PRÁCTICA 11

Procesamiento de imágenes

Parte 3

Descripción de la práctica

El objetivo de esta práctica es realizar una aplicación que amplíe la realizada en las prácticas 8, 9 y 10 introduciendo nuevas operaciones definidas por el estudiante. Concretamente, deberá incluir las siguientes nuevas funcionalidades:

- Mostrar bandas
- Cambio de espacio de color
- Operador Sepia

El aspecto visual de la aplicación será el mostrado en la Figura 1. En la parte inferior, además de lo ya incluido en la práctica 10, se incorporará un botón para la operación "sepia", así como un área para trabajar con los espacios de color que incluya (1) un botón para mostrar las bandas de la imagen seleccionada y (2) una lista desplegable para transformar la imagen al espacio de color seleccionado.

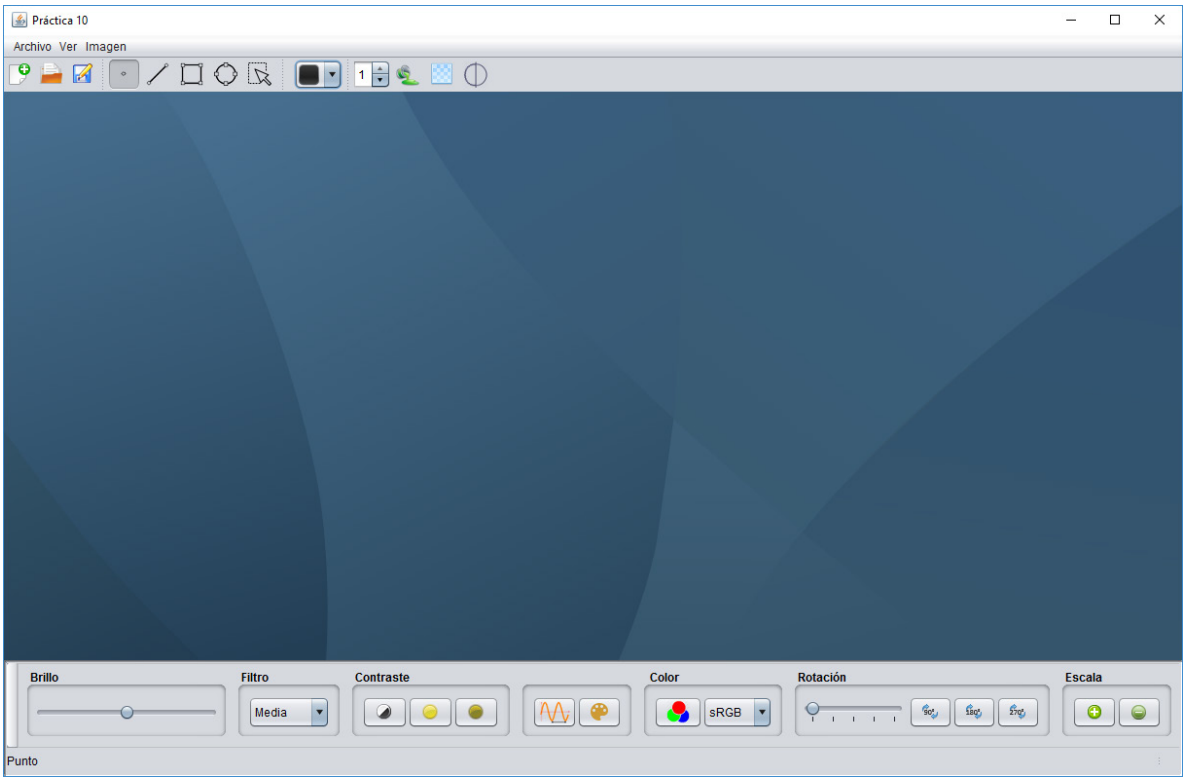

Figura 1: Aspecto de la aplicación

Pruebas iniciales

En una primera parte, probaremos los operadores "`BandCombineOp`" y "`ColorConvertOp`" usando parámetros fijos. Para ello, y en la misma línea que prácticas anteriores, incluiremos las correspondientes opciones en el menú "Imagen" cuya selección implicará la aplicación de la correspondiente operación en la imagen seleccionada.

