UNIVERSIDAD DE GRANADA

Sistemas Multimedia (2022-2023)
Grado en Ingeniería Informática

Documentación Práctica Final

Autor:

Mario Guisado García

Índice

1. Objetivo de la práctica

El objetivo es desarrollar una aplicación multimedia, usando el entorno de desarrollo de Apache NetBeans, y programado en Java, que ofrezca las herramientas necesarias para tareas básicas e intermedias de dibujo, procesado de imágenes, audio y vídeo.

2. Requisitos de la aplicación

A continuación, se definen las distintas posibilidades que ofrece la aplicación, divididas en los distintos bloques principales de funcionalidades.

Gráficos

La aplicación permitirá el dibujado de distintas formas y figuras, entre ellas:

1. Línea.
2. Trazo libre.
3. Curva de dos puntos.
4. Rectángulo.
5. Elipse.
6. Cara sonriente.

Asociados a estas formas, encontramos una serie de atributos:

- ➢ Color, a elegir entre distintos disponibles mediante un diálogo.
- ➢ Relleno de la figura.
- ➢ Transparencia, con un deslizador asociado para elegir el grado.
- ➢ Antialiasing.
- ➢ Grosor del trazo.
- ➢ Contorno del propio trazo, a elegir entre líneas contínuas, punteadas, o un patrón de punteado personalizado.

Cabe destacar que todas estas opciones de dibujado será posible llevarlas a cabo sobre una ventana nueva, o sobre una imagen previamente abierta. Además, se nos proporcionará la opción de volcar sobre la imagen las figuras dibujadas a nuestra elección.

Imagen

El procesado de imágenes será una parte importante de la totalidad de las funcionalidades ofrecidas por la aplicación. Las opciones en cuestión serán las siguientes:

1. Botón de duplicado de imagen.
2. Modificación del brillo y contraste mediante deslizadores.
3. Filtros de emborronamiento -de distintos tamaños-, así como enfoque y relieve.
4. Alteración de contrastes, en distintas formas (normal, iluminado u oscurecido).
5. Opción para invertir los colores de la imagen.
6. Extracción de bandas y conversión a espacios RGB, YCC y GRAY.
7. Aumentos y disminuciones sobre la imagen, así como rotaciones fijas o libres.
8. Tintados y filtro sepia.
9. Ecualización y posterización de la imagen.
10. Resaltado del rojo, así como permitir la variación del tono de los colores.
11. Además, se incluirán diversas operaciones basadas en las funciones spline lineal, cuadrática, un operador tipo LookUp propio y una operación pixel a pixel propias, todas ellas con parámetros libremente modificables por el usuario.

Sonido

También se ofrecen una serie de posibilidades para trabajar con archivos de sonido. En concreto, se deberá:

1. Reproducir y pausar archivos de audio, pudiendo ser elegidos y añadidos a una lista de ficheros.
2. Grabar un audio propio, pudiendo almacenarlo en el sistema y realizar lo visto en el punto anterior.

Vídeo

En cuanto a las opciones de vídeo:

1. Se podrá reproducir y pausar archivos de vídeo almacenados en nuestro sistema, de la misma forma que se hacía con los audios.
2. Podremos mostrar la imagen proporcionada por la webcam en tiempo real.
3. Se podrá realizar una captura de pantalla a la imagen ofrecida por la cámara, y se tratará como una imagen sobre la que podemos realizar las mismas operaciones que se realizan sobre imágenes estáticas.

Interfaz

Para que el usuario pueda interactuar adecuadamente con la aplicación, se proporcionará una interfaz gráfica en la que se mostrarán una serie de botones, sliders y listas desde las que realizar todas las operaciones. Esta interfaz será separada en partes claramente distinguibles y se denotará a través de un “tooltip text” el objetivo de cada herramienta.

3. Análisis

Vistos los requisitos, vamos a ver las estructuras y clases de las que disponemos para llevar a cabo la aplicación. Si bien en un principio, pareciese que java proporciona la mayoría de clases necesarias para implementarla, descubrimos que gran parte de los requisitos de la misma conllevan necesariamente la creación de clases, atributos y métodos propios para poder cumplirlos en su totalidad y con plena funcionalidad.

Gráficos

Para resolver la parte de los gráficos en Java disponemos de la biblioteca Shape, la cual proporciona toda una serie de objetos que heredan de esta clase y que permiten instanciar diversas figuras entre las que encontramos rectángulos, elipses o curvas.

Si bien estas clases son útiles en un contexto básico, más aún teniendo en cuenta que se proporcionan junto a métodos específicos a cada una para realizar distintas operaciones, no nos será suficiente para el caso práctico en el que nos encontramos. En este contexto, las figuras dibujadas deben tener una serie de atributos asociados a cada una de ellas. Además, para las distintas operaciones que queremos efectuar debemos definir métodos concretos a cada una. Por último, cabe mencionar que incluiremos figuras que no se encuentran en esta biblioteca, sino que son completamente nuevas, como es el caso de la figura de la cara.

