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 anterior, introduciendo nuevas funcionalidades relativas al procesamiento imágenes. Concretamente, deberá incluir las siguientes funcionalidades:

- Posibilidad de variar el brillo y contraste 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 y contraste de la imagen activa, así como una lista desplegable con los distintos filtros que se puedan aplicar.

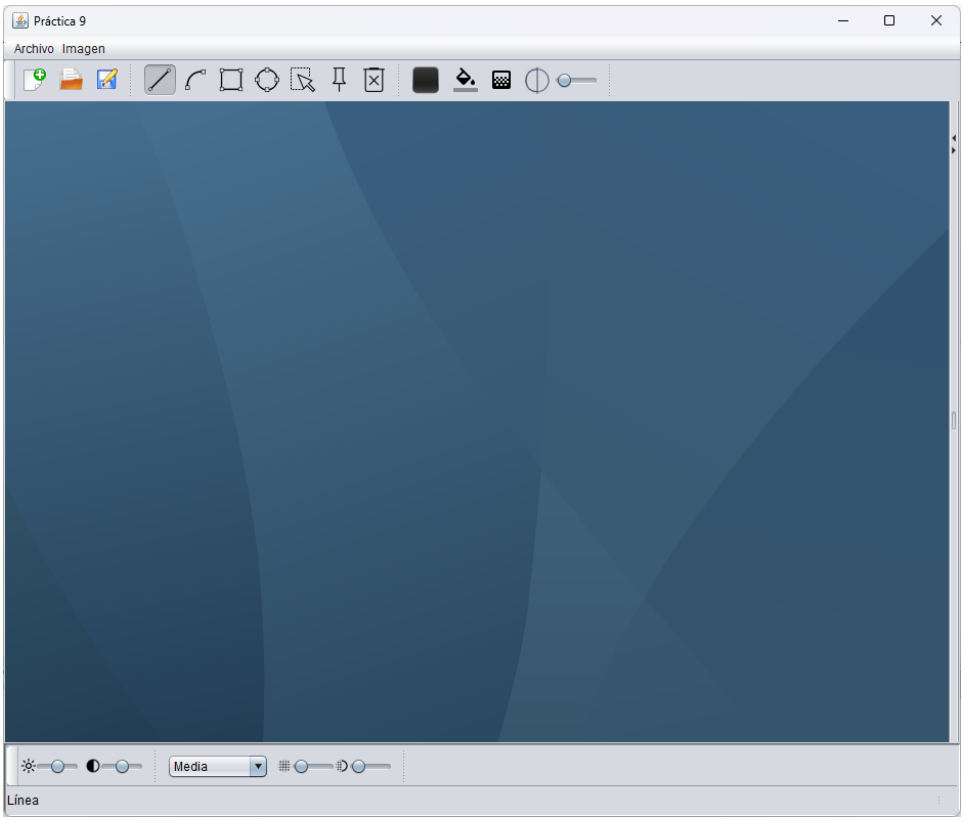

Figura 1: Aspecto de la aplicación

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.getLienzo2D().getImage();
    if(img!=null){
      try{
         RescaleOp rop = new RescaleOp(1.0F, 100.0F, null);
         rop.filter(img, img);
         vi.getLienzo2D().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.getLienzo2D().setImage(imgdest);
  vi.getLienzo2D().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 y contraste

En este apartado modificaremos el brillo y contraste 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/contraste 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.getLienzo2D().getImage().getColorModel();
    WritableRaster raster = vi.getLienzo2D().getImage().copyData(null);
    boolean alfaPre = vi.getLienzo2D().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:

```
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.getLienzo2D().getImage());
```

Por último, recordemos que tanto el brillo como el contraste se variarán mediante el operador `RescaleOp`. En el caso del brillo, se variará el factor suma, fijando a 1.0 el factor escala. En el caso del contraste, se variará el factor de escala[6,7], fijando a 0.0 el factor suma.

---

[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.getLienzo2D().getImage()` teniendo en cuenta que `filter` devolvería una nueva imagen que habría que asignar al lienzo con el `setImagen()`.

[6] Recordemos que, si el factor de escala está entre 0 y 1, reduce el contraste; si está por encima de 1, lo aumenta.

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 (realce o perfilado)
    - 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[8]:

    ```
    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[9].

El reto final…

Por último, se proponen los siguientes retos que te permitirán confirmar si has entendido bien la idea de filtro/convolución[10]:

1. **Emborronamiento mediante deslizador**. Incluir un deslizador que permita emborronar la imagen de forma interactiva, esto es, aplicará un emborronamiento que será mayor o menor en función del valor del deslizador. Para ello, se recomienda usar un filtro media cuya máscara tenga un tamaño igual al valor del deslizador (asumimos una máscara cuadrada)[11]. Nótese que será necesario calcular los valores de la máscara de convolución en función de su tamaño[12]. Al igual que con el brillo, la imagen sobre la que se aplica la operación ha de ser la original: cada nuevo valor del deslizador implicará ir aplicando el emborronamiento sobre la imagen original e ir mostrando el resultado en el lienzo[13].

