UNIVERSIDAD DE GRANADA

Práctica final de Sistemas Multimedia:

Memoria de la práctica

Por Germán Castilla López

ÍNDICE

INTRODUCCIÓN

En este documento procederé a explicar y enumerar los requisitos con los que cumple la aplicación adjunta, como he abordado la implementación de la funcionalidad requerida y explicaré el diseño de clases de creación propia.

Nota: para ayudar a la evaluación de la documentación, en los requisitos he añadido una pequeña nota a aquellos requisitos opcionales o que requieran una implementación de creación propia desde cero (por ejemplo, el operador píxel a píxel de creación propia).

Nota 2: Esta aplicación ha sido desarrollada en Netbeans 10.0.

Nota 3: Para la realización de esta aplicación se ha seguido los ejemplos de código y las indicaciones de los guiones de prácticas ofrecidos por el profesor de la asignatura además de los vídeos realizados por él. Los requisitos que estos guiones traían implementados han sido desarrollados con la implementación del guión de prácticas.

Requisitos

En este bloque procederé a explicar todos los requisitos con los que cumple el programa desarrollado:

Requisitos de carácter general

1. El programa contará con un escritorio central.
2. En el escritorio se podrá alojar ventanas internas de diferentes tipos, es decir, imagen, vídeo, cámara.
3. Si se crea una nueva ventana interna, tendrá de nombre “Nueva”.
4. En la barra de menú habrá un menú archivo con las opciones “Nuevo”, “Abrir”, y “Guardar”. Estas opciones estarán también en la barra de herramientas con un icono asociado y un ToolTipText.
    a. Al pulsar sobre la opción “Nuevo” se crea una nueva imagen.
    b. Al pulsar la opción “Abrir” se abrirá un archivo elegido por el usuario (ver requisito 5).
    c. Al pulsar la opción “Guardar” se guardará la imagen con las figuras dibujadas.
5. Con la opción de Abrir, se podrá abrir tanto imagenes, sonidos o vídeos.
6. En la barra de menú habrá un menú “Ver” que permitirá ocultar las diferentes barras de herramientas.
7. En la barra de menú habrá un menú “Ayuda”, con la opción “Acerca de” que lanzará un diálogo con el nombre del programa, versión y autor.
8. En el diálogo para abrir archivos habrá filtros para que solo se muestren archivos con una extensión permitida.
9. Si se produce algún error al abrir un archivo se lanzará un diálogo que informe del problema.

10. En el diálogo para guardar archivos habrá filtros para que solo se muestren archivos con una extensión permitida.
11. **[Opcional]** El usuario, al crear una nueva imagen, podrá elegir las dimensiones de la imagen mediante un diálogo.

Requisitos de dibujo

1. Cada figura tendrá sus propios atributos, independientemente del resto de figuras.
2. Solo se podrá dibujar sobre el área de la imagen.
3. El lienzo mantendrá todas las figuras que se vayan dibujando.
4. Se podrá dibujar diferentes figuras.
5. Figuras obligatorias: línea, rectángulo y elipse.
6. El usuario podrá elegir el color del trazo de las figuras.
7. El usuario podrá elegir al crear una figura si quiere que se le realice alisado de contornos o no.
8. El usuario podrá elegir el grosor de la figura a crear.
9. El usuario podrá elegir si las figuras creadas serán con contornos continuos o discontinuos.
10. El usuario podrá elegir si crear las figuras rellenas o no.
11. El usuario podrá elegir el color de relleno.
12. El usuario podrá elegir si el relleno será degradado.
13. El degradado se hará con los colores elegidos (trazo y relleno).
14. El usuario podrá elegir si el degradado se aplica vertical u horizontalmente.
15. El usuario podrá elegir el grado de transparencia de la figura que va a dibujar.
16. El usuario podrá editar las figuras ya dibujadas cuando marque el botón de edición:
    a. El usuario podrá elegir que figura va a editar haciendo clic sobre ella.
    b. El usuario podrá mover la figura seleccionada con una operación de arrastrar y soltar con el ratón.
    c. La figura seleccionada debe indicarse mediante un rectángulo discontinuo.
    d. Se podrá editar cualquiera de los atributos de la figura seleccionada en la barra de dibujo.
17. El usuario podrá elegir el color del trazo o del relleno de la figura mediante un diálogo de selección de colores.
18. Al cambiar de una ventana interna a otra, los botones marcados deberán activarse conforme a los atributos de la ventana activa.
19. **[Opcional]** El usuario podrá añadir rectángulos redondeados.
20. **[Opcional]** Se dispondrá de más direcciones de degradado.

Requisitos de procesamiento de imágenes

1. Cada ventana mostrará una sola imagen.

2. Si una imagen no cabe entera en la ventana interna, se incluirá barras de desplazamiento para moverse sobre la imagen.
3. Si se abre una imagen, el nombre de la ventana que la contiene tendrá como título el nombre de la imagen.
4. El usuario podrá modificar el brillo de una imagen mediante un deslizador.
5. El usuario tendrá a disposición opciones de contraste normal, iluminado y oscurecido.
6. El usuario podrá modificar el escalado de la imagen (aumentar o disminuir).
7. El usuario podrá aplicar un filtro sepia.
8. El usuario podrá aplicar filtros de emborronamiento, enfoque y relieve.
9. El usuario podrá aplicar un filtro de ecualización.
10. El usuario pordrá rotar la imagen libremente mediante un deslizador.
11. El usuario podrá convertir una imagen a los espacios RGB, YCC y GRAY.
12. El usuario tendrá la opción de extracción de bandas.
13. El usuario podrá realizar operaciones de tintado con el color que este seleccionado, pudiendo indicar el grado de mezcla.
14. El usuario podrá aplicar un filtro para ver la imagen seleccionada en negativo.
15. El usuario podrá elegir mediante un botón de dos posiciones si los filtros que desea aplicar se aplican a las figuras dibujadas o no.
16. El usuario tendrá la opción de crear una copia de la imagen seleccionada.
17. En el título de la ventana que contiene a la imagen debe indicarse entre corchetes el espacio de color en el que se encuentra o, si la imagen está asociada a una banda, indicar a que banda.
18. El usuario podrá realizar la operación cuadrática, indicando con un deslizador el valor de m.
19. El usuario podrá aplicar la operación de posterización mediante deslizador sobre la imagen seleccionada.
20. El usuario podrá aplicar la operación de resaltado de rojo sobre la imagen seleccionada.
21. **[Píxel a píxel propio]** Habrá una operación píxel a píxel de diseño propio con al menos un parámetro que se aplique sobre la imagen seleccionada.
22. **[LookUp propio]** Habrá una operación LookUp de diseño propio con al menos un parámetro que se aplique sobre la imagen seleccionada.
23. **[Opcional]** El usuario podrá elegir el umbral de la operación de resaltado de rojo mediante un deslizador.
24. **[Opcional]** El usuario podrá enverdecer la imagen mediante una operación de extracción de bandas.
25. **[Opcional]** Otra función de diseño propio.

Requisitos de sonido

1. Habrá un botón para reproducir, otro para parar la reproducción o la grabación y otro para grabar, aparte de una lista de reproducción que será una lista despegable.

2. Los ficheros abiertos de audio se añadirán a la lista de reproducción, sin reproducirse.
3. Cuando se pulse el botón de reproducir, se reproducirá el sonido que esté seleccionado en la lista de reproducción.
4. Cuando se pulse el botón de parar, se dentendrá la reproducción de audio.
5. Cuando se pulse el botón de grabación, se grabará el sonido recogido por el micrófono del sistema y se guardará en un archivo cuyo nombre será elegido por el usuario.
6. Cuando se pulse el botón de parar, se dentendrá la grabación de audio.

Requisitos de video

1. Al abrir un vídeo, la ventana interna tendrá de título el nombre del fichero.
2. El usuario podrá reproducir los vídeos que abra y podrá controlar su reproducción.
3. El usuario podrá ver la secuencia que esté captando la webcam.
4. El usuario podrá crear una captura de la imagen que muestre la webcam o el vídeo que se esté reproduciendo y se mostrará en una nueva ventana.
5. La ventana con la captura realizada deberá tener como título “Captura”.
6. Debe haber una lista despegable con las cámaras del sistema.
7. El vídeo captado por webcam será el correspondiente a la cámara que elija el usuario.
8. Debe haber una lista despegable con las resoluciones de la cámara de la que el usuario pueda elegir con que resolución se abrirá la cámara.