Imagen

Para las operaciones sobre imágenes, Java cuenta con bibliotecas como la que nos proporciona BufferedImageOp, con métodos para realizar distintos tipos de transformaciones (LookupOp, BandCombineOp, ConvolveOp…). Se usarán estas clases como base para implementar el resto de funcionalidades que requiere la aplicación.

Sonido

En el caso del sonido, en Java contamos con clases como SMPlayer, con la que podremos asociar un reproductor, que nos proporcionará los métodos necesarios para reproducir y pausar audio de un fichero previamente seleccionado.

De la misma manera, contamos con otra clase, SMRecorder, para las tareas relacionadas con la grabación de audio.

Vídeo

Para las funcionalidades relacionadas con la captura de la webcam, aunque se cuenta con un entorno oficial para llevarlas a cabo, JMF, este se encuentra desactualizado y fuera de uso. Por ello, en esta práctica se ha optado por Webcam Capture API, la cual ofrece todas las herramientas necesarias, así como soporte a códecs modernos.

Interfaz

Para facilitar la interacción del usuario con la aplicación, Apache NetBeans proporciona una sencilla interfaz gráfica mediante la cual podemos añadir elementos como botones, diálogos o paneles a nuestra aplicación. Con ellos, implementaremos toda la parte gráfica o visual, con la que el usuario interactúa directamente. Sin embargo, para que la aplicación funcione correctamente, se deberán implementar clases y métodos adicionales que servirán de intermediario entre el usuario y las acciones tomadas, y las clases importadas y creadas.

4. Diseño

Analizada la aplicación, y sabiendo las clases y bibliotecas con las que trabajaremos, se comentarán las opciones por las que se han optado para llevar a cabo la aplicación, así como el aspecto final de la misma.

Gráficos

Para resolver la falta de funcionalidad ofrecida por la clase Shape, se optará por definir una interfaz (“figuras”) para definir los métodos propios necesitados, y se crearán clases nuevas para cada una de las figuras, que implementarán dicha interfaz. De esta manera solucionaremos también el problema de almacenar los atributos asociados a cada figura, almacenándolos como variables de clase.

Cada una de estas clases extenderá la clase Shape correspondiente.

Concretamente se tendrá:

- Cara, que extiende Area.
- Curva, que extiende QuadCurve2D.
- Elipse, que extiende Ellipse2D.
- Línea, que extiende Line2D.
- Rectangulo, que extiende Rectangle2D.
- Trazo, que extiende Path2D.

```
/**
 * Clase que define la curva. Extiende QuadCurve2D.
 * @author Mario Guisado García
 */
public class curva extends QuadCurve2D.Double implements figuras{
```

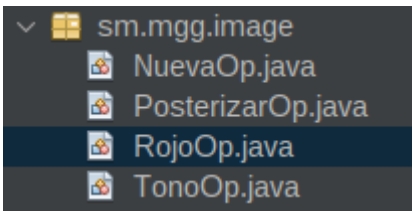
Las clases propias extenderán objetos Shape e implementarán la interfaz “figuras”.

Imagen

En el caso de las operaciones con imágenes, trabajaremos con clases que heredan de BufferedImageOpAdapter, y en las que definiremos parámetros auxiliares y métodos propios empleados para realizar las transformaciones necesarias.

A la hora de trabajar con las imágenes del sistema, se nos permitirá abrir o guardar ficheros correspondientes a un filtro previamente establecido, y asociar lo leído a través de diálogos -usando la clase JFileChooser-, a objetos BufferedImage, que más tarde podremos asociar y plasmar a las ventanas internas gracias a métodos específicos, contenidos en clases como “Lienzo”, que hará de intermediaria con el usuario y nuestras clases propias.

Concretamente, se han creado cuatro nuevas clases, con las que implementaremos las funciones vistas en las prácticas PosterizarOp, RojoOp y TonoOp, así como la operación pixel a pixel de diseño propio, NuevaOp.



Sonido

Asociado a los objetos de las clases SMPlayer y SMRecorder, que serán las empleadas para llevar a cabo este apartado, crearemos nuestros propios manejadores para gestionar los eventos generados, como el inicio o fin de la reproducción de archivos de audio. A través de ellos manejaremos de forma simple todos los eventos relacionados con la reproducción y grabación de audio. En este aspecto, Java nos brinda clases muy completas que nos facilitará enormemente el trabajo, y no será necesario la definición de clases propias.

Vídeo

Usando Webcam Capture API crearemos nuestra propia clase, que almacenará variables de dicha biblioteca (Webcam), con algunos métodos para definirlas correctamente y añadir lo capturado por la webcam al espacio de trabajo.

