UNIVERSIDAD POLITECNICA DE VALENCIA

ESCUELA POLITECNICA SUPERIOR DE GANDIA

I.T. Telecomunicación (Sonido e Imagen)

"Desarrollo de un software para la edición de imágenes digitales."

TRABAJO FINAL DE CARRERA

Autor/es:
David Grau López

Director/es:
Herranz Herruzo, José Ignacio

GANDIA, 2013

CONTENIDO

0. OBJETIVOS 6
1. INTRODUCCIÓN AL TRATAMIENTO DIGITAL DE IMAGEN 7
   1.1. Imagen digital 7
   1.2. El ojo humano 7
   1.3. Modelando el ojo humano 9
   1.4. Adquisición de la imagen 10
   1.5. Color 11
   1.6. Almacenando el color en una imagen digital 13
   1.7. Segmentación de imágenes 14
   1.8. Aplicaciones 14
2. FUNDAMENTOS TEÓRICOS 16
   2.1. Realzado de imágenes 16
      2.1.1. Realzado por procesado del punto 16
      2.1.2. Filtrado espacial 19
      2.1.3. Filtros para segmentación 23
      2.1.4. Filtros morfológicos 26
      2.1.5. Diferenciación de objetos 30
3. DESCRIPCIÓN DEL SOFTWARE 33
   3.1. Clases utilizadas 34
   3.2. Interfaz de usuario 36
   3.3. La clase utilidades 38
      3.3.1. Variables 40
      3.3.2. Funciones varias 41
   3.4. Procesado de imágenes 48
      3.4.1. Negativo 48
      3.4.2. Ecualizar histograma 48
      3.4.3. Brillo 49
      3.4.4. Contraste 50

3.4.5. Margen dinámico ... 51
3.4.6. Umbral simple ... 51
3.4.7. Umbral doble ... 52
3.4.8. Voltear horizontal ... 52
3.4.9. Voltear vertical ... 53
3.4.10. Rotaciónes fijas ... 53
3.4.11. Rotación libre ... 54
3.4.12. Cambiar tamaño del lienzo ... 55
3.4.13. Redimensionar imagen ... 56
3.4.14. Filtros predefinidos ... 59
3.4.15. Mediana ... 60
3.4.16. Filtro de usuario ... 60
3.4.17. Gradientes ... 62
3.4.18. Erode ... 62
3.4.19. Dilate ... 64
3.4.20. Close ... 65
3.4.21. Open ... 65
3.4.22. Cálculo de áreas ... 66
3.4.23. Cálculo de perímetro ... 66
3.4.24. Compacidad ... 66
3.4.25. Ruido ... 67
3.5. Dibujando ... 68
3.5.1. Paleta de colores ... 68
3.5.2. Herramientas de dibujo ... 68
3.5.3. Portapapeles ... 68
3.6. Formatos de imagen ... 69
3.6.1. Procesado por lotes ... 69
3.7. Captura ... 70
3.7.1. Captura de pantalla ... 70
3.7.2. Captura por webcam ... 70
3.8. Transformada de Fourier ... 71
3.9. Configuración ... 73

4. CONCLUSIONES 74
5. LÍNEAS FUTURAS 75
6. BIBLIOGRAFÍA 76

0. OBJETIVOS

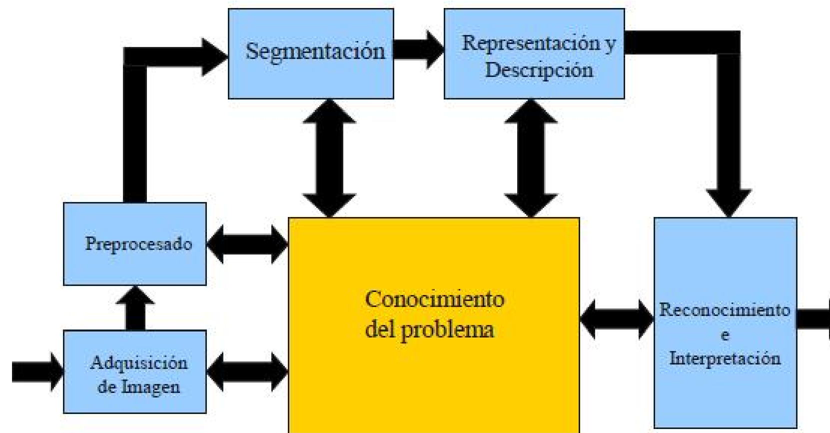
El primer objetivo de este proyecto es adquirir un conocimiento más profundo sobre el tratamiento de imagen digital y sobre programación en Windows y su API.

El otro objetivo es desarrollar un programa similar al utilizado en las prácticas de la asignatura Tratamiento Digital de Imagen (PDI32) pero adaptado a los nuevos sistemas operativos y que aproveche las capacidades y los recursos de los sistemas actuales.

PDI32 es una buena herramienta pero fue programada para sistemas operativos antiguos lo cual hace que se produzcan errores durante su uso sobre los actuales.

Esta versión utiliza un nuevo sistema de menús, más actual y accesible además de ayuda en formato CHM, compatible con sistemas a partir de Windows Vista.

1. INTRODUCCIÓN AL TRATAMIENTO DIGITAL DE IMAGEN



1.1. Imagen digital

Una imagen digital es una imagen que ha sido discretizada tanto en sus coordenadas espaciales como en su luminosidad. El resultado es una función *f(x,y)* donde x, y son las coordenadas espaciales y el valor de la función es el nivel de luminosidad en este punto.

Cuando se trata de una imagen en color tendremos tres planos de color para rojo, verde y azul.

Esta imagen se puede considerar como una matriz en la que se guarda el nivel de luminosidad. Cada uno de estos puntos se llamará *pixel.*

El tratamiento de la imagen sigue tres pasos:

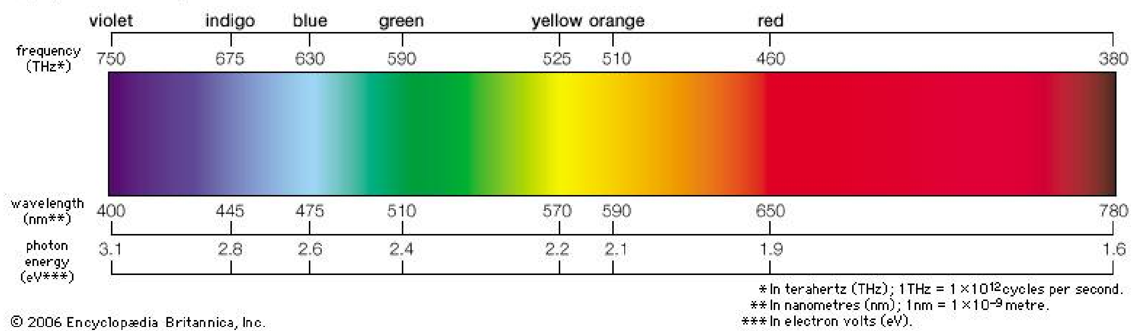
- Adquisición.
- Procesado.
- Representación.

1.2. El ojo humano

Primero hay que ver un poco como funciona la visión humana. Saber cómo se perciben las imágenes será de ayuda a la hora de tratar con imágenes digitales.

La luz visible es una radiación electromagnética cuya longitud de onda corresponde a una zona limitada del espectro electromagnético.

Las longitudes de onda normales en nuestra visión oscilan entre 390nm y 780nm.



La luz tiene tres dimensiones básicas:

- Intensidad: Es el brillo que se recibe o el flujo luminoso que despide o refleja un cuerpo.
- Frecuencia: El ojo la percibe como color.
- Polarización: O ángulo de vibración. No es perceptible por los humanos.

Los ojos pueden distinguir miles de colores diferentes pero sin embargo solo unas dos docenas de niveles de gris.

La luz atravesará el iris, que actúa como diafragma para limitar el brillo. Atraviesa la lente (cristalino) que tenemos en los ojos llega a los receptores que están alojados en la retina.

Los receptores de color son los conos de los que tenemos de 6 a 7 millones (muchos menos que bastones) y cada uno está conectado a su propia terminación nerviosa.

Para recibir el brillo se utilizan los bastones entre 75 y 150 millones. Tienen menos detalle y suelen compartir terminaciones nerviosas.

La principal diferencia entre la lente de un ojo y una lente óptica de una máquina es que la lente del ojo es flexible. Por medio de los músculos del ojo esta lente se hará más o menos plana para enfocar los objetos según la distancia a la que estén.

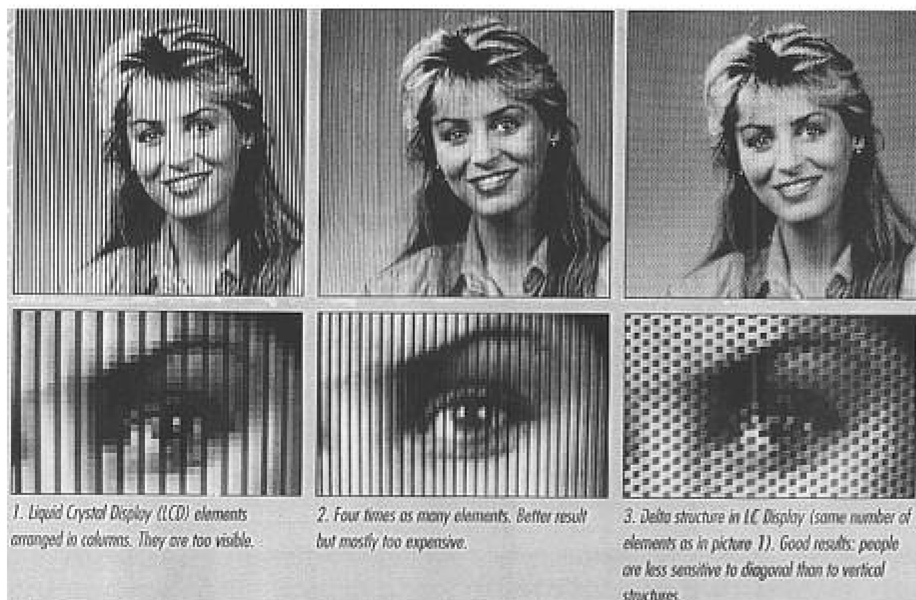
1.3. Modelando el ojo humano

El ojo se puede modelar como un sistema lineal invariante en el tiempo *LIT.*

Para empezar vamos a ver el comportamiento del ojo en el tiempo. El ojo es incapaz de distinguir sucesiones de imágenes muy rápidas. Al observar su respuesta en frecuencia se puede ver que corresponde con un filtro paso bajo con la frecuencia de corte entre 24 y 30 imágenes por segundo. Esta respuesta en frecuencia puede variar según el nivel de brillo que nos llega. Si el nivel de brillo es bajo la respuesta en frecuencia es menor.

Una buena forma de entender el comportamiento espacial del ojo es fijarnos en el funcionamiento de las pantallas de cristal líquido *LCD.*

Estas pantallas están compuestas por cristales. En sus inicios los cristales se ponían en vertical, el resultado era una imagen mezclada con líneas verticales. Se encontraron dos soluciones. Una era aumentar la frecuencia de la pantalla y reducir el tamaño de los cristales. La segunda, y más barata, era disponer los cristales en diagonal de forma que quedaran agrupados en hexágonos.



1. Liquid Crystal Display (LCD) elements arranged in columns. They are too visible.
2. Four times as many elements. Better result but mostly too expensive.
3. Delta structure in LC Display (same number of elements as in picture 1). Good results: people are less sensitive to diagonal than to vertical structures.

La nitidez es la ausencia de ruido, el cual existe tanto a altas como a bajas frecuencias. Los valores que más interesan están a baja frecuencia por lo que se suelen filtrar las imágenes pero es curioso que el ojo humano prefiere una imagen con un poco de ruido que una imagen filtrada desde un principio, de forma que es más cómodo a la hora de ver una imagen que se filtre solo si esta tiene mucho ruido.

