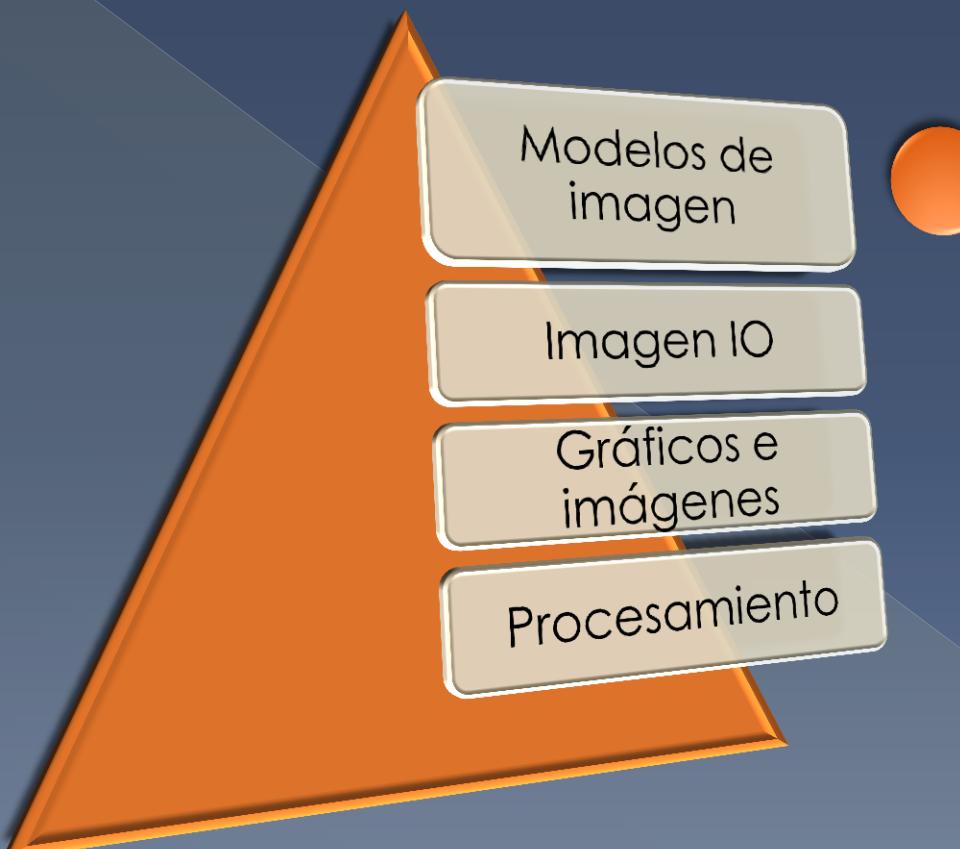


# JAVA 2D Imaging

# Índice



Modelos de  
imagen

Imagen IO

Gráficos e  
imágenes

Procesamiento

# Modelos de imagen en Java

- Productor/consumidor
  - > Proviene de la primera versión de Java
  - > Principales clases/interfaces:
    - Image, ImageProducer, ImageConsumer, ImageObserver
- Modo inmediato
  - > Introducido en el Java 2D™
  - > Principales clases/interfaces:
    - BufferedImage, BufferedImageOp, ImageIO
- Modelo pipeline
  - > Introducido en la JAI
  - > Principales clases/interfaces:
    - TitleImage, PlanarImage...

# Modelos de imagen en Java

Modelo productor/consumidor



Lectura de Image

```
Toolkit tk = Toolkit.getDefaultToolkit();
Image img = tk.getImage("...");
```

Visualización de Image

```
public void paint(Graphics g){
    super.paint(g);
    g.drawImage(img,x,y,this);
}
```



# BufferedImage

Raster

SampleModel

DataBuffer

ColorModel

ColorSpace

Representa la matriz rectangular de pixeles

Define cómo y en qué espacio se codifica el color

## Raster

SampleModel

DataBuffer

Permite acceder a los pixeles de una imagen, ofreciendo diversas alternativas a la hora de organizar la forma en la que se almacenan los componentes de cada pixel

Almacena los valores de los componentes de cada pixel



# BufferedImage

Raster

SampleModel

DataBuffer

ColorModel

ColorSpace

# DataBuffer

## DataBuffer

DataBufferByte

DataBufferInt

DataBufferShort

DataBufferUShort

DataBufferFloat

DataBufferDouble

# SampleModel

## SampleModel

ComponentSampleModel

SinglePixel  
Packed  
SampleModel

MultiplePixel  
Packed  
SampleModel

Banded  
SampleModel

PixelInterleaved  
SampleModel

# BufferedImage

Raster

SampleModel

DataBuffer

ColorModel

ColorSpace

# ColorModel

ColorModel

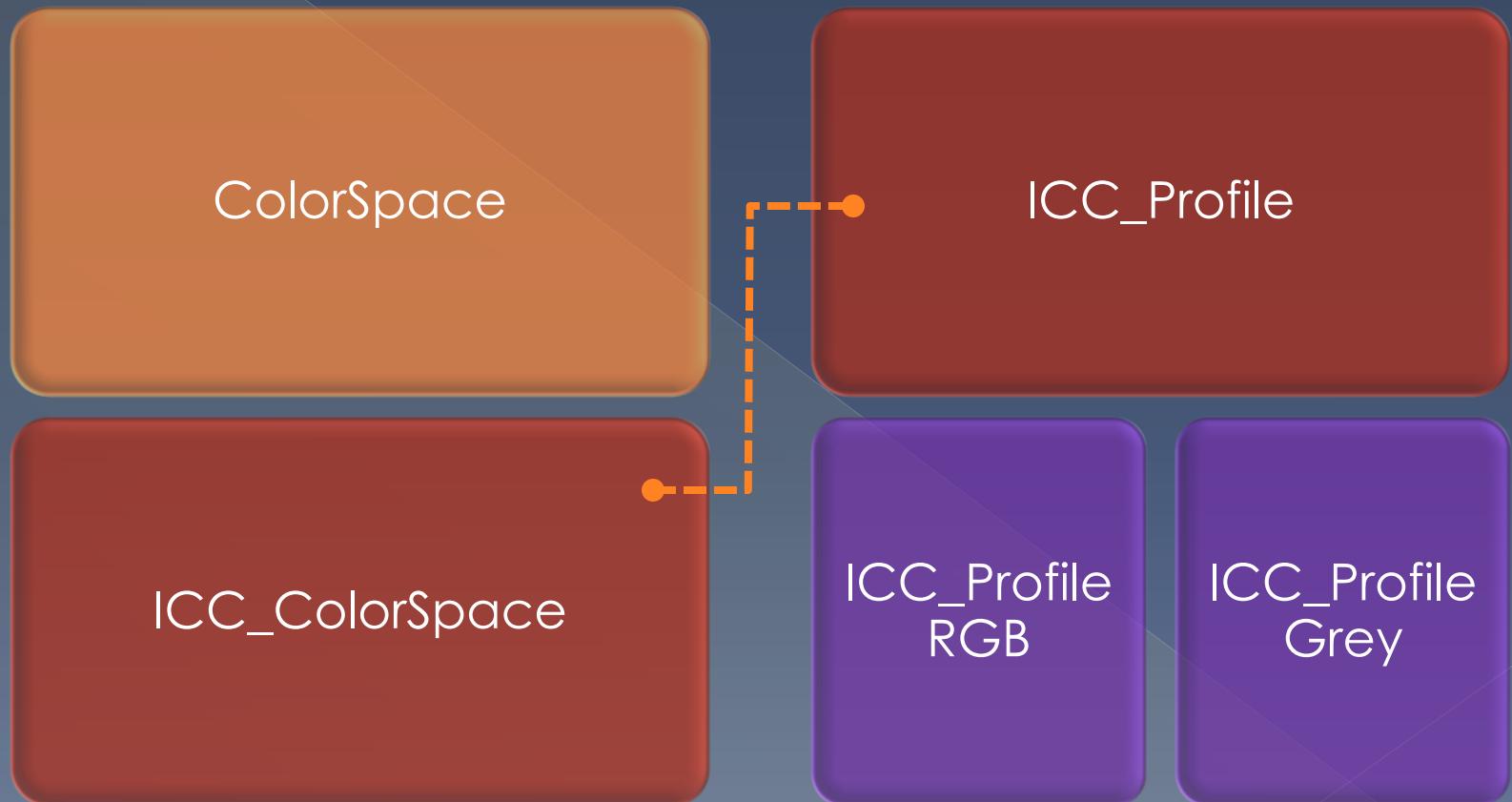
Component  
ColorModel

Indexed  
ColorModel

Packed  
ColorModel

Direct  
ColorModel

# ColorSpace



# BufferedImage

Raster

SampleModel

DataBuffer

ColorModel

ColorSpace

# BufferedImage



Algunos métodos

- *getRaster()*
- *getColorModel()*
- *getSampleModel()*
- *getSubImage(...)*
- *getWidth()*
- *getHeight()*
- *getRGB(int x, int y)*
- *setRGB(int x, int y, int rgb)*



Acceso directo:

```
r = (cint >> 16) & 0xFF;  
g = (cint >> 8) & 0xFF;  
b = cint & 0xFF;
```

Usando Color:

- Color c = new Color(cint);
- cint = c.getRGB();

Usando el ColorModel

# Raster



Algunos métodos

- `getDataBuffer()`
- `getSampleModel()`
- `getWidth()`
- `getHeight()`
- `getNumBands()`
- `(set) getPixel(...)`
- `(set) getSample(...)`
- Métodos “create”

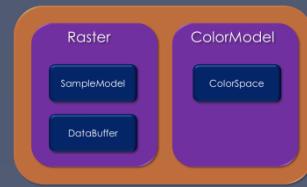


# Raster



Algunos métodos

- `getDataBuffer()`
- `getSampleModel()`
- `getWidth()`
- `getHeight()`
- `getNumBands()`
- `(set) getPixel(int x, int y, int []d)`
- `(set) getSample(int x, int y, int b) [, int s])`
- Métodos “create”

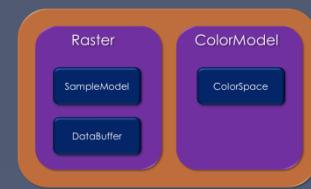


# ColorModel



Algunos métodos

- getColorSpace()
- hasAlpha()
- getRed(int pixel)
- getGreen(int pixel)
- getBlue(int pixel)
- getAlpha(int pixel)
- createCompatibleWriteRaster(...)



# ColorSpace



Algunos métodos

- `toRGB(float []color)`
- `fromRGB(float []rgb)`
- `getInstance(int tipo)`

ColorSpace.CS\_sRGB  
ColorSpace.CS\_LINEAR\_RGB  
ColorSpace.CS\_GRAY  
ColorSpace.CS\_YCC  
ColorSpace.CS\_CIEXYZ

# Creación de BufferedImage

1. **new** BufferedImage(**int width, int height, int imageType**)

TYPE\_INT\_RGB

TYPE\_INT\_ARGB

TYPE\_INT\_ARGB\_PRE

TYPE\_INT\_BGR

TYPE\_BYTE\_BINARY

TYPE\_BYTE\_GRAY

TYPE\_BYTE\_INDEXED

TYPE\_3BYTE\_BGR

TYPE\_4BYTE\_ABGR

TYPE\_4BYTE\_ABGR\_PRE

TYPE USHORT\_555\_RGB

TYPE USHORT\_565\_RGB

TYPE USHORT\_GRAY

TYPE\_CUSTOM

# Creación de BufferedImage

1. **new** BufferedImage(int width, int height, int imageType)

```
BufferedImage img;  
img = new BufferedImage(500, 500, BufferedImage.TYPE_INT_RGB);
```

# Creación de BufferedImage

2. **new** BufferedImage(**int width, int height, int imageType, IndexColorModel cm**)

|                   |                     |
|-------------------|---------------------|
| TYPE_INT_RGB      | TYPE_3BYTE_BGR      |
| TYPE_INT_ARGB     | TYPE_4BYTE_ABGR     |
| TYPE_INT_ARGB_PRE | TYPE_4BYTE_ABGR_PRE |
| TYPE_INT_BGR      | TYPE USHORT_555_RGB |
| TYPE_BYTE_BINARY  | TYPE USHORT_565_RGB |
| TYPE_BYTE_GRAY    | TYPE USHORT_GRAY    |
| TYPE_BYTE_INDEXED | TYPE_CUSTOM         |



# Creación de BufferedImage

3. **new** `BufferedImage(`  
    Color Model `cm`,  
    WritableRaster `raster`,  
    boolean `isRasterPremultiplied`,  
    Hashtable<?,?> `properties`)