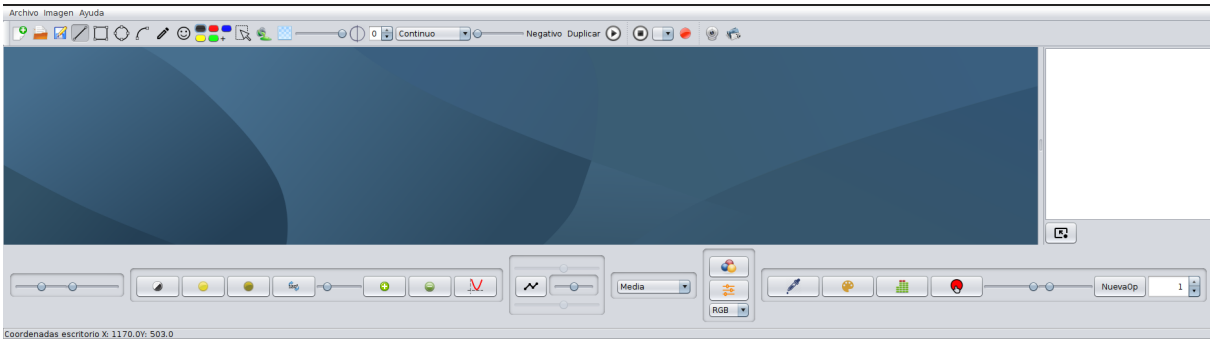
En cuanto a la reproducción de vídeo,usaremos de nuevo, una clase propia, con variables definidas en la biblioteca, en este caso de tipo EmbeddedMediaPlayer, con el que podremos emplear los distintos métodos asociados y definidos en el paquete. Inicializamos los distintos objetos con los que trabajaremos en el constructor (que recibirá el archivo a visualizar) y crearemos métodos para reproducir o parar el vídeo en cuestión y para obtener una instancia de la clase.

```
public class VentanaInternaVideo extends javax.swing.JInternalFrame {

    private EmbeddedMediaPlayer vlcplayer = null;
    private File fMedia;

    private VentanaInternaVideo(File f) {
        initComponents();
        fMedia = f;
        EmbeddedMediaPlayerComponent aVisual = new EmbeddedMediaPlayerComponent();
        getContentPane().add(comp:aVisual, constraints: java.awt.BorderLayout.CENTER);
        vlcplayer = aVisual.getMediaPlayer();
    }

    public static VentanaInternaVideo getInstance(File f) {
        VentanaInternaVideo v = new VentanaInternaVideo(f);
        return (v.vlcplayer != null ? v : null);
    }

    public void play() {
        if (vlcplayer != null) {
            if (vlcplayer.isPlayable()) {
                //Si se estaba reproduciendo
                vlcplayer.play();
            } else {
                vlcplayer.playMedia(string: fMedia.getAbsolutePath());
            }
        }
    }
```

Parte de la clase para la reproducción de vídeo. Haremos uso de un objeto tipo EmbeddedMediaPlayer para aprovechar las funcionalidades de su clase.

Interfaz

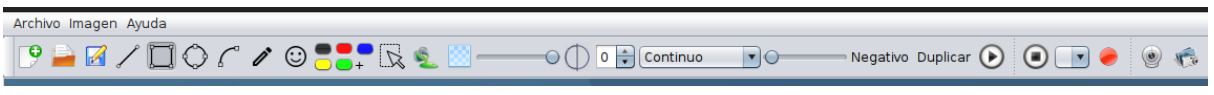
Por último, se definirán varias clases que heredarán de elementos swing de java, concretamente JInternalFrame y JFrame, lo cual nos facilitará la tarea de mostrar al usuario lo realizado y nos permitirá definir variables y métodos adicionales para controlar la interacción de las distintas clases definidas, así como lo introducido por el usuario a través de los distintos botones y herramientas proporcionados por la aplicación. Concretamente, definiremos una clase VentanaInterna para las imágenes, una VentanaInternaCamara para la webcam, y otra VentanaInternaVideo para la reproducción de vídeo. Además, se contará con una clase que realizará la función de ventana principal, mostrando la aplicación al usuario.



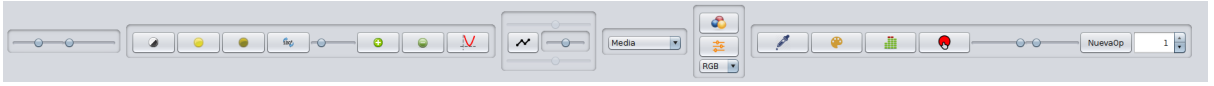
Apariencia inicial de la aplicación.

Además, definiremos una clase Lienzo, que hará de intermediaria entre el usuario y lo mostrado en pantalla, y las clases necesarias para llevar a cabo las distintas operaciones, como las pertenecientes a las figuras y transformaciones de imágenes.

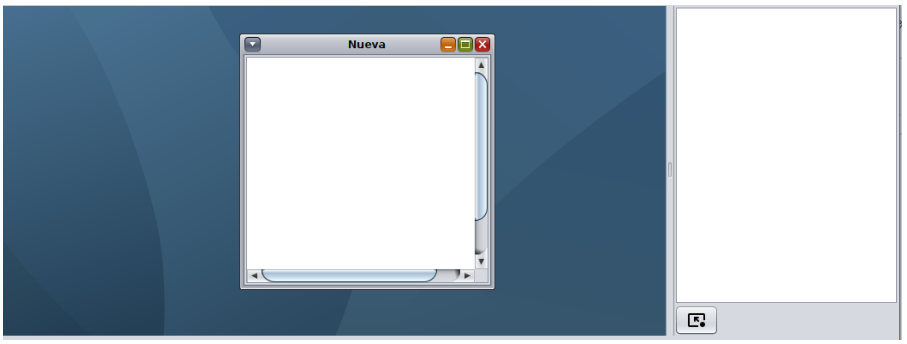
Para que el usuario pueda realizar todas las operaciones vistas, se proporcionará una interfaz, con una barra de herramientas en la parte superior, donde se tendrán a disposición los distintos botones, deslizadores y spinners para interactuar. En ella encontraremos las distintas formas de dibujado, atributos asociados o los botones de reproducción y captura de vídeo y audio entre otras opciones.



