EVALUACIÓN FINAL

Alumno: Álvaro Maximino Linares Herrera
DNI: 76669401M
Asignatura: Sistemas Multimedia
Profesor: Jesús Chamorro

Contenidos

1. Requisitos.

1.1. Requisitos funcionales.

En este apartado se verá que hace el programa.

El programa es capaz de reproducir audio, siempre que el formato de audio se “wav” y a su vez será capaz de grabar también en dicho formato. Hay dos tipos de ventanas de audio, la de reproducción con dos botones, uno para reproducir y otro para detenerlo, y la ventana de grabación, también con dos botones, uno para iniciar la grabación y otro para detenerla. Para grabar se necesita previamente un archivo, que se pedirá al usuario, con terminación “.wav”, que en caso de no existir será creado al finalizar la grabación.

El programa es capaz de reproducir vídeo, abrir la webcam y tomar instantáneas del vídeo y de la webcam, pudiendo guardarlas como imagen. Para la reproducción del vídeo es necesario que la terminación del archivo sea “.avi”. La ventana de vídeo mostrará el vídeo y debajo del mismo un pequeño panel de control, en el que se podrá seleccionar el volumen, iniciar y parar el vídeo y una barra de control para saber porque segundo va la reproducción. Si decidimos tomar una instantánea, esta será abierta en una ventana de tratamiento de imágenes, en la cual se podrá aplicar las distintas operaciones de imágenes implementadas en el programa, así como dibujar sobre ella.

En lo referente al tratamiento de imágenes, el programa está dotado con diferentes operaciones, siendo estas las siguientes:

- Duplicar, que crea una imagen nueva a partir de la del lienzo seleccionado. Esta operación es realizada de una forma muy sencilla, ya que lo único que tiene que hacer es saber que ventana interna esta seleccionada y obtener su imagen, crear una nueva ventana interna, meterle la imagen y hacer a esta ventana interna visible.
- Modificar el brillo mediante un deslizador.
- Contraste normal, iluminado y oscurecido, mediante tres botones se podrá cambiar el contraste de la imagen tantas veces como queramos.
- Filtros de emborronamiento enfoque, relieve y fronteras, se aplicarán a la imagen seleccionando la opción que se quiera en el panel de máscaras.
- Negativo, invertir los colores, mediante un botón en la segunda página del panel tabulado de operaciones.
- Transformación a niveles de gris, que hará que la imagen se vea en el llamado popularmente como “blanco y negro”.
- Giro libre mediante deslizador, mediante un deslizador se podrá girar la imagen desde 0 hasta 360º. El programa también esta dotado de tres botones que harán que la imagen gire 90º, 180º y 270º, respectivamente.
- Escalado, manejado por dos botones, uno para aumentar y otro para disminuir. Cabe destacar que si se disminuye la imagen y luego se amplia se puede observar como se pierde calidad de imagen.
- Umbralización, sobre niveles de gris y color, con deslizador para modificar el umbral. El deslizador va acompañado de un “checkbutton”, el cual si está activo hará que la umbralización sea sobre el color principal seleccionado en la barra de dibujo, así pues si el

color seleccionado es el rojo veremos y vamos modificando el deslizador, veremos como los elementos de la imagen que sean de color rojo van poniéndose blancos al principio y luego los demás elementos dependiendo del nivel de rojo que tengan van siendo iluminados también. En caso de que el "checkbutton" no este seleccionado se umbralizará sobre gris.

- Operación de diseño propio, que será descrita a continuación.

Con respecto a la opción de dibujar, se podrá elegir entre cinco figuras:

- Punto.
- Linea.
- Curva con punto de control.
- Rectángulo.
- Elipse.

Estos cinco elementos tienen distintos parámetros que se podrán modificar mediante los distintos elementos del panel de dibujo, tales elementos son el grosor y la continuidad del trazo, el color principal, que será el que defina el color del trazo, el color de relleno, que será el que defina el color de relleno en el caso del rectángulo y la elipse, también hay cuatro "togglebutton" que son los que controlarán el relleno, siendo las opciones:

- Sin relleno.
- Relleno liso, que hará que la figura se rellene con el color de relleno.
- Degradado H, que hará un degradado horizontal en el que se irá del color principal al de relleno.
- Degradado V, que hará un degradado vertical en el que se irá del color principal al de relleno.