Extracción y muestra de bandas

En primer lugar incluiremos la opción de mostrar las bandas asociadas a la imagen seleccionada (por ejemplo, para una imagen en RGB, mostraría por separado las bandas correspondientes al rojo, verde y azul). Para ello incluiremos un botón que, al pulsarlo, hará que aparezcan en el escritorio tantas ventanas internas como bandas tenga la imagen; cada una de estas ventanas contendrá una imagen (en niveles de gris) con la correspondiente banda[1].

Como vimos en teoría, no existe un método que nos devuelva una banda concreta para una imagen dada; en su lugar, tendremos que hacer uso de los métodos existentes para, en última instancia, crear nosotros mismos la imagen banda deseada[2,3]:

```
//Creamos el modelo de color de la nueva imagen basado en un espcio de color GRAY
ColorSpace cs = ColorSpace.getInstance(ColorSpace.CS_GRAY);
ComponentColorModel cm = new ComponentColorModel(cs, false, false,
                                                 Transparency.OPAQUE,
                                                 DataBuffer.TYPE_BYTE);

//Creamos el nuevo raster a partir del raster de la imagen original
int bandList[] = {iBanda};
WritableRaster bandRaster = (WritableRaster)src.getRaster().createWritableChild(0,0,
                                    src.getWidth(), src.getHeight(), 0, 0, bandList);

//Creamos una nueva imagen que contiene como raster el correspondiente a la banda
BufferedImage imgBanda = new BufferedImage(cm, bandRaster, false, null);
```

donde `iBanda` es el índice de la banda que queremos extraer. Nótese que el código anterior crea una imagen nueva (en niveles de gris) que comparte parcialmente el buffer de datos de la imagen original (gracias al uso del método `createWritableChild`). Para nuestra aplicación, nos interesa que las imágenes banda estén "vinculadas" a la imagen de la que provienen, esto es, que si modificamos una imagen banda (por ejemplo, su brillo), el cambio se vea reflejado en la imagen original.

Para probar el ejercicio, se aconseja abrir una imagen en RGB, extraer sus bandas y probar a cambiar el brillo en una de ellas; la original responderá aumentando o disminuyendo la influencia de la correspondiente banda[4]. En particular, si llevamos dos de las bandas a cero, se mostrará solo la información aportada por la tercera de las bandas[5]. De igual manera, cualquier operación sobre la imagen original deberá de reflejarse en sus correspondientes bandas[6].

---

[1] Se recomienda que el título de la ventana incluya, entre corchetes, el índice de la banda.

[2] Este código nos permite crear una imagen asociada a una determinada banda; en el ejercicio habría que (1) crear una imagen por cada banda y (2) crear las correspondientes ventanas internas que muestren dichas imágenes banda.

[3] El espacio CS_GRAY introduce una corrección gamma que varía la iluminación de la imagen con respecto a la banda original (si se compara, por ejemplo, con la descomposición en canales de algún otro software, veremos esta variación en la intensidad). Una alternativa que no introduce variaciones es la clase `GreyColorSpace` del paquete `sm.image.color` (bastaría cambiar el código donde se crea la instancia por `ColorSpace cs = new sm.image.color.GreyColorSpace()`)

[4] Esta prueba funcionará solo si está bien implementada la operación de brillo; de no ser así, se recomienda revisar las indicaciones de la práctica 9.

[5] Bug no documentado. Cuando creamos *rasters* hijos (esto es, *rasters* que comparten el buffer de datos con otra imagen), y en particular cuando corresponden a una banda, hay operaciones que no funcionan correctamente al aplicarlas sobre el *raster* hijo. Por ejemplo, los operadores `LookUpOp` no gestionan correctamente el modelo de muestreo (si la imagen es entrelazada, la trata como si fuese por bandas, lo que implica que aplica el operador al primer tercio de cada banda); el mismo problema aparece en el caso del `ConvolveOp`. En el caso de nuestra aplicación, implica que los filtros (suavizado, realce, etc.) y las transformaciones (contrastes, etc.) no funcionarán correctamente al aplicarlas sobre las bandas.

[6] Para ver el efecto, será necesario repintar todo el escritorio y forzar así a que se actualicen tanto la imagen original como las imágenes banda (no es lo más eficiente, pero si suficiente para esta práctica).

Cambio de espacio de color

En este segundo bloque, añadiremos la posibilidad de cambiar de espacio de color. Para ello, incluiremos en nuestra ventana principal una lista desplegable que muestre los distintos espacios de color disponibles (en principio, RGB, YCC y GREY[7]); al seleccionar uno de ellos, deberá de aparecer en el escritorio una nueva ventana interna mostrando la nueva imagen en el espacio de color seleccionado (el objetivo es crear una nueva imagen, no sustituir a la original)[8]. Por ejemplo, según vimos en teoría, para convertir una imagen al espacio YCC[9] sería:

```
if (src.getColorModel().getColorSpace().isCS_sRGB()) {
  ColorSpace cs = ColorSpace.getInstance(ColorSpace.CS_PYCC);
  ColorConvertOp cop = new ColorConvertOp(cs, null);
  imgOut = cop.filter(imgSource, null);
}
```

Nótese que en el ejemplo anterior comprobamos antes de la conversión si el espacio de partida es el adecuado (en el caso del ejemplo, RGB). Estrictamente hablando, no sería necesario, ya que fuese cual fuese el espacio inicial lo convertiría a YCC; no obstante, para evitar cálculos innecesarios, podemos chequear antes el espacio y actuar en consecuencia (por ejemplo, no hacer la conversión si la imagen ya está en el espacio solicitado[10]).

Al mostrarla, el aspecto de la imagen será el mismo que el de la imagen original (recordemos que Java hace automáticamente la conversión a RGB para visualizar las imágenes); no obstante, si mostramos las bandas de la nueva imagen, éstas deben de corresponder al del nuevo espacio de color[11,12].

Una vez implementado el cambio de espacios, probad a aplicar las operaciones de prácticas anteriores sobre las imágenes convertidas a YCC. El resultado, ¿es el que esperabas? ¿Sabrías justificar por qué responde de esa forma? Echadle un vistazo a las bandas; ¿qué ocurre si cambio el brillo solo en la banda Y? ¿Y si aplico un filtro?[5]

Sepia

En este último apartado incorporaremos el operador “sepia”. Se trata de uno de los efectos más clásicos en los programas de edición de imágenes, en el que se modifica el tono y saturación para darle un aspecto de “fotografía antigua”. Hay varias transformaciones que producen este tipo de efecto; una de las más populares se define en base a la siguiente ecuación:

$$\begin{aligned} sepiaR &= \min(255\ ,\ 0.393 \cdot R + 0.769 \cdot G + 0.189 \cdot B) \\ sepiaG &= \min(255,\ 0.349 \cdot R + 0.686 \cdot G + 0.168 \cdot B) \\ sepiaB &= \min(255,\ 0.272 \cdot R + 0.534 \cdot G + 0.131 \cdot B) \end{aligned}$$

---

[7] El espacio CS_GRAY introduce una corrección gamma que varía la iluminación de la imagen. Una alternativa que no introduce variaciones es la clase `GreyColorSpace` del paquete `sm.image.color`.

[8] Se recomienda que el título de la ventana incluya, entre corchetes, las siglas del nuevo espacio de color en el que está representada la imagen (p.e, [YCC]).

[9] El código sería similar para otros espacios sin más que cambiar el tipo de espacio en la llamada a `ColorSpace.getInstance`. También habría que adaptar el *if* si se desea hacer alguna comprobación previa.

[10] Salvo para el caso de sRGB, la clase `ColorSpace` no tiene un método para chequear si se es (o no) de un determinado espacio de color (lo que implica que no siempre es inmediato saber en qué espacio se está). Una opción es trabajar con el método `getType` (común a todos los espacios), pero éste devuelve la familia a la que pertenece el espacio (p.e, TYPE_RGB), no el espacio concreto (p.e., CS_sRGB o CS_LINEAR_RGB).

[11] Si se ha usado el espacio CS_GREY para generar las bandas (que, recordemos, introduce una corrección gamma que varía la iluminación) y si se lleva a cabo la secuencia RGB→YCC→RGB, aunque las tres imágenes se verán iguales, las bandas de la primera y última tendrán diferentes intensidades (si bien deberían de ser las mismas). Como ya se indicó, una alternativa que no introduce variaciones es la clase `GreyColorSpace` del paquete `sm.image.color`.

[12] El operador `ColorConvertOp` realiza un paso intermedio cuando va a efectuar la conversión: transforma al espacio CIEXYZ y, de ahí, al espacio solicitado. Este hecho, además de ralentizar el proceso, hace más complejo la definición de nuevas clases de espacios de color: además de la transformación to/fromRGB, exige la implementación del to/fromXYZ (que no siempre es trivial).

con [R,G,B] el color del pixel original. Nótese que en la ecuación anterior hay que tener en cuenta que si el valor obtenido para un componente es superior a 255, hay que truncarlo a 255[13].

Para ello definiremos nuestra propia clase `SepiaOp` de tipo `BufferedImageOp`; concretamente, crearemos el paquete `sm.xxx.imagen` en nuestra librería (con `xxx` las iniciales del estudiante) y definiremos dentro de dicho paquete la nueva clase. Siguiendo el esquema visto en teoría, haremos que herede de `sm.image.BufferedImageOpAdapter` y sobrecargue el método `filter`:

```
public class SepiaOp extends BufferedImageOpAdapter{


  public SepiaOp () {
  }

  public BufferedImage filter(BufferedImage src, BufferedImage dest){
     if (src == null) {
       throw new NullPointerException("src image is null");
     }
     if (dest == null) {
       dest = createCompatibleDestImage(src, null);
     }

     for (int x = 0; x < src.getWidth(); x++) {
       for (int y = 0; y < src.getHeight(); y++) {

         //Por hacer: efecto sepia

     }
     return dest;

  }
}
```

En el método `filter` recorreremos la imagen y, para cada pixel, aplicaremos la ecuación anterior y le asignaremos valor a la imagen destino en función del resultado. Recordar que, al principio del método `filter`, hemos de comprobar si la imagen destino es `null`, en cuyo caso tendremos que crear una imagen destino compatible (llamando a `createCompatibleDestImage`)[14]. En cualquiera de los casos, recordar que el método `filter` ha de devolver la imagen resultado.

Posibles mejoras para trabajar en casa…

Una vez realizada la práctica, se proponen una serie de mejoras para darle mayor funcionalidad y ampliar las posibilidades en el procesamiento de imágenes:

- Mostrar en la barra de estado el valor (RGB) del pixel sobre el que está situado el ratón.
- Incluir una nueva operación de diseño propio (creando una nueva clase de tipo `BufferedImageOp`).

[13] Si esta condición no fuese necesaria, podríamos optar por usar el operador `BandCombineOp`; el problema está en que dicho operador no trunca a 255 aquellos valores que, al aplicar la operación, superen el 255 (en su lugar, hace un "casting" a byte).

[14] También se aconseja comprobar si la imagen fuente (`src`) es distinta de `null`; en caso de que sea nula, lanzar la excepción `NullPointerException`.