# Creación de BufferedImage

```
ColorSpace cs = ColorSpace.getInstance(ColorSpace.CS_sRGB);
ColorModel cm =
    new ComponentColorModel(cs,false,false,
                           Transparency.OPAQUE,
                           DataBuffer.TYPE_BYTE);

SampleModel sm =
    new BandedSampleModel(DataBuffer.TYPE_BYTE,200,200,3);
DataBuffer db = new DataBufferByte(200*200,3);
WritableRaster wr = WritableRaster.createWritableRaster(sm,db,null);

  
img = new BufferedImage(cm,wr,false,null);
```

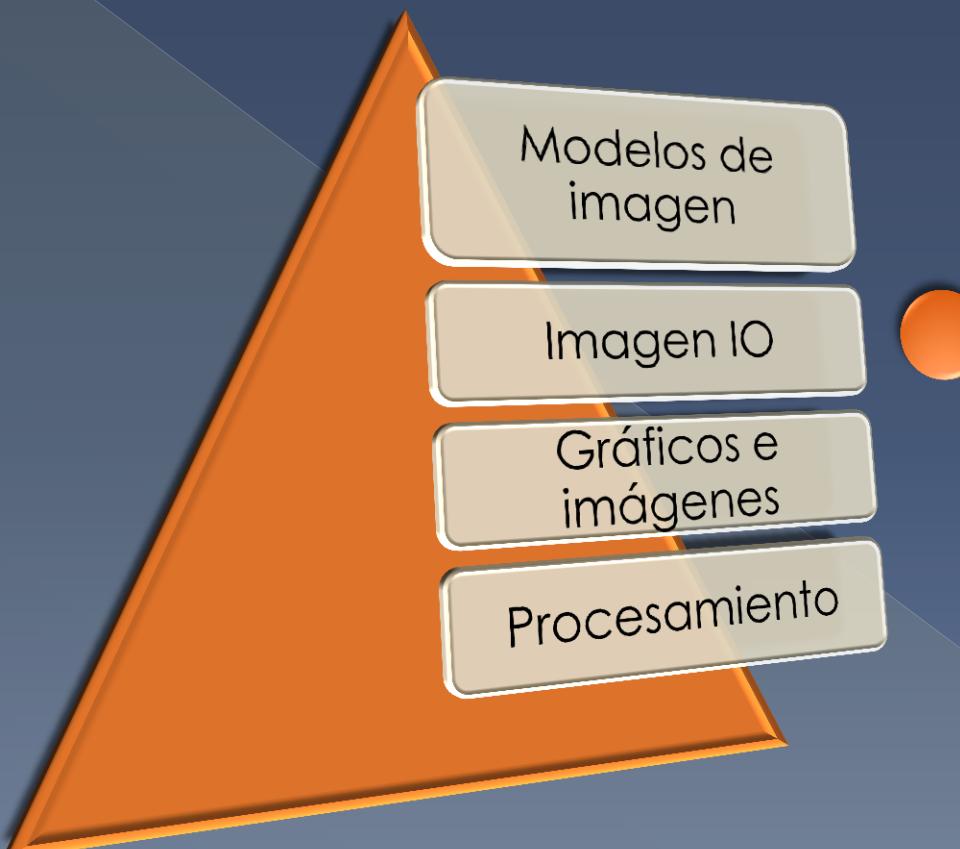


# Ejemplo

Extraer una banda

```
public BufferedImage getImageBand(BufferedImage srclImage, int band) {  
    ColorSpace cs = ColorSpace.getInstance(ColorSpace.CS_GRAY);  
    ComponentColorModel cm = new ComponentColorModel(cs, false, false,  
                                                    Transparency.OPAQUE,  
                                                    DataBuffer.TYPE_BYTE);  
  
    int bandList[] = {band};  
    WritableRaster bandRaster = (WritableRaster) srclImage.getRaster().createWritableChild(0, 0,  
                                         srclImage.getWidth(), srclImage.getHeight(), 0, 0, bandList);  
  
    return new BufferedImage(cm, bandRaster, false, null);  
}
```

# Índice



Modelos de  
imagen

Imagen IO

Gráficos e  
imágenes

Procesamiento

# Lectura de imagen

```
BufferedImage img;  
try{  
    File f = new File("...");  
    img = ImageIO.read(f);  
}catch(Exception ex){  
    System.err.println("Error al leer la imagen");  
}
```

# Lectura de imagen

Ejemplo con diálogo

```
BufferedImage img;  
JFileChooser dlg = new JFileChooser();  
int resp = dlg.showOpenDialog(this);  
if( resp == JFileChooser.APPROVE_OPTION) {  
    try{  
        File f = dlg.getSelectedFile();  
        img = ImageIO.read(f);  
    }catch(Exception ex){  
        System.err.println("Error al leer la imagen");  
    }  
}
```

# Escritura de imagen

```
BufferedImage img = ....;
```

```
try{
```

```
    File f = new File("...");  
    ImageIO.write(img,"jpg",f);  
}catch(Exception ex){  
    System.err.println("Error al leer la imagen");  
}
```

|      |      |
|------|------|
| jpg  | JPG  |
| jpeg | JPEG |
| png  | PNG  |
| gif  | GIF  |

# Escritura de imagen

Ejemplo con diálogo

```
BufferedImage img = ....;
JFileChooser dlg = new JFileChooser();
int resp = dlg.showSaveDialog(this);
if( resp == JFileChooser.APPROVE_OPTION) {
    try{
        File f = dlg.getSelectedFile();
        ImageIO.write(img,"jpg",f);
    }catch(Exception ex){
        System.err.println("Error al leer la imagen");
    }
}
```

# Índice



Modelos de  
imagen

Imagen IO

Gráficos e  
imágenes

Procesamiento

# Gráficos e imagen

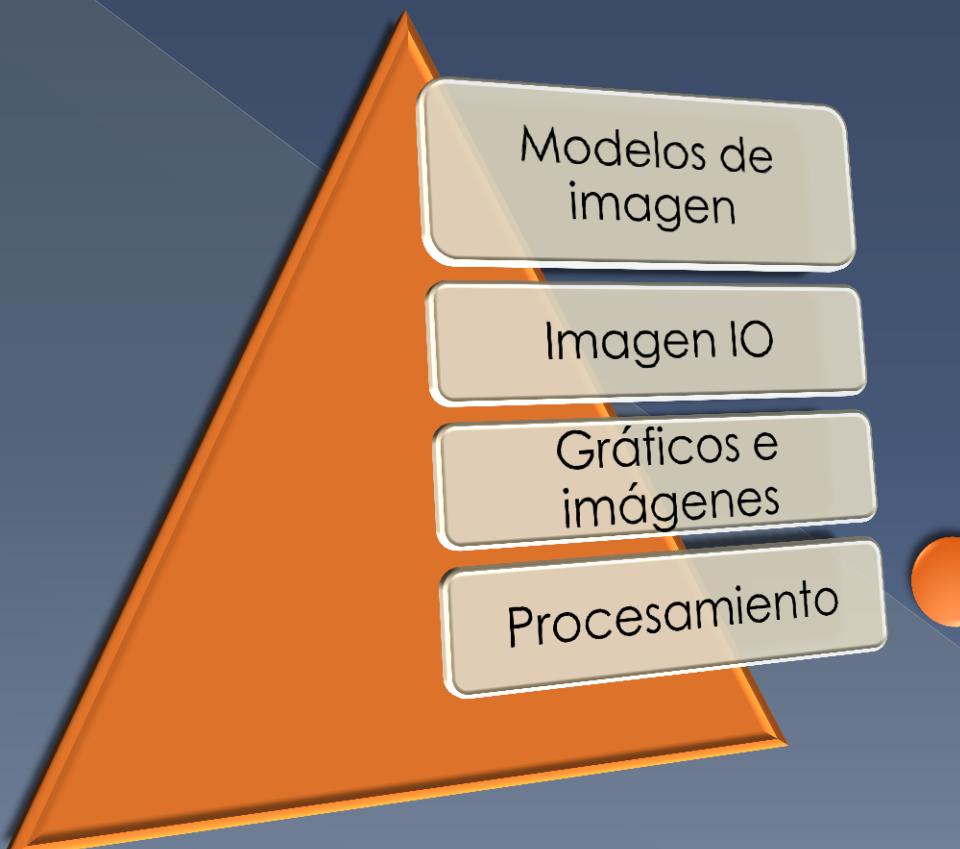
Dibujar en una imagen

- Para dibujar en un *BufferedImage* bastará crear un *Graphics2D* asociado a la imagen y dibujar sobre él

BufferedImage

```
Graphics2D g2d;  
g2d = img.createGraphics();  
  
g2d.draw(s); // con s un Shape
```

# Índice



Modelos de  
imagen

Imagen IO

Gráficos e  
imágenes

Procesamiento

# Operaciones sobre imagen

## Categorías

### Tipo resultado

#### Procesamiento

- Recibe como entrada una imagen y genera como salida otra imagen (procesada)

### Dominio

#### Espacial

- Cada pixel (color o intensidad) está asociado a una localización espacial ( $x, y$ )

### Área de cálculo

#### Punto

- Se procesa solo un pixel (o ventana fija entorno a él) para generar la salida

#### Análisis

- Recibe como entrada una imagen y genera como salida información (estructurada) sobre la imagen

#### Frecuencial

- Representa las amplitudes de frecuencia en el espectro

#### Región

- Se procesa región (agrupación de píxeles) para generar la salida

# Operaciones sobre imagen

## Categorías

### Tipo resultado

#### Procesamiento

- Recibe como entrada una imagen y genera como salida otra imagen (procesada)

#### Análisis

- Recibe como entrada una imagen y genera como salida información (estructurada) sobre la imagen

### Dominio

#### Espacial

- Cada pixel (color o intensidad) está asociado a una localización espacial ( $x, y$ )

#### Frecuencial

- Representa las amplitudes de frecuencia en el espectro

### Área de cálculo

#### Punto

- Se procesa solo un pixel (o ventana fija entorno a él) para generar la salida

#### Región

- Se procesa región (agrupación de píxeles) para generar la salida

# Operaciones sobre imagen

## Categorías

### Tipo resultado

#### Procesamiento

- Recibe como entrada una imagen y genera como salida otra imagen (procesada)

### Dominio

#### Espacial

- Cada pixel (color o intensidad) está asociado a una localización espacial ( $x, y$ )

### Área de cálculo

#### Punto

- Se procesa solo un pixel (o ventana fija entorno a él) para generar la salida

#### Análisis

- Recibe como entrada una imagen y genera como salida información (estructurada) sobre la imagen

#### Frecuencial

- Representa las amplitudes de frecuencia en el espectro

#### Región

- Se procesa región (agrupación de píxeles) para generar la salida

# Operaciones sobre imagen

## Categorías

### Tipo resultado

#### Procesamiento

- Recibe como entrada una imagen y genera como salida otra imagen (procesada)

#### Análisis

- Recibe como entrada una imagen y genera como salida información (estructurada) sobre la imagen

### Dominio

#### Espacial

- Cada pixel (color o intensidad) está asociado a una localización espacial ( $x, y$ )

#### Frecuencial

- Representa las amplitudes de frecuencia en el espectro

### Área de cálculo

#### Punto

- Se procesa solo un pixel (o ventana fija entorno a él) para generar la salida

#### Región

- Se procesa región (agrupación de píxeles) para generar la salida

#### Segmentación

# Procesamiento de imágenes

- › Java 2D implementa una serie de operaciones “punto a punto” sobre la imagen
- › Cada una de estas operaciones es un objeto de la clase *BufferedImageOp* (hay subclases para cada operación)



*BufferedImageOp op = new .....*



-----> La clase dependerá de la operación

# Procesamiento de imágenes

- › Java 2D implementa una serie de operaciones “punto a punto” sobre la imagen
- › Cada una de estas operaciones es un objeto de la clase *BufferedImageOp* (hay subclases para cada operación)
- › Para aplicar la operación se usa el método *filter*:

```
public BufferedImage filter(BufferedImage src, BufferedImage dest)
```

Devuelve un *BufferedImage*  
con el resultado del filtrado

Imagen a la que  
se le aplica el filtro

Imagen en la que almacena el resultado.  
Puede ser null, en cuyo caso el método  
crea internamente la imagen

*BufferedImage* *scr* = .....

*BufferedImageOp* *op*= new .....