Además, para no saturar la parte superior de la aplicación, se dispone de un panel inferior que recopila gran parte de las operaciones sobre imágenes, tales como los cambios de brillo y contraste, o las traslaciones y extracciones de bandas.



El usuario contará además con un panel lateral, situado a su derecha, que almacenará las figuras dibujadas hasta el momento. Este panel permitirá realizar una selección múltiple de dichas figuras y volcarlas sobre la imagen con la ayuda de un botón situado debajo de dicho panel.
En el centro de la pantalla, ocupando la mayor parte del espacio, se encontrará la ventana o ventanas sobre las que se actúa.

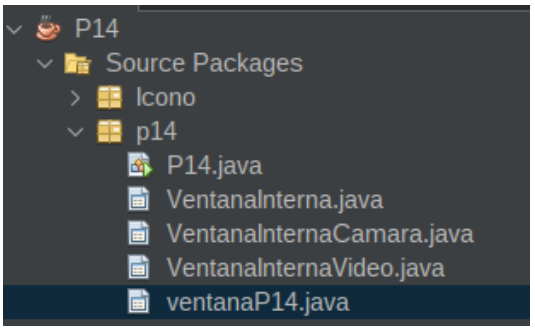


5. Estructura

Para la implementación de esta aplicación, con todos los requisitos propuestos, se establecerá una estructura de clases como la que sigue:

Proyecto

En el propio proyecto de NetBeans encontraremos una clase principal desde la que ejecutaremos el proyecto. En el mismo paquete, encontraremos las clases asociadas a las distintas ventanas internas del proyecto, como son la de imagen, cámara y vídeo, así como la ventana principal:



Como podemos ver, todas estas clases se encuentran bajo el paquete *p14.*

Además, se ha optado por guardar en un paquete llamado “iconos”, aquellos ficheros .png correspondientes a los distintos botones empleados.

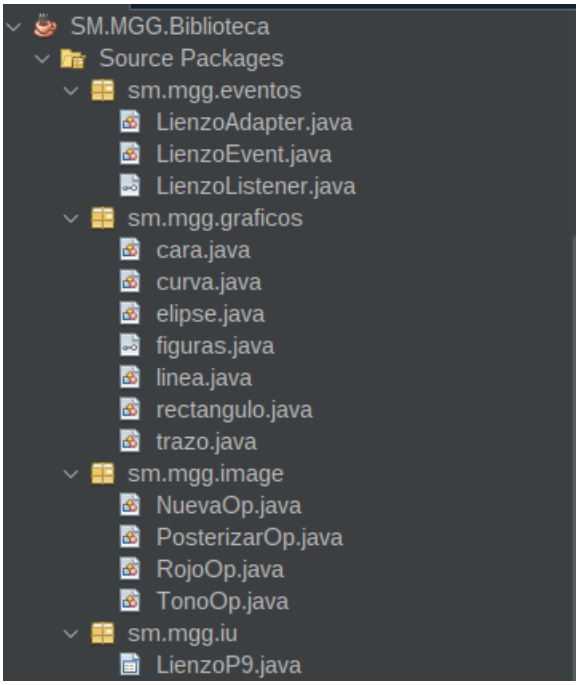
Biblioteca

Definidas las clases del proyecto, haremos uso de una biblioteca propia, dividida en paquetes, en la que definiremos el resto de clases necesarias para llevar a cabo nuestra aplicación.

Concretamente, esta biblioteca se ha dividido en cuatro paquetes:

- ➢ Paquete de eventos, donde definiremos la interfaz, clase asociada a eventos y el adapter propio.
- ➢ Paquete de gráficos, donde definimos todas las clases correspondientes a cada figura dibujable así como la interfaz que implementan.
- ➢ Paquete de imagen, donde se definen las clases correspondientes a distintas operaciones sobre imágenes.
- ➢ Paquete de interfaz, donde se define la clase “Lienzo”, que será con la que interactuamos como usuarios.

De esta manera, la estructura quedaría como sigue:



6. Implementación / Codificación

El código con detalle se encuentra explicado en la documentación generada a través de Javadoc. A fin de evitar un documento excesivamente extenso, y de delegar la explicación concreta del código a la API generada, en esta sección se comentará lo más importante referente a las distintas clases diseñadas, pero no se realizará un análisis extenso de todo lo implementado.

➢ Paquete de eventos

En este paquete se definirá la clase que presentará un evento lanzado por un objeto de la clase Lienzo (LienzoEvent), la interfaz (LienzoListener) y su adapter correspondiente (LienzoAdapter).