Así sabemos que el ojo humano tiene una cierta tolerancia al ruido. Y que tiene limitaciones en cuanto a resolución espacial.

1.4. Adquisición de la imagen

La adquisición se hace por medio de transductores que manipulan y finalmente captan la imagen. Por ejemplo: CCD, ojo humano, película fotográfica...

Los transductores introducen ruido en la imagen, además poseen una resolución limitada y solo pueden hacer un número limitado de tomas.

En el caso digital tenemos una matriz de transductores que tienen que hacer un muestreo espacial de la imagen y luego cuantificar los niveles de brillo convirtiéndolos en niveles de gris.

El muestreo es el proceso de convertir una señal en una secuencia numérica. Esta secuencia se puede convertir (reconstrucción) en una señal parecida a la original si está limitada en frecuencia y la frecuencia de muestreo es el doble del ancho de banda de la señal original.

La cuantificación consiste en convertir los valores de amplitud de la señal en valores numéricos conocidos.

El muestreo se debe hacer con una misma separación entre las muestras tomadas para poder recuperar la señal original. En el caso de imágenes que se hace un muestro espacial (en dos dimensiones) la distancia entre cada muestra será igual para asegurarnos de que los pixeles son cuadrados y del mismo tamaño.

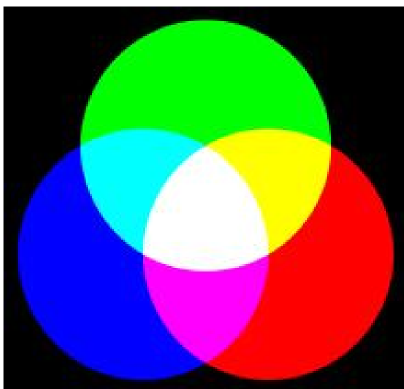
Entre otros objetivos el procesado de la imagen intenta eliminar el máximo de ruido y conseguir una imagen más definida.

Otro factor importante a la hora de adquirir imagen es el rango dinámico, que es la diferencia entre la máxima intensidad de brillo medible y la intesidad mínima detectable. Si se supera, se llega a la saturación, y si se queda por debajo, encontraremos ruido.

Podemos encontrar las siguientes propiedades de los colores:

- Radiancia (watios): Cantidad total de energía que sale de la fuente luminosa.
- Luminancia (lumens): Cantidad de energía percibida por el observador.
- Brillo: Descriptor que da una idea de la noción cromática de la intensidad.
- Tono: Atributo asociado con la longitud de onda (λ) dominante en una mezcla de ondas lumínicas. Sensación de color.
- Saturación: Pureza relativa o cantidad de luz blanca mezclada con un tono, de forma que un color puro estará completamente saturado de color (sin luz blanca).

La síntesis aditiva es obtener un color mediante la suma de colores.



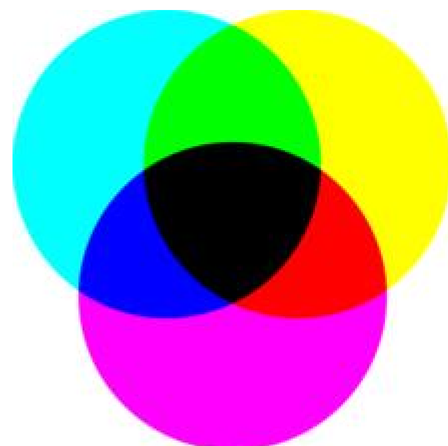
Al sumar los colores rojo, verde y azul se obtiene el color blanco. La ausencia del color dará el negro. Si se suman con diferentes valores de brillo se pueden conseguir el resto de colores.

Si se suman con la misma intensidad se producen los colores secundarios cian, magenta y amarillo.

Los colores primarios no son una propiedad de la luz. Se debe a como los ojos captan los colores ya que los conos tienen receptores específicos para rojo, verde y azul.

Otro ejemplo de uso de estos colores son los monitores y las televisiones.

Esto es la síntesis sustractiva. Todo color que no es aditivo será sustractivo o también se puede decir que todo lo que no es luz directa es luz reflejada por un objeto.

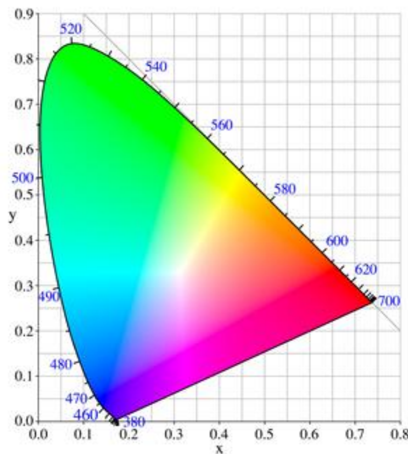


Se utiliza para los pigmentos con los que se pintan los objetos. El color con el que se ven los objetos depende de que partes del espectro son absorbidas por el propio objeto.

Se parte del color blanco (no se quita energía

de radiación). Los colores primarios en este caso son cian, magenta y amarillo. Estos actúan como filtros que impiden pasar ciertos tonos de luz.

Este tipo de síntesis se utiliza en impresión y pintura.

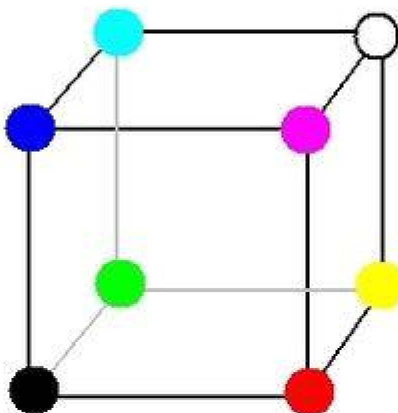


El diagrama de cromaticidad CIE XYY es uno de los primeros espacios de color (1931) definidos matemáticamente.

Como el ojo tiene tres tipos de células receptoras de color una carta con todos los colores visibles deberá ser 3D.

Así podemos dividir el color en dos propiedades:brillo y cromaticidad. En esta carta se ve la cromaticidad. El brillo sería la intensidad con que podemos ver estos colores. Por ejemplo el blanco tiene la misma cromaticidad que el gris, solo varía el brillo.

Los colores puros (totalmente saturados) están en el borde, y en el interior tenemos las mezclas.



Este es el cubo RGB. El modelo de color RGB está basado en la síntesis aditiva. No define como deben ser los colores primarios, cada dispositivo que lo represente lo puede hacer de forma diferente.

El modelo RGB es el utilizado en monitores y será el modelo que adoptaremos para hacer el tratamiento de las imágenes digitales.

Para definir un color hay que dar valor a los componentes rojo, verde y azul. Este valor varía entre 0 (no hay color) y 255 (brillo máximo del color).

El modelo de color CMYK es el utilizado para impresión (modos sustractivos). Se le añade el componente K (negro) ya que al sumar CMY no da un tono de negro puro.

1.6. Almacenando el color en una imagen digital

Ya sabemos cómo funciona el color, cómo se capta la imagen y donde se guarda. Ahora falta saber cómo guardar la información de color o de brillo dentro de la matriz que contiene la imagen muestreada.

Dentro de la matriz que guarda los datos de color podemos utilizar diferentes tipos de datosque definirán la profundidad de color. El primero que se utilizófue el mapa de bits.

Solo se guarda un bit para definir el color siendo cero el negro y 1 el blanco.

La siguiente generación sería el mapa de bytes. Al guardar un byte tenemos 255 posibilidades. Aquí existen dos modos. En la imagen se puede ver el modo de escala de grises. Aquí solo se guarda la información del brillo de la imagen.

La otra posibilidad es el modo de color indexado. En este modo se define una paleta de 256 colores, en esta paleta se indican los valores RGB de cada color. En la imagen tendremos un número que hace referencia a la entrada de la paleta deseada.

El formato High-Color también tiene dos modos:uno con 15 bits y otro con 16.

En el modo de 15 bits se utilizan 5 bits para definir cada valor de RGB más uno de canal alfa. Aquí se pueden conseguir hasta 32768 colores diferentes.

En el modo de 16 bits se utilizan 5 bits para rojo y azul y 6 bits para el verde ya que el ojo humano puede diferenciar más tonos de verde. En este modo hay 65536 colores.

True-Color es el modo de 24 bits. En este caso se han utilizado 8 bits para cada color obteniendo más de 16 millones de colores.

Existen muchos más modos de color como, por ejemplo True-Color de 32 bits que añade 8 bits extra para transparencia.

En este proyecto solo se utilizarán los modos de escala de grises y de 24 bits.

1.7. Segmentación de imágenes

Forma parte del campo de la visión artificial. Es el proceso de dividir una imagen en varias partes u objetos. Así es más fácil localizar objetos o ver los límites de estos dentro de la imagen completa.

Al dividir la imagen en diferentes objetos se simplifica para facilitar su análisis además de dar la posibilidad de etiquetar los diferentes objetos extraídos de la imagen.

Para diferenciar los objetos se buscan pixeles que tengan unas características parecidas.

Más adelante [2.1.3] podremos ver las técnicas utilizadas para la segmentación.

1.8. Aplicaciones

Lo normal es pensar que el tratamiento de imagen digital solo se puede utilizar para aplicaciones artísticas, diseño, cine, publicidad... Pero en realidad hay muchas más posibilidades.

Es posible utilizar el tratamiento de imágenes para la resolución de ciertos problemas por medio del procesado adecuado para cada situación. Veamos algunos ejemplos.

Como ya hemos visto antes la imagen llega con ruido introducido por el sistema de captación aunque también hay otros problemas asociados al sistema de captación como errores en las lentes o errores producidos por quien ha captado la imagen. En estos casos se aplicarán operaciones de realzado de la imagen como: contraste, brillo, nitidez, ecualización, realce de bordes. También se utilizarán filtros de restauración para solucionar desenfoques, movimientos, interferencias, problemas de perspectiva (sobre todo ocasionados por las lentes del sistema de captación).

Hay ciertos tipos de imagen que requerirán un procesado similar al que se acaba de comentar como la mejora de captación de imágenes astronómicas en las que los CCD utilizados introducen mucho ruido y las lentes distorsionan la captura. Se puede aplicar segmentación para diferenciar los objetos captados ya sea por un telescopio o por un satélite.

Otro ejemplo en el que este procesado resulta de gran utilidad es en los sistemas *SIG (*GeographicInformationSystem) utilizados en geografía e

hidrología. Estos sistemas tienen que analizar grandes cantidades de datos asociados a imágenes que se tendrán que mejorar, segmentar y etiquetar. Por ejemplo en un mapa se segmentan las curvas de nivel.

En la actualidad la imagen digital también es utilizada en medicina. Es aplicada en las siguientes tecnologías médicas: fluoroscopía, imagen de resonancia magnética, medicina nuclear, tomografía por emisión de positrones, radiografía, tomografía y ultrasonidos. En este ámbito también se suele utilizar la segmentación de imágenes para detectar volúmenes de tejido, tumores... Y la mejora de imagen es vital para poder hacer cirugía guiada por ordenador.

Estas técnicas también son aplicables a sistemas de seguridad como sensores de huellas digitales, reconocimiento de caras e iris, objetos...

También hay que aplicar un tratamiento y segmentación a las imágenes captadas por sistemas de visión por ordenador. Hay que mejorar las imágenes captadas por las cámaras y separar todos los objetos captados para su posterior análisis. Sus aplicaciones son muy variadas como: control de tráfico, prevención de accidentes, robótica industrial, balance de blancos automático, cuenta de objetos, microorganismos, personas, partículas, meteorología.

Finalmente un tipo de procesado muy utilizado en la actualidad es la compresión de imagen para su almacenamiento o distribución. Dependiendo del resultado que se quiera hay diferentes tipos de compresión y formatos para guardar la información. En algunas aplicaciones convendrá el uso de compresión con pérdidas (que es mayor) de mapas de bits y en otras el uso de vectores que generan trazados cuyo tamaño se puede variar sin perdidas de calidad.