También está la posibilidad de un "checkbutton" editar, el cual nos permitirá cambiar todos los parámetros prefijados de la figura seleccionada.

1.2. Requisitos no funcionales.

Realmente el programa tiene como requisito no funcional, el uso de la webcam, siendo preferible el uso de una webcam por usb, ya que si es la integrada del portátil la mayoría de veces no será capaz de abrirla.

Los códecs para reproducción están incluidos en la JMF y la Java Sound API que son incluidos en la carpeta del proyecto de netbeans.

2. Análisis.

2.1. Reconocimiento del problema.

Queremos implementar un programa que sea capaz de trabajar con imágenes, sonido y vídeo. Este programa a de constar de un escritorio multiventana, que pueda trabajar con distintos tipos de ventanas:

- Dibujo e imágenes.
- Reproducción de sonido.
- Grabación de sonido.
- Vídeo.
- Webcam.

2.2. Especificaciones.

Se quiere que en las ventanas de dibujo e imágenes se puedan dibujar distintos tipos de figura, y que estas tengan parámetros y sean independientes, con esto se quiere decir que si se cambia un parámetro y ya hay previamente dibujados otras figuras estas no sean modificadas a menos que las seleccionemos para tal fin.

Se requiere que el programa sea capaz de reproducir vídeos con extensión ".avi", y sonidos con extensión ".wav", pudiendo ser extendido a los códecs de sonido que incorpora JMF.

A la hora de grabar sonido se tendrá que elegir primero el archivo y en caso de que no exista se cree.

También con respecto al trabajo con imágenes se requiere que se puedan realizar como mínimo las operaciones más comunes a la hora de trabajar con imágenes, tales como cambiar el brillo, el contraste, colores negativos... deben de estar disponibles, en el apartado 1.1 Requisitos funcionales se describen más detalladamente que operaciones deben de estar.

3. Diseño.

En este apartado de describirá brevemente como está organizado todo.

Para empezar habrá una barra de menú que contendrá un menú **Archivo** que constará de cuatro elementos, **Nuevo** que abrirá un nuevo lienzo en el que podremos dibujar libremente, **Abrir** que será un menú desplegable en el que se podrá elegir que ventana y con que archivo se abrirá, **Guardar** para guardar las imágenes que hayamos modificado, seleccionando nosotros el nombre y **Abrir cámara** que nos permitirá abrir la cámara.

También habrá en la barra de menú un menú **Edición** que nos permitirá elegir si queremos mostrar

u ocultar los distintos tipos de paneles que tenemos, en el caso de que los paneles de estado y de operaciones sobre imágenes estén ocultos el panel que contiene a ambos se ocultará, dejando más espacio en el escritorio.

Habrá un menú **Imagen** que nos permitirá nos permitirá aplicar unas pocas aplicaciones que no estaban incluidas en los requisitos y que se han implementado.
Por último habrá un menú **Ayuda** en el cual debe de haber una sección **Acerca de...** que abrirá un dialogo en el que se muestre el nombre del autor, la versión del programa y el nombre del programa.

El programa está dividido en tres secciones, parte superior, escritorio y parte inferior.

En la parte superior nos encontramos con dos paneles, el primero un panel general en el cual están las opciones "**Nuevo", "Abrir", "Guardar", "Abrir cámara" y "Tomar instantánea"**. Las cuatro primeras opciones son iguales que la del menú **Archivo**, siendo la última la novedosa, que nos permitirá tomar una instantánea de un vídeo o de la cámara.
El segundo panel de la parte superior es un panel tabulado, con dos partes, la primera de formas, en la que podremos elegir la forma en la que queremos dibujar y la segunda será la parte de los atributos, donde se encontrarán los distintos colores principales, además de dos botones, uno de color principal y otro de relleno que al pulsarlos nos abrirá una paleta en la que podremos elegir colores que no sean solamente los principales y estos dos botones tendrán como color de fondo el color elegido en ese momento, también nos encontraremos botones para elegir el tipo de relleno y también el tipo de trazo y su grosor, así como el botón de editar.

El escritorio es un entorno multiventana que es capaz de contener distintos tipos de ventanas:

- Dibujo e imágenes.
- Reproducción de sonido.
- Grabación de sonido.
- Vídeo.
- Webcam.

