PRÁCTICA 11.b

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.a incluyendo las siguientes nuevas funcionalidades:

- Umbralización
- Operador gradiente “Sobel”

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.a, se incorporará un deslizador para umbralizar la imagen y un botón asociado al operador gradiente “Sobel”.

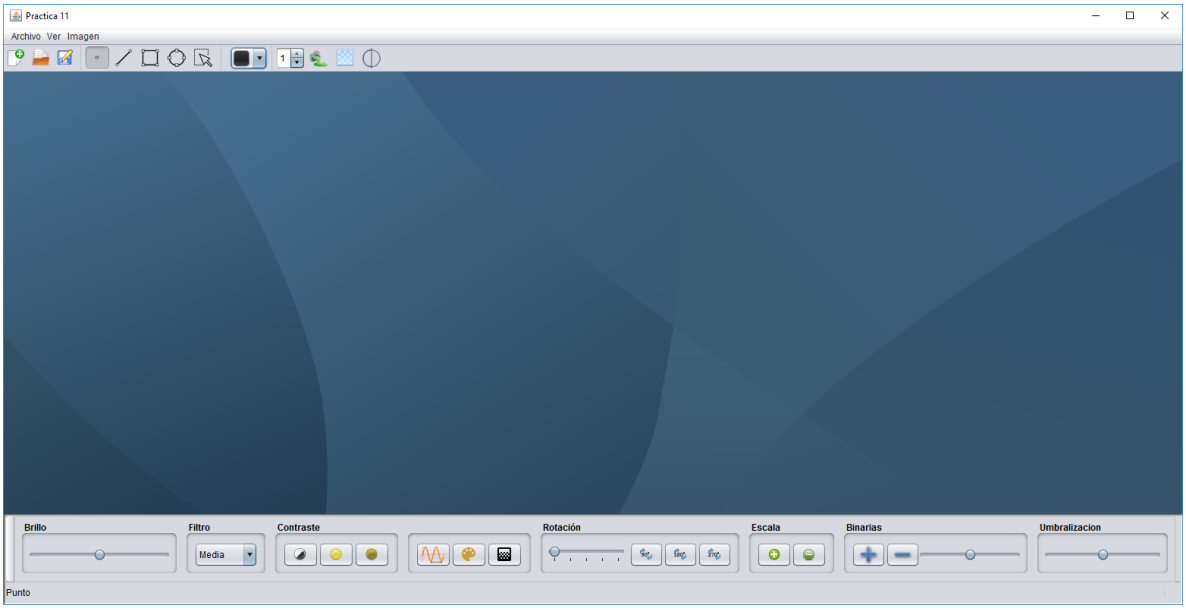

Figura 1: Aspecto de la aplicación

Umbralización

En este primer bloque, incorporaremos un operador de umbralización (véanse transparencias de teoría). Para ello definiremos nuestra propia clase[1] `UmbralizacionOp` de tipo `BufferedImageOp` en el paquete `sm.xxx.imagen` de nuestra librería (con `xxx` las iniciales del estudiante). Esta clase tendrá como propiedad principal el valor umbral (que definiremos como una variable) y, siguiendo el esquema visto en teoría, haremos que herede de `sm.image.BufferedImageOpAdapter` y sobrecargue el método `filter`:

[1] El paquete `sm.image` contiene la clase `ThresholdOp` que permite realizar umbralizaciones basadas en intensidad (por defecto), en color (si se le pasa un color en el constructor) o por bandas. El objetivo de esta práctica no es usar dicha clase, sino crear una propia e implementar la operación.

```
public class UmbralizacionOp extends BufferedImageOpAdapter{
  private int umbral;

  public UmbralizacionOp(int umbral) {
    this.umbral = umbral;
  }

  public BufferedImage filter(BufferedImage src, BufferedImage dest){

    //Código de umbralización

  }
}
```

