PRÁCTICA 9

Procesamiento de imágenes

Parte 1

Descripción de la práctica

El objetivo de esta práctica es realizar una aplicación que amplíe la realizada en la práctica 8, introduciendo nuevas funcionalidades relativas al procesamiento imágenes. Concretamente, deberá incluir las siguientes funcionalidades:

- Posibilidad de variar el brillo de la imagen.
- Aplicación de filtros básicos (emborronamiento, enfoque, relieve, fronteras, etc.)

El aspecto visual de la aplicación será el mostrado en la Figura 1. El menú incluirá, además de la opción "Archivo", una nueva opción "Imagen" en la cual iremos incorporando ítems asociados a operaciones básicas (con parámetros fijos); para esta práctica, esta opción incluirá los ítems "ResacaleOp" y "ConvolveOp". En la parte inferior, se incluirá un deslizador que permita modificar el brillo de la imagen activa, así como una lista desplegable con los distintos filtros que se puedan aplicar.

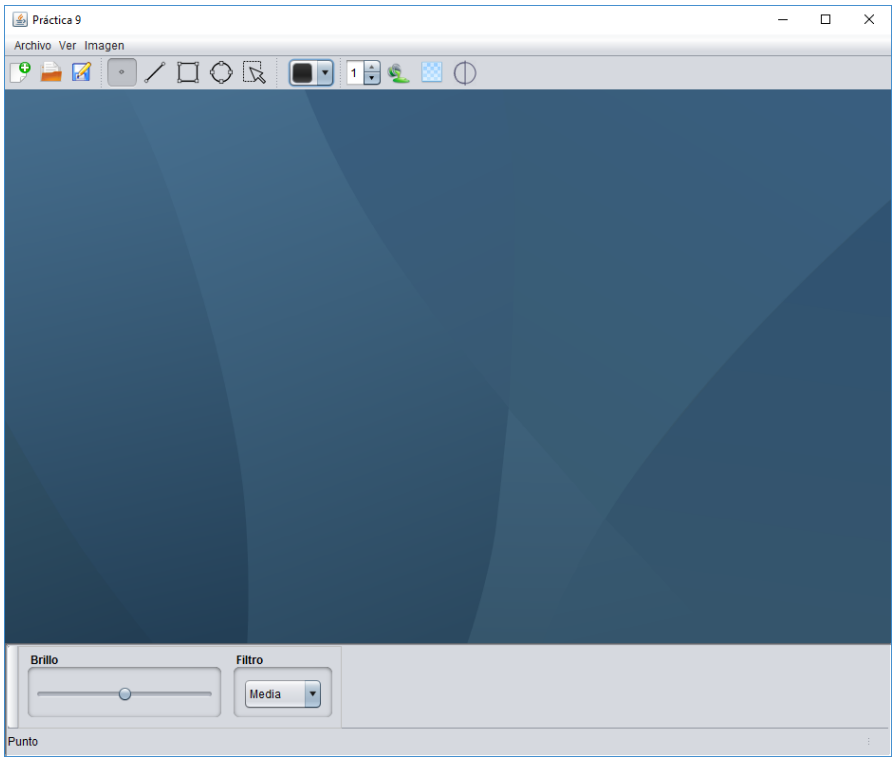

Figura 1: Aspecto de la aplicación

Pruebas iniciales

En una primera parte, probaremos los operadores "`RescaleOp`" y "`ConvolveOp`" usando parámetros fijos (i.e., sin interacción del usuario para definir sus valores). Para ello, incluiremos dos opciones en el menú "Imagen" cuya selección implicará la aplicación de la correspondiente operación en la imagen seleccionada.

En estos casos, se probará el código mostrado en las transparencias de teoría (que incluiremos en el manejador del evento "acción" asociado al menú), teniendo en cuenta que dicha operación se aplicará sobre la imagen mostrada en la ventana activa. Esto supondrá que, al código visto en teoría, habrá que añadirle las sentencias para el acceso y actualización de la imagen; por ejemplo, para el caso de la operación "RescaleOp":

```
private void menuRescaleOpActionPerformed(ActionEvent evt) {
  VentanaInterna vi = (VentanaInterna) (escritorio.getSelectedFrame());
  if (vi != null) {
    BufferedImage img = vi.getLienzo().getImage();
    if(img!=null){
      try{
         RescaleOp rop = new RescaleOp(1.0F, 100.0F, null);
         rop.filter(img, img);
         vi.getLienzo().repaint();
      } catch(IllegalArgumentException e){
         System.err.println(e.getLocalizedMessage());
      }
    }
  }
}
```

