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
- Sepia
- Ecualización
- 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, ecualización, sepia y filtro rojo, así como un deslizador para la posterización.

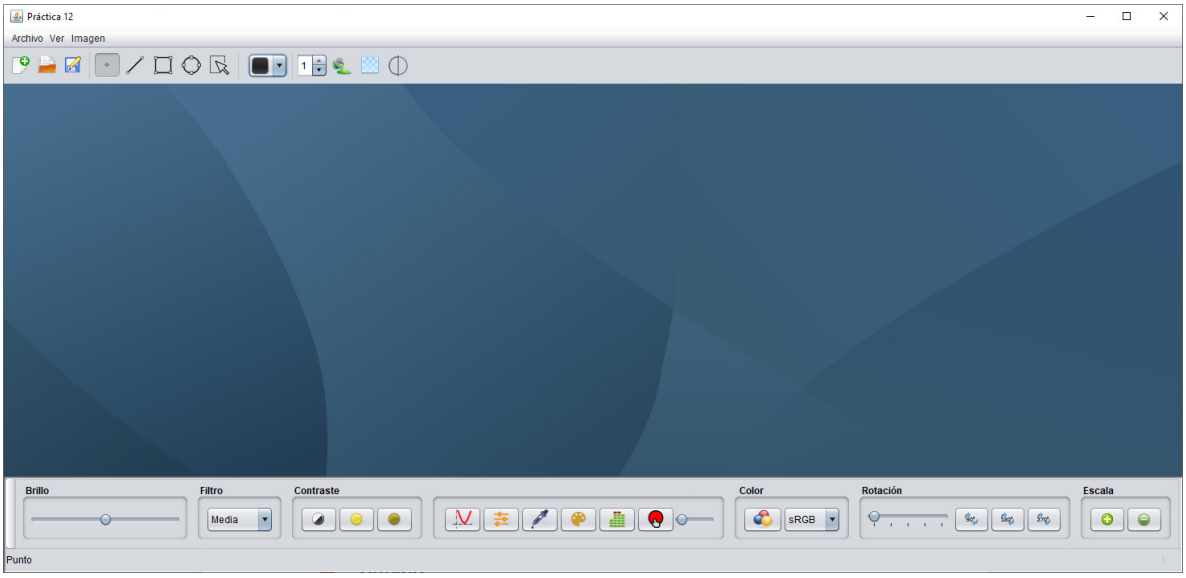

Figura 1: Aspecto de la aplicación

Tintado

En primer lugar incorporaremos la posibilidad de "tintar" la imagen del color seleccionado en 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(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.

Ecualización

En tercer 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(img,img);
```

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]

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[3] y viene dada por la siguiente ecuación:

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

con $\lfloor \cdot \rfloor$ la función suelo[4] y $K = {}^{256}/_{N}$, siendo $N \in [1,256]$ un parámetro que indica el número de niveles al que se reduce la correspondiente banda.

---

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

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);
          }
       }
     }
     return dest;
  }

}
```

Como propiedad de la clase, tendremos que incorporar el parámetro asociado al operador, esto es, los niveles de color de las 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`)[5]. 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)[6]. 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[7].

---

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

[5] 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).

[6] 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.

[7] 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.

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 pixel es predominantemente rojo, se comprueba que la aportación de la banda roja frente a las demás es significativa[8]; 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)[9]. En este operador existe un único parámetro: el umbral para la selección de pixel rojo.

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 RedOp extends BufferedImageOpAdapter{
  private int umbral;

  public RedOp (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;
  }

}
```

[8] 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).

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

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 pixel, aplicaremos la ecuación anterior y le asignaremos valor a la imagen destino en función del resultado. Nótese que, en este caso se trata de una iteración "pixel a pixel", ya que para el cálculo del valor destino hace falta conocer simultáneamente los tres componentes rojo, verde y azul.

El reto final…

¡Haz tu propio operador! Define una nueva operación de diseño propio, creando para ello tu propia clase que herede de `BufferedImageOp`. Tienes libertad para elegir la operación[10]; 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[11]; (2) el operador ha de tener, al menos, un parámetro.

Incluye la nueva clase en el paquete `sm.xxx.imagen`. Finalmente, añade un botón (o lo que necesites) en tu barra de herramientas asociado a el nuevo operador.

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.
- Incluir un deslizador para poder ir variando el umbral de selección de tono rojo en la operación de resaltado de rojo.
- Definir un nuevo operador de tintado, al que llamaremos "`TintadoAutoOp`", que reciba como parámetro el color a tintar (como en el caso de `TintOp`), pero que calcule automáticamente el grado de mezcla. Dicho valor se calculará para cada pixel como $\alpha(x,y) = I(x,y)/255$, siendo $I(x,y) = (r(x,y) + g(x,y) + b(x,y))/3$ el nivel de gris del pixel (calculado como la media de las bandas roja, verde y azul). De esta forma, se tintarán con mayor grado aquellos píxeles con un alto nivel de gris (más claros), mientras que los de tonalidad oscura verán reducido su nivel de tintado.
- Crear un diálogo que muestre el histograma[12].
- Al igual que se propuso en la práctica 7 en lo relativo a la gestión de las herramientas de dibujo, también en este caso se propone mejorar la barra de herramienta vinculada a operaciones sobre imágenes. Con carácter general, se propone simplificarla dándole un aspecto "lineal" sencillo. Por ejemplo, un posible diseño sería algo en la siguiente línea[13]:

[10] No te preocupes si no sale un efecto chulo, lo importante es que hagas tu propio operador (aunque sea sencillo).
[11] El operador que resalta el rojo es un ejemplo de operador "pixel a pixel".
[12] 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).
[13] Los iconos de este ejemplo están descargados del repositorio de google https://material.io/icons/.