2. FUNDAMENTOS TEÓRICOS

En la introducción ya se ha visto como se capta la imagen, cómo funciona el color, como almacenarla y para qué sirve hacer su tratamiento digital. Ahora es el momento de empezar a repasar las técnicas utilizadas para procesar la imagen desde un punto de vista teórico antes de entrar en como lo hace el programa.

Los valores utilizados serán 1.0 para brillo máximo y 0.0 para color negro (valores normalizados) ya que esta será una explicación teórica.

2.1. Realzado de imágenes

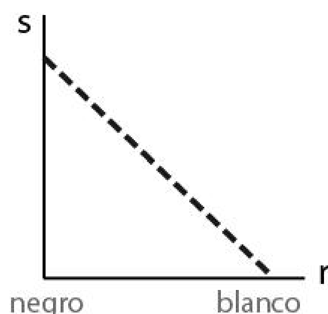
Con el realzado de imagen se adapta la imagen captada a nuestras necesidades según la aplicación que se le vaya a dar.

Dependiendo de la aplicación habrá que utilizar unos procesos u otros ya que un procesado útil para una aplicación puede ser un desastre para otra. Estos procesos se aplicarán en el dominio del espacio y en el de la frecuencia.

2.1.1. Realzado por procesado del punto

El resultado de estas operaciones solo depende del nivel de brillo del punto de entrada. Se utiliza para la manipulación de pixeles.

2.1.1.1. Negativo



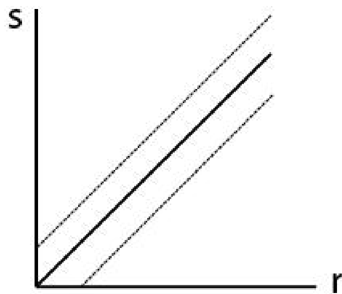
$$s = T[r] = 1 - r$$

Se recorre la imagen punto por punto y se invierte el valor de cada uno restándoselo a 1.

El resultado es el negativo de la imagen original.

En imagen en color se repite este proceso en cada canal de color.

2.1.1.2. Brillo



$$s = r + K$$

Con k=0 tenemos el brillo original de la imagen (la línea continua). Si se modifica su valor se desplazará la recta.

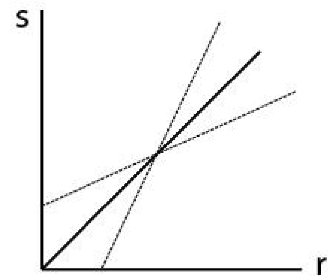
Al desplazar la recta se cambia la respuesta y se modifica el brillo. Si $k<0$ se disminuye el brillo y si es mayor se aumenta.

2.1.1.3. Contraste

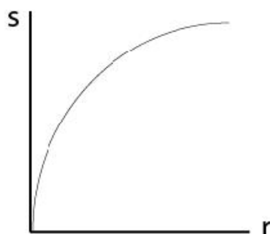
$$s = m \cdot r + k$$

El contraste es la diferencia entre lo mas claro y lo mas oscuro.

Esta diferencia se consigue variando la pendiente de la curva (valor m).



2.1.1.4. Compresión de rango dinámico



$$s = c \cdot \log(1 + |r|)$$

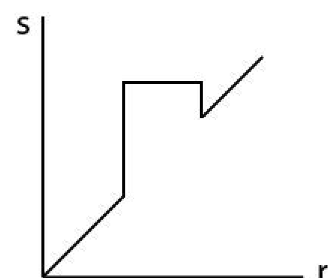
Se utiliza en imágenes con un gran margen dinámico para diferenciar niveles de gris muy alejados.

Utilizado para visualizar la transformada de Fourier.

2.1.1.5. Umbralización

Resalta un rango de grises determinado. Se utiliza para segmentación de imágenes.

La umbralización simple pondrá a 0 los niveles de gris por debajo del seleccionado y a 1 los que estén por encima. Si la umbralización es doble (como en la imagen) se pondrán a 1 los valores seleccionados



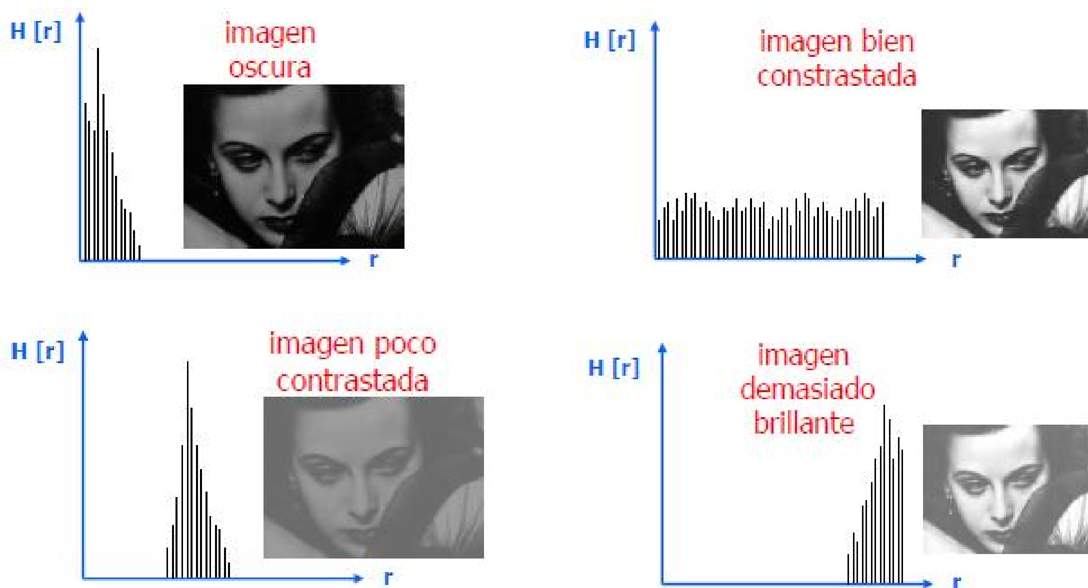
dentro del rango.

2.1.1.6. Procesado del histograma

El histograma es un gráfico de barras que describe la distribución de niveles de gris de la imagen de forma rápida y muy visual.

Los datos están contenidos en un vector. Cada posición corresponde con un nivel de gris y su valor es el número de pixeles de ese nivel de gris que contiene la imagen. Se puede observar rápidamente si existe un nivel de gris y en qué proporción.

Es útil para saber cómo mejorar la imagen ya que en un solo golpe de vista se sabe si la imagen es clara, oscura o está mal contrastada. Estos datos pueden ser muy subjetivos al ver la imagen pero el histograma proporciona una información más precisa.



$$v(i,j) = \frac{Fu(u(i,j) - Fu(a))}{1 - Fu(a)}(L-1) + 0.5$$

La ecualización del histograma pretende obtener una imagen de salida con un histograma con distribución uniforme (imagen bien contrastada) de forma automática. El objetivo es conseguir el mismo número de pixeles para cada nivel de gris.

2.1.1.7. Operaciones con imágenes

En este proyecto se utilizan dos operaciones:la resta y la multiplicación.

La resta se hace aplicando la operación booleana XOR. Se aplica a las imágenes para obtener sus diferencias de niveles de gris. Por ejemplo sus aplicaciones pueden ser en medicina para radiografías, en visión por ordenador para detectar movimiento y para detectar ruido por comparación entre varias tomas de la imagen.

La multiplicación se consigue mediante la operación booleana AND. Se utilizará para detección de características de la imagen o para analizar formas. Por ejemplo, si se quiere obtener el borde de una imagen se multiplicará esta por una versión más grande y con colores invertidos.

2.1.2. Filtrado espacial

Toda imagen está compuesta por un conjunto de frecuencias. Las frecuencias altas se corresponden con las zonas en las que la imagen tiene cambios de brillo muy rápidos. Entonces las frecuencias bajas se corresponden con las zonas de la imagen en las que no hay casi cambios, por ejemplo zonas con colores planos.

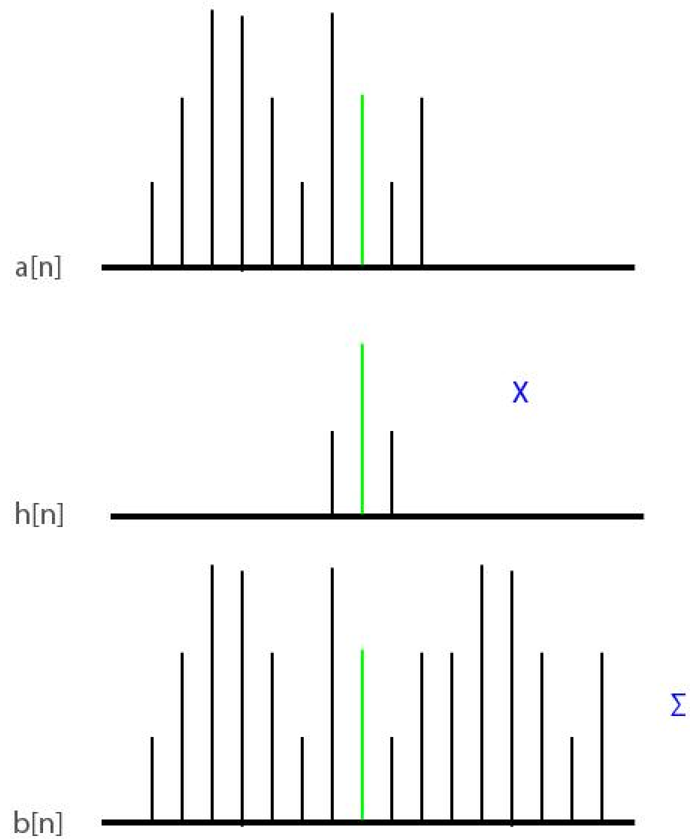
Para eliminar bandas de frecuencias se utiliza el filtrado espacial. Este se hace por medio de procesamiento por grupo de pixeles, se hacen operaciones sobre un grupo de pixeles en torno a un pixel central.

Los filtros se van a generar mediante una operación conocida como convolución espacial. Esta es una operación típica en los sistemas LIT.

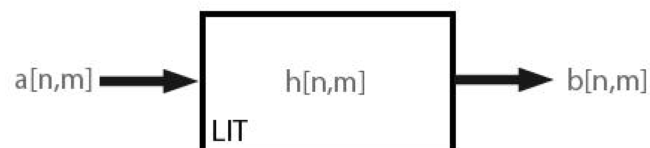
Convolución 1D

Teniendo una señal muestreada a[n] que pasa por el sistema h[n] se obtiene la señal de salida b[n] mediante la convolución de la señal de entrada por h[n].

La convolución se obtendrá mediante el desplazamiento del filtro invertido a la posición de la muestra de entrada. Se multiplica cada coeficiente del filtro por cada una de las muestras de origen y se suman los resultados.



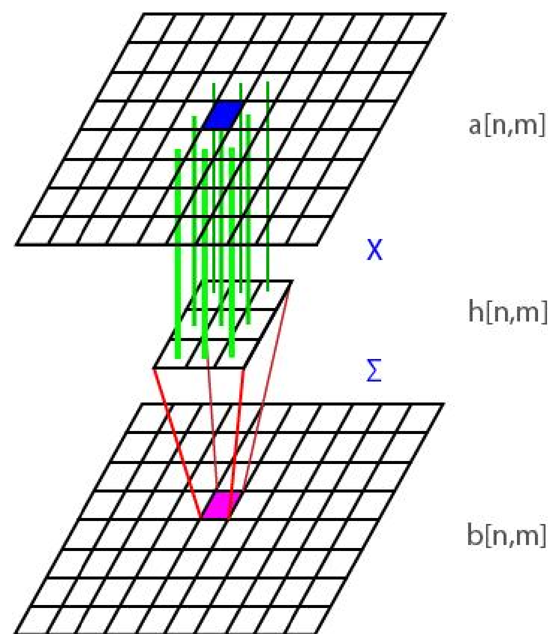
Convolución 2D



Como ya se ha visto al principio la convolución utiliza la suma ponderada de un pixel central y los que le rodean. A estos pixeles se les llama *kernel*. Los valores que contenga el kernel (pesos de la suma) determinarán la respuesta del filtro.