Por último nos encontramos con la parte inferior que contiene dos tipos de paneles, uno de estado en el cuál se nos da información del tipo de figura que tenemos seleccionada. El segundo panel es un panel tabulado que contiene todas las distintas operaciones que pueden hacerse sobre las imágenes, aunque no están las operaciones de "Multiplicación" y "Resta" que están alojadas en el menú **Imagen** de la barra de menús.

3.1. Operación propia.

En el programa se debía de incluir una operación propia, pero en este caso se ha optado por incluir dos.

3.1.1. 2/3UmbralizacionOp

Esta operación es muy sencilla y la idea surgió a través del efecto que imprime una aplicación Android, llamada "Pixlr Express", aunque modificándola un poco haciendo uso de la umbralización en lugar del blanco y negro.

En la imagen de la izquierda se puede ver el efecto que hace la aplicación Android, y en la imagen de la derecha se puede ver el efecto que hace el programa. Como se puede ver se hace la umbralización en dos tercios de la imagen, de ahí el nombre de la operación.

Es una operación sencilla en la que se va iterando sobre la imagen, sobre el primer tercio se aplica la umbralización, en el segundo tercio no se aplica nada aunque hay que asignar a la imagen de destino el valor del "sample" de la imagen principal, de lo contrario se verá negro, y en el último tercio se aplica la umbralización de nuevo.

El umbral se le pasa mediante un parámetro, por lo cual se podría modificar la forma de aplicarse de estos momentos, que es mediante un botón y cambiarlo por un "slider" y de esta forma poder umbralizar de una forma mucho más libre.

Otra posible modificación sería pasarle dos umbrales en la creación y así poder modificar de forma distinta los dos tercios de la imagen que hacen uso de la umbralización, pero para eso habría que modificar la clase.

3.1.2. MarilynOp

La idea de esta operación surgió del "PopArt" o Arte Pop, donde una de las imágenes más representativas es la de Marilyn Monroe, en la cual se ve, en la misma pintura, cuatro veces su cara con distintos colores, pensando en esa imagen tuve la idea de coger una imagen hacerla la mitad de ancha y estrecha y repetirla cuatro veces, y en el "filter" de la operación a la hora de asignarle el "sample" a la imagen de destino crear cuatro "samples" para asignárselos a las cuatro "imágenes" y dependiendo de la banda en la que se encuentre el bucle cambiar uno u otro y aplicarselo a una imagen exclusivamente. Se entiende mejor con una imagen de ejemplo.

En la imagen de la izquierda se puede ver un ejemplo de las muchas imágenes de “popart” que se han realizado sobre Marilyn Monroe, y a la derecha el resultado de aplicar la operación “MarilynOp” a una imagen. Como se puede observar el resultado es bastante bueno, dando un estilo parecido al del “popart”.

3.2. Clases de diseño propio de figuras.

Todas las figuras que se pueden crear heredan de una clase abstracta común llamada “FiguraGeometrica”, que a su vez hereda de “Shape” para tener los métodos que tienen todos los shape e incrementar su versatilidad añadiendole parámetros como el color, el color de relleno, si es discontinuo...

3.2.1. Linea

La clase Linea es usada para crear puntos y lineas, tiene un Line2D el cual se usa para devolver lo que devuelven los métodos que FiguraGeometrica tiene que implementar de Shape, tales como getBounds() y demás. Además tiene dos métodos propios de Line2D, que son setLine(Point2D p1, Point2D p2) y ptSegDist(Point2d p1) que serán usados para mover y crear las lineas y puntos que queramos. Más adelante se verá su codificación.

3.2.2. Rectángulo

La clase Rectangulo es usada para crear rectángulos, tiene un Rectangle2D el cual se usa para devolver lo que devuelven los métodos que FiguraGeometrica tiene que implementar de Shape, tales como getBounds() y demás. Además tiene un método propio de Rectangle2D, que es setFrameFromDiagonal(Point2D p1, Point2D p2) que será usado para mover y crear los rectángulos que queramos. Más adelante se verá su codificación.

3.2.3. Elipse

Tiene los mismos métodos que la clase Rectangulo, pero en lugar de Rectangle2D tiene un Ellipse2D.