Análisis

En este bloque procederé a explicar que nos ofrece java para abordar los requisitos anteriormente comentados, y si lo que nos ofrece java es insuficiente o tiene algún problema para abordar el requisito explicaré como solucionaré esas insuficiencias.

Nota: la numeración de los análisis es exactamente igual a la numeración de los requisitos, por ejemplo, en el punto de análisis 13 de requisitos de dibujo, correspone al requisito de dibujo 13.

Análisis de requisitos de carácter general

1. Javanos ofrece añadir directamente un JDesktopPane, que actua como escritorio en la ventana principal.
2. Javasolo nos ofrece una clase de ventana interna genérica, por lo que crearé una superclase VentanaInternaSM de la que dependan los diferentes tipos de ventanas internas.

3. He llamado a la función asociada a la ventana interna setTitle, indicándole que el nombre será “Nueva”.
4. Para el menú simplemente he añadido un swing menú, proporcionado por java, llamado “MenuBar”, donde al primer item (“Menu”) le he puesto “Archivo” y le he añadido tres “Menu item” (también proporcionado por netbeans) con los nombres que se piden. Para los botones, he añadido tres swing controls, llamados “button” (proporcionado por netbeans) y a cada uno, en el panel de propiedades, le he quitado el texto, le he añadido su icono correspondiente y le he puesto el tool tip text adecuado. En la opción de nueva simplemente crea una nueva ventana y la pone visible. Para guardar, en el lienzo he creado una función getImage a la que se le pasa un valor booleano, que si es true hace una copia de la imagen que hay en el lienzo y después le vuelca las figuras dibujadas llamado al paint.
5. Para el requisito número 7, he creado tres funciones (isImagen, isSonido, isVideo), para, en cada una, comprobar la extensión del fichero a abrir y ver a qué tipo de fichero corresponde. Esto ha sido posible gracias a:
    a. En el caso de las imágenes, la calse ImageIO (propia de java), contiene una función llamada getReaderFileSuffixes, que devuelve un array de strings con todos los sufijos de los archivos permitidos. Además, la calse File contiene un getName, que seguido con un endsWith(extensión) compara directamente la extensión pasada con el final del nombre del archivo. Así que solo hay que hacer un bucle hasta que coincidan.
    b. En el caso de los sonidos, se ha usado las librerías propias de java AudioFileFormat y AudioSystem para comprobar que ficheros son aceptados por java, usando la función getAudioFileTypes, que devuelve un array de Type (de la librería AudioFileFormat). A cada objeto de este tipo se le puede hacer un getExtension que devuelve un string con la extensión, por lo que con un bucle idéntico al del caso anterior se puede comprobar si el fichero es un sonido.
6. A la barra del menú he añadido el item “Ver” con tres menu item (leyenda, barra de herramientas superior y barra de herramientas inferior). En cada opción, compruebo si el item que se selecciona está visible (isVisible(), función que da java): si lo está, lo hago invisble (setVisble(false), función que da java), si no está visible, lo pongo visible (setVisible(true)).
7. He añadido otro item a la barra de menú llamado “Ayuda”, al que he añadido un menuItem llamado “Acerca de”, a cuyo action performed le he asignado que lancce un showMessageDialog(), función de JOptionPane (todo incluido ya en java, importando su correspondiente paquete).
8. Java nos ofrece la clase FileNameExtensionFilter, cuyo constructor acepta dos parámetros, uno con el nombre del filtro y otro con un array de strings con los ficheros reconocidos. He creado tres filtros usando esta clase, uno para cada tipo (imágenes, sonidos y vídeos) y se los he aplicado al diálogo de abrir con la función setFileFilter() que ya incorpora Java para los JFileChooser.

9. Java nos ofrece la posibilidad de lanzar diálogos con texto con la clase JOptionPane y su función showMessageDialog. Si el programa no reconoce el tipo de archivo se lanzará una ventana con el mensaje "Error al abrir el archivo: Formato desconocido", si ocurre cualquier otro error el mensaje será "Error al abrir el archivo".
10. Java nos ofrece la clase FileNameExtensionFilter, cuyo constructor acepta dos parámetros, uno con el nombre del filtro y otro con un array de strings con los ficheros reconocidos. He creado tres filtros usando esta clase, uno para cada tipo (imágenes, sonidos y vídeos) y se los he aplicado al diálogo de guardar con la función setFileFilter() que ya incorpora Java para los JFileChooser.
11. **[Opcional]** Java nos ofrece un tipo de diálogo en donde información con la clase JOptionPane y su método showInputDialog. Esta ventana solo recoge texto (string) por lo que hay que convertir el texto en un entero para la creación de la imagen. Esto es posible gracias a la función que trae la clase Integer de Java parseInt() que convierte el string pasado en entero. Como esta ventana solo admite el paso de una variable, al crear la nueva imagen se lanzan dos ventanas, una para indicar los píxeles del ancho y otra para los píxeles del alto. Estos dos valores serán los usados después en el constructor del BufferedImage.

Análisis de requisitos de dibujo

1. Java no nos proporciona la solución, ya que las clases de las diferentes formas geométricas no tienen atributos de color, trazo, grosor... Es decir, no se les podía cambiar los atributos de forma individual. Para solucionar este problema, he creado una superclase llamada ShapeSM, de la que dependeran las diferentes clases de las diferentes figuras.
2. En el lienzo he creado una figura Shape que se inicializa como un rectángulo. Este rectángulo se le pasa al lienzo cada vez que se crea una ventana nueva o se abre una imagen, con las dimensiones de la zona imagen. En el paint del lienzo se llama a la función clip() del g2d, a la que se le pasa este rectángulo.
3. En la clase del lienzo he creado un array de mi clase propia ShapeSM, y en el método paint se recorre este array y se va dibujando las figuras una a una.
4. He creado un enumerado llamado Formas. En la ventana principal he creado una variable de este enumerado, que irá cambiando según el usuario seleccione una forma u otra. Para ello, cada vez que se selecciona una forma, se llama a una función que cambia otra variable en el Lienzo. Una vez aquí, en las funciones para crear y actualizar las figuras del lienzo, primero se comprueba que forma está guardada en la variable.
5. He creado tres clases (Linea, Rectangulo y Elipse). Las tres hacen extends de ShapeSM. En cada una, en los constructores se crea la respectiva forma (Line2D, Rectangle2D y Ellipse2D), y cada clase tiene una función paint con la función draw (para pintar las figura). Aparte tienen una función setFigura (donde Figura es el nombre de dicha figura) que hace uso de funciones de java

para poder colocar la figura (por ej: setLine). Cada vez que se pinta una nueva figura, se crea una nueva instancia de cada clase, para respetar los atributos individuales de cada figura.

6. He creado un comboBox, el cual tiene como objetos en la lista Color en vez de String. Después, he cambiado el código asociado al comboBox de colores para que se inicialice con seis colores predeterminados. Para evitar que en la lista salieran sus valores RGB, he creado una clase llamada ColorCelRender, la cual asigna a cada color su forma (un cuadrado negro, un cuadrado rojo...). El color que elija el usuario se le pasa al lienzo, y este al shape (cuyo constructor he cambiado para que se cree con el color que se le pase), así que se crea la figura con el color seleccionado.
7. Como tengo que mantener las propiedades diferentes, he usado las clases que he creado anteriormente de ShapeSM y de las figuras, ya que con lo que nos ofrece java sería imposible. A la clase ShapeSM se le pasa el color (setColor) y si está alisado o no. En las figuras que pueden ser alisadas (es decir, las figuras curvas), he hecho uso de este constructor, quitando el anterior, así que cuando el usuario pulsa el botón de alisado, la siguiente figura se creará con el anitialiasing (suavizado de bordes) que nos ofrece java activado.
8. Java nos ofrece un tipo de botón llamado spinner, en el cual se le puede asignar números y aumentarlos o decrementarlos mediante dos flechitas. Para el grosor, java nos ofrece la clase BasicStroke, al que se le pasa un entero. Al llamar al Graphics2D se le puede asignar el stroke con el que queremos que dibuje, así que solo es necesario añadir al constructor de las figuras (ShapeSM) un BasicStroke y asignárselo a su Graphics2D.
9. Para el requisito 13.h, en la barra inferior del programa el usuario puede marcar si quiere que las figuras sean discontinuas. Esto se lo pasa al lienzo. La clase mencionada anteriormente BasicStroke, la cual nos ofrece java, no solo tiene el constructor al que se le pasaba el grosor, tiene varios constructores, y en uno se puede configurar la discontinuidad con la que una figura se va a dibujar. En el constructor usado se le puede pasar el grosor, la forma en la que acaban los segmentos discontinuos (he elegido de forma redondeada), la decoración cuando dos segmentos se encuentran, o lo que más nos importa en este apartado, el patrón de discontinuidad.
10. He añadido un botón de dos posiciones (JToggleButton) para que el usuario pueda indicar si quiere la figura rellenada o no. Esto se le pasa al lienzo, para asignarle este valor a los constructores de las figuras. En cada clase de cada figura, se comprueba si debe ser rellenado o no. En caso afirmativo, en el paint de la clase de la figura se hace uso de la función fill() de la clase Graphics2D para rellenar la figura, en caso contrario, se usa la función draw().
11. He creado un comboBox igual que el del color para el trazo. Al elegir un color en el comboBox, se le pasa al lienzo, y en la creación de las figuras rellenables se le debe pasar el color de relleno. En cada figura rellenable, si se tiene que rellenar, en el paint pintará con la función draw() de java la figura (para pintar los bordes), luego se usa la función setPaint() de java, a la que se le pasa el

color de relleno, y después se hace uso de la función fill() para rellenar la figura. Java generaba problemas de visualización cuando el borde y el relleno era de color diferente y el grosor del trazo era 1, por lo que todas las figuras tendrán de grosor de trazo mínimo 2.