El proceso en 2D es similar al proceso en 1D pero teniendo en cuenta que se trabaja con una imagen y un kernel en dos dimensiones.

El kernel tendrá las dos dimensiones del mismo tamaño. El tamaño de este será de números impares (3x3, 5x5) para no perder simetría.



2.1.2.1. Filtro paso bajo o media

Es un filtro lineal de suavizado.Este tipo de filtro realza las frecuencias bajas atenuando las altas. El efecto de este filtro es un suavizado o emborronamiento de la imagen.

Sirve para eliminar los detalles pequeños (antes de extraer un objeto más grande), para rellenar espacios pequeños entre líneas o curvas y para atenuar el ruido granular.

El kernel utilizado para este filtro es así:

$$\begin{pmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{pmatrix}$$

Como se puede observar la suma de todos los pesos vale 1 y todos los pesos son positivos.

Al aumentar el peso del pixel central se disminuye el efecto de los pixeles laterales.

2.1.2.2. Filtro paso alto

Este es otro filtro lineal pero de realce. El objetivo de este filtro es realzar las altas frecuencias dejando las bajas como están. Se utiliza para intensificar los detalles finos que se han perdido al capturar o al procesar la imagen.

El kernel de este filtro puede ser así:

$$\begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$$

En el centro siempre habrá un coeficiente positivo y este estará rodeado de coeficientes negativos. La suma valdrá 1.

2.1.2.3. Filtro mediana

Ahora estamos ante un filtro no lineal. Se utiliza para eliminar el ruido impulsivo (pixeles aleatorios muy separados entre sí).

Para aplicarlo se toma la mediana de la máscara alrededor del pixel central. La mediana se calcula ordenando los valores de entrada y quedándonos con el central. Al quedarse estos pixeles de ruido en los bordes de la mediana nunca se elegirán como resultado de forma que se pueden eliminar sin afectar a la imagen original.

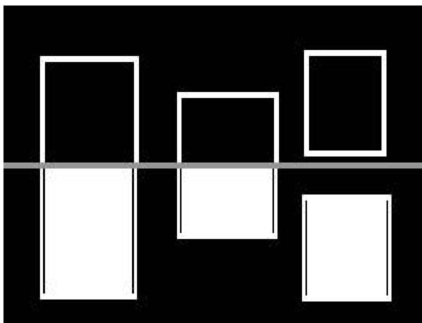
Si el ruido impulsivo tiene un tamaño mayor de 3 o 4 pixeles habrá que utilizar una máscara mayor que la de 3x3.

2.1.3. Filtros para segmentación

Este tipo de filtros se utilizan para segmentación por detección de discontinuidades, la cual sirve para análisis o reconocimiento de objetos. Reducen la imagen de forma que solo se vean los límites de los objetos que están contenidos en ella.

Hay tres filtros que pueden realizar estas operaciones: desplazamiento y diferencia, gradiente y laplaciano.

2.1.3.1. Filtros de detección de líneas



Son unos sencillos filtros lineales que dependiendo del kernel escogido pueden detectar un tipo u otro de líneas dentro de la imagen.

Para extraer los límites verticales desplaza la imagen un pixel a la izquierda y hace la diferencia con la imagen original.

Para los límites horizontales hace un desplazamiento vertical de la imagen y luego saca la diferencia con la original.

Los kernels utilizados para la detección de bordes en horizontal y vertical son:

$$\begin{pmatrix} -1 & -1 & -1 \\ 2 & 2 & 2 \\ -1 & -1 & -1 \end{pmatrix} \text{Horizontal} \qquad \begin{pmatrix} -1 & 2 & -1 \\ -1 & 2 & -1 \\ -1 & 2 & -1 \end{pmatrix} \text{Vertical}$$

El tamaño de la línea a detectar depende del tamaño del kernel utilizado. Por ejemplo si queremos detectar líneas de 5 pixeles de largo la matriz será de 5x5.

2.1.3.2. Filtros de detección de bordes

Destacan con gran precisión los bordes en una dirección determinada. Se centra en las diferencias de intensidad pixel a pixel. Con estos filtros se pueden extraer fácilmente los contornos de los objetos y así separarlos. Tienen un bajo costo computacional.

Pero antes de empezar hay que saber que es un gradiente y como aplicarlo a una señal discreta o imagen.

Gradiente

Utilizado en cálculo vectorial. El gradiente ∇f de un campo escalar f es un campo vectorial. El vector gradiente f evaluado en un punto genérico x del dominio f ($\nabla f(x)$) indica la dirección en la que el campo varía más rápido y su módulo representa la velocidad de variación de f en la dirección de dicho vector gradiente.

El gradiente de una imagen es un vector de dos dimensiones Gx y Gy que contiene los gradientes en cada dirección.

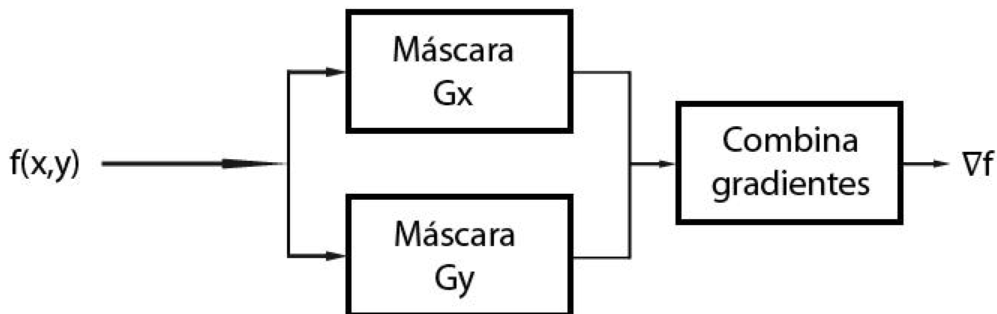
$$f = \begin{bmatrix} Gx \\ Gy \end{bmatrix} = \begin{bmatrix} \frac{\delta f}{\delta x} \\ \frac{\delta f}{\delta y} \end{bmatrix}$$

Su módulo y su fase serán:

$$\textit{módulo}: \quad |\nabla f| = \sqrt{Gx^2 + Gy^2} \qquad \text{aproximación}: \quad |\nabla f| = |Gx| + |Gy|$$

$$fase: \quad \angle f = arctan \frac{Gx}{Gy}$$

Todo esto traducido a procesado de la imagen se puede realizar siguiendo este esquema:



Las máscaras Gx, Gy se obtienen por medio de kernels como los utilizados en el resto de filtros. Para calcularlas tenemos dos opciones la gradiente de Prewitt y la gradiente de Sobel. Estos son sus kernels:

Prewitt:

$$Vertical: \begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix} \quad Horizontal: \begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix}$$

Sobel:

$$Vertical: \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} \quad Horizontal: \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$$

El último paso será combinar las máscaras. Para esa operación también tenemos dos posibilidades. Cálculo por raíz cuadrada que se aproxima más al verdadero resultado del gradiente y cálculo por valor absoluto que aunque no proporciona el verdadero valor del gradiente requiere menos tiempo de cálculo.

Las expresiones para la combinación son:

$$Raíz = \sqrt{vertical^2 + horizontal^2}$$

$$Valor\ absoluto = |vertical| + |horizontal|$$

Laplaciano

Este filtrado realza los bordes en todas las direcciones, el resultado es similar a la suma de los filtros de detección de líneas. En este caso se trabaja con la segunda derivada que obtiene mejores resultados pero añade ruido a la imagen.

Para utilizar este tipo de filtrado solo hay que escoger el kernel apropiado que es el siguiente:

$$\begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$$

2.1.4. Filtros morfológicos

En matemáticas se utiliza la palabra morfología para designar una herramienta que se utiliza para extraer los componentes de una imagen empleados para representar y describir la forma de una región.

Estas operaciones se aplican a imágenes en blanco y negro ya que se utilizan funciones booleanas para realizarlas.

Este tipo de filtros cambian el tamaño de la imagen procesada con el objetivo de suavizar los contornos de los objetos que componen la imagen por medio de la eliminación de ciertas estructuras como agujeros, bultos... Así se simplifican las imágenes preservando las características esenciales de forma y eliminando las irrelevancias

Pero antes de entrar en este tipo de filtrado hay que repasar algunas de las operaciones que se van a realizar.

Operaciones lógicas

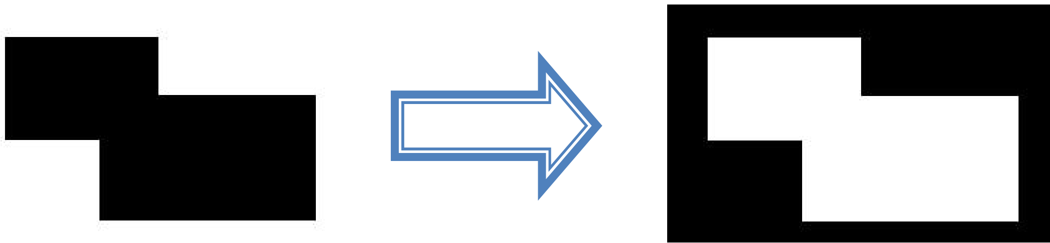
Este tipo de operaciones forman parte del álgebra de Boole. El álgebra de Boole es un sistema matemático para formular proposiciones lógicas mediante símbolos así se pueden resolver estos problemas como si se tratara de álgebra normal.

Este tipo de álgebra se utiliza para el análisis y diseño de sistemas digitales. Las variables booleanas solo pueden tener el valor 0 o 1.

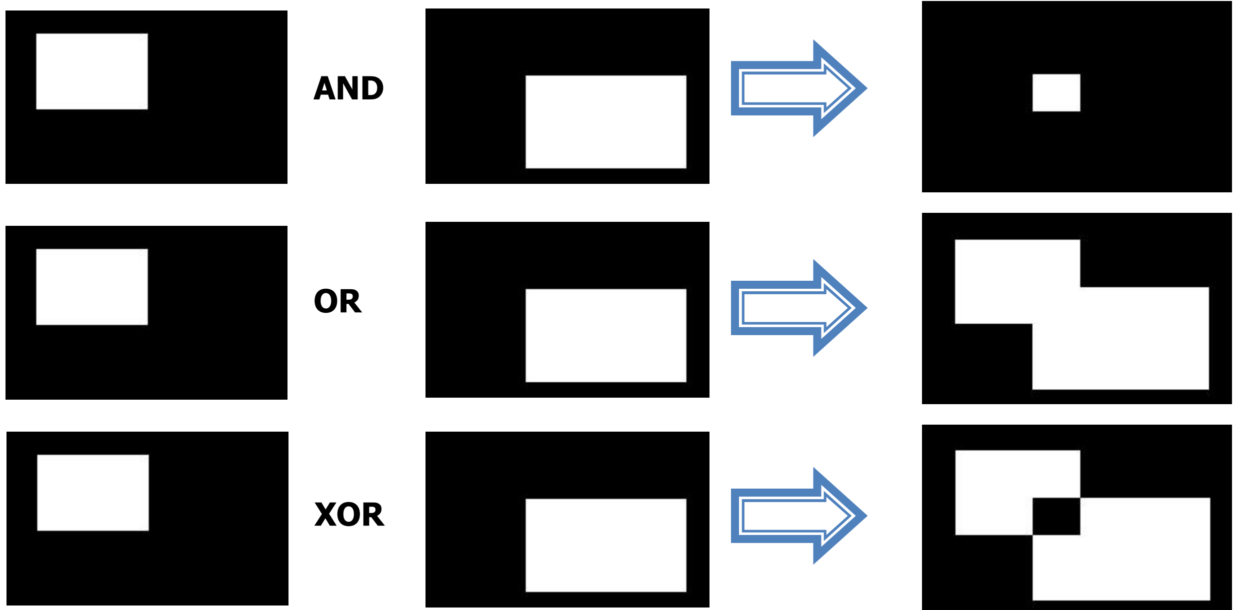
Las operaciones lógicas se implementan por medio de circuitos lógicos que son los bloques básicos que al combinarse se convierten en sistemas digitales. En este caso en vez de utilizar circuitos lógicos se hará uso de las operaciones booleanas que contiene el ordenador.