Cabe destacar que en Shape Rectangle2D y Ellipse2D son considerados RectangularShapes por lo tanto se podrían haber implementado los dos en una sola clase, pero de esta forma lo veo más claro.

3.2.3. Curva

La clase Curva es usada para crear curvas con puntos de control, tiene un QuadCurve2D el cual se usa para devolver lo que devuelven los métodos que FiguraGeometrica tiene que implementar de Shape, tales como getBounds() y demás. Además tiene dos métodos propios de Line2D, que son setCurve(Point2D p1, Point2D p2, Point2D p3) y setCurve(double d, double d1,double d2,double d3,double d4,double d5) que serán usados para mover y crear las curvas que queramos. Más adelante se verá su codificación.

4. Codificación.

En este apartado se verán algunas de las clases de diseño propio por encima, para ampliar información se deberá de ver el código fuente y el javadoc.

4.1. Codificación de FigurasGeometricas.

```
public abstract class FiguraGeometrica implements Shape {

   private Color color = Color.BLACK;
   private Color colorRelleno = Color.BLACK;
   private Stroke stroke = new BasicStroke(1.0f);
   private boolean relleno = false;
   private boolean degradadoH = false;
   private boolean degradadoV = false;
   private boolean discontinuo = false;
   /**
    * Cambia el tipo de color de la figura.
    *@param c objeto de tipo Color.
    */
   public void setColor(Color c){
      this.color=c;
   }

   /**
    *@return Nos devuelve el color que tenemos asignado en la figura.
    */
   public Color getColor(){
      return this.color;
```

```
};
/**
 * Cambia el tipo de color de relleno de la figura.
 *@param c objeto de tipo Color.
 */
public void setColorRelleno(Color c){
    this.colorRelleno=c;
}
/**
 *
 *@return Nos devuelve el color de relleno que tenemos asignado en la figura.
 */
public Color getColorRelleno(){
    return this.colorRelleno;
};
/**
 * Cambia el tipo de trazo de la figura.
 *@param str objeto de tipo float.
 */
public void setStroke(float str){
    this.stroke=new BasicStroke(str);
}
/**
 *
 *@return Nos devuelve el tipo de Stroke que tenemos asignado en la figura.
 */
public Stroke getStroke(){
    return stroke;
}
/**
 * Cambia el tipo de trazo de la figura.
 *@param str objeto de tipo Stroke.
 */
public void setStroke(Stroke str){
    stroke=str;
}
/**
 * Modifica el parametro de relleno de la figura.
 *@param r objeto de tipo boolean.
 */
public void setRelleno(boolean r){
```

```
        this.relleno=r;
    }
    /**
     *
     *@return Nos devuelve un tipo boolean que nos indica si tenemos asignado el relleno de la figura.
     */
    public boolean getRelleno(){
        return relleno;
    }
    /**
     * Modifica el parametro de relleno degradado horizontal de la figura.
     *@param d objeto de tipo boolean.
     */
    public void setDegradadoH(boolean d){
        this.degradadoH=d;
    }
    /**
     *
     *@return Nos devuelve un tipo boolean que nos indica si tenemos asignado el relleno con degradado horizontal de la
figura.
     */
    public boolean getDegradadoH(){
        return degradadoH;
    }
    /**
     * Modifica el parametro de relleno degradado vertical de la figura.
     *@param d objeto de tipo boolean.
     */
    public void setDegradadoV(boolean d){
        this.degradadoV=d;
    }
    /**
     *
     *@return Nos devuelve un tipo boolean que nos indica si tenemos asignado el relleno con degradado vertical de la
figura.
     */
    public boolean getDegradadoV(){
        return degradadoV;
    }
    /**
     * Nos devuelve la forma a la cual pertenece la figura.
     * @return int, entre 0 y 4
```

```
     */
    public abstract int getForma();

    public void setDiscontinuo(boolean d){
        this.discontinuo=d;
    }
}
```

Como se puede ver los parámetros de dibujo que tiene cada figura son los parámetros que se pueden cambiar mediante el panel de dibujo, haciendo así que cada figura sea independiente, en la clase Lienzo también se encuentran estos parámetros pero esos parámetros son los relativos a los seleccionados de cada lienzo, no de cada figura.

Es una clase muy sencilla y con la cual obtendremos un mayor control sobre todo lo que podemos hacer con las figuras, ampliando así el uso que se le pueden dar a los shapes.