*BufferedImage* *dest* = *op.filter(src, null)*;

La clase dependerá de la operación

# Procesamiento de imágenes

- › Java 2D implementa una serie de operaciones “punto a punto” sobre la imagen
- › Cada una de estas operaciones es un objeto de la clase *BufferedImageOp* (hay subclases para cada operación)
- › Para aplicar la operación se usa el método *filter*:

```
public BufferedImage filter(BufferedImage src, BufferedImage dest)
```

Devuelve un *BufferedImage*  
con el resultado del filtrado

Imagen a la que  
se le aplica el filtro

Imagen en la que almacena el resultado.  
Puede ser null, en cuyo caso el método  
crea internamente la imagen

*BufferedImage scr* = .....

*BufferedImage dest* = .....

*BufferedImageOp op* = new .....  
*op.filter(src, dest);*

La clase dependerá de la operación

# Procesamiento de imágenes

- › Java 2D implementa una serie de operaciones “punto a punto” sobre la imagen
- › Cada una de estas operaciones es un objeto de la clase *BufferedImageOp* (hay subclases para cada operación)
- › Para aplicar la operación se usa el método *filter*:

```
public BufferedImage filter(BufferedImage src, BufferedImage dest)
```

Devuelve un *BufferedImage*  
con el resultado del filtrado

Imagen a la que  
se le aplica el filtro

Imagen en la que almacena el resultado.  
Puede ser null, en cuyo caso el método  
crea internamente la imagen

*BufferedImage* *scr* = .....

*BufferedImageOp* *op*= new .....  
*op.filter(src, src);*



La clase dependerá de la operación  
No siempre está permitido, depende de la operación

# Procesamiento de imágenes

## BufferedImageOp

RescaleOp

ConvolveOp

AffineTransformOp

LookupOp

BandCombineOp

ColorConvertOp

# Procesamiento de imágenes

BufferedImageOp

RescaleOp

ConvolveOp

AffineTransformOp

LookupOp

BandCombineOp

ColorConvertOp

# RescaleOp

- Para cada componente de cada pixel  $f(x,y)$ , aplica la transformación:

$$g(x,y) = \textcolor{orange}{a} * f(x,y) + \textcolor{orange}{b}$$

donde  $f(x,y)$  representa el valor del pixel en la coordenada  $(x,y)$  de la imagen original (p.e,  $f(3,3) = [255,0,0]$ ). La salida  $g(x,y)$  se ajusta al rango  $[0,255]$

- Parámetros
  - $\textcolor{orange}{a}$  : factor de escala (o ganancia). Influye en el contraste: si  $0 < a < 1$  lo reduce, si  $a > 1$  lo aumenta
  - $\textcolor{orange}{b}$  : desfase (offset). Influye en el brillo: si  $b < 0$  lo reduce, si  $b > 0$  lo aumenta
- Construcción:
  - `new RescaleOp(float a, float b, RenderingHints hints)`
  - `new RescaleOp(float[] a, float[] b, RenderingHints hints)`

```
new RescaleOp(float a, float b, RenderingHints hints)
new RescaleOp(float[] a, float[] b, RenderingHints hints)
```

# RescaleOp

Brillo

a

b

RenderingHints

*new RescaleOp(1.0F, 100.0F, null)*



# RescaleOp

Brillo

*new RescaleOp(1.0F,-100.0F,null)*



# RescaleOp

Brillo

*new RescaleOp(1.5F, 0.0F,null)*



# RescaleOp

Brillo

*new RescaleOp(0.5F, 0.0F,null)*



# RescaleOp

Código ejemplo

```
try{
    RescaleOp rop = new RescaleOp(1.0f,100.0f, null);
    BufferedImage imgOut = rop.filter(img,null);
} catch(Exception e){
    System.err.println("Error");
}
```

# Procesamiento de imágenes

BufferedImageOp

RescaleOp

ConvolveOp

AffineTransformOp

LookupOp

BandCombineOp

ColorConvertOp

# ConvolveOp

- Para cada componente de cada pixel  $f(x,y)$ , aplica la operación de convolución dada por:

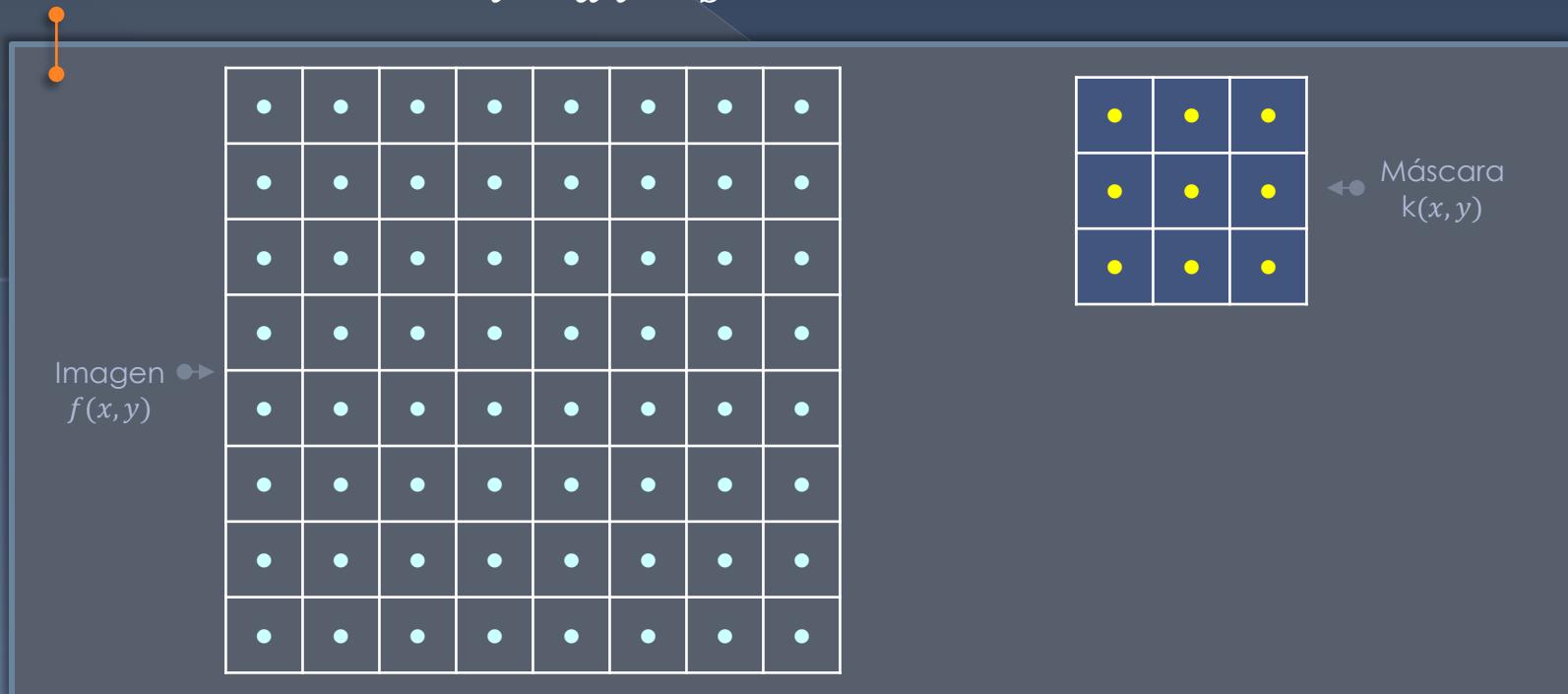
$$g(x, y) = \sum_{i=-a}^a \sum_{j=-b}^b k(i, j) * f(x - i, y - j)$$

- Parámetros
  - $\mathbf{k}$ : máscara de convolución de tamaño  $m \times n$

# ConvolveOp

- Para cada componente de cada pixel  $f(x,y)$ , aplica la operación de convolución dada por:

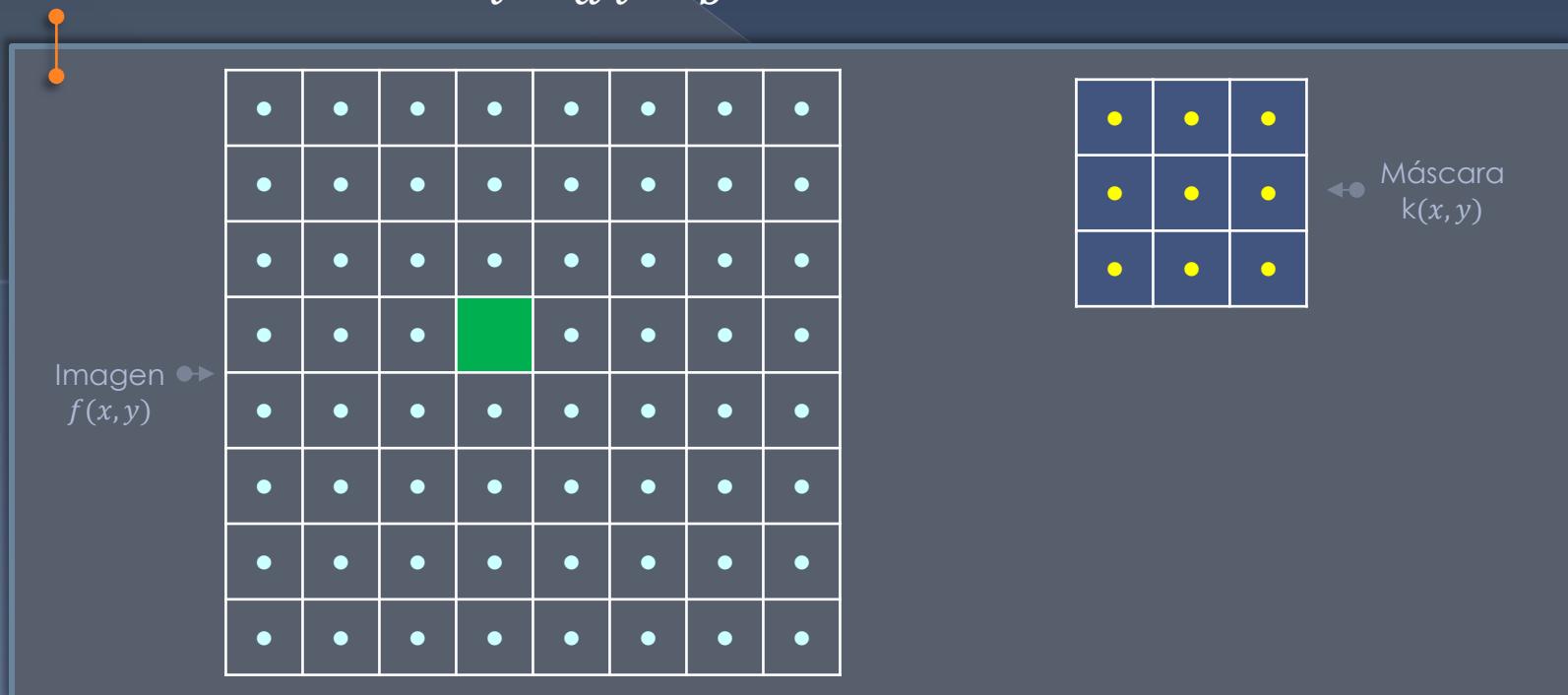
$$g(x, y) = \sum_{i=-a}^a \sum_{j=-b}^b k(i, j) * f(x - i, y - j)$$



# ConvolveOp

- Para cada componente de cada pixel  $f(x,y)$ , aplica la operación de convolución dada por:

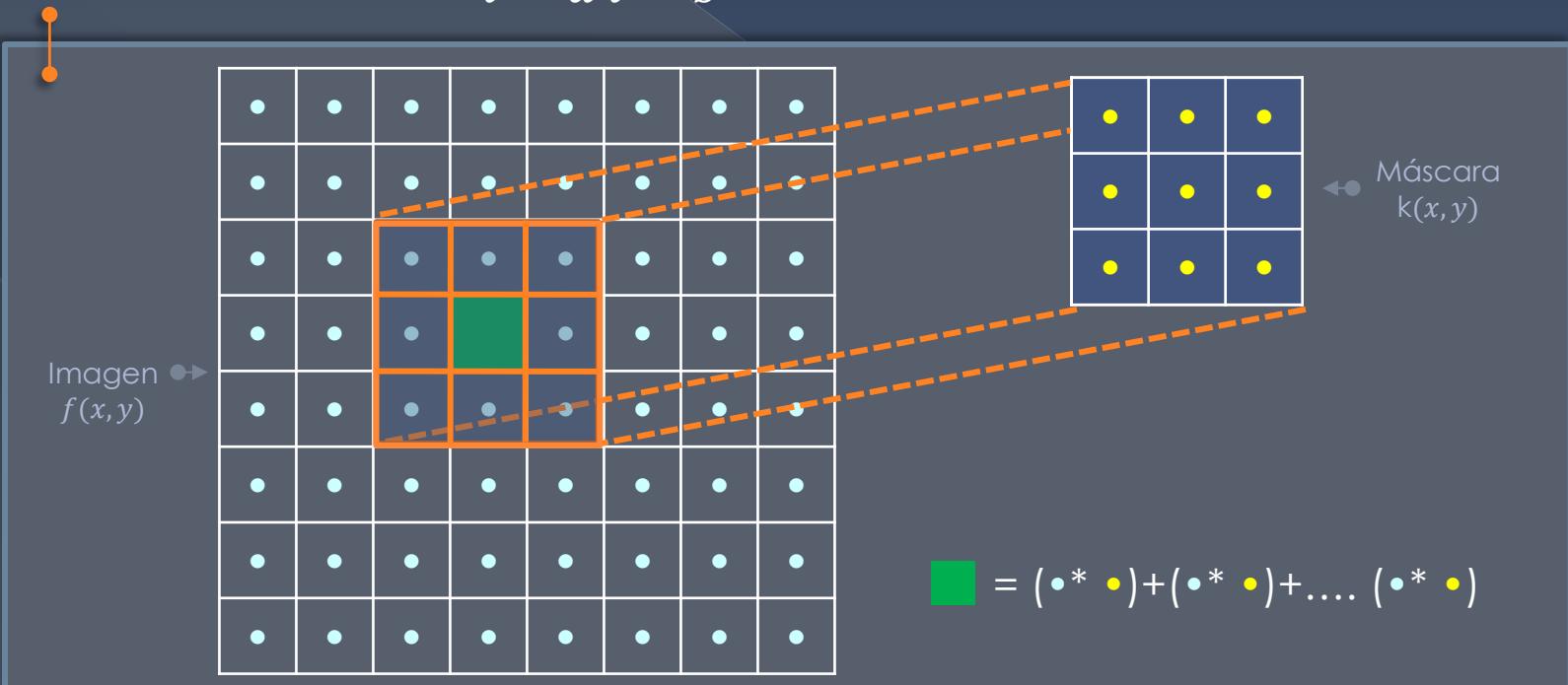
$$g(x, y) = \sum_{i=-a}^a \sum_{j=-b}^b k(i, j) * f(x - i, y - j)$$



# ConvolveOp

- Para cada componente de cada pixel  $f(x,y)$ , aplica la operación de convolución dada por:

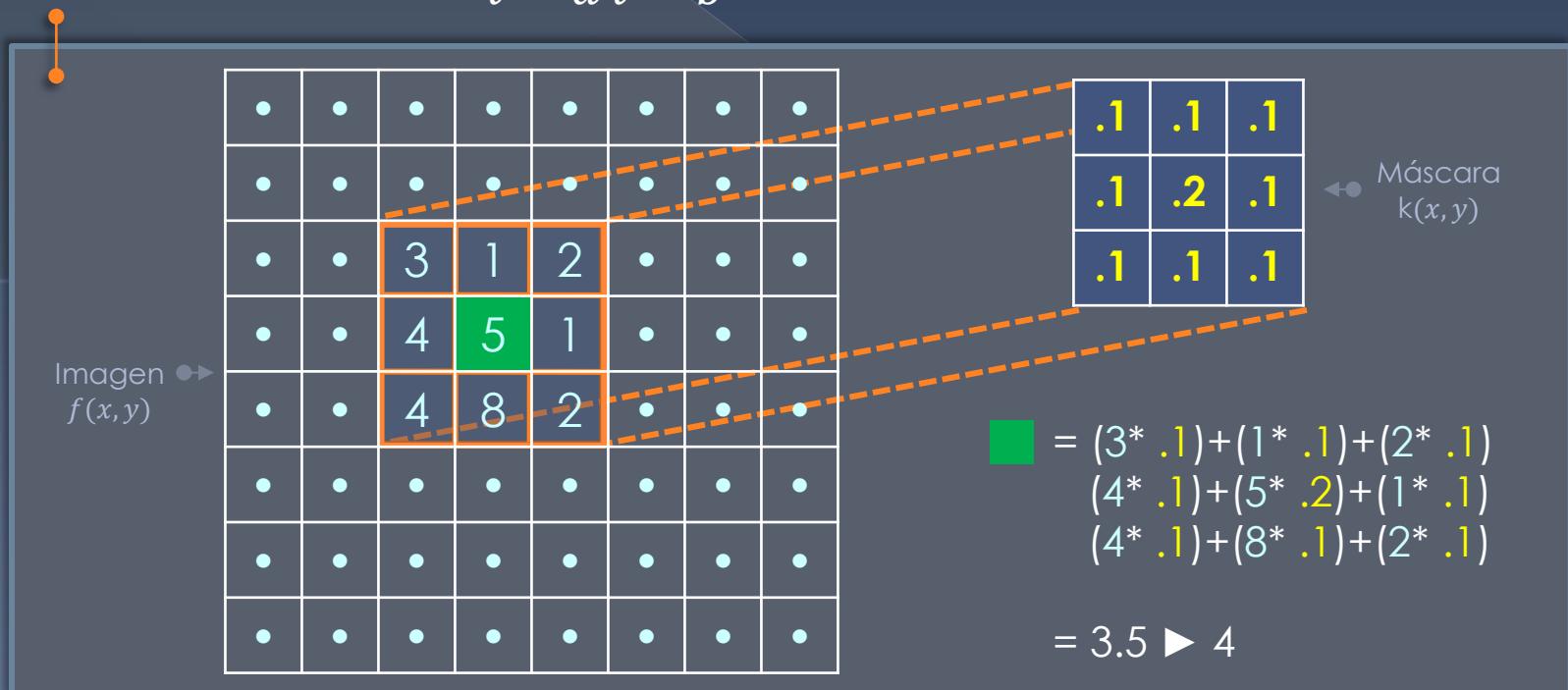
$$g(x, y) = \sum_{i=-a}^a \sum_{j=-b}^b k(i, j) * f(x - i, y - j)$$



# ConvolveOp

- Para cada componente de cada pixel  $f(x,y)$ , aplica la operación de convolución dada por:

$$g(x, y) = \sum_{i=-a}^a \sum_{j=-b}^b k(i, j) * f(x - i, y - j)$$



# ConvolveOp

- Para cada componente de cada pixel  $f(x,y)$ , aplica la operación de convolución dada por:

$$g(x, y) = \sum_{i=-a}^a \sum_{j=-b}^b k(i, j) * f(x - i, y - j)$$

- Parámetros
  - $\mathbf{k}$ : máscara de convolución de tamaño  $m \times n$
- Construcción:

```
float m[] = {0.1f, 0.1f, 0.1f, ..., 0.1f,}
Kernel k = new Kernel(3,3,m);
ConvolveOp cop = new ConvolveOp(k);
```

# ConvolveOp

Suavizado



|     |     |     |
|-----|-----|-----|
| 1/9 | 1/9 | 1/9 |
| 1/9 | 1/9 | 1/9 |
| 1/9 | 1/9 | 1/9 |

Filtro media

# ConvolveOp

Suavizado



1/16 \*

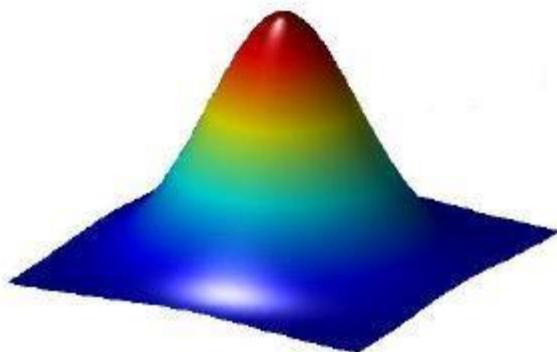
|   |   |   |
|---|---|---|
| 1 | 2 | 1 |
| 2 | 4 | 2 |
| 1 | 2 | 1 |

Filtro binomial

# ConvolveOp

Suavizado

|        |        |        |        |        |
|--------|--------|--------|--------|--------|
| 0.0030 | 0.0133 | 0.0219 | 0.0133 | 0.0030 |
| 0.0133 | 0.0596 | 0.0983 | 0.0596 | 0.0133 |
| 0.0219 | 0.0983 | 0.1621 | 0.0983 | 0.0219 |
| 0.0133 | 0.0596 | 0.0983 | 0.0596 | 0.0133 |
| 0.0030 | 0.0133 | 0.0219 | 0.0133 | 0.0030 |



$$f(x, y) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{x^2 + y^2}{2\sigma^2}\right]$$

Filtro gaussiano

# ConvolveOp

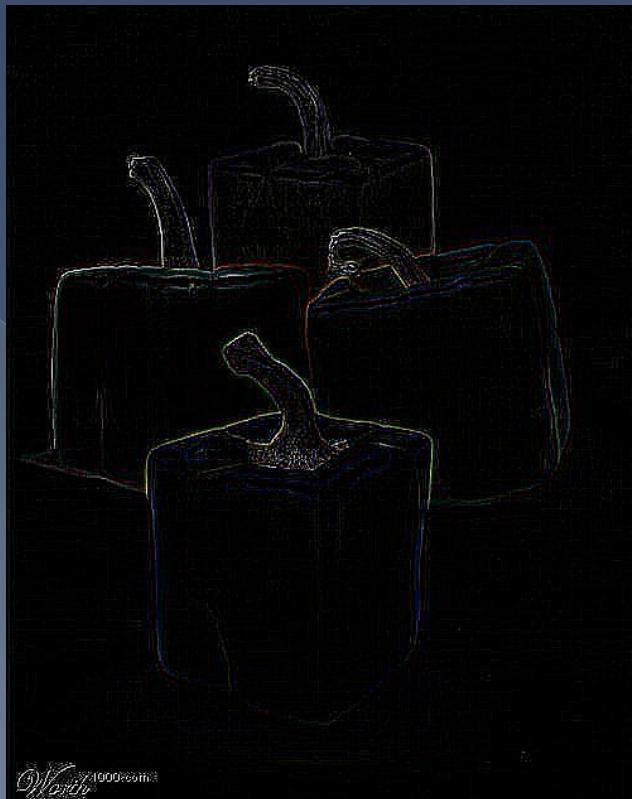
Realce

|    |    |    |
|----|----|----|
| 0  | -1 | 0  |
| -1 | 5  | -1 |
| 0  | -1 | 0  |



# ConvolveOp

Fronteras



|   |    |   |
|---|----|---|
| 1 | 1  | 1 |
| 1 | -8 | 1 |
| 1 | 1  | 1 |

Filtro laplaciano

# ConvolveOp

Código ejemplo

```
try{  
    float filtroMedia[] = { 0.1f, 0.1f, 0.1f,  
                           0.1f, 0.2f, 0.1f ,  
                           0.1f, 0.1f, 0.1f } ;  
    Kernel k = new Kernel(3,3,filtroMedia);  
    ConvolveOp cop = new ConvolveOp( k );  
    BufferedImage imgOut = cop.filter(img,null);  
}catch(Exception e){  
    System.err.println("Error");  
}
```

# Procesamiento de imágenes

BufferedImageOp

RescaleOp

ConvolveOp

AffineTransformOp

LookupOp

BandCombineOp

ColorConvertOp

# AffineTransformOp

- › Realiza una transformación geométrica (rotación, escalado, etc.) de la imagen:

$$\begin{bmatrix} x_g \\ y_g \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & d_x & t_x \\ d_y & s_y & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_f \\ y_f \\ 1 \end{bmatrix}$$

con  $(x_f, y_f)$  coordenadas de la imagen original y  $(x_g, y_g)$  de la imagen transformada. No transforma los valores de los píxeles, sino que los cambia de posición (en algunos casos, deberán de interpolarse valores)

- › Parámetros
  - › **M** : Matriz de transformación

# AffineTransformOp

- Realiza una transformación geométrica (rotación, escalado, etc.) de la imagen:

$$\begin{bmatrix} x_g \\ y_g \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & d_x & t_x \\ d_y & s_y & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_f \\ y_f \\ 1 \end{bmatrix}$$

con  $(x_f, y_f)$  coordenadas de la imagen original y  $(x_g, y_g)$  de la imagen transformada. No transforma los valores de los píxeles, sino que los cambia de posición (en algunos casos, deberán de interpolarse valores)

- Construcción:

`AffineTransform at = ....` → Véase transformaciones afines en el tema 5

`AffineTransformOp op = new AffineTransformOp(at,...);`

Tipo de interpolación aplicada, si procede (p.e., `AffineTransformOp.TYPE_BILINEAR`)

# AffineTransformOp

Rotación



# AffineTransformOp

Escalado



# AffineTransformOp

Shear



# AffineTransformOp

## Código ejemplo

```
try{  
    AffineTransform at = AffineTransform.getScaleInstance(1.25,1.25);  
    AffineTransformOp atop = new AffineTransformOp(at,  
                                              AffineTransformOp.TYPE_BILINEAR);  
    BufferedImage imgdest = atop.filter( img, null);  
}catch(Exception e)  
{  
    System.err.println("Error");  
}
```

concatenate(...)  
translate(...)  
scale(...)  
rotate(...)  
shear(...)

