PRÁCTICA 10

Procesamiento de imágenes

Parte 2

Descripción de la práctica

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

- Operadores de transformación (variaciones de brillo y contraste)
- Rotación y escalado de imágenes

El aspecto visual de la aplicación será el mostrado en la Figura 1. El menú incorporará en su opción “Imagen”, además de lo incluido en la practica 9, los ítems “AffineTransformOp”, “LookupOp”. En la parte inferior, además de los ya incluido en la práctica 9, se incorporarán tres botones asociados a tres tipos de contraste (normal, iluminado y oscurecido), un botón para el efecto “oscurecer zonas claras”, un deslizador para un nuevo operador, un botón para rotación de 180° y dos botones para el escalado (incrementar y reducir).

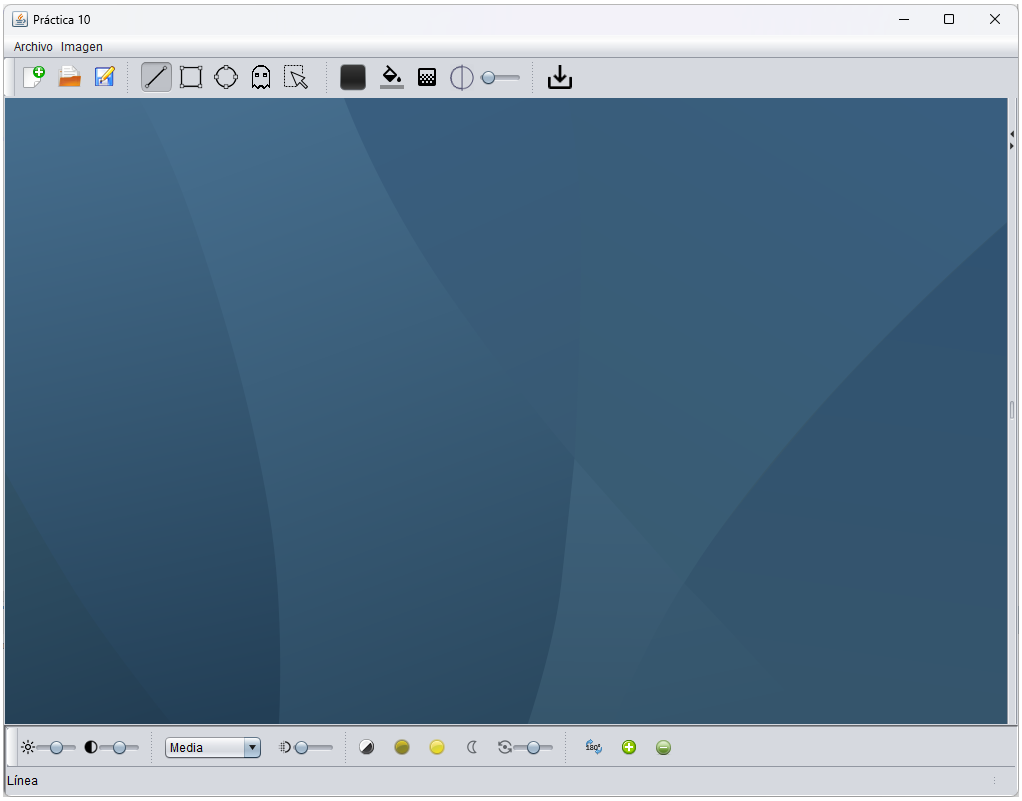

Figura 1: Aspecto de la aplicación

Pruebas iniciales

En una primera parte, probaremos los operadores “`AffineTransformOp`” y “`LookupOp`” usando parámetros fijos (i.e., sin interacción del usuario para definir sus valores). Para ello, incluiremos las correspondientes 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. Al igual que vimos en la práctica 9, 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 "AffineTransformOp" y el escalado:

```
private void menuAffineTransformOpActionPerformed(ActionEvent evt) {
  VentanaInterna vi = (VentanaInterna) (escritorio.getSelectedFrame());
  if (vi != null) {
    BufferedImage img = vi.getLienzo2D().getImage();
    if(img!=null){
      try{
        AffineTransform at = AffineTransform.getScaleInstance(1.5,1.5);
        AffineTransformOp atop = new AffineTransformOp(at,null);
        BufferedImage imgdest = atop.filter(img, null);
        vi.getLienzo2D().setImage(imgdest);
        vi.getLienzo2D().repaint();
      } catch(IllegalArgumentException e){
         System.err.println(e.getLocalizedMessage());
      }
    }
  }
}
```