La clase LienzoEvent hereda de EventObject y tendrá definidas las variables forma y color, objetos tipo Shape y Color respectivamente, para guardar la información asociada a cada evento. Dichas variables serán asignadas en el constructor de la propia clase y además se definirán los get correspondientes:

```
public class LienzoEvent extends EventObject {

    private Shape forma;
    private Color color;

    public LienzoEvent(Object source, Shape forma, Color color) {
        super(source);
        this.forma = forma;
        this.color = color;
    }

    public Shape getForma() {
        return forma;
    }

    public Color getColor() {
        return color;
    }
}
```

La interfaz LienzoListener heredará de EventListener y crearemos en ella los métodos correspondientes a los casos en los que se pueda generar un evento, es decir, en la adición de una figura al lienzo, o en el cambio de alguno de sus atributos.

Por último definiremos el adaptador que implementa la interfaz anteriormente creada: LienzoAdapter.

➢ Paquete de gráficos

Común a todas las clases de este paquete, exceptuando claro está, la interfaz que implementan, encontraremos una serie de atributos y métodos implementados.
En cuanto a los atributos compartidos encontramos atributos para almacenar el color, transparencias, alisados o trazos, así como numerosos booleanos que harán de variables de control.

```
/**
 * El parámetro color será un objeto tipo Color que almacenará la propiedad
 * del color del objeto que estemos tratando en cada instante.
 */
private Color color = Color.black;

/**
 * relleno es un booleano usado para controlar si la(s) figura(s) en
 * cuestión deben estar rellenas o no.
 */
private boolean relleno = false;

/**
 * composicion es un atributo tipo Composite que utilizaremos para definir
 * la transparencia de las figuras. Alpha correspondiente iniciado a 1.
 */
private Composite composicion = AlphaComposite.getInstance(rule:AlphaComposite.SRC_OVER, alpha:1f);

/**
 * antialiasing es un booleano utilizado para pivotar entre los posibles
 * estados de "antialiasing" aplicado a las figuras.
 */
private boolean antialiasing = false;
```

Algunos de los atributos mencionados.

En cuanto a los métodos comunes más a destacar encontramos el método paint, al que proporcionaremos un objeto tipo Graphics con el cual instanciamos Graphics2D a través del objeto g2d, y que emplearemos para realizar los distintos “set” correspondientes a los atributos del objeto. Este método, compartido por cada figura propia será el invocado desde la clase interfaz, Lienzo. Además se contarán con métodos set y get para establecer las distintas propiedades anteriormente vistas.

Pasando a los métodos específicos de cada clase, destacamos el constructor de la clase Cara, que define las distintas figuras que forman la cara y hace uso de los métodos de la clase Area, add y subtract para crearla:

```
public cara(Point2D p){
    Ellipse2D cabeza = new Ellipse2D.Double(x:p.getX(), y:p.getY(), w:100, h:100);
    Ellipse2D ojo1 = new Ellipse2D.Double(p.getX()+25, p.getY()+15, w:20, h:20);
    Ellipse2D ojo2 = new Ellipse2D.Double(p.getX()+55, p.getY()+15, w:20, h:20);
    QuadCurve2D labio = new QuadCurve2D.Double(p.getX()+10, p.getY()+55, p.getX()+50, p.getY()+90, p.getX()+90, p.getY()+55);

    add(new Area(s:cabeza));
    subtract(new Area(s:ojo1));
    subtract(new Area(s:ojo2));
    subtract(new Area(s:labio));
```

En el constructor se creará la forma de la cara, haciendo uso de otras formas como elipses y curvas.

Debemos ser capaces de trasladar la localización de la figura dibujada en la ventana. Por ello, será necesario crear, en algunos casos, métodos específicos para moverla. Por ejemplo, en la clase correspondiente a la curva, el método en cuestión, llamado setLocation, sigue la siguiente implementación:

```
public void setLocation(Point2D pos) {
    double dx = pos.getX() - this.getX1();
    double dy = pos.getY() - this.getY1();
    Point2D newp2 = new Point2D.Double(this.getX2() + dx, this.getY2() + dy);
    Point2D ctrlpoint = new Point2D.Double(this.getCtrlX() + dx, this.getCtrlY() + dy);
    this. setCurve(p1: pos, cp: ctrlpoint, p2: newp2);
```

Donde se calcula la diferencia de posición entre el punto inicial de dibujado y aquel pasado por parámetro. Crea un nuevo punto a partir de esta información y define la curva en el nuevo punto resultante.

También necesitaremos en otros casos determinar si un punto concreto pertenece a nuestra figura, para poder elegir con precisión la figura a mover en caso de tener varias cercanas. Todas estas clases implementarán su método toString correspondiente.