Los métodos no necesitan más comentarios, pues están comentados y se pueden ver en el javadoc con mayor comodidad.

4.2. Codificación de Linea.

```
public class Linea extends FiguraGeometrica {
    private Line2D fig;
    /**
     * Constructor
     * @param p1 Point2D, primer punto de la Linea.
     * @param p2 Point2D, segundo punto de la Linea.
     */
    public Linea(Point2D p1, Point2D p2){
            fig = new Line2D.Float(p1, p2);
    }

    /**
     * Returns an integer Rectangle that completely encloses the Shape.
     * @return an integer Rectangle that completely encloses the Shape.
     */
    @Override
    public Rectangle getBounds() {
        return fig.getBounds();

    }

    /**
     * Returns a high precision and more accurate bounding box of the Shape than the getBounds method.
```

```
     * @return an instance of Rectangle2D that is a high-precision bounding box of the Shape.
     */
    @Override
    public Rectangle2D getBounds2D() {
        return fig.getBounds2D();
    }

    /**
     *
     * @param d the X coordinate of the specified point to be tested
     * @param d1 the Y coordinate of the specified point to be tested
     * @return false because a Line2D contains no area.
     */
    @Override
    public boolean contains(double d, double d1) {
        return fig.contains(d, d1);
    }

    /**
     *
     * @param pd the specified Point2D to be tested
     * @return false because a Line2D contains no area.
     */
    @Override
    public boolean contains(Point2D pd) {
        return fig.contains(pd);
    }

    /**
     *
     * @param d the X coordinate of the upper-left corner of the specified rectangular area
     * @param d1 the Y coordinate of the upper-left corner of the specified rectangular area
     * @param d2 the width of the specified rectangular area
     * @param d3 the height of the specified rectangular area
     * @return true if the interior of the Shape and the interior of the rectangular area intersect, or are both highly likely
to intersect and intersection calculations would be too expensive to perform; false otherwise.
     */
    @Override
    public boolean intersects(double d, double d1, double d2, double d3) {
        return fig.intersects(d, d1, d2, d3);
    }
```

```
    /**
     *
     * @param rd the specified Rectangle2D
     * @return true if the interior of the Shape and the interior of the specified Rectangle2D intersect, or are both highly likely to intersect and intersection calculations would be too expensive to perform; false otherwise.
     */
    @Override
    public boolean intersects(Rectangle2D rd) {
        return fig.intersects(rd);
    }

    /**
     *
     * @param d the X coordinate of the upper-left corner of the specified rectangular area
     * @param d1 the Y coordinate of the upper-left corner of the specified rectangular area
     * @param d2 the width of the specified rectangular area
     * @param d3 the height of the specified rectangular area
     * @return false because a Line2D contains no area.
     */
    @Override
    public boolean contains(double d, double d1, double d2, double d3) {
        return fig.contains(d, d1, d2, d3);
    }

    /**
     *
     * @param rd the specified Rectangle2D to be tested
     * @return false because a Line2D contains no area.
     */
    @Override
    public boolean contains(Rectangle2D rd) {
        return fig.contains(rd);
    }

    /**
     * Returns an iteration object that defines the boundary of this Line2D. The iterator for this class is not multi-threaded safe, which means that this Line2D class does not guarantee that modifications to the geometry of this Line2D object do not affect any iterations of that geometry that are already in process.
     * @param at the specified AffineTransform
     * @return a PathIterator that defines the boundary of this Line2D.
     */
    @Override
```

```
    public PathIterator getPathIterator(AffineTransform at) {
        return fig.getPathIterator(at);
    }

    /**
     *
     * @param at the specified AffineTransform
     * @param d the maximum amount that the control points for a given curve can vary from colinear before a subdivided curve is replaced by a straight line connecting the end points. Since a Line2D object is always flat, this parameter is ignored.
     * @return a PathIterator that defines the boundary of the flattened Line2D
     */
    @Override
    public PathIterator getPathIterator(AffineTransform at, double d) {
        return fig.getPathIterator(at);
    }

    /**
     * Sets the location of the end points of this Line2D to the specified Point2D coordinates.
     * @param p1 the start Point2D of the line segment
     * @param p2 the end Point2D of the line segment
     */
    public void setLine(Point2D p1, Point2D p2){
        fig.setLine(p1, p2);
    }

    /**
     * Returns the square of the distance from a Point2D to this line segment. The distance measured is the distance between the specified point and the closest point between the current line's end points. If the specified point intersects the line segment in between the end points, this method returns 0.0.
     * @param p1 the specified Point2D being measured against this line segment.
     * @return a double value that is the square of the distance from the specified Point2D to the current line segment.
     */
    public double ptSegDist(Point2D p1){
        return fig.ptSegDist(p1);
    }

    /**
     *
     * @return the X coordinate of the start point of this Line2D object.
     */
    public double getX1(){
        return fig.getX1();
```

```
    }
    /**
     *
     * @return the X coordinate of the end point of this Line2D object.
     */
    public double getX2(){
        return fig.getX2();
    }
    /**
     *
     * @return the Y coordinate of the start point of this Line2D object.
     */
    public double getY1(){
        return fig.getY1();
    }
    /**
     *
     * @return the Y coordinate of the end point of this Line2D object.
     */
    public double getY2(){
        return fig.getY2();
    }
    /**
     *
     * @return int con valor 1, que es el número asignado a la forma linea.
     */
    @Override
    public int getForma() {
      return 1;
    }
}
```