Obsérvese que en el código anterior se ha optado por pasarle `null` como segundo parámetro al método `filter`, con lo cual se crea una nueva imagen salida; esto implica que sea necesario actualizar la imagen del lienzo de forma explícita. Por otro lado, en la creación del operador no lo indicamos ningún método de interpolación específico (al usar `null`, emplea el método de "vecino más cercano" por defecto); si queremos mejorar el resultado usando otra interpolación, podría indicarse mediante las variables estáticas que ofrece la clase `AffineTransformOp` (por ejemplo, `AffineTransformOp .TYPE_BILINEAR`. para interpolación bilineal[1]).

Para el caso de la operación "LookupOp", el código será similar al anterior, pero cambiando el operador; por ejemplo, para aplicar una operación de negativo (véanse transparencias) el código sería:

```
byte funcionT[] = new byte[256];
for (int x=0; x<256; x++)
  funcionT[x] = (byte)(255-x);  // Negativo
LookupTable tabla = new ByteLookupTable(0, funcionT);
LookupOp lop = new LookupOp(tabla, null);
```

En relación al operador `LookupOp`, hay que tener en cuenta que para imágenes tipo "BGR" (p.e., `TYPE_INT_BGR`), si no indicamos imagen de salida en la llamada a `filter` (i.e, dejamos a `null` el segundo parámetro), el operador genera una imagen de salida que intercambia[2] los canales B y R. Para abordar este problema, podríamos optar por convertir la imagen fuente a tipo `TYPE_INT_ARGB` para asegurar compatibilidad[3] o pasar en la llamada a `filter` una imagen destino compatible con la fuente (como caso particular, se podría pasar la misma imagen fuente, que en ese caso sería modificada).

---

[1] En este caso, además, emplearía transparencia en la imagen resultado (esto es relevante, por ejemplo, en el caso de rotaciones)

[2] *Bug* no documentado

[3] Por ejemplo, el siguiente código haría la conversión usando un método de la clase `sm.image.ImageTool`:
`img = ImageTools.convertImageType(img, BufferedImage.TYPE_INT_ARGB);`

Variación del brillo y contraste

En este apartado modificaremos el contraste de la imagen aplicando el operador `LookupOp`. Para ello, incluiremos tres botones correspondientes a tres posibles situaciones:

- Contraste "normal", para imágenes en las que la luminosidad esté equilibrada. En este caso, se usan funciones tipo S
- Contraste con iluminación, para imágenes oscuras. En este caso, se usan funciones tipo logaritmo (si es muy oscura), funciones raíz (con cuyo parámetro podemos determinar el grado de iluminación) o correcciones gamma (con gamma entre 0 y 1)
- Contraste con oscurecimiento, para imágenes sobre-iluminadas. En este caso, se usan funciones potencia (con cuyo parámetro podemos determinar el grado de oscurecimiento) o correcciones gamma (con gamma mayor que 1).

Cada una de las operaciones anteriores está asociada a una determinada función (véanse transparencias de teoría). En la clase `LookupTableProducer` del paquete `sm.image` se definen métodos estáticos para crear objetos `LookupTable` correspondientes a funciones estándar (negativo, función-S, potencia, raíz, corrección gamma, etc.); por ejemplo, para crear la función S pasándole los correspondientes parámetros usaríamos el siguiente código[4]:

```
LookupTable lt = LookupTableProducer.sFuction(128.0,3.0);
```

Por otro lado, esta clase define un método `createLookupTable` que devuelve objetos `LookupTable` correspondientes a las funciones clásicas anteriores pero usando parámetros predefinidos. Por ejemplo, el siguiente código crearía un objeto `LookupTable` por defecto asociado a la función-S:

```
LookupTable lt;
lt=LookupTableProducer.createLookupTable(LookupTableProducer.TYPE_SFUNCION);
```

Para los ejemplos que se proponen en este ejercicio, bastaría usar los parámetros por defecto que ofrece el método `createLookupTable`. Por ejemplo, para el caso del contraste normal[5]:

```
private void bContrasteActionPerformed(ActionEvent evt) {
  VentanaInterna vi = (VentanaInterna) (escritorio.getSelectedFrame());
  if (vi != null) {
    BufferedImage img = vi.getLienzo2D().getImage();
    if(img!=null){
      try{
        int type = LookupTableProducer.TYPE_SFUNCION;
        LookupTable lt = LookupTableProducer.createLookupTable(type);
        LookupOp lop = new LookupOp(lt, null);
        lop.filter( img , img); // Imagen origen y destino iguales
        vi.getLienzo2D().repaint();
      } catch(Exception e){
          System.err.println(e.getLocalizedMessage());
      }
    }
  }
}
```

Nótese que en ejemplo anterior se usa como imagen destino la propia imagen fuente.