➢ Paquete de imágenes

Dentro de las clases que heredan de BufferedImageOpAdapter destacamos el método “filter” que devolverá un objeto de tipo BufferedImage. En él, comprobamos si la imagen de origen y destino es válida, y obtenemos los respectivos raster. Recorremos los componentes aplicando los cambios correspondientes a la operación aplicada. Finalmente devolvemos el objeto BufferedImage destino:

```
public BufferedImage filter(BufferedImage src, BufferedImage dest) {
    if (src == null) {
        throw new NullPointerException(s: "src image is null");
    }
    if (dest == null) {
        dest = createCompatibleDestImage(src, destCM: null);
    }
    WritableRaster srcRaster = src.getRaster();
    WritableRaster destRaster = dest.getRaster();
    int sample;
    for (int x = 0; x < src.getWidth(); x++) {
        for (int y = 0; y < src.getHeight(); y++) {
            for (int band = 0; band < srcRaster.getNumBands(); band++) {
                sample = srcRaster.getSample(x, y, b: band);
                int K = (int) 256/ niveles;
                sample = K * (int) (sample/K);
                destRaster.setSample(x, y, b: band, s: sample);
            }
        }
    }
    return dest;
}
```

Método filter correspondiente a la clase PosterizarOp.

➢ Paquete de interfaz

Aquí encontramos la clase Lienzo, en la que se encuentran definidos los principales atributos y métodos necesarios para el dibujado.

En cuanto a los atributos más importantes, destacamos los usados para almacenar los distintos manejadores y figuras o los pinceles usados, que indican la figura a dibujar.

De los métodos destacamos el usado para volcar figuras. En él, procedemos de la misma manera que haríamos en el método paint, pero solo sobre el vector de figuras seleccionadas por el usuario. Dentro de él llamaríamos a los métodos paint correspondientes de cada figura, y las eliminaremos del vector del lienzo, así como de la lista en cuestión.

```
/**
 * Método para el volcado de figuras, es llamado por getImage y realiza la misma
 * función que el método paint, pero para los objetos seleccionados. Una vez pintados
 * los elimina del vector vShape y limpia la lista
 * @param g Objeto graphics necesario para llevar a cabo los métodos sonbre objetos graphics2D
 */

public void volcarFiguras(Graphics g){
   super.paint(g);
   Graphics2D g2d = (Graphics2D)g;
   if (img != null) g2d.drawImage(image: img,i: 0,i1: 0,io: this);
   if (img != null) {
       g2d.setClip(shape: clipArea);
   }
    for (figuras s : seleccionados) {
           s.paint(g);
           vShape.remove(o: s);
       }
       seleccionados.clear();
}
```

También se destaca el método paint, en él se establece el área clip y se llama a cada método paint propio de cada figura registrada en el vector vShape. Además, se contará con métodos para manejar el press y drag del ratón, de manera que crearemos una figura, o moveremos una previamente seleccionada, en función de la acción solicitada.

```
étodo para controla el dragg del ratón una vez pulsado. Si estamos en modo mover calculamos el punto en el que nos encontramos y su diferencia con el punto
espués calculamos el nuevo punto de origen de dibujado de la figura correspondiente que estuviésemos trasladando. En caso de no estar moviendo ninguna figur
reamos aquella indicada por el objeto Shape con las nuevas coordenadas del ratón.
param evt evento del ratón.

ate void formMouseDragged(java.awt.event.MouseEvent evt) {
if(mover){
    Point2D nuevoPuntoOrigen = null;
    if(forma!=null){
        double dx = evt.getX() - puntoInicioDesplazamiento.getX();
        double dy = evt.getY() - puntoInicioDesplazamiento.getY();
        nuevoPuntoOrigen = new Point2D.Double(puntoInicioDibujadoDesplazamiento.getX() + dx, puntoInicioDibujadoDesplazamiento.getY() + dy);
    }
        //Código para el caso del rectángulo
    if (forma!=null && forma instanceof rectangulo)
        ((rectangulo)forma).setFrame(x: nuevoPuntoOrigen.getX(),y: nuevoPuntoOrigen.getY(),w: ((Shape)forma).getBounds2D().getWidth(),h: ((Shape)forma).getBound
        //Código para el caso de la elipse
    if (forma!=null && forma instanceof elipse)
        ((elipse)forma).setFrame(x: nuevoPuntoOrigen.getX(),y: nuevoPuntoOrigen.getY(),w: ((Shape)forma).getBounds2D().getWidth(),h: ((Shape)forma).getBounds2D(
    if (forma!=null && forma instanceof linea)
        ((linea)forma).setLocation(pos: nuevoPuntoOrigen);
    if (forma!=null && forma instanceof curva)
        ((curva)forma).setLocation(pos: nuevoPuntoOrigen);
    if (forma != null && forma instanceof trazo)
        ((trazo)forma).setLocation(pos: nuevoPuntoOrigen);
     if (forma != null && forma instanceof cara)
        ((cara)forma).setLocation(pos: nuevoPuntoOrigen);
}
else{
    if (null != pincel) {
        if (forma instanceof linea) {
            ((linea) forma).setLine(p1: ((linea) forma).getP1(), p2: evt.getPoint());
        }else if(forma instanceof rectangulo){
            ((rectangulo) forma).setFrameFromDiagonal(p1: punto, p2: evt.getPoint());
        }else if(forma instanceof elipse){
            ((elipse) forma).setFrameFromDiagonal(p1: punto, p2: evt.getPoint());
        }else if(forma instanceof trazo){
            ((trazo) forma).lineTo(x: evt.getPoint().getX(), y: evt.getPoint().getY());
        }else if(forma instanceof curva){
            if (punto_control) {
                ((curva) forma).setCurve(p1: punto, cp: punto, p2: evt.getPoint());
                punto_auxiliar = evt.getPoint();
            } else {
                ((curva) forma).setCurve(p1: punto, cp: evt.getPoint(), p2: punto_auxiliar);
```

Método para manejar el arrastrado del ratón.