12. He creado un botón para que el usuario marque si quiere el rellenado degradado o no. Al pulsarlo, se actualiza una variable booleana en el lienzo para usarlo en la figura a crear.
13. Java nos ofrece la clase GradientPaint, que crea el degradado, a la que hay que pasarle el punto donde empieza y acaba el degradado y los dos colores del degradado. Los colores los recoge de los que están seleccionados en el combobox o del diálogo.
14. He creado un botón de doble posición para que el usuario pueda elegir la dirección del degradado. Si se pulsa, será horizontal, si no, vertical. He creado una variable booleana para indicar si el degradado debe ser horizontal o no, y dependiendo del valor de esta variable se creará el GradientPaint con unos puntos u otros.
15. Para que el usuario pueda marcar el grado de transparencia de la figura que va a dibujar a continuación, he hecho uso del swing control proporcionado por java llamado Slider. Esto es una barra deslizante. Mediante el evento focusGained asociado a la barra (evento que ocurre cuando se pincha en la barra), el programa reconoce en que ventana se está interactuando. Mediante el evento stateChanged (que ocurre cuando el valor de la barra cambia), se le pasa al lienzo el valor elegido por el usuario. Finalmente, con el evento focusLost (que ocurre cuando se pincha en otro lugar que no sea la barra), el valor de la barra vuelve al máximo. En el lienzo se le pasa el valor de transparencia en las creaciones de las figuras. Para hacer la figura transparente he usado la clase proporcionada por java Composite, que al crear una instancia de ella se le puede asignar un valor de transparencia. Por defecto le he asignado el valor 1, es decir, opaco, pero cuando se cambia el valor en la barra se actualiza este valor, hasta un 0, que sería invisible.
16. He añadido un botón de dos posiciones (JToggleButton) para que el usuario pueda indicar si quiere optar a funciones de edición. Si se pulsa, una variable booleana llamada “editar” se pondrá a true en el lienzo.
    a. Para que el usuario pueda elegir la figura pinchando sobre ella he creado una función en el lienzo que recorre el vector de figuras y comprueba si alguna figura contiene el punto seleccionado. Para ello, he creado una función en cada clase de cada figura llamada contains(), a la que se le pasa un punto. Dentro, solo he usado la función contains() que nos proporciona java para cada tipo de figura sobre la figura de esa clase. Si alguna figura contiene el punto, se guarda en una variable.
    b. En el método updateShape, si está la variable editar a true, se llamará a una función de creación propia llamada setLocation a la que se le pasa la figura a mover y el nuevo punto donde ponerla. Dentro de la función setLocation, se comprueba que tipo de figura es la que se va a mover, y

se llama al setLocation de la figura pertinente, función a la que se le pasa el punto donde colocar la figura. Dentro de cada setLocation de cada figura se trabaja con la clase creada para calcular el nuevo posicionamiento de la figura.

c. He creado un rectángulo con la clase de Java Rectangle2D, a la cual se le pasa la esquina superior izquiera y el ancho y alto del rectángulo. Para sacar esta información de la figura seleccionada he modificado las clases propias que he creado para las figuras, donde he creado cuatro getters para sacar los cuatros atributos necesarios. Dentro de estos getter he hecho uso de los getters para cada figura que nos ofrece Java. Para evitar problemas de superposición he establecido el grosor del rectángulo a 5 y he establecido el color del marco a negro.
d. Simplemente he creado funciones update para cada atributo, donde se cambia el valor del atributo a modificar y luego se hace un repaint para ver el resultado.

17. Java nos ofrece directamente la opción de hacer aparecer una ventana para la selección de colores, con la clase JColorChoose y su método showDialog(). Esto devuelve un color. Para que el usuario pueda elegir mediante este diálogo los colores de trazo y relleno he creado dos botones, uno para el diálogo del color del trazo y otro para el diálogo del color de relleno.
18. He tenido que crear una nueva clase que extienda de la clase de Java InternalFrameAdapter, modificando su función internalFrameActivated. Dentro de esta función se comprueba si hay una ventana interna seleccionada. En caso afirmativo, se recoge la información de esa ventana y se le va pasando a los diferentes botones con las funciones de java para cambiar el estado de los botones. En caso negativo, se dejan todos los botones en su estado inicial.
19. **[Opcional]** Siguiendo el sistema de clases creado para la creación de nuevas figuras, he creado otra clase para el rectángulo redondeado, llamada RectanguloRedondeado, que también extiende a la clase ShapeSM. El contenido de la clase es idéntico a las demás, pero aquí se hace uso de la clase de Java RoundRectangle2D, con la única peculiaridad de que hay que indicarle el redondeado de las esquinas con la función setRoundRect(). Yo lo he inicializado a 25 tanto en el eje X como en el Y.
20. **[Opcional]** He creado un botón para que el usuario pueda elegir que el degradado sea de esquina a esquina. Para ello cuando se crea el degradado se comprueba si la variable booleana correspondiente está true. Esta opción está por encima de la dirección horizontal o vertical.

Análisis de requisitos de procesamiento de imágenes

1. Cada vez que se abre una nueva imagen se crea una ventana nueva, por lo que no es posible que se cargue en una misma ventana dos imágenes.
2. Hañadido un scrollPanel en la ventana interna imagen, opción que ya nos ofrece java.

3. Lo deja hacer java directamente, ya que guardamos el archivo abierto en un objeto tipo File, que tiene una función getName(), que devuelve el nombre, y ese resultado se lo pasamos a la función de la ventana interna setTitle, función que también viene ya definida en java para las ventanas internas.
4. El deslizador es Swing Control ofrecido por java llamado spinner. En el evento focusGained (que ocurre cuando se pincha sobre la barra), se recoge diversos datos sobre la imagen a la que se le va a modificar el brillo, y se crea una nueva imagen con estos datos. Tras esto, en el evento stateChanged (que ocurre cuando se modifica el valor de la barra), se recoge el valor que ha elegido el usuario, se hace uso de la clase proporcionada de java RescaleOp, utilizada para hacer diversos cambios a las imagenes, al que se le pasa el valor del deslizador y se aplica, mediante la función de java filter(), a la imagen de la ventana. Finalmente, con el evento focusLost (que ocurre cuando se pincha en otro lugar que no sea la barra), el valor de la barra vuelve al inicial.
5. Java nos ofrece la posibilidad de aplicar filtro Lookup a las imagenes con la clase Lookup, así que he utilizado esta clase, he creado una función a la que se le pasa una LookupTable (también proporcionado por java), y se le aplica a la imagen de la ventana seleccionada. Para cada tipo de contraste he creado un botón y he usado la librería proporcionada de sm.image para crear las tablas LookupTable, mediante la clase LookupTableProducer. Para el contraste normal he usado la tabla sfuncion, para el contraste iluminado he usado la tabla logarithm, y para el contraste oscurecido la tabla power, todo proporcionado por la librería.
6. He creado dos botones, uno para aumentar la escala y otro para disminuirla. En los dos se hace un llamamiento a la función escalar(). A esta función se le pasa un double (en caso de aumentar sería 1.5 y en caso de disminuir sería 0.5). Dentro de esta función se hace uso se la clase de java AffineTransform y AffineTransformOp para crear el filtro de escalado con el double mencionado anteriormente. Una vez creado el filtro, se le aplica a la imagen de la ventana seleccionada con la función filter() de java.
7. He creado un botón para que el usuario pueda aplicar el filtro sepia cuando lo desee. Para aplicar el filtro, he hecho uso de la biblioteca proporcionada sm.image, que ya viene con una clase llamada SepiaOp, que genera el filtro sepia, y al que se le puede aplicar la función filter() pasándole la imagen sobre la que se va a realizar el filtro.
8. He creado un comboBox con cuatro items, dos filtros de emborronamiento (media y binomial), enfoque y relieve. Dentro del comboBox, según el filtro elegido se crea la máscara correspondiente, usando la clase de java Kernel. Este kernel se le pasa a la clase ConvoleOp, también de java, y se le aplica a la imagen seleccionada con la función filter. Para crear el kernel específico he creado una función getKernel, donde hago uso de la biblioteca sm.image, con el uso de la clase KernelProducer, el cual ya tiene algunas máscaras definidas. Más concretamente, para el filtro de emborronamiento media uso la máscara TYPE_MEDIA_3X3, para el emborronamiento binomial uso la máscara

