# PRÁCTICA 12.b

## Procesamiento de imágenes

**Parte 5**

## Descripción de la práctica

El objetivo de esta práctica es realizar una aplicación que amplíe la realizada en la práctica 12 incluyendo las siguientes nuevas funcionalidades:

- Operadores binarios
- 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 12, se incorporará un botón asociado al operador gradiente “Sobel”, así como un área para operaciones binarias que incluya dos botones correspondientes a la suma y a la resta.

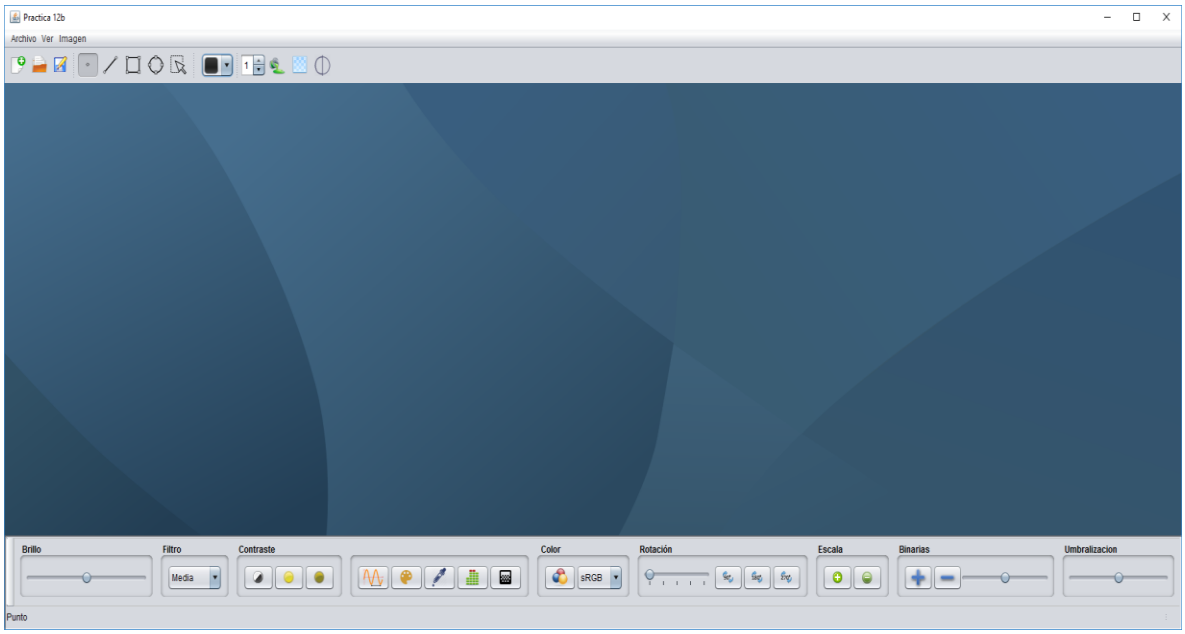


*Figura 1: Aspecto de la aplicación*

## 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)[1].

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`):

[1] 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`”.

```
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());
        }
      }
    }
  }
}
```

## Cálculo de bordes: operador Sobel

En segundo lugar, incluiremos el operador Sobel para la detección de contornos[2]. 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:

- 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,

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

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.

## 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 mezcla de dos imágenes (*blending*) de forma interactiva aplicando la suma ponderada. Para ello se usará de nuevo la clase `BlendOp`, si bien en este caso variaremos el valor alfa de la mezcla[1] (véanse transparencias de teoría). Para ello, incluiremos un deslizador en el panel de imágenes que permita seleccionar el valor de alfa[3,4].
- 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).

Para mejorar la interfaz:

- Al igual que se propuso en la práctica 7 en lo relativo a la gestión de las herramientas de dibujo, también en este caso se propone mejorar la barra de herramienta vinculada a operaciones sobre imágenes. Con carácter general, se propone simplificarla dándole un aspecto "lineal" sencillo. Por ejemplo, un posible diseño sería algo en la siguiente línea[5]:


En el ejemplo anterior los deslizadores están asociados a un botón de dos posiciones de diseño propio[6]:


si bien podría optarse por otro tipo de solución basada en los componentes estándar ya existentes:


[3] 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`)

[4] 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). Una alternativa a declarar variables miembro en la ventana principal para las imágenes fuentes (como hicimos en otros casos), sería crearse una ventana interna específica para la operación mezcla (que tendría como variables miembro las imágenes a mezclar).

[5] Los iconos de este ejemplo están descargados del repositorio de google https://material.io/icons/.

[6] Implicaría crear una clase propia que herede de `JToggleButton` que incorpore como funcionalidad el lanzamiento, al activarse, de una ventana (`JWindow`) con un deslizador. En este caso, la incorporación de este tipo de componente no podría hacerse usando las herramientas visuales del NetBeans.