7. Operadores propios

Para el desarrollo de esta aplicación se han desarrollado dos operaciones nuevas, la primera, se basará en la aplicación de una función sobre los valores de una imagen, y la segunda, será una modificación de los píxeles que la componen.

Función LookUp

Para esta operación se han seguido las siguientes funciones:

$$f(x) = \sqrt{-x + 128}$$

$$g(x) = \frac{(x - 128)^2}{n}$$

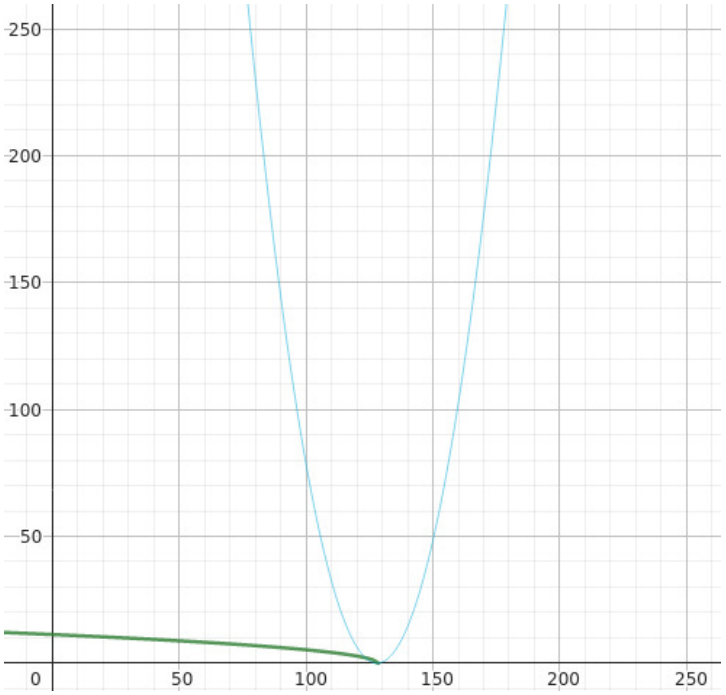
Para x <= 128: √ (-x+128)
En otro caso: ((x-128)^2)/n
Siendo n el parámetro definido por el propio usuario
a través de un slider.

Para llevar a cabo esta operación, debemos hacer uso de un objeto LookupTable al que le pasaremos como datos un array de bytes en el que se guardará el resultado de hacer la operación a cada elemento. Después crearemos un objeto LookupOp con ayuda de esta tabla, y lo asignaremos a la ventana actual con SetImage.

La implementación es la siguiente:

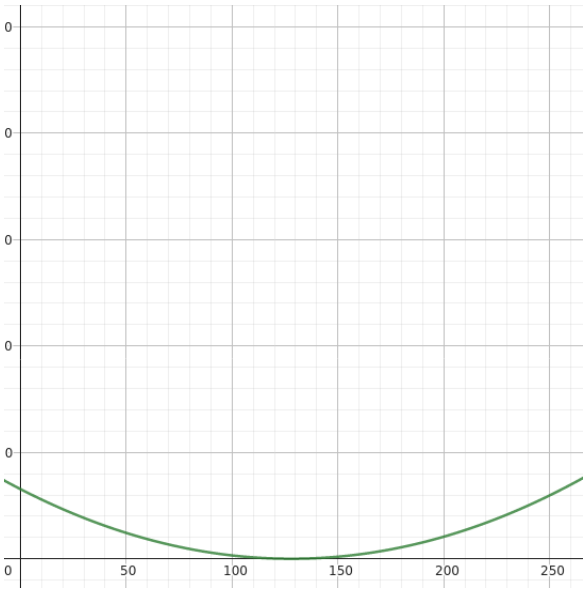
```
private void sliderLookupPropioStateChanged(javax.swing.event.ChangeEvent evt) {
    VentanaInterna vi = (VentanaInterna) (escritorio.getSelectedFrame());
    if (vi != null) {
        if (imgFuente != null) {
            try {
                byte funcionT[] = new byte[256];
                for (int x = 0; x < 256; x++) {
                    if (x <= 128) {
                        funcionT[x] = (byte) sqrt(-x + 128);
                    } else {
                        funcionT[x] = (byte) ((byte) (pow((x - 128), b: 2)) / (float) sliderLookupPropio.getValue());
                    }
                }
                LookupTable tabla = new ByteLookupTable(offset: 0, data: funcionT);
                LookupOp lop = new LookupOp(lookup: tabla, hints: null);
                BufferedImage imgdest = lop.filter(src: imgFuente, dst: null);
                vi.getLienzo().setImage(img: imgdest);
                vi.getLienzo().repaint();
```

La gráfica para n = 10 quedaría de la siguiente manera:

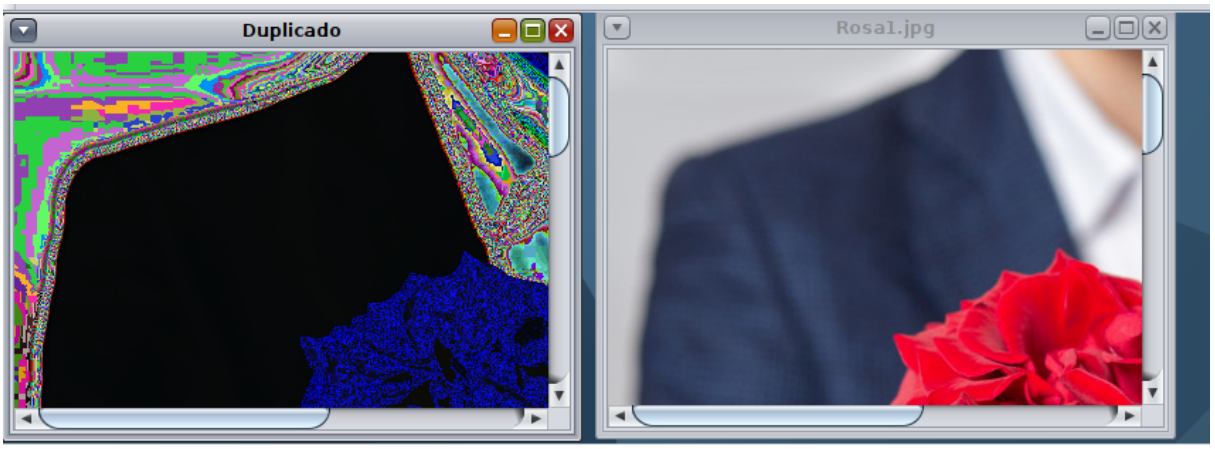


Podemos ver que los componentes inferiores a 128 son apagados hasta llegar al 0, mientras que los superiores sufren una redistribución en sus valores comenzando en 0 y ascendiendo para los más altos, algo que se suaviza si el usuario introduce un valor (n) mayor.

Para n = 500, vemos como este incremento de los valores a partir de 128 se suaviza:



En un ejemplo práctico, observaremos como los colores que no superen el umbral establecido se oscurecen hasta casi llegar al negro, aquellos en valores intermedios son igualmente oscurecidos aunque en menor medida, y los valores más altos son distribuidos a lo largo del trazo ascendente descrito por la función ((x-128)^2)/n, donde solo los blancos puros tendrían los valores más altos de toda la imagen.



Operación pixel a pixel

Para la operación pixel a pixel propia, se ha creado una clase que, al igual que las anteriores, hereda de BufferedImageOpAdapter. El parámetro que guardará esta clase será uno introducido por el propio usuario y que determinará la cantidad de aleatoriedad existente a la hora de llevar a cabo la transformación.

```
public class NuevaOp extends BufferedImageOpAdapter {

    private int aleatoriedad;

    public NuevaOp(int valor) {
        this.aleatoriedad = valor;
    }
```

En el método filter, generamos un número aleatorio cada vez que analizamos y transformamos un píxel. En la mayoría de los casos, la transformación aplicada a cada píxel será una multiplicación -distinta para cada banda-, a los componentes del mismo. En otros casos, más improbables, se aplicará una reducción de los valores de cada componente, teniendo en cuenta los valores de otras componentes distintas a la que se aplicará la transformación. El usuario podrá variar la probabilidad con la que se da este último caso en función del valor introducido.

Así pues, describimos el comportamiento de la siguiente forma:

Para cada valor RGB de cada píxel:
- R = R * 2
- G = G * 3
- B = B * 4

Existirá una probabilidad 1/N en la que el valor adquirido será:
- R = B / 2
- G = R / 3
- B = G / 4

La implementación es la siguiente:

```
Random rand = new Random();
int numeroAleatorio;

WritableRaster srcRaster = src.getRaster();
WritableRaster destRaster = dest.getRaster();
int[] pixelComp = new int[srcRaster.getNumBands()];
int[] pixelCompDest = new int[srcRaster.getNumBands()];
for (int x = 0; x < src.getWidth(); x++) {
    for (int y = 0; y < src.getHeight(); y++) {
        srcRaster.getPixel(x, y, iArray: pixelComp);

        numeroAleatorio = rand.nextInt(bound: aleatoriedad);

        if (numeroAleatorio == 0) {
            pixelCompDest[0] = pixelComp[2] / 2;
            pixelCompDest[1] = pixelComp[0] / 3;
            pixelCompDest[2] = pixelComp[1] / 4;
        } else {
            pixelCompDest[0] = pixelComp[0] * 2;
            pixelCompDest[1] = pixelComp[1] * 3;
            pixelCompDest[2] = pixelComp[2] * 4;
        }

        destRaster.setPixel(x, y, iArray: pixelCompDest);

    }
}
```

En la siguiente imagen se muestra un ejemplo con un valor introducido de 10, es decir, hay un 10% de probabilidades de que se aplique la división de los valores de cada banda, y un 90% de que se multipliquen: