PRÁCTICA 11.a

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:

- Operador Sepia
- Operadores binarios

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 operaciones binarias que incluya (1) dos botones asociados a la suma y resta, y (2) un deslizador para mezclar dos imágenes.

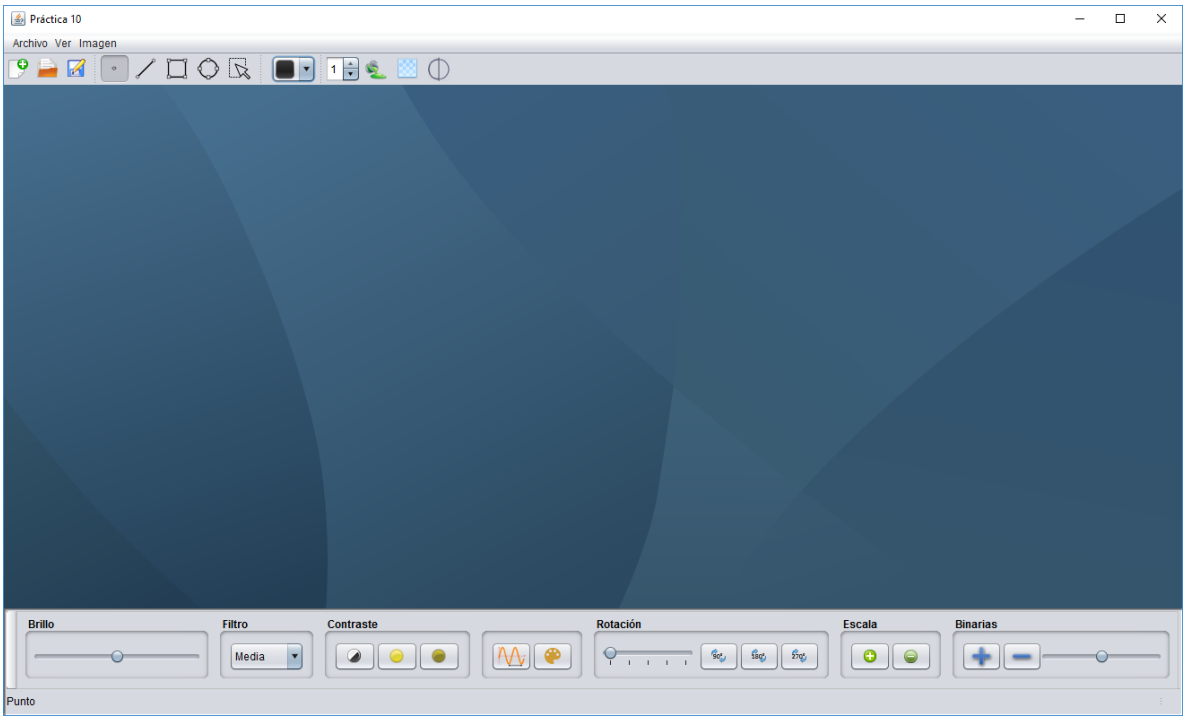

Figura 1: Aspecto de la aplicación

Sepia

En este primer bloque, 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:

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

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[1].

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);
     }

     BufferedImagePixelIterator.PixelData pixel;
     for(BufferedImagePixelIterator it=new BufferedImagePixelIterator(src); it.hasNext();) {
        pixel = it.next();

        //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. Para recorrer la imagen pixel a pixel usaremos el iterador `BufferedImagePixelIterator` (véase plantilla explicada en teoría)[2]. 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.

---

[1] Si esta condición no fuese necesaria, podríamos optar por usar el operador `ConvolveOp`; 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).

[2] Recordemos que, en cada iteración, el iterador nos dará un objeto de la clase `BufferedImagePixelIterator.PixelData`. Dicha clase tiene asociadas cuatro variables: `public int col` (que almacena la columna del píxel, o coordenada x), `public int row` (que almacena la fila del píxel, o coordenada y), `public int numSamples` (que almacena el número de componentes del pixel) y `public int[] sample` (que almacena los valores de los componentes del píxel). Así, por ejemplo, para conocer el valor del componente R dado un `PixelData p` (en el caso de una imagen de tipo RGB), sería `p.sample[0];` de igual forma, para saber las coordenadas x e y del pixel sería `p.col` y `p.row` (respectivamente).

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

Operadores binarios: suma y resta

En este bloque incorporaremos dos operadores aritméticos: la suma ponderada y la resta. Para ello, usaremos las clases `BlendOp` y `SubtractionOp` definidas en el paquete `sm.image` (ambas clases están implementadas según lo visto en teoría)[4].

Para simplificar la interacción, y dado que estas operaciones requieren de dos imágenes, el operador se aplicará sobre la imagen de la ventana seleccionada y la imagen de la ventana anterior a la seleccionada (a la que accedemos mediante el mensaje `selectFrame`):

```
private void botonSumaActionPerformed(ActionEvent evt) {
  VentanaInterna vi = (VentanaInterna) (escritorio.getSelectedFrame());
  if (vi != null) {
    VentanaInterna viNext = (VentanaInterna) escritorio.selectFrame(false);
    if (viNext != null) {
      BufferedImage imgRight = vi.getLienzo().getImage();
      BufferedImage imgLeft = viNext.getLienzo().getImage();
      if (imgRight != null && imgLeft != null) {
        try {
          BlendOp op = new BlendOp(imgLeft);
          BufferedImage imgdest = op.filter(imgRight, null);
          vi = new VentanaInterna();
          vi.getLienzo().setImage(imgdest);
          this.escritorio.add(vi);
          vi.setVisible(true);
        } catch (IllegalArgumentException e) {
            System.err.println("Error: "+e.getLocalizedMessage());
        }
      }
    }
  }
}
```

Mezcla de dos imágenes

En este bloque mezclaremos dos imágenes (*blending*) de forma interactiva aplicando la suma ponderada. De nuevo usaremos la clase `BlendOp`, si bien en este caso variaremos el valor alfa de la mezcla[4] (véanse transparencias de teoría). Para ello, incluiremos un deslizador en el panel de imágenes que permita seleccionar el valor de alfa[5].

Al igual que ha ocurrido en otras operaciones basadas en deslizador, hay que tener en cuenta que las imágenes sobre la que se aplica la operación han de ser las originales: cada nuevo valor del deslizador implicará calcular la imagen (temporal) resultado de aplicar la mezcla sobre las dos imágenes originales (la imagen resultado se irá mostrando en la ventana mientras el usuario mueve el deslizador)[6].

---

[4] Por defecto, la suma ponderada `BlendOp` usa como factor de mezcla alfa=0.5. Si se desea usar otro valor, puede indicarse en el constructor o modificarse mediante el método "`setAlpha`".

[5] Alfa ha de estar entre 0 y 1, si bien el deslizador no admite valore flotantes. Para ello, pondremos como valores mínimo y máximo del deslizador 0 y 100, respectivamente, calculando el valor de alfa como el valor del deslizador dividido por 100 (`getValue()/100`)

[6] En este caso no bastará, como en prácticas anteriores, con definir una sola variable de tipo `BufferedImage` en la ventana principal: serán necesarias dos, una por cada imagen fuente. Nótese que en este caso, además de asignar valor a las variables, hay que lanzar una ventana interna (si seguimos la filosofía de otras prácticas, esto lo haremos cuando el deslizador de mezcla gane el foco). Otra opción, como alternativa a declarar variables miembro en la ventana principal para las imágenes fuentes, sería crearse una ventana interna específica para la operación mezcla (que tendría como variables miembro las imágenes a mezclar).

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.
- Incorporar la operación de "tintado"[7].
- Incorporar nuevas operaciones binarias (por ejemplo, la multiplicación). Para ello, definir en el paquete `sm.xxx.imagen` nuevas clases, una por operador, que hereden de `sm.image.BinaryOp` y sobrecargarán el método `binaryOp` (véanse trasparencias de teoría).

[7] Implementada en `sm.image.TintOp`.