Como se puede ver dentro de las clases propias de las formas no hay ningún parámetro nuevo, por lo que todos serán controlados por la superclase FigurasGeometricas, siendo así mucho más cómodo a la hora de meter en Lienzo las figuras en un vector, pues no tendremos que hacer castings, sino que podremos meter las figuras directamente cada una con sus parámetros distintos.

Nota: Los comentarios de los métodos de esta clase Linea, y de las otras tres clases de formas están en inglés, pues han sido cogidos de la API de Java, de esta forma no se pierden matices al traducir. Las otras tres clases son como esta, por lo cual no las comentaré en esta documentación, pues van comentadas en el javadoc.

4.3. Codificación de 2/3UmbralizacionOp.

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

   /**
    * Constructor, al cual se le pasará mediante el parámetro el grado de umbralización que queremos.
    * @param umb int
    */
   public DosTerciosUmbralizacionOp(int umb){
      umbral = umb;
   }

   /**
    * Aplica la umbralización a dos tercios de la imagen, de forma vertical, se quedaría pues con la primera parte
    * la imagen umbralizada, la parte del medio normal, y la ultima parte quedaría umbralizada.
    * @param src BufferedImage, de donde se coje la imagen y sus valores.
    * @param dest BufferedImage, imagen de destino, donde se guardará la imagen ya cambiada.
    * @return BufferedImage con la umbralización ya realizada.
    */
   @Override
   public BufferedImage filter(BufferedImage src, BufferedImage dest) {
      if (dest == null) {
         dest = createCompatibleDestImage(src, null);
      }

      WritableRaster srcRaster = src.getRaster();
      WritableRaster destRaster = dest.getRaster();
      int x = srcRaster.getWidth();
      for (int row = 0; row < srcRaster.getHeight(); row++) {
         for (int col = 0; col < (x/3); col++) {
            for (int band = 0; band < srcRaster.getNumBands(); band++) {
               int sample = srcRaster.getSample(col, row, band);
               if(sample>=umbral){
                  sample=255;
               }else{
                  sample=0;
               }
               destRaster.setSample(col, row, band, sample);
            }
         }
         for (int col = x/3; col < ((2*x)/3); col++) {
```

```
            for (int band = 0; band < srcRaster.getNumBands(); band++) {
                int sample = srcRaster.getSample(col, row, band);
                destRaster.setSample(col, row, band, sample);
            }
        }
        for (int col = ((2*x)/3); col < x; col++) {
            for (int band = 0; band < srcRaster.getNumBands(); band++) {
                int sample = srcRaster.getSample(col, row, band);
                if(sample>=umbral){
                    sample=255;
                }else{
                    sample=0;
                }
                destRaster.setSample(col, row, band, sample);
            }
        }
    }
    return dest;
  }
}
```

Como se comento anteriormente lo único que hace esta operación es redefinir filter, hace dos operaciones dependiendo de si estamos en el primer y último tercio de la imagen o si estamos en el medio. En el primer caso, de 0 a x/3 y de (2*x)/3 a x cambiamos el sample que le asignamos a la imagen de destino, en el segundo caso simplemente asignamos el sample de la imagen fuente a la imagen destino, sin cambiar nada, de esta forma obtendremos un resultado parecido a este:

4.4. Codificación de MarilynOp.

