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 las prácticas anteriores incluyendo las siguientes nuevas funcionalidades:

- Tintado
- Sepia
- Posterización
- Filtro rojo

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 incorporarán cuatro botones para las operaciones de tintado, sepia y filtro rojo, así como un deslizador para la posterización.

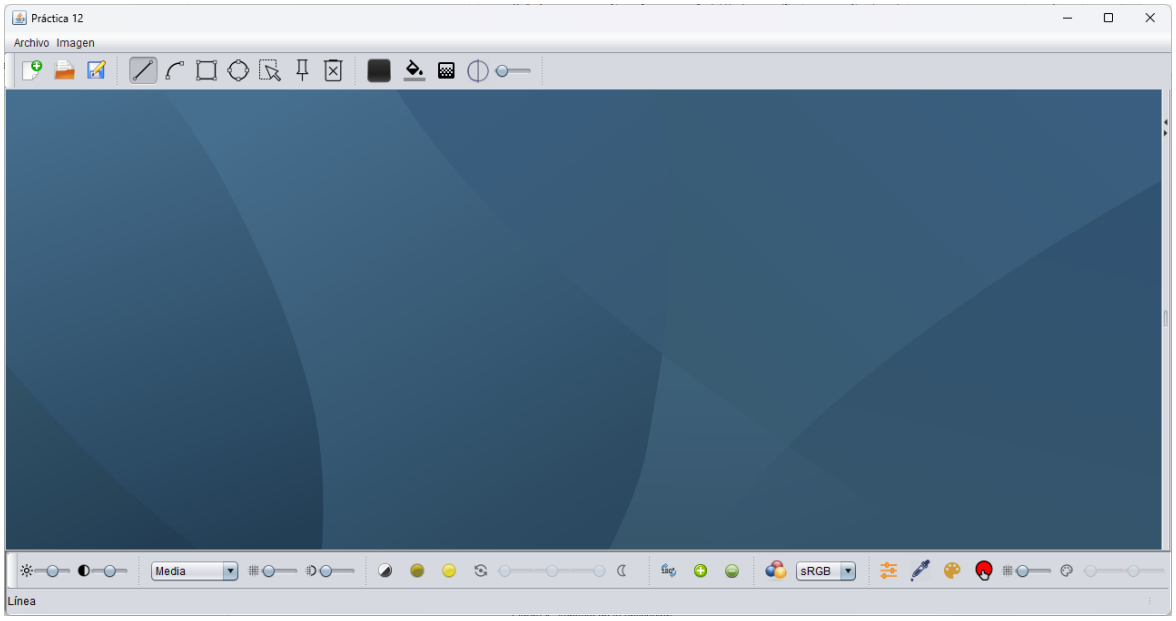

Figura 1: Aspecto de la aplicación

Tintado

En primer lugar, incorporaremos la posibilidad de "tintar" la imagen del color seleccionado en el selector 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(img,img);
```

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. Nótese que el operador de tintado admite que imagen fuente y destino sean la misma; en este ejemplo hacemos uso de esta posibilidad, lo que permite actualizar la imagen del lienzo de una forma rápida y eficiente.

Sepia

En segundo lugar, incorporaremos la opción de aplicar el efecto "sepia" a la imagen seleccionada. 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". Para ello usaremos la clase `SepiaOp` del paquete `sm.image` que implementa el operador desarrollado en teoría (véanse trasparencias). En este caso no hay parámetros asociados, por lo que el código para aplicar este efecto sería:

```
SepiaOp sepia = new SepiaOp();
sepia.filter(img,img);
```

De nuevo, hacemos uso de la posibilidad que ofrece el operador de que imagen fuente y destino sean la misma.

Posterizar: reduciendo el número de colores

En este apartado incorporaremos un operador de diseño propio que aplique el efecto de "posterizar", consistente en reducir el número de colores de una imagen a un número específico de niveles. La operación se aplica componente a componente[1] y viene dada por la siguiente ecuación:

$$comp_{poterizado} = K * \lfloor comp_{original} / K \rfloor$$

con $\lfloor \cdot \rfloor$ la función suelo[2] y $K = {}^{256}/_{N}$, siendo $N \in [1,256]$ un parámetro que indica el número de niveles al que se reduce la correspondiente banda.

Para abordar este operador tendremos que definir nuestra propia clase `PosterizarOp` 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 PosterizarOp extends BufferedImageOpAdapter{
  private int niveles;

  public PosterizarOp (int niveles) {
      this.niveles = niveles;
  }

  public BufferedImage filter(BufferedImage src, BufferedImage dest){
     if (src == null) {
       throw new NullPointerException("src image is null");
     }
     if (dest == null) {
       dest = createCompatibleDestImage(src, null);
     }
     WritableRaster srcRaster = src.getRaster();
     WritableRaster destRaster = dest.getRaster();
     int sample;

     for (int x = 0; x < src.getWidth(); x++) {
       for (int y = 0; y < src.getHeight(); y++) {
          for (int band = 0; band < srcRaster.getNumBands(); band++){
            sample = srcRaster.getSample(x, y, band);

           //Por hacer: efecto posterizar

            destRaster.setSample(x, y, band, sample);
          }
```

[1] Por ejemplo, para una imagen RGB se aplicaría de forma independiente en cada componente R, G y B

[2] Devuelve la parte entera del argumento (p.e., $\lfloor 2.3 \rfloor = 2$). A nivel de programación, equivale a hacer el *casting* a entero.

```
            }
        }
        return dest;
    }

}
```

Como propiedad de la clase, tendremos que incorporar el parámetro asociado al operador, esto es, los niveles de color de la posterización. En el constructor habrá que pasarle dicho parámetro para inicializar la correspondiente variable. En el método `filter` recorreremos la imagen y, para cada componente, aplicaremos la ecuación de posterizado. Nótese que, en este caso se trata de una iteración "componente a componente", ya que para el cálculo del valor destino no hace falta conocer simultáneamente los tres componentes rojo, verde y azul. 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`)[3]. En cualquiera de los casos, recordar que el método `filter` ha de devolver la imagen resultado.

A nivel de interfaz de usuario, incorporaremos un deslizador con el que poder variar el valor del número de niveles al que se reducen las bandas (el parámetro *N* de la ecuación)[4]. Al igual que ocurrió con las operaciones de cambio de brillo y giro, tendremos que tener en cuenta que la imagen sobre la que se aplica la operación ha de ser la original, y no la obtenida temporalmente durante el proceso[5].

Resaltando el tono rojo…

En este apartado incorporaremos un operador de diseño propio que resalte el tono rojo, dejando el resto en niveles de gris. Para ello se propone una operación muy sencilla que considera los tres componentes rojo, verde y azul de la imagen original para calcular el valor del pixel destino. Concretamente, la operación responde a la siguiente ecuación:

$$g(x,y) = \begin{cases} f(x,y) & si\ \ R_f - G_f - B_f \geq T \\ \dfrac{R_f + G_f + B_f}{3} & en\ otro\ caso \end{cases}$$

donde $f(x,y) = [R_f, G_f, B_f]$ es la imagen original y $g(x,y)$ la imagen resultado. La interpretación de la ecuación anterior es la siguiente: para detectar si un píxel es predominantemente rojo, se comprueba que la aportación de la banda roja frente a las demás es significativa[6]; para ello, se mide la diferencia del componente rojo frente a la suma de los componentes verde y azul y, si ésta es superior a un umbral, se considera que el pixel en cuestión es rojo. En caso de que el pixel sea considerado rojo, se mantendrá su color actual (es decir, el mismo que tenía en la imagen original); si, por el contrario, no se considera rojo, se le asignará a cada componente de la imagen destino el valor medio de las tres bandas (lo que equivale a transformarlo en nivel de gris)[7]. En este operador existe un único parámetro: el umbral para la selección de pixel rojo.

---

[3] También se aconseja comprobar si la imagen fuente (`src`) es distinta de `null`; en caso de que sea nula, lanzar la excepción `NullPointerException`. Adicionalmente, se pueden incorporar otras comprobaciones como, por ejemplo, que está en el espacio de color adecuado (en este caso, RGB).

[4] Aunque el rango de *N* está entre 1 y 256, se aconseja poner un límite superior en el deslizado más bajo (p.e., 20), ya que con valores altos el efecto es prácticamente imperceptible. Como límite inferior, se aconseja 2.

[5] Al igual que en la práctica 9, se recomienda usar el evento "ganar foco" para identificar el comienzo de la operación (donde se creará una copia de la imagen que haya en lienzo) y el evento "perder foco" para identificar el final de la operación. Véase práctica 9 para más detalles.

[6] Nótese que no bastaría con comprobar que el valor de la banda roja es elevado, ya que hay colores que son resultado de mezclar el rojo con otros componentes (por ejemplo, el amarillo, que resulta de combinar el rojo y el verde).

[7] El valor de cada componente de la imagen destino será el mismo, es decir, se cumplirá que $g(x,y) = [meida, media, media]$ con $media = \frac{R_f+G_f+B_f}{3}$.

Para abordar este operador tendremos que definir nuestra propia clase `RojoOp` de tipo `BufferedImageOp`, que incluiremos en el paquete `sm.xxx.imagen` en nuestra librería (con `xxx` las iniciales del estudiante). Al igual que en el operador anterior, y siguiendo el esquema visto en teoría, definiremos la nueva clase heredando de `sm.image.BufferedImageOpAdapter`:

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

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

  public BufferedImage filter(BufferedImage src, BufferedImage dest){
     if (src == null) {
       throw new NullPointerException("src image is null");
     }
     if (dest == null) {
       dest = createCompatibleDestImage(src, null);
     }
     WritableRaster srcRaster = src.getRaster();
     WritableRaster destRaster = dest.getRaster();
     int[] pixelComp = new int[srcRaster.getNumBands()];
     int[] pixelCompDest = new int[srcRaster.getNumBands()];

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

        //Por hacer: efecto resaltar rojo

        destRaster.setPixel(x, y, pixelCompDest);
       }
     }
     return dest;
  }

}
```

Como propiedad de la clase, tendremos que incorporar el parámetro asociado al operador, esto es, el umbral de selección de la tonalidad roja. En el constructor habrá que pasarle dicho parámetro para inicializar la correspondiente variable. En el método `filter` recorreremos la imagen y, para cada píxel, aplicaremos la ecuación anterior y le asignaremos valor a la imagen destino en función del resultado. Nótese que estamos ante una iteración "pixel a pixel", ya que para el cálculo del valor destino hace falta conocer simultáneamente los tres componentes R, G y B.

El reto final…

Por último, se propone implementar un operador que permita modificar el tono de un color de la imagen seleccionado (manteniendo su saturación y brillo). Los parámetros del operador serán (1) el color "C" que se quiere cambiar, (2) un umbral $T \in [0,360]$ que indique el margen de aceptación para que el tono de un color dado se considere similar a "C" y (3) el desplazamiento de tono $\varphi \in [0,360]$ a aplicar para modificar el color. Para ello, en cada píxel P_{xy} haremos la siguiente secuencia de operaciones:

1. Convertimos el color RGB a HSB. Para ello se aconseja utilizar el método estático `RGBtoHSB`[8] de la clase `Color`.

[8] El método `RGBtoHSB` recibe como parámetros los valores R, G y B en el rango [0,255]; admite como cuarto parámetro un vector donde almacenar el resultado de la transformación (si es `null`, crea internamente el vector y lo devuelve). Los valores H, S y B devueltos por este método están en el rango [0,1], por lo que habrá que multiplicarlos por la constante adecuada para llevarlos al rango deseado. En particular, el valor de H hay que multiplicarlo por 360 para que $H \in [0,360]$; nótese que la operación definida en este ejercicio asume que H está en el rango [0,360], por lo que será necesaria hacer esta multiplicación antes de aplicarla.

2. Si se cumple que el tono H_{Pxy} del píxel P_{xy} es similar al tono H_C del color "C", entonces modificaremos el tono del píxel P_{xy} (si no, lo dejaremos igual); concretamente, se aplicará la siguiente transformación:

$$H_{Resultado} = \begin{cases} (H_{Pxy} + \varphi) \bmod 360 & si\ distancia(H_C, H_{Pxy}) \leq T \\ H_{Pxy} & en\ otro\ caso \end{cases}$$

donde $\varphi \in [0,360]$ y $T \in [0,360]$ son los parámetros del operador. Los valores S (saturación) y B (brillo) los dejamos igual (solo se modifica el tono, que viene dado por el componente H).

Para comprobar si dos tonos son similares, calcularemos su distancia teniendo en cuenta que el dominio es circular. La distancia entre dos tonos $H_1, H_2 \in [0,360]$ viene dada por la siguiente ecuación:

$$distancia(H_1, H_2) = \begin{cases} |H_1 - H_2| & si\ |H_1 - H_2| \leq 180 \\ 360 - |H_1 - H_2| & en\ otro\ caso \end{cases}$$

3. Convertimos el nuevo color HSB a RGB. Para ello se aconseja utilizar el método estático `HSBtoRGB`[9] de la clase `Color`.

En la aplicación incluiremos un botón de dos posiciones para activar/desactivar este operador (véase Figura 1)[10]; junto al botón, se incluirán dos deslizadores: uno para el parámetro $\varphi \in [0,360]$ y otro para el parámetro $T \in [0,360]$. Al activar la operación, el usuario podrá ir variando los valores de los parámetros e ir viendo el efecto sobre la imagen que haya en el lienzo al comienzo de la operación; el resultado se dará por definitivo cuando se pulse de nuevo el botón y se desactive el operador.

Nótese que, como caso particular, si el valor de T es 360, lo que implica que cualquier color será similar a "C", al cambiar el valor del parámetro estaremos cambiando el tono de todos los píxeles de la imagen. Pruébalo, te resultará curioso.

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.

---

[9] El método `HSBtoRGB` recibe como parámetros los valores H, S y B en el rango [0,1]. Esto implica que, si están en otro rango, habrá que hacer la correspondiente división para llevarlo al rango adecuado. En particular, si la H está en el rango [0,360] (como es nuestro caso en este operador), será necesario dividirla por 360 antes de pasarla como parámetro al método `HSBtoRGB`. Por otro lado, nótese que este método devuelve el color RGB codificado como un entero; como ya vimos en teoría, existen varias formas para extraer los componentes R, G y B a partir de ese entero (véanse transparencias). Por ejemplo, usando operadores binarios:

```
r = (colorint >> 16) & 0xFF;
g = (colorint >> 8) & 0xFF;
b =  colorint & 0xFF;
```

con `colorint` el color RGB codificado como un entero.

[10] Nótese que, en este caso, el inicio y fin de operación estarán asociados al evento de *actionPerformed* del botón, distinguiéndose el inicio o fin en función de que esté o no seleccionado. Recordemos que, dado que es una operación "interactiva", en el inicio se creará una copia de la imagen y en el fin se liberará.

- Incluir un deslizador para poder ir variando el umbral de selección de tono rojo en la operación de resaltado de rojo.
- Define una nueva operación de diseño propio, creando para ello tu propia clase que herede de `BufferedImageOp`. Tienes libertad para elegir la operación; inspírate en las que hemos visto en clase, deja volar tu imaginación e ¡invéntate algo! Solo hay dos condiciones: (1) la operación ha de ser del tipo que hemos llamado “pixel a pixel”, es decir, para obtener el nuevo valor de un pixel se han de considerar todos los componentes del pixel origen en el cálculo; (2) el operador ha de tener, al menos, un parámetro