AffineTransform.**getTranslateInstance**(double tx, double ty)  
AffineTransform.**getScaleInstance**(double sx, double sy)  
AffineTransform.**getRotateInstance**(double theta,...)  
AffineTransform.**getShearInstance**(double shx, double shy)

# Procesamiento de imágenes

BufferedImageOp

RescaleOp

ConvolveOp

AffineTransformOp

LookupOp

BandCombineOp

ColorConvertOp

# LookupOp

- Para cada componente de cada pixel  $f(x,y)$ , aplica la transformación:

$$g(x,y)_i = \mathbf{T}(f(x,y)_i)$$

con  $\mathbf{T}$  una función definida como:

$$\mathbf{T} : [0,255] \rightarrow [0,255]$$

Puede ser la misma para todos los componentes, o una distinta para cada uno

- Parámetros

- $\mathbf{T}$  : La función de transformación

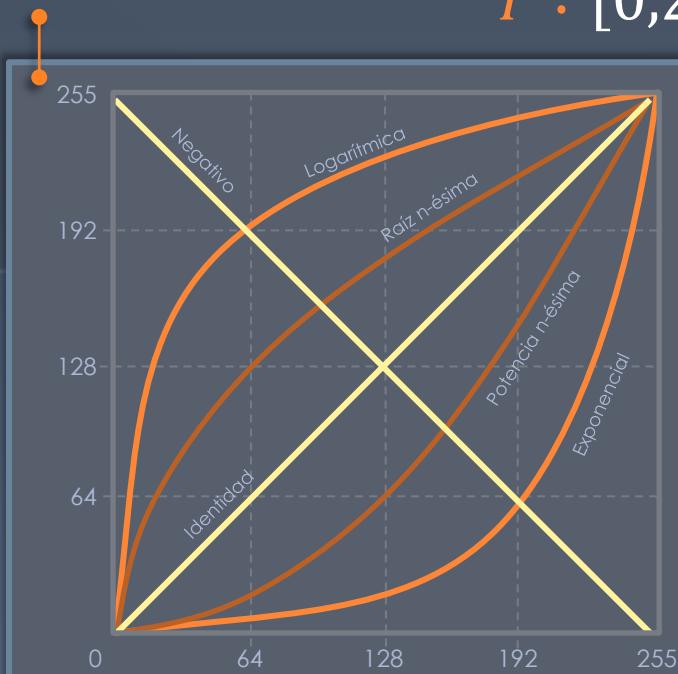
# LookupOp

- Para cada componente de cada pixel  $f(x,y)$ , aplica la transformación:

$$g(x,y)_i = \mathbf{T}(f(x,y)_i)$$

con  $\mathbf{T}$  una función definida como:

$$\mathbf{T} : [0,255] \rightarrow [0,255]$$



Identidad:  $T(x) = x$

Negativo:  $T(x) = 255 - x$

|                       |     |     |
|-----------------------|-----|-----|
| 100                   | 250 | 230 |
| 90                    | 73  | 106 |
| [255,<br>100,<br>180] | 27  | 48  |

$f(x,y)$

|                     |     |     |
|---------------------|-----|-----|
| 155                 | 5   | 35  |
| 165                 | 182 | 149 |
| [ 0,<br>155,<br>75] | 228 | 207 |

$g(x,y)$

# LookupOp

- Para cada componente de cada pixel  $f(x,y)$ , aplica la transformación:

$$g(x,y)_i = \mathbf{T}(f(x,y)_i)$$

con  $\mathbf{T}$  una función definida como:

$$\mathbf{T} : [0,255] \rightarrow [0,255]$$



# LookupOp

- Para cada componente de cada pixel  $f(x,y)$ , aplica la transformación:

$$g(x,y)_i = \mathbf{T}(f(x,y)_i)$$

con  $\mathbf{T}$  una función definida como:

$$\mathbf{T} : [0,255] \rightarrow [0,255]$$

- Parámetros
  - $\mathbf{T}$  : La función de transformación en forma de tabla

- Construcción:

```
byte funcionT[] = {200, 190, 89, 38, 177, ..., 100};
```

```
LookupTable table = new ByteLookupTable(0, funcionT);
```

```
LookupOp lop = new LookupOp(table, ...);
```

# LookupOp

- Para cada componente de cada pixel  $f(x,y)$ , aplica la transformación:

$$g(x,y)_i = \mathbf{T}(f(x,y)_i)$$

con  $\mathbf{T}$  una función definida como:

$$\mathbf{T} : [0,255] \rightarrow [0,255]$$

- Parámetros
  - $\mathbf{T}$  : La función de transformación en forma de tabla

- Construcción:

```
byte funcionT[] = new byte[256];
for (int x=0; x<256; x++) {
    funcionT[x] = (byte)(255-x); // Negativo
}
```

$$T(x) = 255 - x$$

# LookupOp

- Para cada componente de cada pixel  $f(x,y)$ , aplica la transformación:

$$g(x,y)_i = \mathbf{T}(f(x,y)_i)$$

con  $\mathbf{T}$  una función definida como:

$$\mathbf{T} : [0,255] \rightarrow [0,255]$$

- Parámetros
  - $\mathbf{T}$  : La función de transformación en forma de tabla

- Construcción:

```
byte funcionT[] = new byte[256];
for (int x=0; x<256; x++) {
    funcionT[x] = (byte)(Math.log(1.0+(double)x)); // Logaritmo
}
```

$$T(x) = \log(1 + x)$$

No se mueve en el rango [0,255]  
(el valor máximo que alcanza es 5)

# LookupOp

- Para cada componente de cada pixel  $f(x,y)$ , aplica la transformación:

$$g(x,y)_i = \mathbf{T}(f(x,y)_i)$$

con  $\mathbf{T}$  una función definida como:

$$\mathbf{T} : [0,255] \rightarrow [0,255]$$

- Parámetros
  - $\mathbf{T}$  : La función de transformación en forma de tabla

- Construcción:

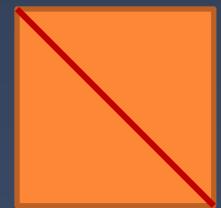
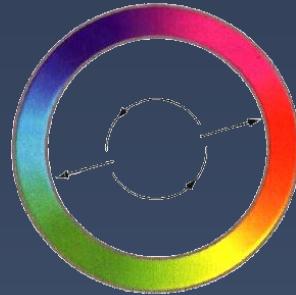
```
byte funcionT[] = new byte[256];
double K = 255.0/Math.log(1.0+255.0); // Cte. normalización  $\frac{255}{MAX}$ 
for (int x=0; x<256; x++) {
    funcionT[x] = (byte)(K*Math.log(1.0+(double)x)); // Logaritmo
}
```

$$T(x) = \log(1 + x)$$

$T(x)$

# LookupOp

Negativo

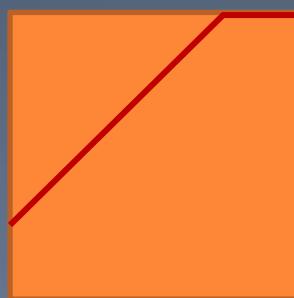


Worth<sup>1000.com</sup>

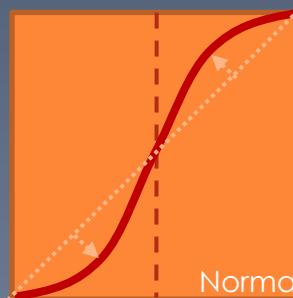
# LookupOp

Brillo y contraste

- Se busca ajustar el brillo y contraste para obtener mayor detalle en un rango de intensidades dado sin cambiar los colores



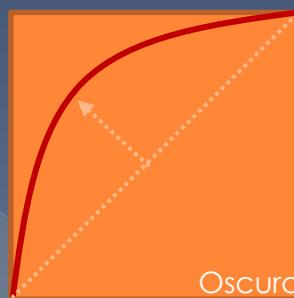
Brillo



Normal



Iluminada



Oscura

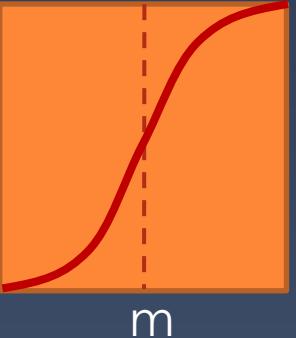
Brillo y contraste

# LookupOp

Brillo y contraste



$$T(x) = \frac{1}{1 + (m/x)^E}$$

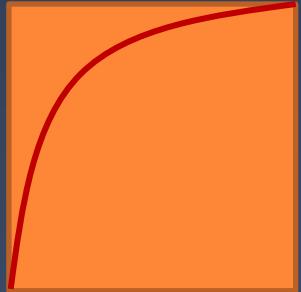


# LookupOp

Gamma



$$T(x) = A \cdot x^\gamma$$



# LookupOp

Código ejemplo

```
try{
    byte fT[] = new byte[256];
    for (int x=0; x<256; x++)
        fT[x] = (byte)(255-x); // Negativo
    ByteLookupTable tabla = new ByteLookupTable(0, fT);
    LookupOp lop = new LookupOp(tabla, null);
    BufferedImage imgdest = lop.filter(img,null);
}catch(Exception e){
    System.err.println("Error");
}
```



# Procesamiento de imágenes

BufferedImageOp

RescaleOp

ConvolveOp

AffineTransformOp

LookupOp

BandCombineOp

ColorConvertOp

# BandCombineOp

- Realiza una combinación lineal de las bandas de una imagen. Para cada píxel  $f(x,y) = [b1_f, b2_f, b3_f]$  de la imagen original, aplica la transformación:

$$\begin{bmatrix} b1_g \\ b2_g \\ b3_g \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{bmatrix} \begin{bmatrix} b1_f \\ b2_f \\ b3_f \end{bmatrix}$$

$$\text{con } g(x,y) = [b1_g, b2_g, b3_g]$$

- Parámetros
  - $\mathbf{M}$  : La matriz de combinación lineal

$$\begin{aligned} b1_g &= m_{00}b1_f + m_{01}b2_f + m_{02}b3_f \\ b2_g &= m_{10}b1_f + m_{11}b2_f + m_{12}b3_f \\ b3_g &= m_{20}b1_f + m_{21}b2_f + m_{22}b3_f \end{aligned}$$

- Construcción:

```
float [][] matriz = {{1.0F,0.0F,0.0F},{0.0F,0.0F,1.0F},{0.0F,1.0F,0.0F}};
```

```
BandCombineOp op = new BandCombineOp(matriz, ... );
```

# BandCombineOp

|   |   |   |
|---|---|---|
| 1 | 0 | 0 |
| 0 | 0 | 1 |
| 0 | 1 | 0 |



# BandCombineOp

## Código ejemplo

```
try{  
    float [][] matriz = {{1.0F,0.0F,0.0F},{0.0F,0.0F,1.0F},{0.0F,1.0F,0.0F}};  
    BandCombineOp bcop = new BandCombineOp(matriz , null );  
    WritableRaster rasterdest = bcop.filter(img.getRaster(),null);  
    BufferedImage imgdest = new BufferedImage(img.getColorModel(),  
                                              rasterdest,false,null);  
}  
catch(Exception e){  
    System.err.println("Error");  
}
```

Solo se puede aplicar sobre el raster

# Procesamiento de imágenes

BufferedImageOp

RescaleOp

ConvolveOp

AffineTransformOp

LookupOp

BandCombineOp

ColorConvertOp

# ColorConvertOp

- › Realiza una conversión de espacio de color pixel a pixel
- › Parámetros
  - › Espacio de color destino
- › ¿Cómo se representa el color en un ordenador?
  - › Por un vector en un espacio de color (típicamente 3D) ➔ RGB ~ [255,120,70]
  - › Hay diferentes espacios de color, cada uno con una interpretación particular de los componentes del vector