Estas son las operaciones lógicas que vamos a utilizar:

not



In	Out
1	0
0	1



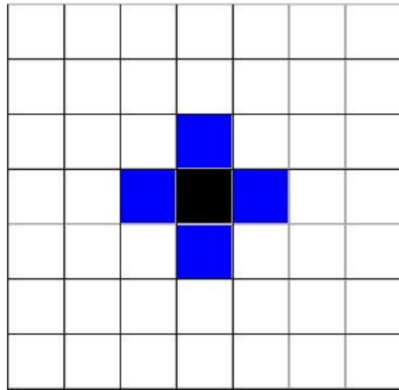
AND		
In 1	In 2	Out
0	0	0
0	1	0
1	0	0
1	1	1

OR		
In 1	In 2	Out
0	0	0
0	1	1
1	0	1
1	1	1

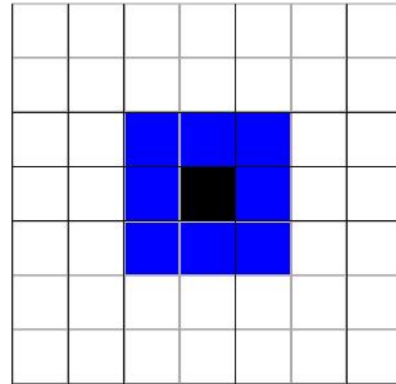
XOR		
In 1	In 2	Out
0	0	0
0	1	1
1	0	1
1	1	0

Vecindad

La vecindad hace referencia a los pixeles que rodean a un pixel central. A éstos se les llama vecinos. Hay varios tipos de vecindad pero nos vamos a centrar en dos de ellos el 4-vecinos y el 8-vecinos. La forma más fácil de entender su distribución es con imágenes, se definirá el pixel central de color negro y los vecinos de color azul:



4-vecinos

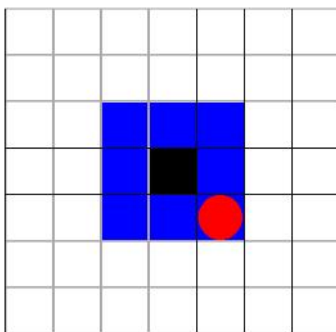


8-vecinos

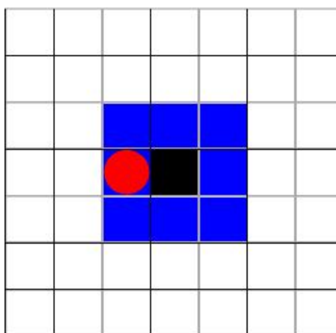
Conectividad

Un pixel o un grupo de estos están conectados al central si están dentro del conjunto definido por la vecindad. Por ejemplo si tenemos un pixel a la derecha del central si al poner encima la imagen de los vecinos cae dentro de un punto azul estará conectado.

Veamos algunos ejemplos utilizando los colores de los ejemplos anteriores, ahora se marcará como un punto rojo el pixel del cual queremos saber si hay conectividad:



Este pixel tiene conectividad 8.

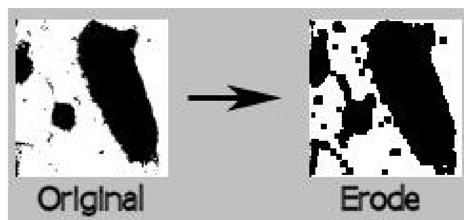


Este pixel tiene conectividad 4.

2.1.4.1. Erosión

Es un operador sustractivo que disminuye el tamaño de los objetos. Elimina penínsulas delgadas y objetos demasiado pequeños.

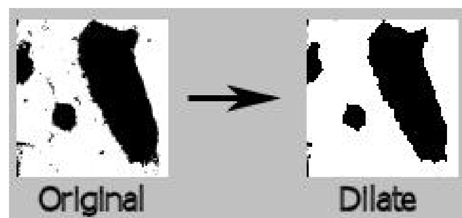
Este filtro no utiliza un kernel se basa en los datos que recoge de los pixeles 8-vecinos que encuentra mientras recorre la imagen. Entonces, lo que se hace ahora es mediante una ventana de 3x3 se recorre la imagen y se pone a uno el pixel central de esta si todos sus vecinos están también a uno. Esta comprobación se hace con la operación booleana AND.



2.1.4.2. Dilatación

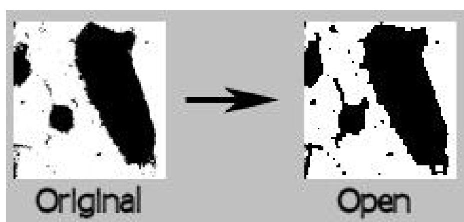
Este otro es un operador aditivo y aumentará el tamaño de los objetos. Elimina discontinuidades y pequeños huecos o entrantes.

Como en la erosión ahora se recorre la imagen con una ventana de 3x3 pixeles pero en este caso se pondrá a uno el valor central cuando al menos uno de sus ocho vecinos este a 1. Esta comprobación se realiza por medio de la operación booleana OR.



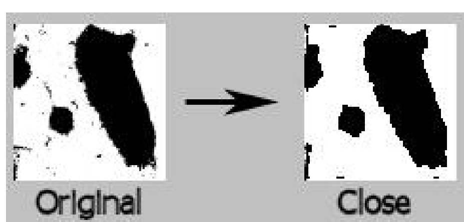
2.1.4.3. Open

Suaviza la imagen mediante la eliminación del "ruido negro" (puntos aleatorios) y salientes estrechos. Este filtro simplemente se basa en aplicar erosión y después dilatación. Es posible repetirlo varias veces.



2.1.4.4. Close

Suaviza la imagen eliminando el "ruido blanco" y rellenando los salientes estrechos. Se tiene que aplicar dilatación y después erosión. También es posible repetirlo varias veces.



2.1.4.5. Observaciones sobre los filtros morfológicos

La dilatación y la erosión no son operaciones inversas pero sí que son inversas aproximadas y su concatenación permite recuperar aproximadamente el tamaño de los objetos (open y close) y algunos de los huecos que no hayan sido eliminados. Además, suavizan el contorno de los objetos y modifican su tamaño.

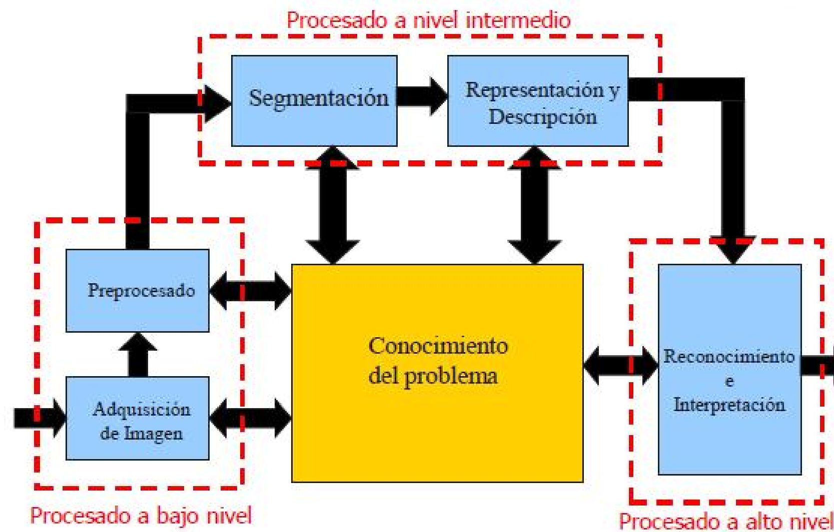
Los filtros open y close tienen las mismas propiedades de suavizado que la dilatación y la erosión. Generalmente no afectan al tamaño de los objetos si son muy grandes. Estos filtros son unidireccionales, solo pueden quitar un tipo de ruido.

2.1.5. Diferenciación de objetos

Todos los procesados de imagen que se han visto servirán de ayuda para los que se verán ahora. Estos últimos procesos se utilizarán para un sencillo sistema de visión por ordenador que distingue entre objetos redondos o alargados.

Actualmente no se pueden hacer sistemas de visión por ordenador totales, esto quiere decir que solo se pueden aplicar a fines muy concretos, en algunos casos pueden superar las capacidades de visión humanas aunque en otras tener peor

percepción pero servir de ayuda para trabajos automáticos. Existen muchos métodos y muchas aplicaciones pero nos centraremos solo en una.



Una vez se ha adquirido la imagen hay que prepararla para poder empezar el proceso de reconocimiento de formas. La imagen de entrada debe estar lo más limpia posible y en blanco y negro para tener una iluminación uniforme.

Entonces el primer paso será segmentar la imagen y de paso pasarla a blanco y negro umbralizando. El umbral se elegirá mirando el histograma.

El objeto debe estar dibujado en blanco sobre un fondo negro así que si es necesario se aplicará el proceso de negativización.

Para eliminar posibles elementos que se confunda con el fondo se aplicará la operación morfológica close y para eliminar los posibles saliente se aplicará open.

2.1.5.1. Operaciones

El sistema de reconocimiento necesita unos datos determinados para poder hacer el cálculo que le dirá la forma del objeto.

Área

El área serán todos los pixeles blancos que hay en el interior del objeto. Como ya se ha preparado es una operación sencilla. Se recorrerá el objeto contando los pixeles blancos.

Perímetro

Para este cálculo hay dos posibles métodos. Por gradientes y con morfología.

El método por gradientes consiste en sacar la gradiente horizontal y la vertical. Luego se aplica la fórmula de valor absoluto [2.1.3.2] y se umbraliza para que los pixeles del perímetro tengan una anchura de un pixel.

La forma utilizada en este proyecto es con morfología. El primer paso es erosionar el objeto, ahora se hace la operación booleana XOR [2.1.4] entre la imagen original y la erosionada.

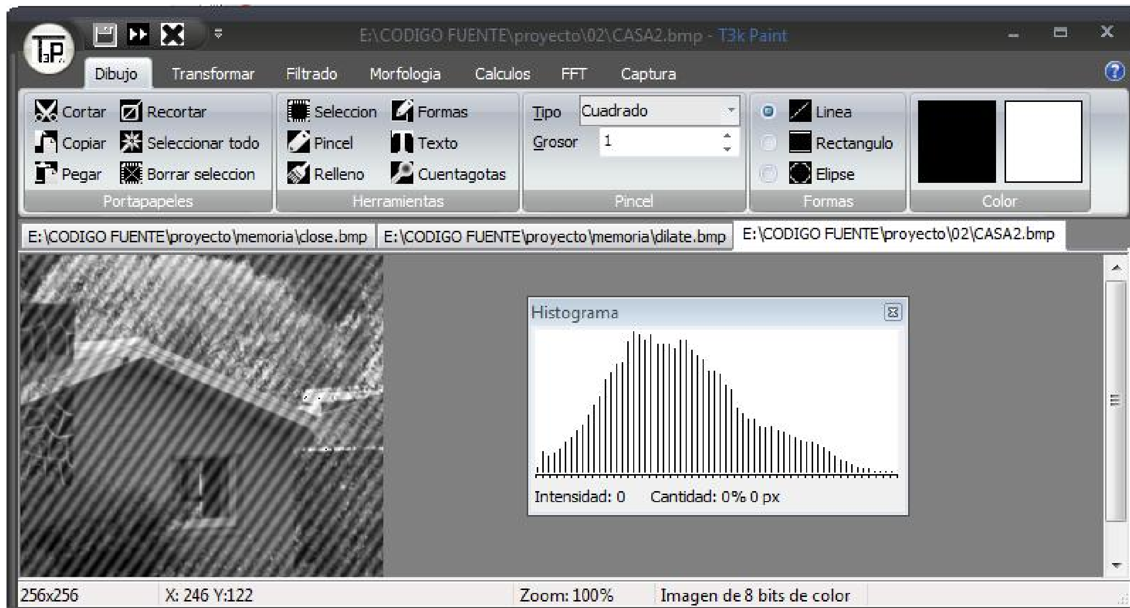
Se siga el camino que se siga el paso final será contar los pixeles blancos que representan el perímetro de la imagen.