TYPE_BINOMIAL_3X3, para el enfoque uso la máscara TYPE_ENFOQUE_3X3, y para el relieve uso la máscara TYPE_RELIEVE_3X3.

9. He creado un botón para que el usuario pueda aplicar el filtro de ecualización cuando lo desee. Para aplicarlo he hecho uso de la biblioteca sm.image, que trae la clase EqualizationOp, que crea el filtro de ecualización y se le aplica a la imagen seleccionada mediante la función filter().
10. He creado un slider para que el usuario pueda rotar la imagen cuando lo desee con valores entre 0 y 360. En el evento stateChanged y focusGained (ya se ha explicado cuando se activan estos eventos), se realiza exactamente lo mismo. Se hace uso de la clase AffineTransform de java y de su función getRotateInstance, a la que se le pasa los grados a girar en radianes y el centro de la imagen como punto de rotación. Tras esto, se hace uso de la clase AffineTransformOp, también de java, para crear la operación de rotación. Con esto ya solo falta usar la función filter() sobre la imagen seleccionada. En el evento focusLost simplemente se pone la barra a 0.
11. He creado un comboBox para que el usuario pueda elegir la conversión deseada para la imagen. Para la conversión he usado la clase de java ColorSpace, que ya tiene estas tres operaciones implementadas, simplemente llamado a su getInstance(). En el caso de conversión a RGB he elegido la opción de CS_sRGB, para YCC he elegido la opción de CS_PYCC y para GRAY la de CS_GRAY, que se crearán dependiendo de cual se seleccione en el comboBox. Una vez elegido cual de las tres conversiones se va a realizar, he hecho uso de la clase de java ColorConvertOp, para crear la operación de conversión según la opción elegida con anterioridad. Tras esto se aplica el filtro a la imagen con la función filter() y el resultado se guarda en una nueva imagen que se muestra en una ventana.
12. He creado un botón para que el usuario pueda aplicar el la extracción de bandas cuando lo desee. Para la extracción de bandas he creado una función llamada getImageBand a la que se le pasa una imagen y un entero que referencia a una banda. Dentro de la función se hace uso de la clase de java ComponentColorModel, para crear el modelo de color, pasándole un espacio de color creado con la clase de java ColorSpace, dentro del espacio GRAY. Después se hace uso de la clase de java WritableRaster para crear un raster apartir del raster de la imagen original usando el método ya implementado por java de createWritableChild, para que los cambios realizados en el raster original se vean reflejados en el raster hijo. Tras esto, se crea la imagen con el raster y el modelo de color creados. Esta función se hace uso dentro de un ciclo que se repite tantas veces como bandas tenga la imagen seleccionada. Simplemente se crea una imagen para cada banda y se asigna a una nueva ventana.
13. He creado un botón para que el usuario pueda aplicar la operación de tintado cuando lo desee. Al pulsar el botón, se hace uso de la clase TintadoOp, incluida en la librería sm.image. En su constructor se le pasa el color con el que se le quiere hacer el tintado y el grado de mezlca. Tras esto simplemente se usa la

función filter() sobre la imagen. Para que el usuario pueda elegir el grado de mezcla he creado un deslizador junto al botón con el único evento de stateChanged, para que actualice el valor cuando la barra se mueve. Si no se toca la barra, el grado de mezcla por defecto que he puesto es de 0.5, es decir, la mitad.

14. He creado un botón para que el usuario pueda aplicar el filtro negativo cuando lo desee. Este filtro ya viene implementado en la librería sm.image, por lo que no hay que hacer ninguna operación para generar la tabla LookUp. Simplemente hay que usar la clase LookUpTableProducer de la librería y llamar a su función createLookupTable, indicándole que la tabla deseada es la correspondiente al filtro negativo. Tras esto llamo a la función aplicarLookUp, que aplica con la función filter() la operación sobre la imagen seleccionada.
15. He creado un botón de dos posiciones para que el usuario pueda elegir si aplicar los filtros incluyendo a las figuras o no. Para ello, he creado una función que es llamada cada vez que el usuario aplica algún tipo de filtro. Dentro de esta función se comprueba si el botón que he comentado antes está pulsado y en caso positivo se llama a una función al lienzo que vuelca las figuras sobre la imagen. Para que volcar las imagenes simplemente se crea una nueva imagen exactamente igual a la que hay en el lienzo y se coloca en él, cambiándo a la anterior. Esto hace que el programa reconozca a las figuras dibujadas como parte de la imagen original. Tras esto se borra todo el contenido del vector de figuras.
16. He creado un botón para que el usuario pueda duplicar la imagen cuando lo desee. La clase de java BufferedImage, usada para el manejo de imágenes, ofrece tres tipos de constructores diferentes, y en este caso, para crear la copia, nos interesa el constructor que pide el modelo de color (clase ColorModel de java), el raster (clase WritableRaster de java), un boolean isRasterPremultiplied, y una tabla hash, que la dejamos a null en este caso. Para conseguir todos estos atributos, primero recogemos la imagen del lienzo (volcando las figuras en la imagen para que se guarden en la copia) y vamos sacando los atributos con las funciones que nos ofrece java (getColorModel(), isAlphaPremultiplied y copyData). Tras la creación de la nueva imagen copia, la asignamos a una nueva ventana interna y la hacemos visible.
17. En el caso de las bandas, al ser un bucle for el que genera las bandas y las ventanas, cada vez que se genera una ventana he puesto en la función title() (de java) que ponga el nombre de la foto y entre corchetes “Banda:” y el número de la iteración del bucle. Para el espacio de color, cuando se genera con el filtro se crea un string con el espacio de color generado y al generar la ventana interna con la imagen en la función setTitle() he pasado el nombre de la imagen y entre corchetes el string con el espacio de color correspondiente.
18. He creado un botón para que el usuario pueda activar la operación cuando desee, y un deslizador al lado para que indique el valor de m mediante el evento stateChanged.

Ni java ni el paquete sm.image nos ofrece ninguna clase ni método para crear la operación cuadrática, por lo que he creado una función llamada cuadrática() a la que se le pasa como parámetro un double m.
La operación cuadrática responde ante la siguiente función:

$$f(x) = \frac{1}{100}(x - m)^2$$

donde m es un valor entre 0 y 255, que es el parámetro que indica donde se hace 0 la función (ya que tiene forma convexa, tocando el eje x en un punto, donde se hace 0).
Para poder realizar la operación, antes debo calcular la constante K, que será la constante de normalización para que el resultado de la operación de la función de entre 0 y 255. K se calcula con la operación 255/Max, siendo máximo el punto más elevado de la función. Al tratarse de una función que no crece de una forma monótona el valor máximo lo alcanzará en dos puntos diferentes: 0 si m es mayor o igual que 128, y 255 en el caso contrario. Por lo tanto para calcular el máximo usamos la función y colocamos en la x el valor 0 o 255, dependiendo de cuanto valga m. Tras esto ya podemos calcular K.
Para poder aplicar la operación necesitaremos una ByteLookUpTable (clase definida por java), la cual recibe una tabla de bytes. En esta tabla de bytes es donde tenemos que poner los valores que corresponderán a cada valor de x. Es decir, si la imagen original en x valía 20, en la tabla en la posición 19 (20-1) tendrá el valor del resultado que de la operación cuadrática. Para ello creamos un bucle for que itere de 0 a 255 y vamos asignando a la tabla (que es un array de byte) el valor que de la función, usando como x el número de la iteración y como m el valor pasado por parámetro.
Tras esto podemos crear el objeto de la clase LookUpTable (de java) que recibe como parámetro el ByteLookUpTable anterior y aplicar la tabla. Para ello creamos un objeto de la clase LookupOp (de java) que recibe como parámetro la tabla creada, que con la función filter() aplica la operación sobre la imagen.