# ColorConvertOp

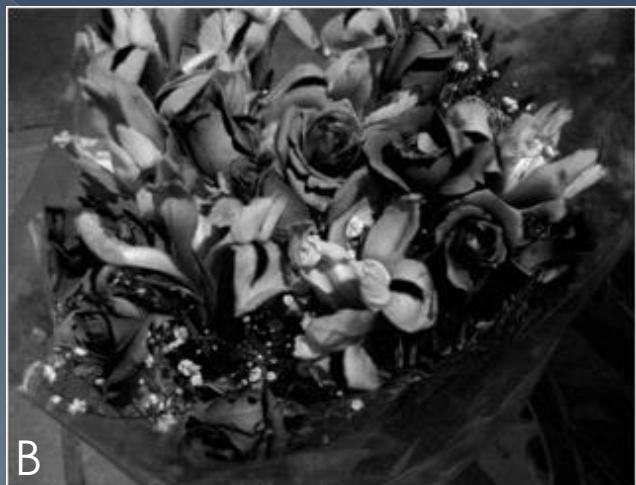
RGB



R



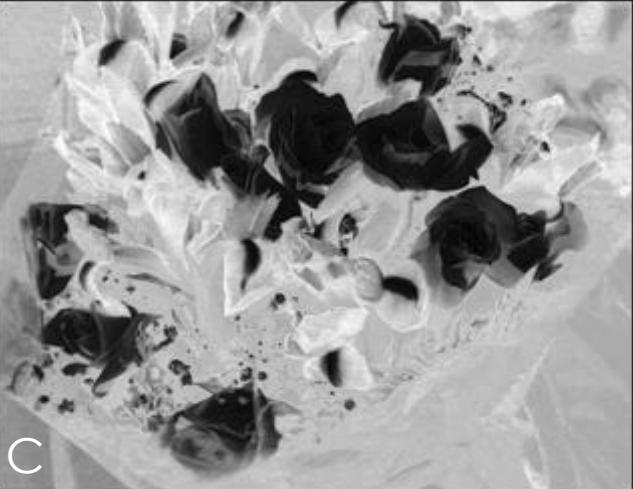
G



B

# ColorConvertOp

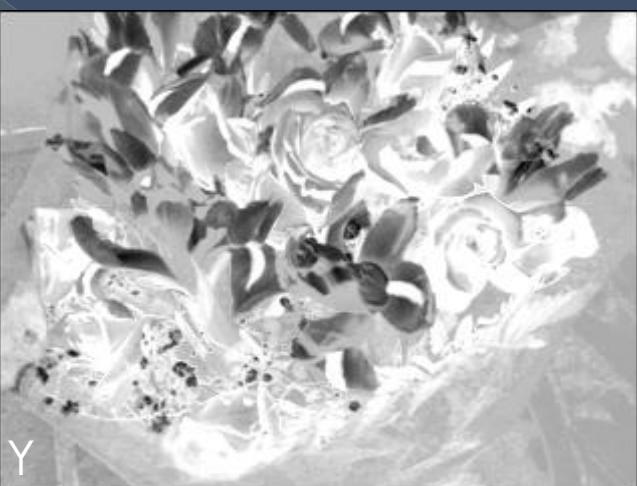
CMY



M



Y



# ColorConvertOp

YCbCr



Y



Cr

Cb

# ColorConvertOp

- › Realiza una conversión de espacio de color pixel a pixel
- › Parámetros
  - › Espacio de color destino
- › ¿Cómo se representa el color en un ordenador?
  - › Por un vector en un espacio de color (típicamente 3D)
  - › Hay diferentes espacios de color, cada uno con una interpretación particular de los componentes del vector



# ColorConvertOp

- › Realiza una conversión de espacio de color pixel a pixel
- › Parámetros
  - › Espacio de color destino
- › Construcción:

- `ColorSpace cs = ColorSpace.getInstance(ColorSpace.CS_GRAY);  
ColorConvertOp op = new ColorConvertOp(cs, ...);`

- `ColorSpace.CS_sRGB`  
`ColorSpace.CS_LINEAR_RGB`
- `ColorSpace.CS_GRAY`
- `ColorSpace.CS_YCC`  
`ColorSpace.CS_CIEXYZ`

```
ICC_Profile ip = ICC_Profile.getInstance(ColorSpace.CS_GRAY);  
ColorSpace cs = new ICC_ColorSpace(ip);
```

# ColorConvertOp



Worth<sup>1000.com</sup>

Worth<sup>1000.com</sup>

# ColorConvertOp

Código ejemplo

```
try{
    ColorSpace cs = ColorSpace.getInstance(ColorSpace.CS_GRAY);
    ColorConvertOp op = new ColorConvertOp(cs,null);
    BufferedImage imgdest = op.filter(img,null);
}catch(Exception e){
    System.err.println("Error");
}
```

# Procesamiento de imágenes

## BufferedImageOp

RescaleOp

ConvolveOp

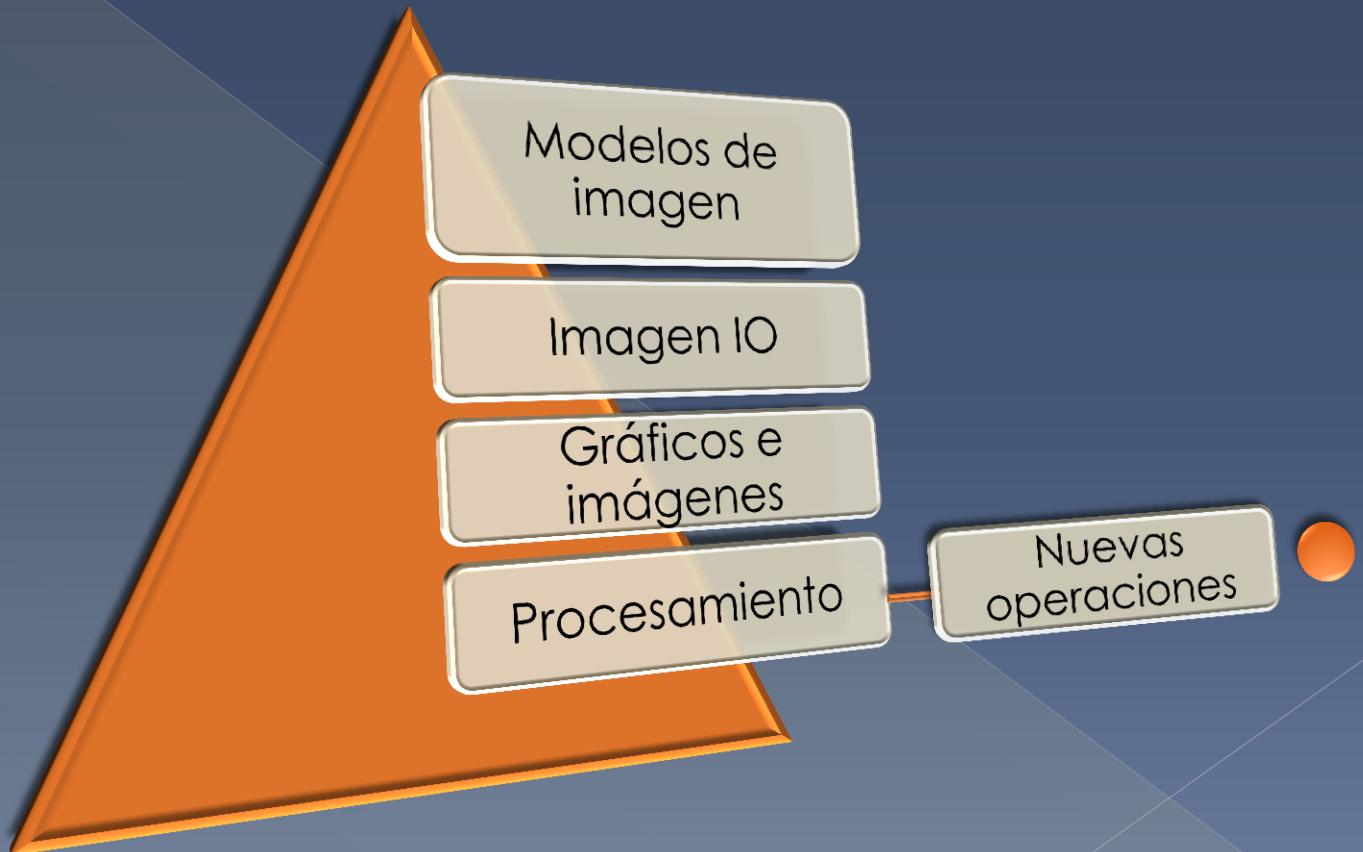
AffineTransformOp

LookupOp

BandCombineOp

ColorConvertOp

# Índice



# Procesamiento de imágenes

## BufferedImageOp

RescaleOp

ConvolveOp

AffineTransformOp

LookupOp

BandCombineOp

ColorConvertOp

Personalizada

:

Personalizada

# Nuevas operaciones

Cómo abordarlo con Java

- Es posible definir nuevas operaciones para el procesamiento de imágenes creando clases que implementen el interfaz *BufferedImageOp*
- Implicará implementar cinco métodos:

**filter**

Aplica la operación

**getBounds**

Bounding box de image dest.

**createCompatibleDestImage**

Crea imagen compatible

**getPoint2D**

Relaciona punto fuente con dest.

**getRenderingHints**

Rendering hints de la operación

# Nuevas operaciones

## Plantilla ejemplo

```
public class ImageOp implements BufferedImageOp {  
  
    public BufferedImage filter(BufferedImage src, BufferedImage dest){  
    }  
  
    public Rectangle2D getBounds2D(BufferedImage src) {  
    }  
  
    public BufferedImage createCompatibleDestImage(BufferedImage src, ColorModel destCM) {  
  
    }  
  
    public Point2D getPoint2D(Point2D srcPt, Point2D dstPt) {  
  
    }  
  
    public RenderingHints getRenderingHints() {  
    }  
}
```

# Nuevas operaciones

## sm.image BufferedImageOpAdapter

```
public class BufferedImageOpAdapter implements BufferedImageOp {  
  
    public BufferedImage filter(BufferedImage src, BufferedImage dest){  
        }  
  
    public Rectangle2D getBounds2D(BufferedImage src) {  
        return src.getRaster().getBounds();  
    }  
  
    public BufferedImage createCompatibleDestImage(BufferedImage src, ColorModel destCM) {  
        if (destCM == null) destCM = src.getColorModel();  
        WritableRaster wr = destCM.createCompatibleWritableRaster(src.getWidth(),src.getHeight());  
        return new BufferedImage(destCM,wr,destCM.isAlphaPremultiplied(), null);  
    }  
  
    public Point2D getPoint2D(Point2D srcPt, Point2D dstPt) {  
        if(dstPt==null) dstPt=(Point2D)srcPt.clone();  
        else dstPt.setLocation(srcPt);  
        return dstPt;  
    }  
  
    public RenderingHints getRenderingHints() {  
        return null;  
    }  
}
```

# Nuevas operaciones

sm.image.BufferedImageOpAdapter

```
public abstract class BufferedImageOpAdapter implements BufferedImageOp {  
  
    public abstract  
  
    public Rectangle getBounds(BufferedImage src){  
        return src.get  
    }  
  
    public BufferedImage createCompatibleDestImage(BufferedImage src, ColorModel destCM){  
        if(destCM == null)  
            destCM = src.getColorModel();  
        if(destCM instanceof WritableRaster)  
            return new BufferedImage(src.getWidth(), src.getHeight(), destCM);  
        else  
            return new BufferedImage(src.getWidth(), src.getHeight(), destCM);  
    }  
  
    public Point2D getCenterPoint(BufferedImage src){  
        Point2D dstPt = new Point2D.Double((src.getWidth() - 1) / 2, (src.getHeight() - 1) / 2);  
        if(dstPt == null)  
            dstPt = new Point2D.Double(0, 0);  
        else dstPt.setLocation((src.getWidth() - 1) / 2, (src.getHeight() - 1) / 2);  
        return dstPt;  
    }  
  
    public RenderingHints getRenderingHints(){  
        return null;  
    }  
  
    public BufferedImage filter(BufferedImage src, BufferedImage dest){  
        return dest;  
    }  
}
```



# Nuevas operaciones

## Iterar sobre la imagen

```
public class ImageOp extends BufferedImageOpAdapter{  
    public BufferedImage filter(BufferedImage src, BufferedImage dest){  
        }  
    }
```

# Nuevas operaciones

Iterar sobre la imagen: componente a componente

```
public class ImageOp extends BufferedImageOpAdapter{  
  
    public BufferedImage filter(BufferedImage src, BufferedImage dest){  
  
        if (dest == null) {  
            dest = createCompatibleDestImage(src, null);  
        }  
        WritableRaster srcRaster = src.getRaster();  
        WritableRaster destRaster = dest.getRaster();  
  
        for (int x = 0; x < srcRaster.getWidth(); x++) {  
            for (int y = 0; y < srcRaster.getHeight(); y++) {  
                for (int band = 0; band < srcRaster.getNumBands(); band++) {  
                    int sample = srcRaster.getSample(x, y, band);  
                    .  
                    .  
                    .  
                    destRaster.setSample(x, y, band, sampleDest);  
                }  
            }  
        }  
        return dest;  
    }  
}
```