```
public class MarilynOp extends BufferedImageOpAdapter{
   /**
    * Constructor
    */
   public MarilynOp(){


   }


   /**
    * Crea una imagen que sea la mitad de alta y la mitad de ancha que la original, y sobre la de destino la va
    * copiando mediante la asignación del sample y la modificación del mismo dependiendo de en que banda estemos operando.
    * @param src BufferedImage, de donde se coje la imagen original y sus valores.
    * @param dest BufferedImage, donde se van almacenando los resultados.
    * @return
    */
   @Override
   public BufferedImage filter(BufferedImage src, BufferedImage dest) {
      if (dest == null) {
         dest = createCompatibleDestImage(src, null);
      }
      //Creo lienzoImagen y le asigno la imagen fuente para cambiarle el tamaño.
      BufferedImage img = src;


      VentanaInterna vi = new VentanaInterna();
      vi.getLienzo().setImagen(img, img.getWidth()/2, img.getHeight()/2);
      src = vi.getLienzo().getImagen();


      WritableRaster srcRaster = src.getRaster();
      WritableRaster destRaster = dest.getRaster();



      int x = srcRaster.getWidth();
      int y = srcRaster.getHeight();


      for (int row = 0; row < y; row++) {
         for (int col = 0; col < (x/2); col++) {
            for (int band = 0; band < srcRaster.getNumBands(); band++) {
               int sample = srcRaster.getSample(col,row,band);
               int sample2=sample, sample3=sample, sample4=sample;
```

```
        if(band==0){
           sample/=255;
           sample2/=255;
        }
        if(band==1){
           sample4/=255;
        }
        if(band==2){
           sample3/=255;
           sample2/=255;
        }

        destRaster.setSample(col, row, band, sample);
        destRaster.setSample(col+(x), row, band, sample2);
        destRaster.setSample(col, row+(y), band, sample3);
        destRaster.setSample(col+(x), row+(y), band, sample4);
    }
}
for (int col = x/2; col < x; col++) {
    for (int band = 0; band < srcRaster.getNumBands(); band++) {
        int sample = srcRaster.getSample(col,row,band);
        int sample2=sample, sample3=sample, sample4=sample;

        if(band==0){
           sample/=255;
           sample2/=255;
        }
        if(band==1){
           sample4/=255;
        }
        if(band==2){
           sample3/=255;
           sample2/=255;
        }
        destRaster.setSample(col, row, band, sample);
        destRaster.setSample(col+(x), row, band, sample2);
        destRaster.setSample(col, row+(y), band, sample3);
        destRaster.setSample(col+(x), row+(y), band, sample4);
    }
}
```

```
        }
        return dest;
    }
}
```

Esta operación da un estilo “popart” a la imagen a la que se le aplique, lo primero que hacemos es convertir la imagen fuente a una imagen la mitad de ancha y de alta que la original, después en los bucles vamos recorriendo la imagen fuente y le vamos asignando distintos samples a la imagen destino, concretamente cuatro distintos, en los que se modificara el sample dependiendo de la banda en la que estemos, consiguiendo así colores distintos, en las cuatro imágenes más pequeñas. El resultado es una imagen bastante “curiosa”.

En la parte superior vemos las imágenes originales, y en la parte inferior vemos las imágenes con la operación aplicada.

Como se puede observar es un efecto bastante llamativo.

5. Bibliografía.

La gran parte de la documentación ha sido sacada de la API de Java, el resto de los guiones de prácticas de la asignatura y de las transparencias de teoria.

http://docs.oracle.com/javase/7/docs/api/javax/sound/sampled/LineListener.html

http://docs.oracle.com/javase/7/docs/api/java/awt/geom/Line2D.html

http://docs.oracle.com/javase/7/docs/api/java/awt/geom/Rectangle2D.html

http://docs.oracle.com/javase/7/docs/api/java/awt/geom/Ellipse2D.html

http://docs.oracle.com/javase/7/docs/api/java/awt/geom/QuadCurve2D.html