19. Ni java ni el paquete sm.image nos ofrece algún método para la creación de esta operación, por lo que he creado una clase para ella. Aunque el paquete sm.image no incorpora esta operación, sí que incorpora una clase base para crear operaciones para las imagenes, la clase BufferedImageOpAdapter, que incluye varios métodos ya implementados y uno sin implementar, el método filter(), que es el que nos interesa. Por lo tanto, la clase de la operación posterización que he creado hace un extends de la clase del paquete sm.image BufferedImageOpAdapter. Esta operación responde a la función

$$comp\_posterizado = K * [comp\_original/K]$$

Siendo K = 256/N, y N un número entre 1 y 256, que indica el número de niveles al que se reduce la banda.
Por ello, dentro de la clase he creado una variable niveles que controle ese número, que será la que se le pase por el constructor, la elegida por el usuario en el deslizador.

Dentro de la función filter (que recibe la imagen original y la destino) creo el operador. Para ello, compruebo si la imagen destino es null, y en caso afirmativo creo una nueva imagen con la función createCompatibleDestImage, ya implementada en la clase BufferedImageOpAdapter. Tras esto solo debo seguir la estructura de las operaciones componente a componente (como es este caso), recorriendo con un triple bucle toda la imagen (eje X, eje Y y las bandas) y aplicando la función mencionada anteriormente. Para ello he creado la variable K (256/niveles) y la variable srcRaster de la clase WritableRaster para consefuir el comp_original de la función, con el método getSample() de su clase, volcando el resultado en otra variable llamada sample. Este resultado se le aplica a la imagen destino con el método setSample.

20. Como en el caso de la operación de posterización, ni java ni el paquete sm.image ofrece algún método para aplicar esta operación, por lo que he creado una clase para ella, llamada RedOp, que también extiende a la clase BufferedImageOpAdapter.

    En este caso, la operación responde a la siguiente fórmula:

$$g(x,y) = \begin{cases} f(x,y) & si\ \ R_f - G_f - B_f \geq T \\ \dfrac{R_f + G_f + B_f}{3} & en\ otro\ caso \end{cases}$$

    donde f(x,y) es la imagen original y g(x,y) es la imagen destino. Es decir, dado un umbral T, si en un píxel predomina el rojo (R es mayor que G+B) el píxel se queda como en la imagen original. Si no predomina el color rojo, el píxel se lleva al gris, con el valor medio de las tres bandas.

    Para llevar a cabo esta operación he seguido el mismo esquema que en la operación de posterización, creando una variable para el umbral también, con la diferencia de que esta operación es una operación píxel a píxel, no componente a componente. Esto significa que vamos a necesitar la información de cada píxel (que se consigue creando una array de enteros que guarde la información de los valores de las bandas de cada píxel, gracias a la función getPixel de la clase WritableRaster). También siginifica que será necesario un bucle doble, para recorrer cada píxel de la imagen. Dentro de este bucle doble se comprueba cual es el resultado de R-B-G, y si es mayor o igual que el umbral, se iguala el array de píxeles original con el array de píxeles destino. En caso de que sea menor, a cada componente del array se le asigna el valor devuelto por la operación matemática (R+G+B)/3. Tras esto, se aplica el resultado a la imagen destino con el método setPixel sobre el WritableRaster de la imagen destino. Estas operaciones se hacen para cada píxel de la imagen.

    Para que el usuario pueda activar esta operación he creado un botón que crea un objeto de esta clase y le aplica el filtro con un valor de umbral de 30.

21. **[Píxel a píxel propio]** Para la creación de la clase para la operación píxel a píxel propio he seguido los mismos pasos que los explicados anteriormente, heredando de la clase BufferedImageOpAdapter del paquete sm.image.
En este caso la operación responde a la siguiente función:

$$g(x,y)\begin{cases} 130\%\ f(x,y) & si\ Rf+Gf+Bf \geq T \\ 70\%\ f(x,y) & en\ otro\ caso \end{cases}$$

donde g(x,y) es la imagen destino y f(x,y) es la imagen final. Es decir, dado un umbral T, si la suma de los componentes R+G+B es mayor a ese umbral, esos componentes ganarán en claridad un 30% (es decir, se multiplicarán por 1.3, nunca sobrepasando el valor 255 cada componente). Si por el contrario, no se consigue pasar ese umbral sumando los tres componentes, esos componentes perderán un 30% de claridad. Con esto lo que se intenta conseguir es un resaltado de las sombras de la imagen, dependiendo de el umbral será un cambio más brusco o más suave.
Para llevar a cabo estas operaciones he seguido el esqueleto de la operación píxel a píxel anterior (creación de arrays, doble bucle, variable umbral, etc.). También he creado una variable suma, que, dentro del doble bucle, recorre cada píxel sumando sus componentes. Comprueba si la suma es mayor que el umbral, y en caso afirmativo se aplica el valor del componente actual al valor del componente destino, multiplicándolo por 1.3, y se comprueba si este valor supera el 255. Si lo supera, se deja a 255, si no, se deja el resultado de la multiplicación. En caso de que no supere la suma el umbral, se hace la misma multiplicación pero por 0.7.
Dependiendo del umbral el efecto puede variar mucho, así que he creado un botón que lanza un diálogo donde el usuario puede elegir el número de este umbral, y posteriormente se aplica la operación a la imagen.
Ejemplo en imágenes:

Imagen original.

Imagen con umbral 100.

Imagen con umbral 350.

Como se puede comprobar, si se aplica la operación con un umbral pequeño, solo entran en juego los píxeles más oscuros, por lo que realza las sombras dejándolas más oscuras y las zona de color adquieren colores más vivos. Sin embargo, si el umbral es mayor, solo los píxeles con colores más vivos y claros se salvan, dejando un efecto en el que gran parte de la imagen es muy oscura, y resaltando muchísimo los colores más vivos y claros, como se puede comprobar observando la rosa, a la que casi se le ve solo el contorno.

Para entender este efecto hay que tener claro que, en una imagen RGB (por ejemplo), cada componente de cada píxel puede tener un valor de 0 a 255. Entonces, si la suma de los tres valores es pequeña, el píxel correspondiente a esos tres componentes es oscuro, mientras que si es muy alto, el píxel es muy claro o tiene colores muy vivos. Con esto es con lo que juega esta operación.

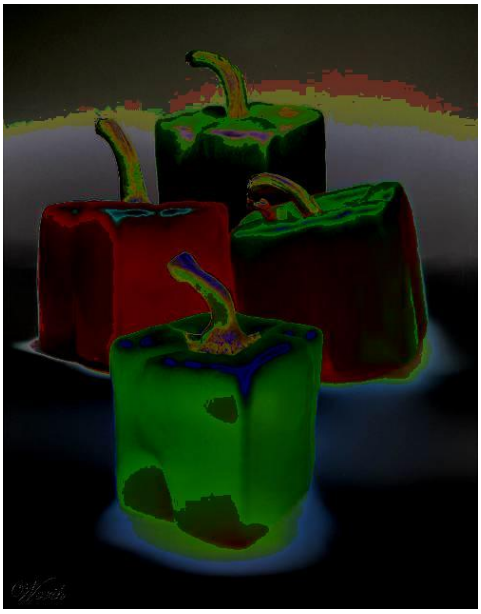
22. [**Lookup propio]** Para la creación de la operación LookUp de diseño propio he querido implementar en la imagen una especie de semi-negatividad con oscurecimiento. La función a la que correspondería esta operación sería la siguiente:

$$y\begin{cases} x-50 & si\ x < m \\ (255-x)-50 & si\ x \geq m \end{cases}$$

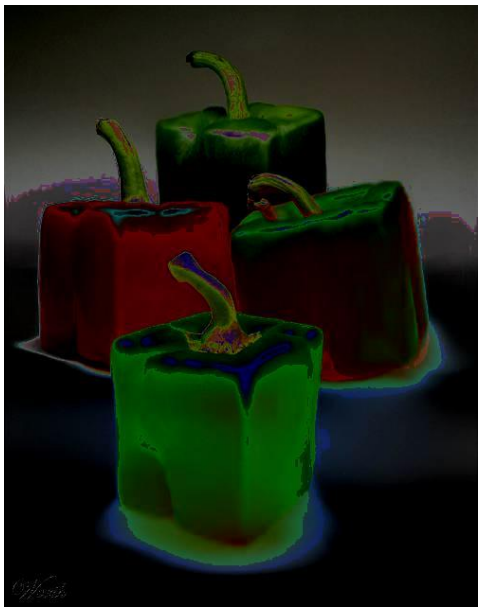
donde m es un valor entre 0 y 255 que indica donde fluctúa la gráfica. La representación gráfica es:

El punto donde la gráfica deja de crecer y empieza a decrecer sería m.
Para poder aplicar esta operación es necesario crear un objeto de la clase ByteLookupTable, por lo que he creado una función que lo crea, la cual recibe como parámetro la m. Para crear una operación Lookup es necesario generar una tabla de 256 en la que se recoja el valor que va a tener y en cada valor x. Es decir, en la gráfica anterior, todo valor de x por debajo de 50 tendrá un valor de 0. Después, cada y tendrá el valor de x-50 hasta el punto m. Apartir de ese punto, cada valor de y tendrá el de (255-x)-50, hasta llegar a 0.
Para poder construir esta tabla he creado un array de bytes de 256 posiciones. He creado un bucle for desde 0 hasta m para la parte creciente donde aplico a cada posición y la operación x-50, y otro bucle for desde m hasta 255 donde aplico para cada y la operación (255-x)-50. Para que el usuario pueda elegir el valor de la m, al pulsar el botón del filtro, se lanza un diálogo donde el usuario puede poner la m.
Esta tabla se le aplica al ByteLookupTable, y este a la clase LookupOp, tras lo cual se usa la función filter() para aplicar la operación sobre la imagen.
El resultado en imágenes sería este:

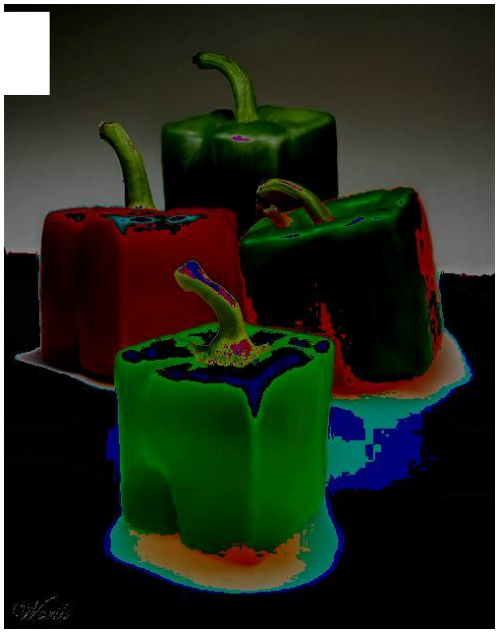
Imagen original



m=100



m=150



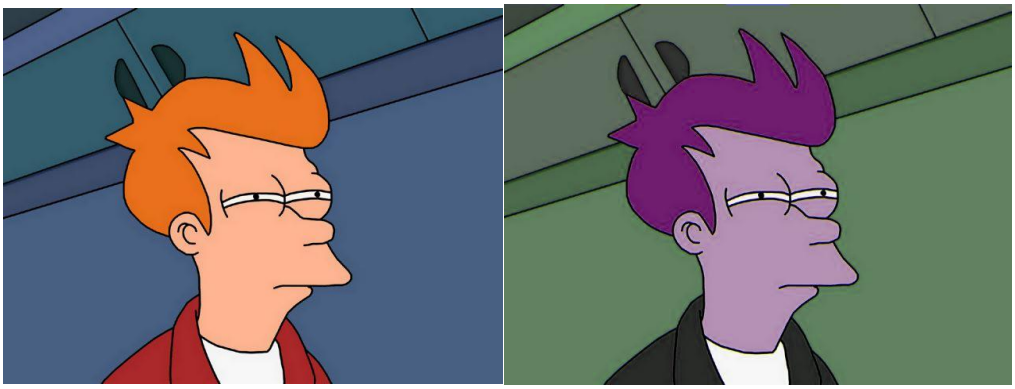
m=200

Se puede comprobar comprobar que en todas las imágenes hay un grado de oscurecimiento continúo, pero que, dependiendo del umbral m, la semi-negatividad se verá afectada de una forma u otra.

23. **[Opcional]** He creado una variable que guarde el umbral que se le pasará a la clase RedOp para la operación de resaltado del rojo. He creado al lado del botón de esta operación un deslizador donde cambiar el valor de este umbral. El deslizador está incializado al valor 30, y puede ir de 10 a 100.

24. **[Opcional]** He creado un botón para que el usuario pueda enverdecer la imagen cuando lo desee. Para ello, al pulsar el botón se llevará a cabo una operación de combinación de bandas. Para ello es necesario usar la clase de java BandCombineOp, a la que se le debe pasar una matriz. La estructura de esta matriz es simple, es de 3x3, cada fila y columna corresponde a una banda y se le puede asignar los valores deseados. Para esta operación he optado por quitar la banda del rojo, la del verde dejarla al 100%, y la del azul dejarla al 70%. He llegado a esta conclusión después de varias pruebas, con esta configuración ha quedado un resultado más deseable:

25. **[Opcional]** Mientras desarrollaba mis funciones de diseño propio, me entró curiosidad por ver cuál sería el resultado si en cada píxel intercambio los valores de los componentes. Es decir, a un píxel le cambio su valor en el componente R por el del valor del componente G. En el componente G pongo el valor de B y en el B el de R (hablando de imagenes RGB). Aunque es una operación extremadamente sencilla, puede tener resultados curiosos, por ello la he incluido aquí. A continuación muestro un ejemplo:



Análisis de requisitos de sonido

1. Lo deja hacer java directamente, ya que guardamos el archivo abierto en un objeto tipo File, que tiene una función getName(), que devuelve el nombre, y ese resultado se lo pasamos a la función de la ventana interna setTitle, función que también viene ya definida en java para las ventanas internas.

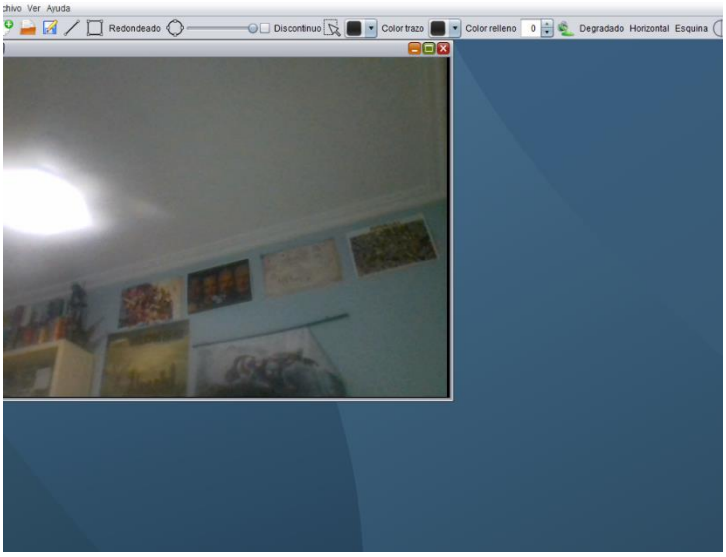
2. He añadido a la barra de herramientas 3 botones de los dados por java, les he quitado el texto y les he asignado su correspondiente icono (del paquete iconos). Para la lista, he añadido un comboBox, también dado por netbeans, he eliminado su contenido por defecto y he cambiado el tipo de objeto que almacenaba (String) por el tipo File.
3. Al abrir un fichero de audio lo he añadido al combo box con la función addItem, seguida de la función setSelectedItem (para que salga como seleccionado el último fichero abierto), a las que ambas se le pasa el fichero. Ambas funciones ya vienen en java para los combo box. El problema que nos ocurre aquí, es que la función addItem(f) usa la función toString del File que se le pasa como parámetro, y dicha función devuelve el path completo del fichero, quedando como resultado una visualización poco amigable. Para subsanar este problema, simplemente he utilizado la opción propuesta en el guión de prácticas sobre sonido, usar una clase anónima sobreescribiendo el File f que ya estaba creado (f = new File(path)). Al hacer esto, se puede sobrecargar la función toString de la clase File, para indicar que solo queremos el nombre, no el path completo.
4. He hecho uso de la biblioteca proporcionada (smsound). He creado un objeto SMClipPlayer, que está inicializado a null. Dentro del actionPerformed del botón de reproducir simplemente he cogido el elemento de la lista con la función getSelectedItem, propia de java para los comboBox, y este fichero se lo he pasado al constructor para asignarlo al objeto SMClipPlayer creado anteriormente. Tras esto simplemente he llamado a la función play, que viene en la biblioteca para SMClipPlayer.
5. He vuelto a usar la biblioteca proporcionada smsound. Simplemente en el actionPerformed el botón parar compruebo si el objeto SMClipPlayer, explicado en el apartado anterior, no es nulo, y si no lo es, llamo a la función stop() del SMClipPlayer y lo dejo nulo.
6. He usado la biblioteca proporcionada SMSoundRecorder, y he creado un objeto tipo SMSoundRecorder, inicializado a null, llamado recorder. Dentro del actionPerformed del botón de grabar he usado el diálogo de guardado de archivo, ya implementado por java, y tras esto le he pasado el archivo creado por el usuario al objeto recorder, para inicializarlo, y si tras esto no está nulo, llamo a la función record() del objeto recorder, ya implementada en la biblioteca SMSoundRecorder.
7. He seguido unos pasos parecidos a los pasos seguidos para cumplir el requisito 4 de los requisitos de sonido, con la diferencia de que los pasos se realizan sobre el objeto recorder.
8.