# Nuevas operaciones

Iterar sobre la imagen: componente a componente

```
public class ImageOp extends BufferedImageOpAdapter{  
  
    public BufferedImage filter(BufferedImage src, BufferedImage dest){  
  
        if (dest == null) {  
            dest = createCompatibleDestImage(src, null);  
        }  
        WritableRaster srcRaster = src.getRaster();  
        WritableRaster destRaster = dest.getRaster();  
        int sample;  
        for (int x = 0; x < srcRaster.getWidth(); x++) {  
            for (int y = 0; y < srcRaster.getHeight(); y++) {  
                for (int band = 0; band < srcRaster.getNumBands(); band++) {  
                    sample = srcRaster.getSample(x, y, band); // sample definido fuera del bucle  
                    .  
                    .  
                    .  
                    destRaster.setSample(x, y, band, sampleDest);  
                }  
            }  
        }  
        return dest;  
    }  
}
```

# Nuevas operaciones

Iterar sobre la imagen: pixel a pixel

```
public class ImageOp extends BufferedImageOpAdapter{  
  
    public BufferedImage filter(BufferedImage src, BufferedImage dest){  
  
        if (dest == null) {  
            dest = createCompatibleDestImage(src, null);  
        }  
        WritableRaster srcRaster = src.getRaster();  
        WritableRaster destRaster = dest.getRaster();  
  
        for (int x = 0; x < srcRaster.getWidth(); x++) {  
            for(int y = 0; y < srcRaster.getHeight(); y++) {  
                int[] pixelComp=null;  
                pixelComp = srcRaster.getPixel(x, y, pixelComp);  
                .  
                .  
                .  
                destRaster.setPixel(x, y, pixelCompDest);  
            }  
        }  
        return dest;  
    }  
}
```

# Nuevas operaciones

Iterar sobre la imagen: pixel a pixel

```
public class ImageOp extends BufferedImageOpAdapter{  
  
    public BufferedImage filter(BufferedImage src, BufferedImage dest){  
  
        if (dest == null) {  
            dest = createCompatibleDestImage(src, null);  
        }  
        WritableRaster srcRaster = src.getRaster();  
        WritableRaster destRaster = dest.getRaster();  
        int[] pixelComp = new int[srcRaster.getNumBands()];  
        for (int x = 0; x < srcRaster.getWidth(); x++) {  
            for(int y = 0; y < srcRaster.getHeight(); y++) {  
  
                srcRaster.getPixel(x, y, pixelComp); // pixelComp creado fuera del bucle  
                .  
                .  
                .  
                destRaster.setPixel(x, y, pixelCompDest);  
            }  
        }  
        return dest;  
    }  
}
```

# Procesamiento de imágenes

## BufferedImageOp

RescaleOp

ConvolveOp

AffineTransformOp

LookupOp

BandCombineOp

ColorConvertOp

TintadoOp

SepiaOp

# Tintado

sm.image.TintOp

- Para cada píxel  $f(x, y) = [r, g, b]$  de la imagen original, se aplica la transformación:

$$g(x, y) = \alpha \cdot C + (1 - \alpha) f(x, y)$$

con  $g(x, y) = [r, g, b]$  la imagen resultado

- Parámetros

- $\alpha$  : El grado de tintado, cumpliéndose que  $\alpha \in [0,1]$  ( $\alpha = 0$  lo deja igual,  $\alpha = 1$  lo pinta todo de color  $C$ )
- $C$  : El color con el que se tinta la imagen, siendo  $C = [r, g, b]$



# Tintado

## sm.image.TintOp

```
public class TintOp extends BufferedImageOpAdapter{
    public BufferedImage filter(BufferedImage src, BufferedImage dest){
        if (dest == null) {
            dest = createCompatibleDestImage(src, null);
        }
        WritableRaster srcRaster = src.getRaster();
        WritableRaster destRaster = dest.getRaster();
        int sample;

        for (int x = 0; x < srcRaster.getWidth(); x++) {
            for (int y = 0; y < srcRaster.getHeight(); y++) {
                for (int band = 0; band < srcRaster.getNumBands(); band++) {
                    sample = srcRaster.getSample(x, y, band);
                    sample = (int) ( alpha* color[band] + (1.0f - alpha)* sample );
                    destRaster.setSample(x, y, band, sample);
                }
            }
        }
        return dest;
    }

    public TintOp(Color color, float alpha) {
        this.alpha = alpha;
        this.color = color.getColorComponents(null);
        for(int b=0; b< this.color.length; b++) this.color[b] *=255;
    }

    float alpha, color[];
}
```



Incluir los chequeos al inicio para asegurar que la operación se puede aplicar



Se aplica al canal alfa; adaptar si no se quiere así

Se está asumiendo RGB

$$\mathbf{g}(x, y) = \alpha \cdot \mathbf{c} + (1 - \alpha)\mathbf{f}(x, y)$$

# Sepia

sm.image.SepiaOp

- Para cada píxel  $f(x, y) = [r, g, b]$  de la imagen original, se aplica la transformación:

$$sepiaR = \min(255, 0.393 \cdot r + 0.769 \cdot g + 0.189 \cdot b)$$

$$sepiaG = \min(255, 0.349 \cdot r + 0.686 \cdot g + 0.168 \cdot b)$$

$$sepiaB = \min(255, 0.272 \cdot r + 0.534 \cdot g + 0.131 \cdot b)$$

con  $g(x, y) = [sepiaR, sepiaG, sepiaB]$  la imagen resultado

- Sin parámetros



# Sepia

## sm.image.SepiaOp

```
public class SepiaOp extends BufferedImageOpAdapter{  
    public BufferedImage filter(BufferedImage src, BufferedImage dest){  
        if (dest == null) {  
            dest = createCompatibleDestImage(src, null);  
        }  
        WritableRaster srcRaster = src.getRaster();  
        WritableRaster destRaster = dest.getRaster();  
        int[] pixelSrc = new int[srcRaster.getNumBands()];  
        int[] pixelSepia = new int[srcRaster.getNumBands()];  
  
        for (int x = 0; x < srcRaster.getWidth(); x++) {  
            for (int y = 0; y < srcRaster.getHeight(); y++) {  
                srcRaster.getPixel(x, y, pixelSrc);  
                pixelSepia[0] = Math.min( 255, pixelSrc[0]*0.393 + pixelSrc [1]*0.769 + pixelSrc [2]*0.189 );  
                pixelSepia[1] = Math.min( 255, pixelSrc[0]*0.349 + pixelSrc [1]*0.686 + pixelSrc [2]*0.168 );  
                pixelSepia[2] = Math.min( 255, pixelSrc[0]*0.272 + pixelSrc [1]*0.534 + pixelSrc [2]*0.131 );  
                destRaster.setPixel(x, y, pixelSepia);  
            }  
        }  
        return dest;  
    }  
    public SepiaOp(){  
    }  
}
```

Incluir los chequeos al inicio para asegurar que la operación se puede aplicar

No se está considerando el canal alfa (habría que adaptarlo)

Se está asumiendo RGB

$$sepiaR = \min(255, 0.393 \cdot r + 0.769 \cdot g + 0.189 \cdot b)$$
$$sepiaG = \min(255, 0.349 \cdot r + 0.686 \cdot g + 0.168 \cdot b)$$
$$sepiaB = \min(255, 0.272 \cdot r + 0.534 \cdot g + 0.131 \cdot b)$$

# Histograma

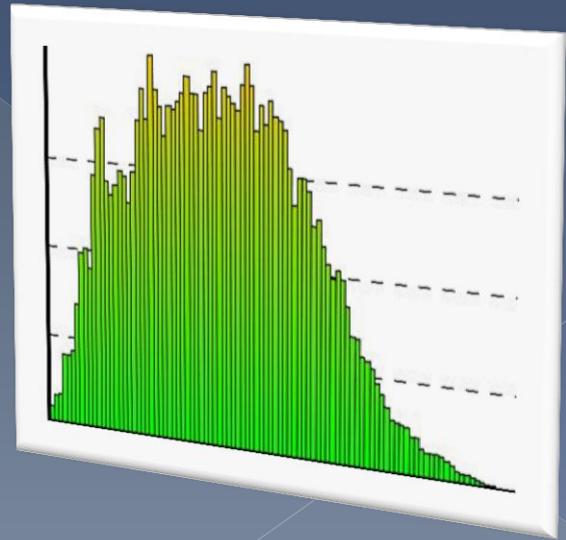
## Definición y propiedades

- El histograma es un conteo del número de veces que un valor aparece en un conjunto.

$$h(i) = n_i \quad i = 0, \dots, N$$

- En el caso de imágenes:

- Si es de niveles de gris, se cuenta cuantas veces aparece cada nivel (para una imagen estándar de 8-bits por muestra, el tamaño del histograma será de 256 bins)
- Si es de color, el conteo se hace por bandas (lo que nos daría un histograma por cada banda) o por colores



# Histograma

Definición y propiedades

- El histograma es un conteo del número de veces que un valor aparece en un conjunto.

$$h(i) = n_i \quad i = 0, \dots, N$$

- El histograma suele normalizarse de forma que la suma de todos sus valores sea 1. Para una imagen de tamaño  $W \times H$  se calcula como:

$$\tilde{h}(i) = h(i)/_{WH} = n_i/_{WH} \quad i = 0, \dots, N$$

$\tilde{h}$  nos da una distribución de probabilidad

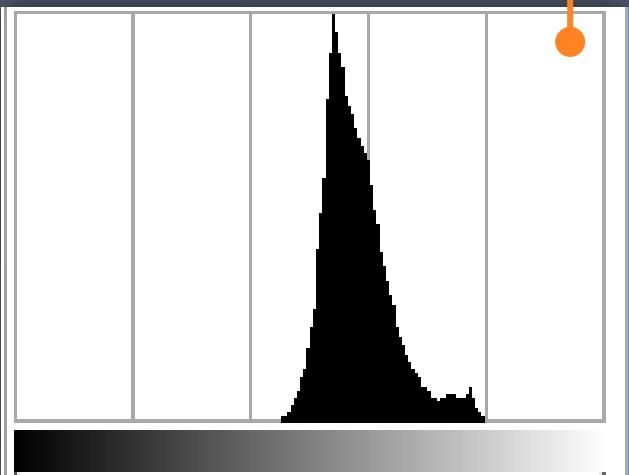
# Histograma

## Definición y propiedades

- La distribución de valores a lo largo del histograma está directamente relacionada con el **contraste** de la imagen:
  - Un histograma con distribuciones “estrechas” es indicativo de una imagen poco contrastada
  - Un histograma “ancho” indica mayor contraste
- Con carácter general, el histograma de una imagen “**subexpuesta**” tendrá una distribución estrecha con un pico desplazado hacia la izquierda (zona oscura), mientras que en una **sobreexposición** el pico estará desplazado a la derecha (zona clara)

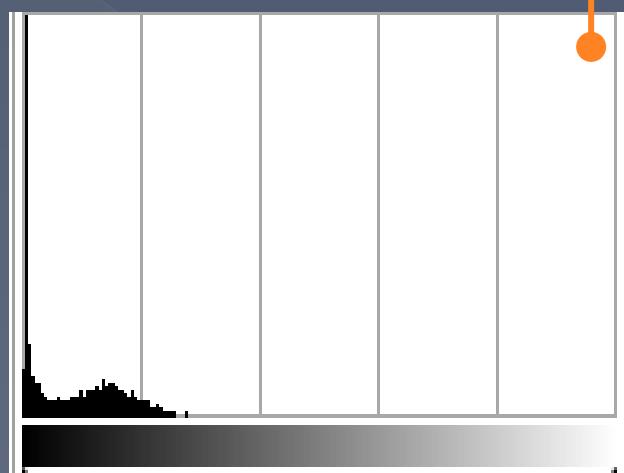
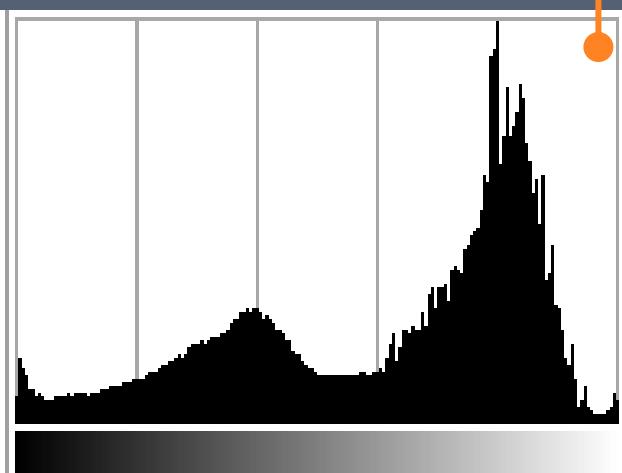
# Histograma