2.1.5.2. Detección

Una vez obtenidos los valores de área y perímetro ya se puede detectar la forma. Para ello se aplicará un descriptor conocido como densidad de una región o compacidad.

$$C = \frac{perímetro^2}{área}$$

Si el valor de C es 4π la forma será redonda, si el valor es mayor la forma será alargada.



El nombre del programa que se describe en este proyecto es T3k Paint. Ha sido programado en C++ utilizando el entorno Embarcadero Rad Studio-x con el compilador C++ Builder.

T3k Paint está hecho para ofrecer compatibilidad total con las nuevas versiones de Windows tanto en diseño como en su ayuda que utiliza el nuevo formato CHM (Microsoft Compiled HTML Help) ya que los antiguos archivos de ayuda en formato HLP ya no son compatibles con las nuevas versiones del sistema operativo. Además su código está optimizado para utilizar el juego de instrucciones de Pentium Pro y MMX con lo cual se aprovechan mejor los recursos del sistema.

Visualmente está adaptado a la actualidad ofreciendo el sistema de menús Ribbon de Microsoft y la inclusión de pestañas para cambiar rápidamente entre documentos aunque otro importante punto del diseño de su interfaz es que proporciona toda la información relativa al documento abierto de forma clara haciendo más cómodo el trabajo con imágenes.

El programa está divido en varias partes debido a como se ha implementado. El tratamiento de la imagen se hace a través de una clase llamada utilidades de la que se hablará más adelante [3.3]. Las opciones de dibujo están incluidas dentro de la misma clase pero en cierto modo van a parte ya que se han utilizado funciones del Win32 API.

En resumen, la parte visual del programa es un interfaz para controlar la clase utilidades que es la que se encargará de todos los cálculos y de hacer el trabajo.

3.1. Clases utilizadas

Antes de empezar a analizar con profundidad la clase utilidades vamos a repasar los objetos comunes de Windows y los proporcionados por la librería del compilador ya que algunos también se utilizarán dentro esta.

TCanvas

Viene de la clase Graphics y nos provee una zona de dibujo abstracta. Es un lienzo con una serie opciones para dibujar y controlar los colores será muy utilizada por las herramientas de dibujo de T3k Paint y para dibujar en el interfaz de usuario.

Tcanvas lleva dentro todas las herramientas de dibujo necesarias para dibujar líneas, rectángulos, óvalos, hacer rellenos...

TColor

Este es el objeto que maneja los colores dentro del canvas y de las ventanas y sus controles, no se utiliza para nada en el procesado de las imágenes.

Los colores se especifican en un formato de 4 bytes en hexadecimal. El primer byte se dejará siempre a cero ya que se va a utilizar la paleta de 24bits del sistema y no hace falta especificar ninguna. Los bytes siguientes son los valores RGB. Es posible también utilizar nombres de colores predefinidos como clBlack para el negro, clRed para el rojo...

TPen

Es uno de los objetos que nos proveen para dibujar dentro de TCanvas. No es una herramienta de dibujo en sí. Sirve para configurar los parámetros de los lápices.

Los lápices son los encargados de pintar líneas rectas, dibujar los contornos de las figuras o polígonos...

A este objeto se le puede especificar cuál va a ser grosor del lápiz *width*, el color *TColor* y el estilo.

Tenemos varios estilos para dibujar líneas, el normal es *psSolid* para un relleno solido pero también se pueden hacer líneas discontinuas o definir nuestro propio estilo. El estilo será muy útil para indicar los rectángulos de selección.

TBrush

Con TPen definimos el lápiz para dibujar los bordes, con este objeto lo que hacemos es definir el relleno.

Como en el otro caso también se indicará su color y su estilo pero también se le puede cargar un bitmap aunque esta propiedad no se hará falta.

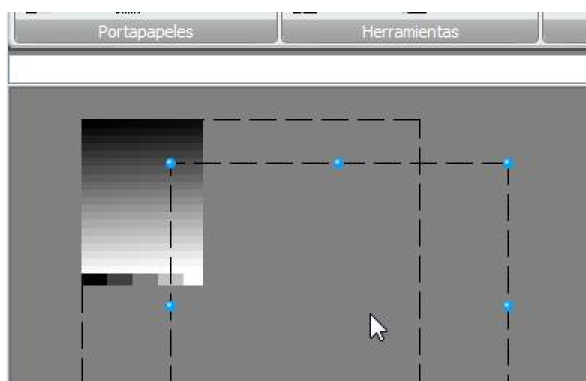
TPaintBox

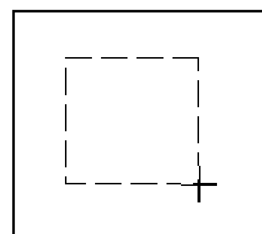
Como su nombre indica es una zona de dibujo. Este es un objeto visual que nos proporciona un TCanvas físico.

Casi cualquier objeto visual tiene un TCanvas asociado pero el paintbox está optimizado para dibujar en él. Esto se nota en la velocidad de refresco y en que se incluye doble buffer para evitar problemas.

Otra diferencia que tiene paintbox con otros objetos visuales es que es transparente lo cual nos permite ponerlo sobre otro objeto y dibujar encima con toda la facilidad que nos da el TCanvas.

En la siguiente imagen se puede ver un paintbox montado sobre otro objeto, en el resultado (al lado) hay un rectángulo que marca una selección, este se dibuja en el paintbox.





TImage

El objeto TImage es un contenedor de imágenes. Siempre que haya que mostrar una imagen en una ventana este es el mejor objeto. Además también contiene un TCanvas para poder dibujar sobre él solo que con una diferencia que es la que los hace diferentes y lo convierte en el más indicado para estos casos.

Dentro de TImage nos encontramos con el objeto TPicture (que veremos ahora) que es el contenedor de las imágenes en sí. Al tener este objeto dentro

el TImage ya no puede ser transparente ya que la imagen (por defecto) tiene siempre un color de fondo por lo que no se puede utilizar para superponerlo con otros objetos.

TPIcture

Aquí tenemos una clase clave para el programa. Dentro de este objeto están los gráficos y sus formatos básicos (bmp, ico, wmf).

Al tener dentro los formatos se puede acceder a ellos y a sus propiedades pero también es la clave para abrir y guardar los archivos ya que contiene las funciones para codificar y almacenar los archivos de gráficos.

TBitmap

Sin esta clase no habría programa. Almacena el mapa de bits, sus dimensiones, cabecera, paleta, canvas y la parte más importante de todo, la función scanline.

La función scanline es la encargada de leer y escribir los pixels de la imagen de la forma más rápida. Gracias a ella se pueden hacer todos los procesados gráficos de esta aplicación.

Otra función de gran importancia dentro de un TBitmap es la Assign. Con esta función se puede copiar imágenes completas entre objetos TBitmap prácticamente sin coste computacional.

Finalmente hay que comentar la propiedad Handle que es la utilizada para pasar un objeto de la clase TBitmap a cualquier función del API.

3.2. Interfaz de usuario

Aunque esta parte se suele explicar con anterioridad, era necesario conocer un poco las clases con las que se van a trabajar antes de hablar de ellas.

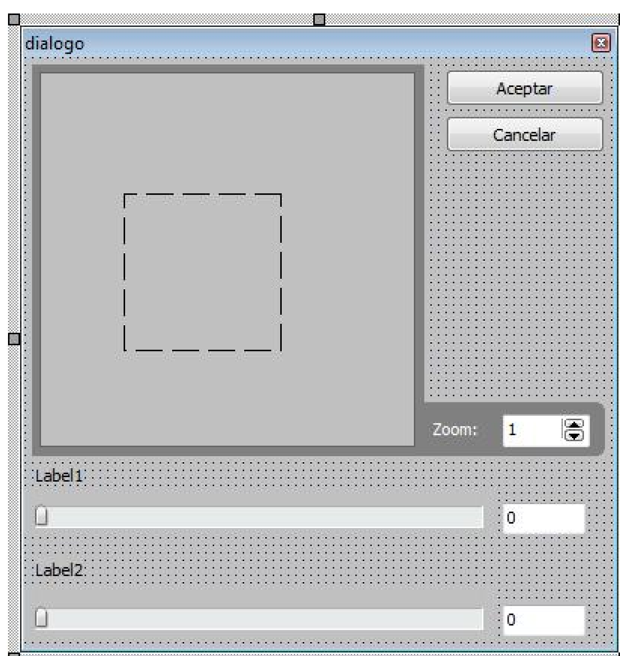
Esta aplicación se compone de 17 ventanas de las cuales hay dos que se comparten, la ventana principal y otra llamada dialogo que se utiliza en gran cantidad de efectos.

A todas las ventanas que se encargan del procesado de la imagen se les va a pasar el documento activo que es un objeto de la clase utilidades. De esta forma tienen acceso directo a la imagen, su histograma y demás propiedades. Este tipo de ventanas incluyen los controles necesarios para iniciar las funciones de filtrado así como una pre visualización de los resultados.

Ventana dialogo

Esta ventana da de que hablar debido a sus características de inicialización. A esta ventana además del documento activo se le pasarán ciertos parámetros extra para configurarla. La recogida de dichos parámetros se debe a que se comparte en gran cantidad de filtros concretamente en los siguientes: brillo, contraste, umbral simple y doble, margen dinámico, erode, dilate, open y close.

Se ha programado así para ahorrar líneas de código y recursos ya que estos procesos solo necesitan uno o dos valores para configurarlos.



Para llamar a esta ventana se utilizan dos líneas la primera es para configurar, su formato es:

```
__fastcallTdialogo::configura(int procesado, booldoble,Stringtitulo,
String etiqueta1, String etiqueta2, int max1, int max2,  int min1, int
min2,int centro1,int centro2,utilidades* documento);
```

Los valores de entrada son:

procesado: Es un entero que indica el tipo de procesado que se va a realizar. Los posibles valores que puede tomar son:

1. 1: Brillo
2. 2: Contraste
3. 3: Umbral simple
4. 4: Umbral doble
5. 5: Margen dinámico

6. 6: Erode
7. 7: Dilate
8. 8: Close
9. 9: Open

doble: De tipo booleano. Si es true se mostrarán dos parámetros de configuración del filtro.

titulo: Es el texto que se muestra en la barra de título de la ventana.

etiqueta1 y 2: El texto de las etiquetas de los parámetros configurables (Label1 y Label2).

max1 y 2: El valor máximo de recorrido de las barras de selección.

min1 y 2: El valor mínimo de recorrido de las barras de selección.

centro1 y 2: Posición en la que se inician las barras de selección.

documento: Puntero al documento activo.

Una vez está configurada la ventana se abrirá mediante la función Show(). Este sería un ejemplo:

```
dialogo-
>configura(7,false,"Dilate:","Repeticiones:","",25,255,1,1,1,255,&docu
mento[actual]);

dialogo->Show();
```

Como se puede ver en el ejemplo es mucho más cómodo escribir estas dos líneas de código que crear una ventana nueva para cada efecto.

3.3. La clase utilidades

Como se dijo al principio de la sección esta clase es la que controla todas las acciones que se realizan en el programa.

El programa contiene 100 copias de esta clase metidas en un vector. Esto quiere decir que se puede abrir hasta 100 documentos y se podrá acceder a ellos con la facilidad que nos dan los vectores.