Análisis de requisitos de video

1. Lo deja hacer java directamente, ya que guardamos el archivo abierto en un objeto tipo File, que tiene una función getName(), que devuelve el nombre, y

ese resultado se lo pasamos a la función de la ventana interna setTitle, función que también viene ya definida en java para las ventanas internas.

2. Para reproducir el vídeo, al dar al botón de play se comprueba que la ventana seleccionada contiene un vídeo (es decir, es de tipo VentanaInternaVideo). Si es así, se llama a la función de la clase de la ventana llamada play(). Dentro de esta clase se ha creado un objeto de la clase EmbeddedMediaPlayer, que hace de reproductor. Esta clase está entregada en las bibliotecas proporcionadas sobre vcl. En la función play() de la clase de la ventana se llama a la función play() del objeto reproductor, o, en caso de que no se pueda reproducir, a la función playMedia() del objeto reproductor a la que se le pasa la ruta del archivo de vídeo. Para parar el vídeo, al dar al botón de parar también se comprueba si la ventana seleccionada contiene un vídeo, y si es así llama a la función de la ventana llamada stop(). Dentro de esta función, en la clase VentanaInternaVideo, se comprueba si el objeto reproductor está reproduciendo el vídeo con la función isPlaying(). Si es así, lo pausa con la función pause(), y si no, lo para con la función stop(). La diferencia entre estas dos funciones es que con pause(), se puede reanudar el vídeo por donde iba, y con stop() se vuelve al principio. Estas tres funciones también vienen implementadas en las bibliotecas proporcionadas.
3. He creado un botón que al pulsarlo crea una ventana interna de tipo VentanaInternaCamara, con border layout que muestra la imagen recogida por la webcam. Para ello se crea un objeto de la clase Webcam, dada en las librerías proporcionadas, a la que se le asigna la cámara elegido. Tras esto se crea un area visual con la clase WebcamPanel, también dada en las bibliotecas, y se asigna a la ventana.



4. He creado un botón para que el usuario pueda tomar capturas cuando lo desee. Cuando se pulsa, se comprueba que tipo de ventana es la que está seleccionada, y si es de tipo cámara o vídeo se crea una nueva imagen, se le asigna el return de la función getImage() de la ventana y se crea una nueva ventana de tipo imagen a la que se le asigna la captura. En el getImage() de la

ventana de tipo cámara se hace uso de la función getImage() de la clase Webcam, función ya implementada en la librería. Para el vídeo, se hace uso de la función getSnapshot() de la clase EmbeddedMediaPlayer, también implementada en la librería.

5. Al crear la ventana para la captura se llama a la función setTitle() de java y se indica que el título sea “Captura”.
6. He creado un comboBox que acepte objetos de tipo Webcam, donde aparece un listado de las webcams que el ordenador tiene conectados. Para ello, he hecho uso de la librería dada para los requisitos de vídeo, que tiene una clase Webcam y un método llamado getWebcams() que devuelve una lista de webcams en el sistema.
7. Para ello he modificado la clase de VentanaInternaCamara, para que en el constructor (y en su función getInstance()) haya que pasarle un objeto Webcam, que será la que esté seleccionada en el comboBox.
8. He creado un comboBox que contiene strings. He personalizado el código con la opción que nos da java y he añadido las resoluciones de la cámara que esté seleccionada en el comboBox de las webcams. El string de la resolución lo he formado con las funciones getWidth() y getHeight() que trae la clase Dimension. El índice que esté seleccionado se le pasa al getInstance cuando se crea la ventana interna de la cámara, para que pueda asignar la resolución adecuada. Esto es posible gracias a las funciones getViewSizes() y setViewSize() de la clase Webcam.

Diseño

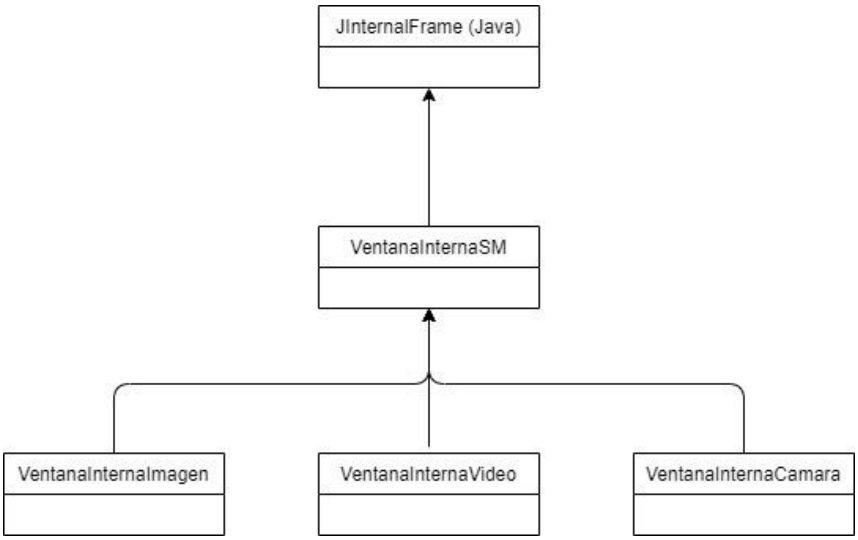
En este apartado explicaré qué clases propias he tenido que crear, el porqué he tenido que crearlas y el como están estructuradas. Para analizarlo lo iré analizando por bloques.

Nota: Todas las clases estarán explicadas más detalladamente en el documento de documentación de codificación.

Clases ventana interna

Dado que la aplicación debía funcionar con ventanas internas para mostrar imagenes, vídeos, o la imagen que este capturando la webcam, he tenido que crear diferentes clases para cada funcionalidad. Esto es debido a que Java solo nos ofrece la opción de crear ventanas internas genéricas, por lo que es obligatorio crear estas clases.

Para llevar a cabo toda la funcionalidad descrita en los requisitos, he optado por crear una clase VentanaInternaSM “padre”, que hereda de JInternalFrame, la clase que java incluye para las ventanas internas. De esta clase padre, heredarán las otras clases de ventana interna: VentanaInternaImagen, VentanaInternaVideo y VentanaInternaCamara.