Obsérvese que en el código anterior hacemos uso de la posibilidad que ofrece el método `filter` de indicarle una imagen destino previamente creada; en este caso, además, es posible que imagen fuente y destino coincidan, por lo que se aprovecha esta circunstancia para actualizar la imagen del lienzo de una forma rápida y eficiente (sin necesidad de llamar al `setImage()`). Si hubiésemos optado por pasarle `null` como segundo parámetro, el método `filter` habría creado una nueva imagen salida, por lo que hubiese sido necesario actualizar la imagen del lienzo de forma explícita[1]:

```
try{
  RescaleOp rop = new RescaleOp(1.0F, 100.0F, null);
  BufferedImage imgdest = rop.filter(img, null);
  vi.getLienzo().setImage(imgdest);
  vi.getLienzo().repaint();
}
```

Nótese que la primera opción es válida en el caso de que el operador permita que la imagen fuente y destino sean la misma en la llamada a `filter` (como, por ejemplo, `RescaleOp`). No obstante, hay que tener en cuenta que esto no siempre es así (por ejemplo, el operador `ConvolveOp` exige que fuente y destino sean distintos).

Para el caso de la operación "ConvolveOp", el código será similar al anterior pero cambiando el operador; por ejemplo, para el filtro de suavizado visto en teoría (véanse transparencias) el código sería:

```
float filtro[] = {0.1f, 0.1f, 0.1f, 0.1f, 0.2f, 0.1f,0.1f, 0.1f, 0.1f};
Kernel k = new Kernel(3, 3, filtro);
ConvolveOp cop = new ConvolveOp(k);
```

Puesto que, como hemos indicado anteriormente, la convolución no permite que fuente y destino sean el mismo, la llamada al método `filter` se aconseja hacerla pasándole `null` como segundo parámetro y actualizar la imagen del lienzo con el resultado devuelto por el operador (véase código arriba).

[1] Para este caso se recomienda la primera opción: no solo mejora la eficiencia (por no crearse una nueva imagen cada vez que cambiemos el brillo), sino que además permitirá el posterior tratamiento por bandas.

Variación del brillo

En este apartado modificaremos el brillo de la imagen aplicando el operador *RescaleOp*, pero sin establecer un valor fijo (como en el apartado anterior), sino permitiendo que éste sea definido por el usuario mediante un deslizador[2].

En este caso hay 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: cada nuevo valor del deslizador[3], implicará calcular la imagen (temporal) resultado de aplicar ese valor de brillo sobre la imagen original (la imagen resultado se irá mostrando en la ventana mientras el usuario mueve el deslizador); esta consideración hay que tenerla en cuenta a la hora de usar los métodos set/get de la clase *Lienzo2D*. Para abordar este aspecto, una posible opción sería definir una variable de tipo *BufferedImage* en la ventana principal (donde se manejan los eventos asociados al deslizador) que represente la imagen fuente original sobre la que se aplicará la operación; a esta variable se le asignará valor cuando se inicie el cambio de brillo (por ejemplo, en el momento que el deslizador adquiere el foco[4], asignándole –¿una copia de?- la imagen del lienzo activo) y será la usada como imagen de entrada en la operación *RescaleOp* (la imagen resultado será mostrada en el correspondiente lienzo).

Hay diversas opciones a la hora de abordar la solución, pero hay que asegurarse que esté "coordinada" con la decisión tomada para en el uso del método *filter* (¿con o sin null?). Si optamos por la primera opción mostrada en el apartado anterior (esto es, pasar como imagen destino la imagen que está vinculada al lienzo), será necesario crear una copia de la imagen original[5]:

```
private void sliderBrilloFocusGained(java.awt.event.FocusEvent evt) {
  VentanaInterna vi = (VentanaInterna)(escritorio.getSelectedFrame());
  if(vi!=null){
    ColorModel cm = vi.getLienzo().getImage().getColorModel();
    WritableRaster raster = vi.getLienzo().getImage().copyData(null);
    boolean alfaPre = vi.getLienzo().getImage().isAlphaPremultiplied();
    imgFuente = new BufferedImage(cm,raster,alfaPre,null);
  }
}
```

donde *imgFuente* estará declarada en la ventana principal. Cuando se pierda el foco, se interpretará como que la operación ha acabado, por lo que pondremos a *null* la imagen fuente[6]:

```
private void sliderBrilloFocusLost(java.awt.event.FocusEvent evt) {
  imgFuente = null;
}
```

Además, se aconseja poner el valor del deslizador a 0 (método *setValue* de la clase *Slider*). Como se ha indicado, el código anterior deberá de estar coordinado con el uso de los parámetros en el método *filter*. En este caso, sería:

```
rop.filter(imgFuente, vi.getLienzo().getImage());
```

---

[2] Recordemos que con el método *getValue()* de la clase *JSlider* se puede acceder al valor que tenga un deslizador en un momento dado.