---

[7] El deslizador solo admite valores enteros, por lo que se aconseja definir el valor mínimo en 1 y el máximo en 20 y, a la hora de usarlo, dividir su valor entre 10 (lo que implica factores de escala entre 0.1 y 2).

[8] 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).

[9] Véanse transparencias de teoría.

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

[11] Por ejemplo, el deslizador podría moverse entre 1 y 31 (lo que nos da máscaras desde 1x1 hasta 31x31).

[12] No corresponde a una máscara fija del paquete `sm.image` (deben implementarse la generación de la máscara en función del tamaño).

[13] Seguir, por tanto, el mismo esquema que con el brillo usando el `FocusGained` y el `FocusLost`.

2. **Perfilado mediante deslizador.** En el apartado anterior se incluyó un filtro que permite hacer un realce o perfilado estándar en una imagen. La máscara 3x3 que permite aplicar este realce de forma parametrizada viene dada por:

0	-**a**	0
-**a**	4**a**+1	-**a**
0	-**a**	0

siendo *a≥1* el parámetro (cuanto mayor sea *a*, mayor será el efecto). Nótese que el caso a=1 es la máscara implementada en los filtros por defecto.

Incluir esta nueva funcionalidad en la aplicación, de forma que el usuario defina el valor del parámetro "a" de forma interactiva. Concretamente, este parámetro se podrá modificar mediante un deslizador[14] que incluiremos en nuestra aplicación (véase Fig. 1). Nótese que será necesario calcular los valores de la máscara de convolución en función del valor del parámetro dado por el deslizador[15]. Recordar que la imagen sobre la que se aplica la operación ha de ser la original: cada nuevo valor del deslizador implicará ir aplicando el perfilado sobre la imagen original e ir mostrando el resultado en el lienzo.

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, ¿se guarda correctamente? Si no es así, posiblemente sea porque se esté almacenando en un formato inadecuado (como JPG): en estos casos hay que usar un formato que permita canal alfa (por ejemplo, PNG). Si se implementó la mejora propuesta en la práctica 8 (usar filtros en el diálogo guardar y obtener el formato a partir de la extensión del fichero), bastará elegir el formato adecuado; si no se hizo, ahora es un buen momento para hacerlo…
- Probar nuevas máscaras de convolución.
- Cuando se seleccione una figura en el lienzo, hacer que los atributos de la ventana principal se activen conforme a los atributos de la figura seleccionada[16].

[14] Por ejemplo, el deslizador podría moverse entre 1 y 15.

[15] No corresponde a una máscara fija del paquete `sm.image` (deben implementarse la generación de la máscara en función del parámetro).

[16] Para abordar esta funcionalidad habrá que gestionar eventos, pero, en este caso, requiere definir eventos propios (que se generen desde el lienzo). El apéndice de este guion muestra el proceso completo para definir eventos propios en el lienzo y gestionarlos desde la ventana principal. No es algo trivial, pero ¡tú puedes!

Apéndice: Definir eventos propios para el lienzo

En ocasiones es necesario que objetos que son instancias de clases propias (definidas por nosotros) notifiquen sucesos específicos de su clase (por ejemplo, en el caso del lienzo, notificar que se ha añadido una nueva figura o que se ha cambiado alguna propiedad). En estos casos, los eventos estándar de Java no nos sirven, siendo necesario definir (1) clases propias de eventos, así como (2) los interfaces asociados que deberán de implementar los manejadores del evento; además, (3) habrá que incorporar la capacidad de generación de eventos en nuestra clase.

A continuación se muestra un ejemplo en el que se define una clase evento propia para la clase `Lienzo2D`, así como el interface del manejador (se aconseja definirlas en nuestra biblioteca dentro de un paquete específico *sm.xxx.eventos*).

• Definición de la clase *LienzoEvent*

En primer lugar, definimos la clase `LienzoEvent` que representará un evento lanzado por un objeto `Lienzo2D`. Dicha clase deberá de heredar de `EventObject` y tendrá definidas variables miembro para guardar la información asociada a dicho evento. Por ejemplo:

```
public class LienzoEvent extends EventObject{

    private Shape forma;

    public LienzoEvent(Object source, Shape forma) {
        super(source);
        this.forma = forma;
    }
    public Shape getForma() {
        return forma;
    }
}
```

representa un evento que llevará asociada como información una figura.

• Definición de la interfaz *LienzoListener*

A continuación, definimos la interfaz `LienzoListener` que deberán cumplir los manejadores de eventos. Dicha interfaz deberá de heredar de `EventListener` y tendrá definidos tantos métodos como "motivos" puedan generar ese evento. Por ejemplo:

```
public interface LienzoListener extends EventListener{
    public void shapeAdded(LienzoEvent evt);
    public void shapeSelected(LienzoEvent evt);
}
```

que indica que todo manejador tipo `LienzoListener` deberá de implementar dos métodos, uno asociado a añadir una nueva figura al lienzo, y otro a seleccionar una existente. Siguiendo la costumbre de Java, podemos definir el "adapter" asociado[17]:

```
public class LienzoAdapter implements LienzoListener{
    public void shapeAdded(LienzoEvent evt){}
    public void shapeSelected (LienzoEvent evt){}
}
```

[17] Un manejador de eventos lienzo podrá definirse (1) implementando el interface `LienzoListener` o (2) heredando de `LienzoAdapter` y sobrecargando el método que nos interese.

• Incorporando en *Lienzo* la capacidad de lanzar eventos

Una vez creadas las clases anteriores, surge la siguiente pregunta: ¿cómo hacemos que el lienzo lance los eventos? Recordemos que, en este caso, el lienzo hace las veces de generador. Además, ¿cómo le asociamos manejadores al lienzo? Para abordar los puntos anteriores habrá que añadir las siguientes funcionalidades a la clase `Lienzo2D`:

1. Declarar como variable miembro de la clase `Lienzo` una lista de elementos `LienzoListener`. Dicha lista tendrá almacenados los manejadores asociados al lienzo:

```
ArrayList<LienzoListener> lienzoEventListeners = new ArrayList();
```

2. Añadir a la clase `Lienzo2D` un método `addLienzoListener` que permita añadir manejadores a la lista

```
public void addLienzoListener(LienzoListener listener){
  if (listener != null) {
     lienzoEventListeners.add(listener);
  }
}
```

3. Desde el lienzo habrá que notificar a los manejadores asociados el lanzamiento de un determinado evento; esto implicará mandar el correspondiente mensaje (según motivo que lo haya ocasionado) a cada uno de los manejadores de la lista anterior (por ejemplo, mandarle el mensaje `shapeAdded` cuando se añada un elemento nuevo en la lista de figuras).

   Para simplificar esta tarea, se aconseja añadir a la clase `Lienz2D` métodos para notificar a los manejadores asociados el lanzamiento de un determinado evento; concretamente, se aconseja definir uno por cada "motivo" que puede ocasionar el evento:

```
private void notifyShapeAddedEvent(LienzoEvent evt) {
  if (!lienzoEventListeners.isEmpty()) {
    for (LienzoListener listener : lienzoEventListeners) {
      listener.shapeAdded(evt);
    }
  }
}

private void notifyShapeSelectedEvent(LienzoEvent evt) {
  if (!lienzoEventListeners.isEmpty()) {
    for (LienzoListener listener : lienzoEventListeners) {
       listener.shapeSelected(evt);
    }
  }
}
```

4. A los métodos definidos en el punto anterior los llamaremos cuando queramos que se lance el evento. Esto se hará desde otros métodos de la clase, según se cumpla o no el motivo que genera el evento. Por ejemplo, el evento que notifique que se ha añadido una nueva figura debería llamarse después de añadir el correspondiente elemento a la lista de figuras:

```
vShape.add(...);
notifyShapeAddedEvent( new LienzoEvent(this,s) );
```

   donde `s` sería la figura (objeto `Shape`) añadido.

- **Y no olvidar…**

Recordar que, como vimos en la práctica 2, para manejar eventos es necesario (1) definir la clase manejadora, en este caso implementando el interface *LienzoListener*, (2) crear el objeto manejador y (3) enlazar el manejador con el generador, en este caso con el lienzo mediante la llamada a *addLienzoListener*.

Para los eventos estándar (*MouseEvent*, *KeyEvent*, etc.), NetBeans realiza las tres tareas anteriores de una forma sencilla; sin embargo, en el caso de eventos propios, no contamos con NetBeans para "hacerlo por nosotros", así que habrá que incluir el código por nuestra cuenta (que, recordemos, es lo habitual en programación avanzada). Por ejemplo, si queremos en nuestra *VentanaPrincipal* muestre un mensaje (en consola o en la barra de estado) cada vez que se añada una figura en algún lienzo, en primer lugar tendremos que definir la clase manejadora[18]:

```
public class MiManejadorLienzo extends LienzoAdapter{
    public void shapeAdded(LienzoEvent evt){
       System.out.println("Figura "+evt.getForma()+" añadida");
    }
}
```

y posteriormente crear un objeto de la clase anterior y asociarlo con el correspondiente manejador:

```
MiManejadorLienzo manejador = new MiManejadorLienzo();
lienzo.addLienzoListener(manejador);
```

donde *lienzo* es una referencia al lienzo que queremos manejar (por ejemplo, el de una ventana interna al que accedemos mediante *vi.getLienzo2D()*)

[18] En este ejemplo, se podría declarar como una clase interna de *VentanaPrincipal*.