero): crea un perfil a partir del fichero cuyo nombre se pasa como parámetro.
- getInstance(ColorSpace cs): constuye un perfil a partir de un espacio de colores absouto. El parámetro cs debe ser una de las constantes de la clase ColorSpace descritas anteriormente. Este método no es necesario realmente ya que si se quiere obtener una instancia de uno de los espacios de colores absolutos, es mejor usar el método getInstance() de la clase ColorSpace.

También se puede obtener un espacio de colores a partir de un perfil, lo que se consigue gracias a la clase ICC_ColorSpace (que hereda de ColorSpace) y a su constructor ICC_ColorSpace(ICC_Profile p). Esto puede ser útil si se tienen perfiles para múltiples dispositivos de entrada y salida, ya que facilita la creación de espacios de colores adecuados para ellos.

En la tabla siguiente se muestran todos los métodos de creación de perfiles:

Método	Descripción
public static ICC_Profile getInstance(byte[] data)	Perfil construido a partir de un *array* de datos
public static ICC_Profile getInstance(InputStream s) throws IOException	Perfil construido a partir de un objeto canal de entrada
public static ICC_Profile getInstance(String fileName) throws IOException	Perfil construido a partir de un fichero
public static ICC_Profile getInstance(int cspace)	Perfil construido a partir de un espacio de colores
public ICC_ColorSpace(ICC_Profile profile)	Espacio de colores construido a partir de un perfil

6.6 La verdad sobre el color

Realmente será difícil que el lector necesite lidiar con las conversiones entre CIEXYZ y sRGB por sí mismo. Todas estas conversiones las puede realizar Java sin nuestra gestión explícita. El control del color en imágenes 2D se puede hacer con un

objeto preconstruido de clase RenderingHint, que se explicó en el epígrafe 2.5.1. El nombre de la preferencia es KEY_COLOR_RENDERING y puede adoptar tres posibles valores, que son:

- VALUE_COLOR_RENDER_QUALITY: con este valor, la API 2D hará todo lo posible por encontrar la representación de colores más fidedigna. Esto se lleva a cabo gracias a que la API busca un perfil de color apropiado para la pantalla. Todo el color que se muestre se pasará a un espacio de colores a partir del perfil creado. La manera en la que Java encuentra el perfil depende del sistema operativo que se esté ejecutando. Esta preferencia no podrá aplicarse si la implementación 2D no la soporta o si no sepuede crear el perfil adecuado.
- VALUE_COLOR_RENDER_SPEED: este valor implica que Java2D renderizará los colores de una manera rápida, por lo que no se tomarán en cuenta los perfiles. Así, por ejemplo, los colores en un espacio de colores sRGB serán transferidos directamente a una pantalla sin preocuparse si es el espacio de colores adecuado.
- VALUE_COLOR_RENDER_DEFAULT: este valor hace que Java renderice el color en base a las preferencias establecidas en el sistema operativo.

Java a Tope:

Java2D

Cómo tratar con Java
figuras, imágenes y texto en dos dimensiones

Java2D es una de las APIs (*Application Program Interface*) más ricas que proporciona la edición estándar de Java (J2SE). El presente texto aborda las tres áreas principales que integran Java2D: figuras geométricas, imágenes y dibujo de texto. Asimismo se introducen en profundidad los conceptos generales que pueden aplicarse a todos ellos, como pueden ser el tratamiento del color y de la transparencia, la gestión de transformaciones geométricas o la manipulación de ficheros JPEG.

A medida que se van introduciendo nuevas clases de la API, éstas se van hilvanando entre sí para proporcionar al programador un mayor control sobre el aspecto de los lienzos que quiera dibujar. De esta forma se estudia, entre otras muchas cosas, el dibujo de figuras sobre fotografías digitales, la sobreimpresión de texto sobre imágenes e incluso la utilización del contorno de un texto en negrita como elemento a través del que ver una imagen de fondo.

Cada uno de los conceptos necesarios se introduce desde cero, aunque se suponen conocimientos del lenguaje Java, y se proporcionan numerosos ejemplos completos que pueden ser puestos en práctica sin más modificación que la de elegir la foto preferida que se desea transformar, o modificar la cadena de texto a visualizar. Cada ejemplo se acompaña de una adecuada explicación sobre las líneas de código vitales que producen el efecto que se desea conseguir en cada caso, por lo que al lector le resultará muy cómodo realizar modificaciones personalizadas y obtener sus propios resultados. La inclusión de las imágenes que resultan de cada uno de los ejemplos comentados también ayuda enormemente a la comprensión del código y facilita la asimilación de conceptos.

ISBN 978-84-690-5677-6

UNIVERSIDAD DE MÁLAGA