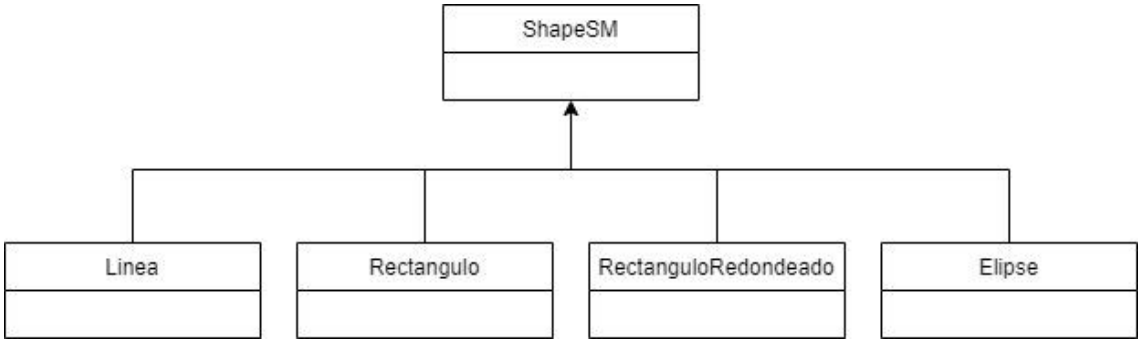
De estas clases solo cabe destacar:

- VentanaInternaSM:
  - Lleva un constructor sin parámetros.
- VentanaInternaImagen:
  - LLeva un constructor sin parámetros.
  - Tiene asignado un lienzo (de la clase propia Lienzo2D).
  - Solo tiene como funciones el getter y el setter del lienzo.
- VentanaInternaVideo:
  - Cuando se llame a esta clase no se hará mediante el constructor, si no mediante el método getInstance().
  - Tanto el método getInstance y el constructor reciben como parámetros el archivo a reproducir.
  - Tiene tres funciones: una para reproducir(play()), una para parar la reproducción (stop()) y una para devolver una captura del vídeo (getImage()).
- VentanaInternaCamara:
  - Cuando se llame a esta clase no se hará mediante el constructor, si no mediante el método getInstance().
  - Tanto el método getInstance y el constructor reciben como parámetros la webcam que debe activarse y la resolución a la que debe estar la cámara.
  - Como funciones tiene el getter y setter de la cámara y un getImage() que devuelve una foto tomada por la cámara.

Clases figuras

Uno de los requisitos de la aplicación era que cada figura debía tener sus propios atributos (color, trazo, etc). Esto me ha obligado a tener que implantar una estructura de clases para poder crear figuras independientes. Esto es así porque la clase que implementa java para las figuras (la clase Shape, o cada clase particular de cada figura), no guarda atributos para cada figura.

Para poder listar y comprobar que figuras se pueden crear he creado un enumerado llamado Formas. Para las figuras, he creado una clase "padre", llamada ShapeSM, de la que hereda cada clase de cada figura particular: Linea, Elipse, Rectangulo y RectanguloRedondeado. He optado por que la clase ShapeSM no herede de ninguna clase ni interfaz de java.



Para la creación de las figuras he decidido hacerlo de la siguiente forma: en la clase ShapeSM habrá un constructor sin parámetros y una lista de variables con todos los atributos que una figura puede tener (color, color del relleno, trazo, si es relleno o no, etc). Dentro de esta clase habrá getters y setters de todos los atributos, y un método paint al que se le pasa un Graphics2D (llamémoslde g2d). Dentro de este paint se le asignará al g2d los atributos que pueden tener todas las figuras (es decir, el color del trazo, el trazo, la transparencia). En las clases de las figuras, tendrán como constructor uno que reciba los parámetros únicos de esa figura que sean los más importantes (es decir, si debe ser rellenada o no, el color del relleno, si debe alisarse) y además unos getters and setters de todos los atributos, incluyendo algunos únicos pero más "secundarios" (por ejemplo, si el relleno debe ser degradado, si el degradado es horizontal o vertical, etc). Y en el paint de estas figuras, añadirle al g2d los atributos únicos de estas figuras.

Con esto, lo que pretendo conseguir es que, a la hora de la creación de las figuras, en una línea aparezca el constructor, corto, con pocos parámetros, pero que fije los atributos básicos de la figura, así además es más fácil ver a primera vista si la figura se puede rellenar, si se puede alisar, etc. Al crear la figura, tras el constructor aparecerá una lista con setters comunes a todas las figuras, fijando el color, el trazo, y demás atributos, y tras esto una lista de setters de los atributos únicos, pero más secundarios, de cada figura. Todo esto vendrá explicado con más detalles en la codificación.

Particularidades a destacar de las clases de figuras:

- Linea:
    - Para la creación de esta figura he hecho uso de la clase Line2D, de java, que se inicializa en el constructor.
    - Al no tener ningún atributo que la diferencie de los demás, el constructor no lleva parámetros.
    - Para las funciones de obtención o modificación de la línea (getX, setLocation, etc) he usado las propias de la clase Line2D de java (más detalles en codificación).

    - En el paint de la clase solo ha sido necesario llamar a la función draw().
- Elipse:
    - Para la creación de esta figura he hecho uso de la clase Ellipse2D, de java, que se inicializa en el constructor.
    - Esta figura tiene de "especial" que puede ser rellenada y alisada, por lo que en el constructor recibe dos booleanos para especificar estos atributos y el color de relleno.
    - Para las funciones de obtención o modificación de la elipse (getX, setLocation, etc) he usado las propias de la clase Ellipse2D de java (más detalles en codificación).
    - En el paint de esta clase ha sido necesario ir comprobando los atributos booleanos comentados anteriormente, para asignar al g2d unos valores u otros.
- Rectangulo:
    - Para la creación de esta figura he hecho uso de la clase Rectangle2D, de java, que se inicializa en el constructor.
    - Esta figura tiene de "especial" que puede ser rellenada, por lo que en el constructor recibe un booleano para especificar este atributo y el color de relleno.
    - Para las funciones de obtención o modificación del rectángulo (getX, setLocation, etc) he usado las propias de la clase Rectangle2D de java (más detalles en codificación).
    - En el paint de esta clase ha sido necesario ir comprobando los atributos booleanos comentados anteriormente, para asignar al g2d unos valores u otros.
- RectanguloRedondeado:
    - Para la creación de esta figura he hecho uso de la clase RoundRectangle2D, de java, que se inicializa en el constructor.
    - Esta figura tiene de "especial" que puede ser rellenada y alisada, por lo que en el constructor recibe dos booleanos para especificar estos atributos y el color de relleno.
    - Para las funciones de obtención o modificación del rectángulo (getX, setLocation, etc) he usado las propias de la clase RoundRectangle2D de java (más detalles en codificación).
    - En el paint de esta clase ha sido necesario ir comprobando los atributos booleanos comentados anteriormente, para asignar al g2d unos valores u otros.

Clase Lienzo2D

Para poder pintar las figuras he tenido que crear una clase lienzo, que hereda de la clase JPanel de de Java. Esta clase es la encargada de delimitar el espacio de dibujado y de representar las figuras y las acciones que el usuario haga.

Esta clase es la encargada de crear las figuras, por lo que tiene todos las variables para cada uno de los atributos posibles de las figuras, junto con sus getters y setters. También tiene la función createShape, donde se crean las figuras y se les asignan sus atributos.

El constructor de la clase no recibe ningún parámetro.

Esta clase también es la encargada de representar la imagen que el usuario abra, por lo que tiene la variable necesaria, junto a sus getters y setters.

También es la encargada de cambiar los atributos de la figura a editar, por lo que están las funciones correspondientes, llamadas update.

Una de las funciones más destacables de la clase es la función paint(). Esta función es propia de la clase JPanel de java, y aquí la sobreescribo para que pueda pintar la imagen y las figuras. Dentro de esta función solo se incluye el drawImage, para pintar la imagen, la función clip(), para delimitar el área de dibujado, crear el marco para la figura seleccionada, con la función de método propio crearMarco() y un bucle for que recorre el array de figuras y llama al paint de cada figura.

Todas las funcionalidades de esta clase estarán descritas en la codificación.

Clase ColorCelRender

El objetivo de esta clase es únicamente que los colores sean vistos de forma apropiada en los comboBox. Para ello la clase implementa a ListCellRenderer<Color> y sobrescribe su función getListCellRendererComponent. Aquí lo único que se hace es determinar como serán los botones de cada color.

Clase PosterizarOp

Uno de los requisitos era que el usuario podía aplicar en cualquier momento la operación de posterización. Para ello, como esta operación no la incluye ni java ni la biblioteca sm.image, he tenido que crear una nueva clase. Lo que se incluye en esta clase y lo que hace se puede comprobar en el punto de análisis referente al requisito de operación de posterización.

Clase RedOp

Uno de los requisitos era que el usuario podía aplicar en cualquier momento la operación de resaltado del color rojo. Para ello, como esta operación no la incluye ni java ni la biblioteca sm.image, he tenido que crear una nueva clase. Lo que se incluye en esta clase y lo que hace se puede comprobar en el punto de análisis referente al requisito de operación de resaltado del rojo.

Clase ResaltadoSombrasOp

Esta clase es la referente a la operación píxel a píxel de creación propia, uno de los requisitos de la práctica. Lo que se incluye en esta clase y lo que hace se puede

comprobar en el punto de análisis referente al requisito de operación píxel a píxel de creación propia.

Clase FuncionPropia

Esta clase es la referente a la función propia explicada en el requisito opcional. Lo que se incluye en esta clase y lo que hace se puede comprobar en el punto de análisis referente al requisito de operación.