PRÁCTICA 12

Procesamiento de imágenes

Parte 4

Descripción de la práctica

El objetivo de esta práctica es realizar una aplicación que amplíe la realizada en la práctica 11 incluyendo las siguientes nuevas funcionalidades:

- Tintado
- Ecualización
- Operador Sepia
- Umbralización

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 11, se incorporará un deslizador para umbralizar la imagen y dos botones asociados a las operaciones de tintado, ecualización y sepia.

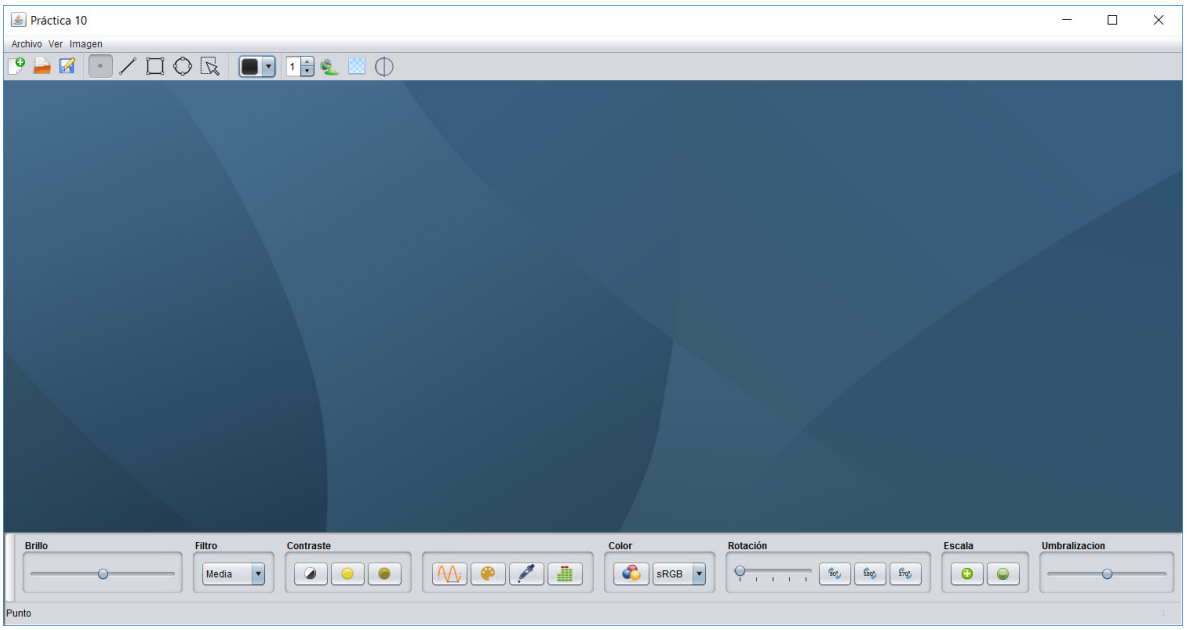

Figura 1: Aspecto de la aplicación

Tintado

En primer lugar incorporaremos la posibilidad de "tintar" la imagen del color seleccionado en la lista desplegable de colores. Para ello usaremos la clase `TintOp` del paquete `sm.image` que implementa el operador de tintado desarrollado en teoría (véanse trasparencias). Dicho operador requiere para su construcción de dos parámetros: el color con el que tintar y el grado de mezcla (valor entre 0.0 y 1.0, con 1.0 indicando máximo tintado). Por ejemplo, para tintar una imagen de color rojo con un grado 0.5 de mezcla, el código sería:

```
TintOp tintado = new TintOp(Color.red,0.5f);
tintado.filter(imgSource, imgSource);
```

donde asumimos que fuente y origen son el mismo. Para esta práctica, el color no sería uno fijo sino el que esté seleccionado en la lista desplegable de colores.

Ecualización

En segundo lugar incorporaremos la posibilidad de ecualizar la imagen seleccionada. Para ello usaremos la clase *EqualizationOp* del paquete *sm.image* que implementa el operador según lo explicado en teoría (véanse trasparencias). En esta implementación se permite ecualizar todas las bandas o indicarle una banda concreta sobre la que trabajar (dejando intactas el resto)[1]; por defecto, aplica la ecualización en todas las bandas. Por ejemplo, para ecualizar una imagen en todas sus bandas, el código sería:

```
EqualizationOp ecualizacion = new EqualizationOp();
ecualizacion.filter(imgSource, imgSource);
```

donde asumimos que fuente y origen son el mismo.