---

[4] Por si resulta útil de cara a implementar otras funciones propias, el código en el paquete `sm.image` correspondiente a la función S es el siguiente:

```
public static LookupTable sFuction(double m, double e){
  double Max = (1.0/(1.0+Math.pow(m/255.0,e)));
  double K = 255.0/Max;
  byte lt[] = new byte[256];
  lt[0]=0;
  for (int l=1; l<256; l++){
    lt[l] = (byte)(K*(1.0/(1.0+Math.pow(m/(float)l,e))));
  }
  ByteLookupTable slt = new ByteLookupTable(0,lt);
  return slt;
}
```

[5] Si se quisiera modificar los parámetros del contraste, habría que llamar directamente a la función S (sin llamar al método `createLookupTable`) pasándole los parámetros correspondientes.

Rotación y escalado

En este apartado incluiremos la posibilidad de que el usuario pueda rotar y/o escalar imágenes usando valores fijos predeterminados. En primer lugar, incluiremos un botón para poder rotar la imagen 180°; para ello haremos uso del operador "`AffineTransformOp`", teniendo en cuenta que la rotación tendrá que hacerse poniendo como eje de rotación el centro de la imagen[6]:

```
double r = Math.toRadians(180);
Point c = new Point(imgSource.getWidth()/2, imgSource.getHeight()/2);
AffineTransform at = AffineTransform.getRotateInstance(r,p.x,p.y);
AffineTransformOp atop;
atop = new AffineTransformOp(at,AffineTransformOp.TYPE_BILINEAR);
BufferedImage imgdest = atop.filter(imgSource, null);
```

Además, permitiremos escalar la imagen mediante dos botones: uno para aumentar el tamaño de la imagen y otro para reducirlo. De nuevo, usaremos un operador "`AffineTransformOp`", en este caso creando un escalado donde fijaremos el factor de escala (por ejemplo, 1.25 para aumentar y 0.75 para reducir).

El reto final…

Por último, se proponen los siguientes retos que te permitirán confirmar si has entendido bien cómo se crea un operador basado en función:

1. **Transformación lineal con un punto de control**. Implementar un operador "LookupOp" que aplique la siguiente transformación T: $[0,255] \rightarrow [0,255]$, con parámetro $a \in [0,255]$:

$$T(x;a) = \begin{cases} \dfrac{a \cdot x}{128} & x < 128 \\ \dfrac{(255-a)(x-128)}{127} + a & x \geq 128 \end{cases}$$

   ¿Sabrías explicar qué efecto tiene sobre la imagen? ¿cómo afecta el parámetro "a"?[7] Incluye la respuesta en la documentación (javadoc) del método que genere esta transformación. En la aplicación incluir un deslizador con el que el usuario podrá ir variando el valor del parámetro "a" e ir viendo el efecto sobre la imagen.

2. **Oscurecer zonas claras**. Implementar un operador "LookupOp" que permita el siguiente efecto sobre una imagen: "las zonas oscuras se mantendrán iguales" mientras que "las zonas claras se oscurecerán". Para delimitar las zonas claras de las oscuras, usar un umbral T (por ejemplo, T=128) de forma que aquellos valores inferiores a T se considerarán oscuros (y el resto, claros).

   Para solucionar este reto has de pensar en qué función habría que aplicar: hay varias opciones, así que intenta primero "esbozar" la gráfica sobre papel (si usas alguna herramienta de representación gráfica de funciones, mucho mejor) y luego trata de crear el LookupTable asociado. Muestra en consola (output) los valores de la tabla (transformación) que se está aplicando.

Para trabajar en casa…

Una vez realizada la práctica, algunas mejoras a considerar podrían ser las siguientes:

- Incluir la operación "negativo".
- Incluir una opción "duplicar" que cree una nueva ventana interna con una copia de la imagen que había en la ventana activa.
- Definir nuevas operaciones "lookup" propias.

---

[6] Tras aplicar el `AffineTransformOp`, la imagen resultado tendrá activo el canal alfa.

[7] Para dar respuesta a estas preguntas, se aconseja dibujar la gráfica para diferentes valores de "a".