PRÁCTICA 11

Procesamiento de imágenes

Parte 3

Descripción de la práctica

El objetivo de esta práctica es realizar una aplicación que amplíe la realizada en las prácticas 8, 9 y 10 introduciendo nuevas operaciones definidas por el estudiante. Concretamente, deberá incluir las siguientes nuevas funcionalidades:

- Umbralización
- Cálculo de bordes
- Operadores binarios

El aspecto visual de la aplicación será el mismo que el de la práctica 10 (véase Figura 1 del correspondiente guion), incorporando al menú "Imagen", además de lo incluido en las practica anteriores, los ítems correspondientes a las nuevas operaciones definidas en esta práctica

Umbralización

En primer lugar, incorporaremos un operador de umbralización[1] (véanse transparencias de teoría); para ello, definiremos una nueva clase "*UmbralizacionOp*" que herede de *sm.image.BufferedImageOpAdapter*, sobrecargue el método *filter* y tenga como propiedad el valor umbral:

```
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, simplificaremos el operador asumiendo que se aplica la misma umbralización en cada banda[2]. Por tanto, recorreremos la imagen (véase plantilla explicada en teoría) y para cada componente aplicaremos la operación:

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

Para recorrer la imagen componente a componente, usaremos el iterador *sm.image.BufferedImageSampleIterator* (véase plantilla explicada en teoría).

---

[1] Para comprobar si el resultado es correcto, se puede comparar con el dado por *sm.image.ThresholdOp*. En este caso, tras crear el objeto con el umbral deseado, hay que indicarle que la umbralización la haga por banda mediante *setType(ThresholdOp.TYPE_EACH_BAND)*. También puede aplicarse a la intensidad o en el dominio del color)

[2] Esto implicará que, para una imagen en color, no obtendremos una imagen binaria

Cálculo de bordes: operador Sobel

En segundo lugar, incluiremos el operador Sobel para la detección de contornos[3]. Para ello, definiremos la clase "`SobelOp`" e implementaremos el operador según lo visto en clase de teoría.

- Como imagen salida, devolveremos la magnitud 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 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:

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

- 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).

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

Operadores binarios: resta y multiplicación

En tercer lugar, incorporaremos dos operadores aritméticos: la resta y la multiplicación[4]. Para ello, definiremos dos nuevas clases "*RestaOp*" y "*MultiplicaciónOp*", una por operador, que heredarán de *sm.image.BinaryOp* (véanse trasparencias de teoría) y sobrecargarán el método *binaryOp*.

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 siguiente a la seleccionada:

```
VentanaInterna vi = (VentanaInterna)escritorio.getSelectedFrame();
if (vi != null) {
  BufferedImage imgRight = vi.getLienzo().getImage();
  VentanaInterna viNext = (VentanaInterna)escritorio.selectFrame(true);
  If(viNext != null){
    BufferedImage imgLeft = viNext.getLienzo().getImage();
    RestaOp op = new RestaOp(imgLeft);
    imgdest = op.filter(imgRight, null);
  }
}
```

[4] Para comprobar si el resultado es correcto, se puede comparar con el dado por *sm.image. SubtractionOp* y *sm.image.MultiplicationOp*.