Una vez incorporada esta herramienta a la aplicación, probar a ecualizar una imagen en color; ¿notas algún efecto extraño? Una vez hecha la prueba, usar de nuevo la misma imagen y convertirla a YCC, extraer sus tres bandas y aplicar la ecualización solo en el canal Y, ¿qué tal ahora?[2]

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:

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

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[3].

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 () {
  }
```

[1] Recordemos que, como vimos en teoría, una ecualización en todas las bandas puede generar efectos no deseados (variaciones en color); en particular, este problema se puede observar si aplicamos una ecualización en las tres bandas RGB. Por este motivo, para ecualizar una imagen en color se recomienda pasarla a un espacio de color que, como el YCC, separe la intensidad (en una banda) de la información cromática (resto de bandas) y aplicar la ecualización solo en la intensidad.

[2] La implementación del espacio CS_YCC de java genera un canal Y que tiende a "iluminar" en exceso la imagen tras la ecualización. Una alternativa que no introduce este efecto es la clase *YCbCrColorSpace* del paquete *sm.image.color*.

[3] 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).

```
  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`)[4]. En cualquiera de los casos, recordar que el método `filter` ha de devolver la imagen resultado.

Umbralización

Por último, incorporaremos un operador de umbralización (véanse transparencias de teoría). Para ello definiremos nuestra propia clase[5] `UmbralizacionOp` de tipo `BufferedImageOp` en el paquete `sm.xxx.imagen` de nuestra librería (con `xxx` las iniciales del estudiante). Esta clase tendrá como propiedad principal el valor umbral (que definiremos como una variable) y, siguiendo el esquema visto en teoría, haremos que herede de `sm.image.BufferedImageOpAdapter` y sobrecargue el método `filter`:

```
public class UmbralizacionOp extends BufferedImageOpAdapter{
  private int umbral;

  public UmbralizacionOp(int umbral) {
    this.umbral = umbral;
  }

  public BufferedImage filter(BufferedImage src, BufferedImage dest){

    //Código de umbralización

  }
}
```

Para esta práctica, implementaremos la umbralización basada en intensidad (véanse transparencias de teoría). Como sabemos, este operador genera como resultado una imagen binaria donde los píxeles que superan un umbral (medido en términos de intensidad) se le asigna un valor (p.e, 255) [6] y al resto 0:

$$g(x,y) = \begin{cases} 255 & si\ I(x,y) \geq T \\ 0 & si\ I(x,y) < T \end{cases}$$

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

[5] El paquete `sm.image` contiene la clase `ThresholdOp` que permite realizar umbralizaciones basadas en intensidad (por defecto), en color (si se le pasa un color en el constructor) o por bandas. El objetivo de esta práctica no es usar dicha clase, sino crear una propia e implementar la operación.

[6] Otra opción sería asignar, en lugar de 255, el color original del pixel (en este caso no obtendríamos una imagen binaria).

con $I(x,y)$ la intensidad del pixel calculada como la media de sus componentes $I(x,y) = (r(x,y) + g(x,y) + b(x,y))/3$

Por tanto, en el método `filter` recorreremos la imagen y, para cada pixel, aplicaremos la operación anterior y asignaremos 0 o 255 en la imagen destino en función del resultado. Al margen de lo anterior, 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`)[7]. 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:

- Incluir un deslizador para poder ir variando el grado de mezcla (valor de alfa) en la operación de tintado.
- Incorporar la umbralización basada en color. En este caso, además del umbral, el usuario deberá elegir el color central de la umbralización (véanse transparencias de teoría)[8].
- Crear un diálogo que muestre el histograma[9].

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

[8] Implementada en `sm.image.ThresholdOp` (no es necesario crear clase propia, se puede comprobar el efecto usando la existente en el paquete `sm.image`).

[9] El cálculo del histograma se encuentra implementado en `sm.image.Histogram`. La complejidad de esta mejora no está, por tanto, en el cálculo del histograma, sino en el dibujo del histograma (usando lo visto en los temas y prácticas de gráficos).