Para esta práctica, implementaremos la umbralización basada en intensidad (véanse transparencias de teoría). Como sabemos, este operador genera como resultado una imagen binaria donde los píxeles que superan un umbral (medido en términos de intensidad) se le asigna un valor (p.e, 255) [2] y al resto 0:

$$g(x,y) = \begin{cases} 255 & si\ I(x,y) \geq T \\ 0 & si\ I(x,y) < T \end{cases}$$

con $I(x,y)$ la intensidad del pixel calculada como la media de sus componentes $I(x,y) = (r(x,y) + g(x,y) + b(x,y))/3$

Por tanto, en el método `filter` recorreremos la imagen y, para cada pixel, aplicaremos la operación anterior y asignaremos 0 o 255 en la imagen destino en función del resultado. Para recorrer la imagen pixel a pixel usaremos el iterador `BufferedImagePixelIterator` (véase plantilla explicada en teoría)[3].

Al margen de lo anterior, 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`)[4]. En cualquiera de los casos, recordar que el método `filter` ha de devolver la imagen resultado.

Cálculo de bordes: operador Sobel

En segundo lugar, incluiremos el operador Sobel para la detección de contornos[5]. Para ello, definiremos la clase "`SobelOp`" e implementaremos el operador según lo visto en clase de teoría. Para ello, tendremos en cuenta las siguientes consideraciones:

---

[2] Otra opción sería asignar, en lugar de 255, el color original del pixel (en este caso no obtendríamos una imagen binaria).

[3] 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).

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

[5] Para comprobar si el resultado es correcto, se puede comparar con el dado por `sm.image.SobelOp`.

- El método `filter` recorrerá la imagen y calculará el gradiente Sobel según la fórmula (véase trasparencias de teoría):

$$\nabla f = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right] = [\nabla_x, \nabla_y] \quad \text{con} \quad \nabla_x = \frac{1}{4}\begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} \quad \text{y} \quad \nabla_y = \frac{1}{4}\begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix}$$

  Para una imagen en color, el vector gradiente en un pixel se calculará sumando los vectores gradiente de cada banda. Una vez obtenido el gradiente, la magnitud y la orientación vendrán dados por:

$$|\nabla| = \sqrt{\nabla_x^2 + \nabla_y^2} \quad , \quad \theta = \tan^{-1}\left(\frac{\nabla_y}{\nabla_x}\right)$$

- Como imagen salida, devolveremos la magnitud $|\nabla|$ del gradiente (recordemos que el operador Sobel calcula el gradiente y, por tanto, asociado a un pixel tendremos un vector).

  Recordemos que dicho valor ha de estar entre 0 y 255. Para ello, lo más correcto sería normalizar la imagen en su conjunto una vez calculada la magnitud (multiplicando por 255/MAX, con MAX el valor máximo de magnitud obtenido). No obstante, y para simplificar, podemos optar por "truncar" la magnitud (si supera el valor 255, se trunca a 255); en este último caso, se puede usar la función `sm.image.ImageTools.clampRange(magnitud,0,255)`.

- El cálculo anterior requiere aplicar dos convoluciones para el cálculo de los gradientes en *x* e *y*. En principio, parece lógico pensar en el uso de `ConvolveOp` para llevar a cabo dicha operación, pero nos vamos a encontrar con un problema: el operador trunca los valores negativos dejándolos a cero. Esto implica, por tanto, que el uso de `ConvolveOp` sólo contabilizará los "saltos" positivos, por lo que sería necesario implementar una nueva convolución que permitiera operar con valores negativos. Para simplificar el ejercicio, usaremos el operador `ConvolveOp`, si bien el resultado que obtendremos no será realmente el correspondiente al gradiente Sobel.

- En este caso recorreremos la imagen pixel a pixel usando el iterador `sm.image.BufferedImagePixelIterator` (véase plantilla explicada en teoría).

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:

- Incorporar la umbralización basada en color. En este caso, además del umbral, el usuario deberá elegir el color central de la umbralización (véanse transparencias de teoría)[6].
- Incorporar la operación de ecualización[7].
- Crear un diálogo que muestre el histograma[8].

[6] Implementada en `sm.image.ThresholdOp` (no es necesario crear clase propia)
[7] Implementada en `sm.image.EqualizationOp` (ecualización por bandas).
[8] 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).