[3] Notificado mediante el evento *stateChange*

[4] En este caso, se aconseja asignarle valor *null* cuando pierda el foco

[5] Si se optase por el uso del *null* como imagen destino, no sería necesario hacer una copia: podríamos asignar directamente *imgSource=vi.getLienzo().getImage()* teniendo en cuenta que *filter* devolvería una nueva imagen que habría que asignar al lienzo con el *setImagen()*.

[6] Se aconseja poner el deslizador a valor cero.

Aplicación de filtros

En este apartado aplicaremos diferentes filtros basados en el operador de convolución. En primer lugar, probaremos máscaras definidas en el paquete `sm.image` (que se adjunta a esta práctica). A continuación, probaremos nuevas máscaras definidas por el estudiante.

1. En la parte inferior de la aplicación incluiremos una lista desplegable con el conjunto de filtros disponibles en nuestra aplicación, de forma que cuando el usuario seleccione uno de ellos, éste se aplicará automáticamente a la imagen seleccionada. Concretamente, se considerarán los siguientes filtros:

    - Emborronamiento media
    - Emborronamiento binomial
    - Enfoque
    - Relieve
    - Detector de fronteras laplaciano

    Cada uno de estos filtros está asociado a una máscara de convolución. En la clase `KernelProducer`, incluida en el paquete `sm.image` que se adjunta a esta práctica, se incluyen varios tipos de máscaras (definidas como variables estáticas), así como un método `createKernel` (también estático) que devuelve objetos `Kernel` correspondiente a máscaras predefinidas. Por ejemplo, el siguiente código crearía una máscara para el filtro media:

    ```
    Kernel k = KernelProducer.createKernel(KernelProducer.TYPE_MEDIA_3x3);
    ```

    Una vez creada la máscara, obtenemos el operador[7]:

    ```
    ConvolveOp cop = new ConvolveOp(k,ConvolveOp.EDGE_NO_OP,null);
    ```

2. Una vez probados los filtros anteriores, aplicar un emborronamiento media con máscaras de tamaño *5x5* y *7x7*. En este caso, las máscaras no están definidas en `sm.image`, por lo que hay que implementar el código para crear la máscara[8].

El reto final…

Por último, se proponen como reto implementar los siguientes filtros (incluir una entrada en la lista desplegable por cada uno de ellos) [9,10]:

- **Emborronamiento horizontal**. Este emborronamiento viene definido por la máscara [0.2, 0.2, 0.2, 0.2, 0.2], siendo su efecto un emborronamiento (tipo media) con una dirección predominante (en este caso, horizontal). ¿Sabrías justificar por qué se produce ese efecto?

  La máscara anterior tiene un tamaño de 5x1, ¿cuáles serán los valores si el tamaño fuese de 7x1? ¿y su fuese de 10x1? Cuanto mayor sea el tamaño, mayor será en emborronamiento, ¡haz la prueba!

- **Emborronamiento diagonal**. En este caso, se quiere un emborronamiento en la misma línea de ejercicio anterior, si bien la dirección ha de ser diagonal de 45°. ¿Cuál sería la máscara a utilizar? Piensa la respuesta y pruébala, ¡a ver si lo consigues!

---

[7] En el ejemplo activamos la opción `EDGE_NO_OP`, de forma que los pixeles del borde (en los que no se puede aplicar la convolución) se copiarán de la original. Si se optase por la opción `EDGE_ZERO_FILL`, el borde se pondría a cero (ésta es la opción por defecto).

[8] Véanse transparencias de teoría.

[9] No se incluyen en los vídeos. El objetivo es que se analice en qué grado se domina lo aprendido, ¡tú puedes!

[10] Estos filtros (máscaras) no se encuentran definidas en el paquete `sm.image` (deben implementarse).

Para trabajar en casa…

Una vez realizada la práctica, se proponen las siguientes mejoras:

- Cuando se cree una imagen nueva, crearla con canal alfa (es decir, de tipo `TYPE_INT_ARGB`). Este tipo de imágenes permitirá trabajar con transparencia.
- Cuando tenemos imágenes con canal alfa, ¿funciona el brillo? En caso negativo, ¿cuál es el motivo? ¿Y su solución?[11]
- Probar nuevas máscaras de convolución.
- Incluir la opción de aplicar las operaciones de imágenes a las formas pintadas. Para ello, añadir un casilla de activación en el menú de archivo y/o un botón de dos posiciones en la barra de herramientas; en caso de que esté activada, se deberán de volcar las figuras en la imagen[12] antes de aplicar las operaciones (el efecto será que los cambios de brillo y los filtros se verán también reflejados en las formas). Al volcar las figuras, éstas deberán de eliminarse del vector.

[11] Véase clase `RescaleOp` y considérese la posibilidad de usar un vector de *offsets* (y el correspondiente constructor) en lugar de un único valor.

[12] Recuérdese lo visto en teoría para dibujar en una imagen a través de su `Graphics2D` (p.e., en la práctica 8 se usó para volcar las formas antes de guardar la imágen).