## Ejemplos



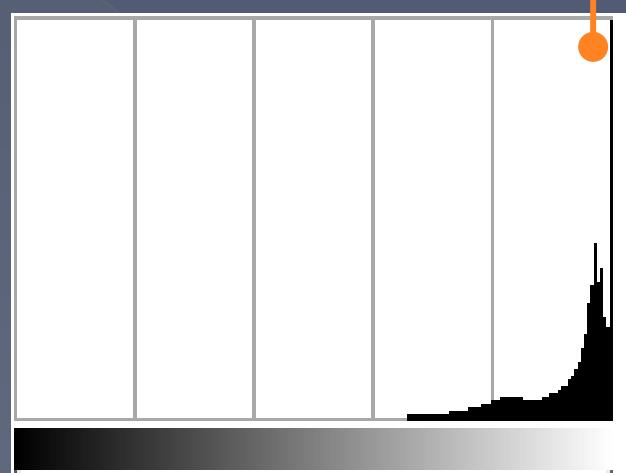
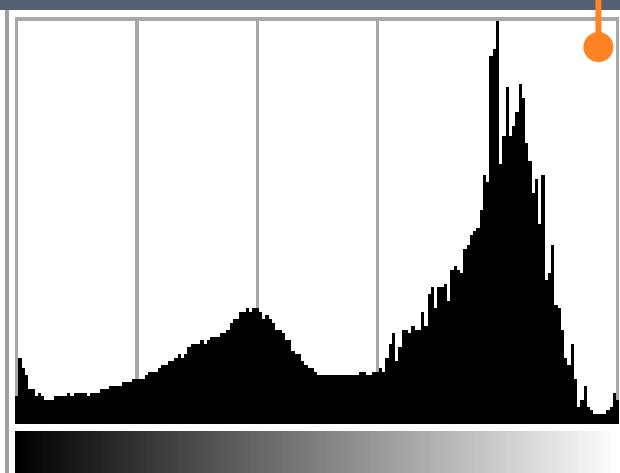
# Histograma

## Ejemplos



# Histograma

## Ejemplos



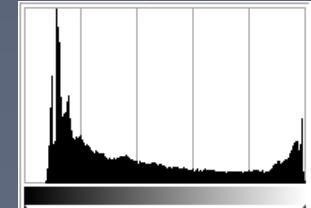
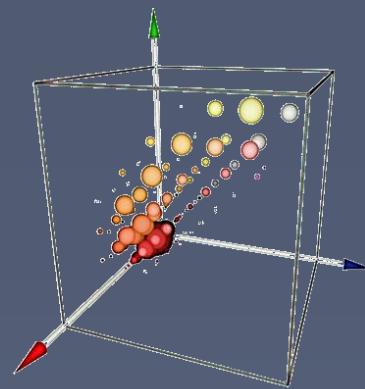
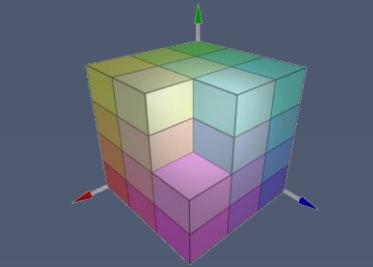
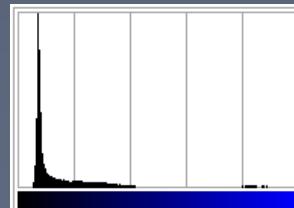
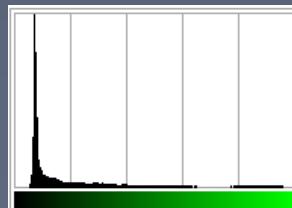
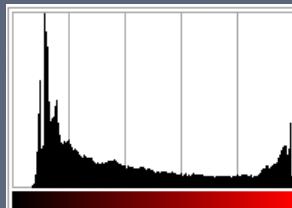
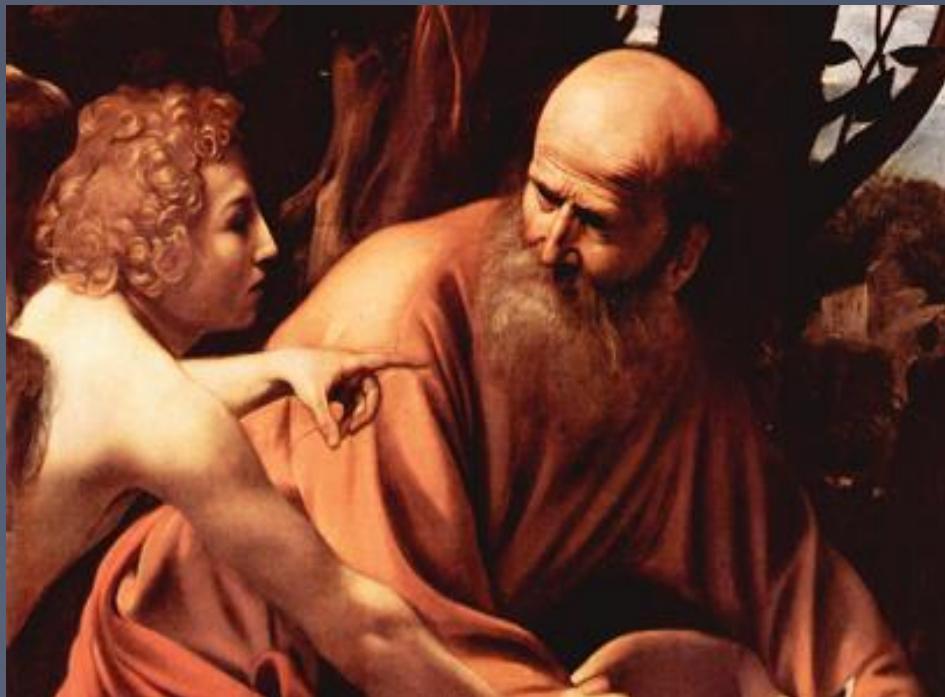
# Histograma

## El caso del color

- Cuando la imagen es en color, se puede:
  - Calcular un histograma por cada banda
  - Calcular el *histograma de color* trabajando sobre el dominio de los colores
    - Un histograma de color tendrá tantas dimensiones como el espacio de color (normalmente, 3D), de forma que cada “bin” estará asociado a un color (3D)
    - En este caso, no es práctico considerar todo el dominio de colores para calcular el histograma; en su lugar se suele optar por soluciones como cuantizar el espacio, esto es, dividirlo en bins 3D cada uno de los cuales representa un rango de colores
  - Calcular el histograma de un “*pseudocanal*” obtenido a partir de los componentes (p.e., la media o el máximo, interpretado como medida de intensidad). No es un canal de color real

# Histograma

El caso del color



# Histograma

`sm.image.Histogram`

- La clase `sm.image.Histogram` representa el histograma de una imagen dada
- Para imágenes en color, calcula el histograma para cada banda

**`Histogram(BufferedImage img)`**

Crea el histograma (uno por banda)

**`getValue(int bin, int band)`**

Devuelve el valor del histograma para un bin y banda dados

**`getNumBins()`**

Devuelve el número de bins del histograma

**`getNumBands()`**

Devuelve el número de bandas.

**`getConts()`**

Devuelve una copia del histograma (conteo)

**`getConts(int band)`**

**`getNormalizedHistogram()`**

Calcula el histograma normalizado

**`getNormalizedHistogram(int band)`**

**`getCumulativeHistogram()`**

Calcula el histograma acumulado (rango [0,1])

**`getCumulativeHistogram(int band)`**

# Ecuación

## Definición

- La ecuación es una transformación que pretende obtener un histograma con una **distribución uniforme** (en el caso ideal, que el conteo de píxeles sea el mismo en cada bin)
  - Trata de obtener un histograma lo más “plano” posible, mejorando con ello el **contraste**



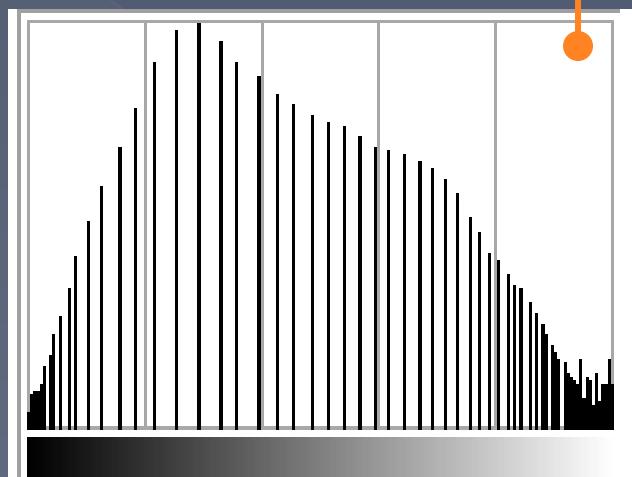
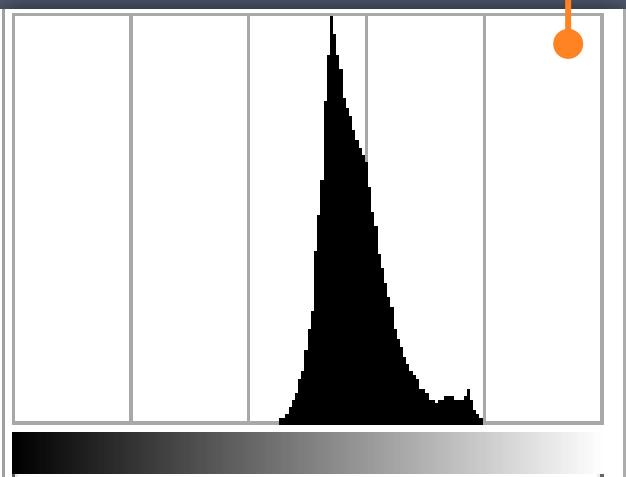
# Ecuación

## Definición

- La ecuación es una transformación que pretende obtener un histograma con una **distribución uniforme** (en el caso ideal, que el conteo de píxeles sea el mismo en cada bin)
  - Trata de obtener un histograma lo más “plano” posible, mejorando con ello el **contraste**
- Concretamente, se aplica la transformación
$$g(x, y) = \tilde{H}(f(x, y))$$
 donde  $\tilde{H}(j) = \sum_{i=0}^j \tilde{h}(i)$ es la función (rescalada) de distribución de probabilidad acumulada

# Ecuación

## Ejemplo



# Ecualización

## El caso del color

- Cuando la imagen es en color:

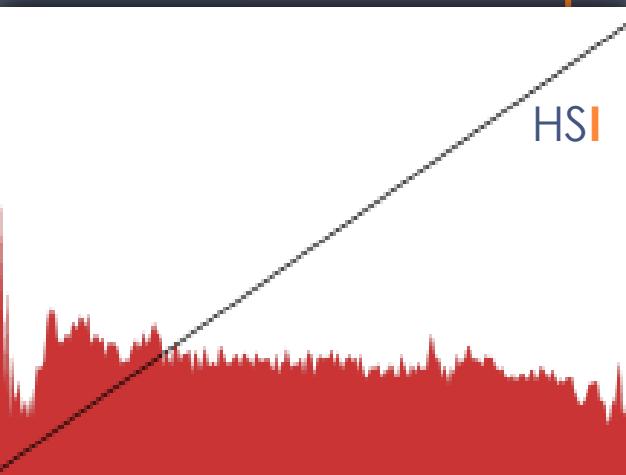
- › Se puede hacer la ecualización **en cada banda**, pero si las distribuciones son muy distintas, puede producir alteraciones en los patrones de color



- › Lo correcto es hacer la ecualización **solo en el canal de intensidad** usando para ello espacios de color que separen la parte cromática de la acromática (YCbCr, YUV, HSV, etc.)

# Ecuación

Ejemplo



# Ecuación

sm.image.EqualizationOp

- La *sm.image.EqualizationOp* ecualiza la imagen por bandas (una o todas)

- > No trabaja sobre el dominio del color
- > Si la imagen está en el espacio de color adecuado, puede usarse para ecualizar solo el canal de intensidad

---

**EqualizationOp(int band)**

Constructor

**filter**

Aplica la operación

**getBounds**

Bounding box de image dest.

**createCompatibleDestImage**

Crea imagen compatible

**getPoint2D**

Relaciona punto fuente con dest.

**getRenderingHints**

Rendering hints de la operación

# JAVA 2D Imaging