A continuación se puede ver el diseño de la clase.

utilidades

Attributes

+ imagen:TPicture*
+ procesada:TPicture*
- altura:int
- anchura:int
- area:int
- color1:TColor
- color2:TColor
- datos1:int**
- datos2:int**
- datosB1:int**
- datosB2:int**
- datosG1:int**
- datosG2:int**
- datosR1:int**
- datosR2:int**
- funcionT:int[]
- funcionTB:int[]
- funcionTG:int[]
- funcionTR:int[]
- iExtension:UnicodeString
- modo:int
- nombreI:UnicodeString
- perimetro:int
- temporal:TPicture*
- tipoFondo:int
- TIPORECORTE:int
- undo:TPicture*

Methods

+ ~utilidades
+ brillo:void
+ calc_area:int
+ calc_perimetro:int
+ cambiatama:void
+ circulo:void
+ clear:void
+ close:void
+ contraste:void
+ creaNuevo:void
+ cuentagotas:TColor
+ dameAltura:int
+ dameAnchura:int
+ dameArea:int
+ dameCanvas:TCanvas*
+ dameExtension:UnicodeString
+ dameFinal:TPicture*
+ dameHandle:HDC__*
+ dameModo:int
+ damePerimetro:int
+ dameProcesada:TPicture*
+ dameTipoFondo:int
+ dameTipoRecorte:int
+ dilate:void
+ eqhistograma:void
+ erode:void
+ escalado:void
+ estiloBrocha:void
+ estiloLapiz:void
+ filtroUsr:void
+ finLinea:void
+ getPixel:unsigned char
+ getPixel24:tagRGBTRIPLE
+ gradientes:void
+ hazUndo:void
+ horizontal:void
+ HPF:void
+ inicia:int
+ iniLinea:void
+ laplaciano:void
+ LPF:void
+ luminosidad:int
+ margendinamico:void
+ media:void
+ mediana:void
+ negativo:void
+ nombre:UnicodeString
+ open:void
+ pincel:void
+ pintaImagen:void
+ ponImagen:void
+ ponTexto:void
+ putPixel:int
+ putPixel24:int
+ putPixel24p:int
+ quicksort:void
+ recortaPixel:int
+ recortaPixel24:tagRGBTRIPLE
+ recortaSimple:int
+ rectangulo:void
+ redondeo:float
+ reinicia:void
+ rellenar:void
+ rota180:void
+ rota270:void
+ rota90:void
+ rotarLibre:void
+ ruido:void
+ setColores:void
+ setFont:void
+ setGrosor:void
+ setRecorte:void
+ setTama:void
+ setTipoFondo:void
+ umbral1:void
+ umbral2:void
+ utilidades
+ validarImagen:void
+ vertical:void
+ voltearH:void
+ voltearV:void
+ XOR:TPicture*

3.3.1. Variables

La variable más importante de la clase es la que contiene la imagen que estamos tratando. Para alojar las imágenes se utiliza la clase TPicture [3.1]. La clase trabaja internamente con tres copias de la imagen.

Para guardar la imagen con la que se está trabajando tenemos la variable *imagen*. Esta es la copia que se ve en el editor y la que tiene aplicados los efectos que hayamos utilizado, es la versión final.

Otra se llama *procesada*, es una copia intermedia utilizada por los cuadros de dialogo para probar los efectos sin afectar a la versión final. Cuando se acepta el efecto elegido se copia en *imagen*.

Finalmente tenemos otra copia llamada *undo*. Cuando se aplica un efecto a la copia final *(imagen*) se copia esta sobre *undo* para después copiar *procesada* sobre *imagen*. Es una forma muy rápida de tener un nivel de deshacer sin tener que escribir líneas extra al final de cada efecto.

Sea cual sea el formato de archivo que se cargue se convierte en TBitmap y se pasa *imagen*. Cuando se graba se utiliza la variable *iExtension* para sacar el formato de imagen y poder convertirla. Cuando se crea o se carga un documento nuevo su nombre se guarda en *nombreI*.

Como los efectos se aplican dentro de esta case nos encontramos con cuatro vectores de 256 valores llamados *funcionT, funcionTR, funcionTG, funcionTB*. Estos vectores guardan los datos de la función T para escala de grises y para cada plano de color que se utilizarán para calcular la ecualización del histograma.

La función utilizada para aplicar filtros con gradientes también necesita datos. En este caso son arrays dinámicos llamados *datos1*, *datos2, datos1R, datos2R, datos1G, datos2G, datos1B, datos2B*.

Otros datos de la imagen son *anchura*, *altura* y *modo*. Los dos primeros guardan esos datos para hacer los bucles a la hora de procesar y la variable modo nos indica si la imagen está en escala de grises o con paleta de 24bits tomando los valores 8 o 24.

tipoFondo sirve para saber el fondo que habrá que dibujar de fondo al aplicar un efecto ya que algunos procesos dejarán a la vista el fondo de la imagen. Si toma el valor 0 dejará de fondo la imagen original, si es 1 pondrá color blanco y si es 2 color negro.

TIPORECORTE selecciona el tratamiento de números negativos siendo recorte a cero si vale 0, valor absoluto si vale 1 y suma 128 cuando valga 2.

Finalmente las variables *área* y *perímetro* guardan los resultados cuando se hacen estos cálculos.

3.3.2. Funciones varias

Para poder aplicar efectos a las imágenes hay que definir una serie de funciones de ayuda. Aunque se han creado gran cantidad de estas funciones solo nos centraremos en las referentes a procesado de imagen.

3.3.2.1. Putpixel

Una de las funciones más importantes del programa.Todo procesado de imagen requiere que se pinten pixeles en ella.

La clase TCanvas provee una función llamada pixels[][] que hace este trabajo y de hecho en algunos puntos del programa que no tienen demasiada carga de dibujo se ha utilizado pero es realmente lenta como se observa en la siguiente tabla.

Tamaño	100x100	400x400	1600x100
Pixels	4700	81300	81300
ScanLine	31	296	218
GetDibBits	187	1539	1735

En la comparativa se ve la velocidad en milisegundos a la hora de rellenar imágenes de los tamaños expuestos. Se ve claramente que la función pixels hace imposible procesar imágenes aun teniendo un procesador muy potente. GetDibBits es una función API que mejora los resultados pero ScanLine es el claro ganador así que vamos a hacer uso de esta función tanto en putpixel como en getpixel.

ScanLine forma parte del objeto TBitmap y nos devuelve un vector que es la fila completa que le indiquemos siendo aquí donde reside el secreto de su velocidad, solo tiene que hacer un acceso a memoria para recuperar la fila y luego los que hagan falta para alterar las columnas del vector.

Con esta información se han construido tres funciones para escribir pixeles, una para escala de grises y dos para 24 bits: a una se le pasan los valores r,g,b en bytes separados y en otra por medio de la clase TRGBTriple que básicamente lo que hace es encapsular esta información.Se han programado dos opciones

porque según el procesado que apliquemos nos conviene usar una u otra como ya veremos más adelante.

Las tres funciones son así:

```
__fastcallutilidades::putPixel(int x, int y, Byte color, TPicture* imgp){

 Byte *p1;

 p1=(byte *) imgp->Bitmap->ScanLine[y];

p1[x]=color;

}

__fastcallutilidades::putPixel24p(int x, int y, TRGBTriple color, TPicture* imgp){

      TRGBTriple *p;

      p=(TRGBTriple *)imgp->Bitmap->ScanLine[y];

      p[x]=color;

}


__fastcallutilidades::putPixel24(int x, int y, Byte r, Byte g, Byte b, TPicture* imgp){

      TRGBTriple *p;

      p=(TRGBTriple *)imgp->Bitmap->ScanLine[y];

      p[x].rgbtRed=r;

      p[x].rgbtGreen=g;

      p[x].rgbtBlue=b;

}
```

Las variables de entrada serán siempre la posición, el color y la imagen puesto que al tener tres copias de la imagen cada vez interesará acceder a una u otra con facilidad.

3.3.2.2. GetPixel

GetPixel es otra importantísima función. No se pueden hacer cálculos sobre los pixeles si no conocemos su brillo o su color. GetPixel es la encargada de hacer este trabajo.

Esta función es igual que la anterior pero le hemos dado la vuelta, en lugar de asignarle un valor leemos el que nos da, ya que estamos trabajando con un vector es muy sencillo.

Así esta función lo que hace es devolver un byte con el nivel de gris o un TRGBTriple con el color. Las funciones son así:

```
Byte __fastcall utilidades::getPixel(int x, inty,TPicture* imgg){

Byte *p;

 p=(byte *) imgg->Bitmap->ScanLine[y];

return p[x];

}

TRGBTriple __fastcallutilidades::getPixel24(int x, inty,TPicture*
imgg){

      TRGBTriple *p;

      p=(TRGBTriple *) imgg->Bitmap->ScanLine[y];

      return p[x];

}
```

Las variables de entrada son las mismas que en el caso anterior pero sin la información de color ya que ésta es la que se nos devuelve al hacer la llamada.

3.3.2.3. RecortaPixel

RecortaPixel aplica el tratamiento de números negativos al valor que le pasemos. Tenemos tres funciones para esto.

La primera y más básica es recortaSimple que solo comprueba si el número entero de entrada está en el rango de un byte, es decir entre 0 y 255. Si se sale de ese rango devolverá 0 o 255.

```
int __fastcall utilidades::recortaSimple(int valor){

int vuelta;

vuelta=valor;
```

```
if (valor<0) {

      vuelta=0;

  }

if (valor>255) {

      vuelta=255;

  }

return vuelta;

}
```

Las siguientes sí que hacen tratamiento de números negativos. Se les pasará el brillo en un entero o el color en un TRGBTriple y el tipo de tratamiento a aplicar que por lo general estará en la variable *TIPORECORTE* [3.3.1].

Esta es la versión para escala de grises que trabaja con bytes aunque en algunas ocasiones se aplica directamente a los bytes RGB:

```
int __fastcall utilidades::recortaPixel(int valor, int tipo){

      int vuelta;

      vuelta=valor;

      if ((valor>=0)&&(valor<=255)) {

            vuelta=valor;

      } else {

            if (valor>255) {

                  vuelta=255;

            } else {

                  switch(tipo){

                        case 0:vuelta=0;break; //recorte a 0

                        case 1:vuelta=abs(valor);break; //valor
absoluto

                        case 2:if (valor+128<0){

                                          vuelta=0;

                                    } else {

                                          vuelta=valor+128;

                                    };break;
```

```
                  }

            }

            }

      return vuelta;

}
```

Y esta es la versión que utiliza colores guardados en TRGBTriple:

```
TRGBTriple __fastcall utilidades::recortaPixel24(TRGBTriple color, int
tipo){

      TRGBTriple vuelta;

      vuelta=color;

      if ((color.rgbtRed>=0)&&(color.rgbtRed<=255)) {

      vuelta.rgbtRed=color.rgbtRed;

      } else {

            if (color.rgbtRed>255) {

                  vuelta.rgbtRed=255;

            } else {

                  switch (tipo) {

                        case 0:vuelta.rgbtRed=0;break;

                        case 1:vuelta.rgbtRed=abs(color.rgbtRed);break;

                        case 2:if (color.rgbtRed+128<0){

                                          vuelta.rgbtRed=0;

                                    } else {

                              vuelta.rgbtRed=color.rgbtRed+128;

                                    };break;

                  }

            }

      }

      if ((color.rgbtGreen>=0)&&(color.rgbtGreen<=255)) {

      vuelta.rgbtGreen=color.rgbtGreen;

      } else {

            if (color.rgbtGreen>255) {
```

```
                  vuelta.rgbtGreen=255;

            } else {

                  switch (tipo) {

                        case 0:vuelta.rgbtGreen=0;break;

                        case
1:vuelta.rgbtGreen=abs(color.rgbtGreen);break;

                        case 2:if (color.rgbtGreen+128<0){

                                          vuelta.rgbtGreen=0;

                                    } else {
      vuelta.rgbtGreen=color.rgbtGreen+128;

                                    };break;

                  }

            }

      }

      if ((color.rgbtBlue>=0)&&(color.rgbtBlue<=255)) {

      vuelta.rgbtBlue=color.rgbtBlue;

      } else {

            if (color.rgbtBlue>255) {

                  vuelta.rgbtBlue=255;

            } else {

                  switch (tipo) {

                        case 0:vuelta.rgbtBlue=0;break;

                        case
1:vuelta.rgbtBlue=abs(color.rgbtBlue);break;

                        case 2:if (color.rgbtBlue+128<0){

                                          vuelta.rgbtBlue=0;

                                    } else {
      vuelta.rgbtBlue=color.rgbtBlue+128;

                                    };break;

                  }

            }

      }
```

```
      return vuelta;}
```

3.3.2.4. Luminosidad

A veces al aplicar ciertos procesos sobre una imagen de 24 bits será más fácil hacerlo sobre la información de luminosidad que sobre los planos de color de forma independiente como es el caso del umbral.

La función devuelve un entero que se calcula a partir de la siguiente formula:

$$luminosidad = (0.299 * R) + (0.587 * G) + (0.114 \quad B)$$

Quedando la función así:

```
int __fastcall utilidades::luminosidad(TRGBTriple color){

      returnredondeo((0.299 * color.rgbtRed ) + (0.587 *
color.rgbtGreen) + (0.114 * color.rgbtBlue));

}
```

Redondeo es otra función que, aunque será muy utilizada, no hace falta explicar ya que solo se limita a redondear a entero.

3.3.2.5. ValidarImagen

En [3.3.1] ya se explicó que tenemos varias copias de la imagen. Esta función es muy simple pero muy útil ya que es la encargada de copiar la imagen final al buffer de undo y luego copiar la procesada a la final por medio del método Assign [3.1].

El código de la función es:

```
void __fastcall utilidades::validarImagen(){

      undo->Assign(imagen);

      imagen->Assign(procesada);

}
```

3.4. Procesado de imágenes

Ya está todo preparado para empezar a trabajar con las imágenes (clases, variables, funciones...).

Puesto que el programa es un interfaz para la clase utilidades se van a ver todos los procesados junto con su interfaz de usuario y los ejemplos correspondientes.

3.4.1. Negativo

```
void __fastcall utilidades::negativo();
```

Como vimos en la sección 2.1.1.1 este proceso invierto los pixeles de la imagen. Para ello la recorremos y se resta el valor de brillo a 255 para conseguir el valor inverso.

3.4.2. Ecualizar histograma

```
void __fastcall utilidades::eqhistograma();
```

Para aplicar lo visto en 2.1.1.6 el primer paso es calcular el histograma lo cual se hace al cargar la imagen, a continuación se debe obtener la función T de ecualización y finalmente aplicarla.

La función T es de valor único y monótonamente creciente. Para obtenerla se utilizará el siguiente código:

```
for (i=0;i<256;i++)

{

      suma=suma+histo_win->dameValor(i);

      funcionT[i]=(255*suma)/(anchura*altura); }
```

La función se va cargando con este bucle y tiene 256 valores porque cada uno se corresponde con un nivel de gris.

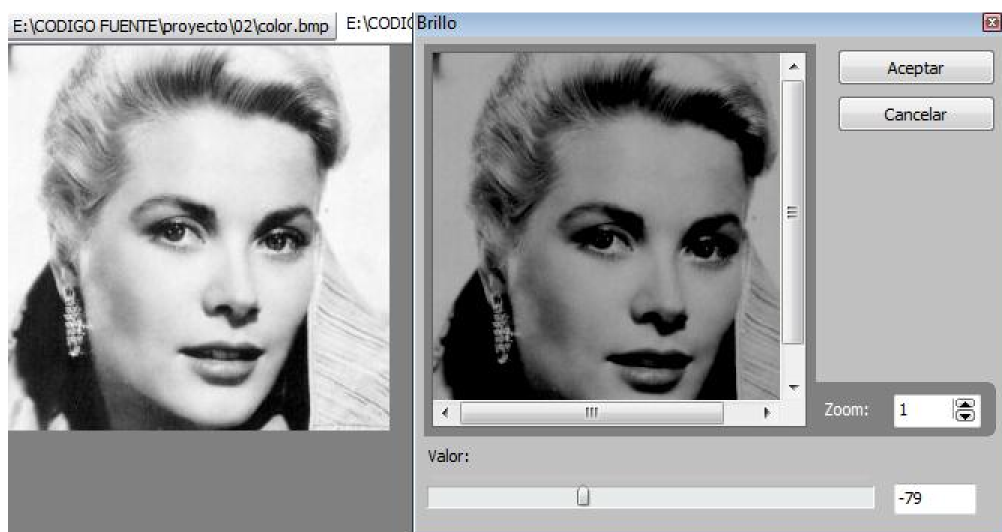
Para aplicar esta función se va a recorrer la imagen tomando el valor de cada pixel. Una vez tenemos su valor este se aplica como índice para leer el resultado de la función T. El funcionamiento es el mismo que encontramos cuando se trabaja con imágenes de color indexado solo que en este caso los valores de la paleta son los calculados en la función.

3.4.3. Brillo

```
void __fastcall utilidades::brillo(int valor);
```

Como ya se ha explicado en 2.1.1.2 para variar el brillo hay que sumar o restar un valor a cada pixel de la imagen.

Esta función utiliza la ventana dialogo [3.2] para recoger el entero que necesita para hacer el cálculo. Como se ve en la captura solo hay habilitada una barra.



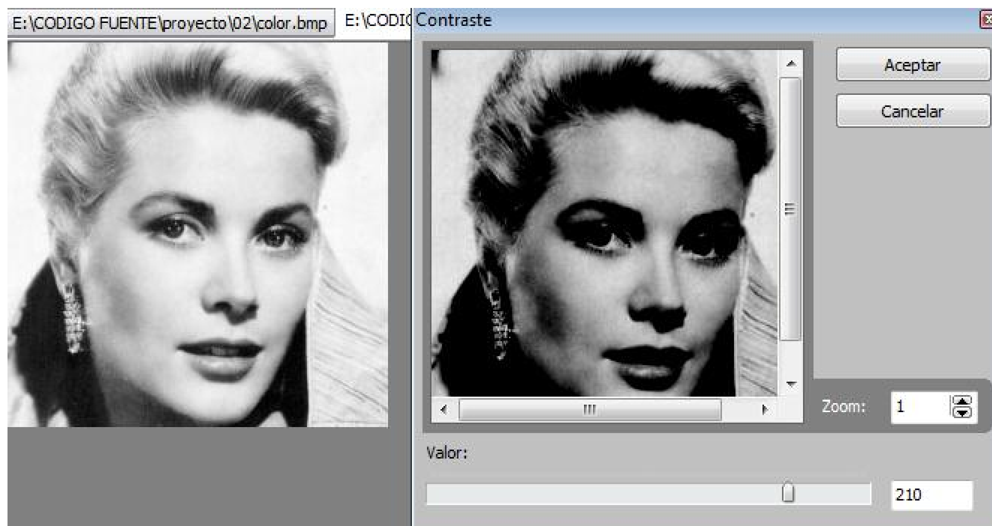
La barra está configurada para tener un rango entre -255 y 255 con un centro a cero.

El proceso es simple, se recorre la imagen y se suma el valor seleccionado al pixel leído. Para evitar problemas de desbordamiento se ha utilizado la función recortasimple [3.2.3.3].

3.4.4. Contraste

```
void __fastcall utilidades::contraste(int valor);
```

Para aplicar variaciones de contraste [2.1.1.3] también se va a utilizar la ventana de dialogo [3.2] para introducir su dato de entrada.



La barra se ha configurado para tener un mínimo igual a cero y un máximo de 255 con el centro fijado a 128.

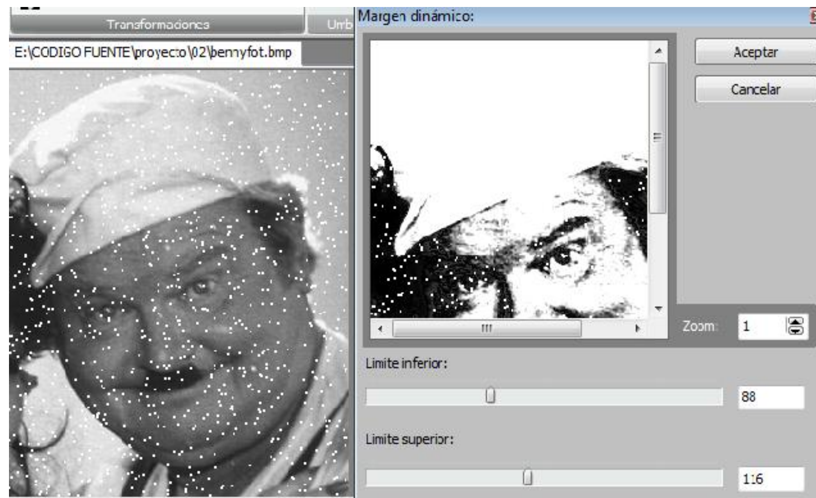
El proceso es similar al del brillo pero con algunas diferencias ya que tenemos que aplicar una fórmula que dará un resultado de tipo float. Viendo el código es más fácil de entender:

```
k=((valor/100.0f)-1.0f)/2.0f;
for (x=0; x <anchura; x++) {
for (y=0; y < altura; y++) {
      color=(valor/100.0f)*((getPixel(x,y,imagen)/255.0f)-k);
      if (color>1.0f) {
            color=1.0f;
      }
      if (color<0) {
            color=0.0;
        }
      putPixel(x,y,(byte) 255*color,procesada);}        }
```

Para empezar se tiene que calcular k que es un número que ya está en float. Como estamos trabajando en este formato hay que normalizar el valor del pixel leído para que su rango se encuentre entre 0 y 1 pero con decimales.

3.4.5. Margen dinámico

```
void __fastcall utilidades::margendinamico(intliminf, intlimsup)
```



El ajuste de margen dinámico es una forma de contraste con márgenes no simétricos. Como se puede ver utiliza la ventana dialogo para introducir el límite superior e inferior.

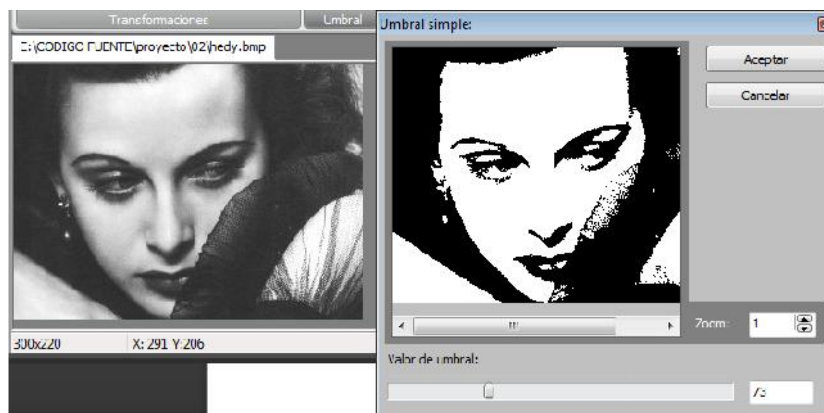
La fórmula que se aplica en este caso es la siguiente:

$$y = m \cdot x - k$$

3.4.6. Umbral simple

```
void __fastcall utilidades::umbral1(int valor);
```

Como ya sabemos el proceso de umbralizar una imagen [2.1.1.5] servirá para segmentar imágenes. Esta es la versión más simple del procesado.



Utilizando la ventana dialogo se le pasará el valor de umbral, si el brillo está por debajo de este se pondrá el pixel a cero, en caso contrario se pondrá a uno.

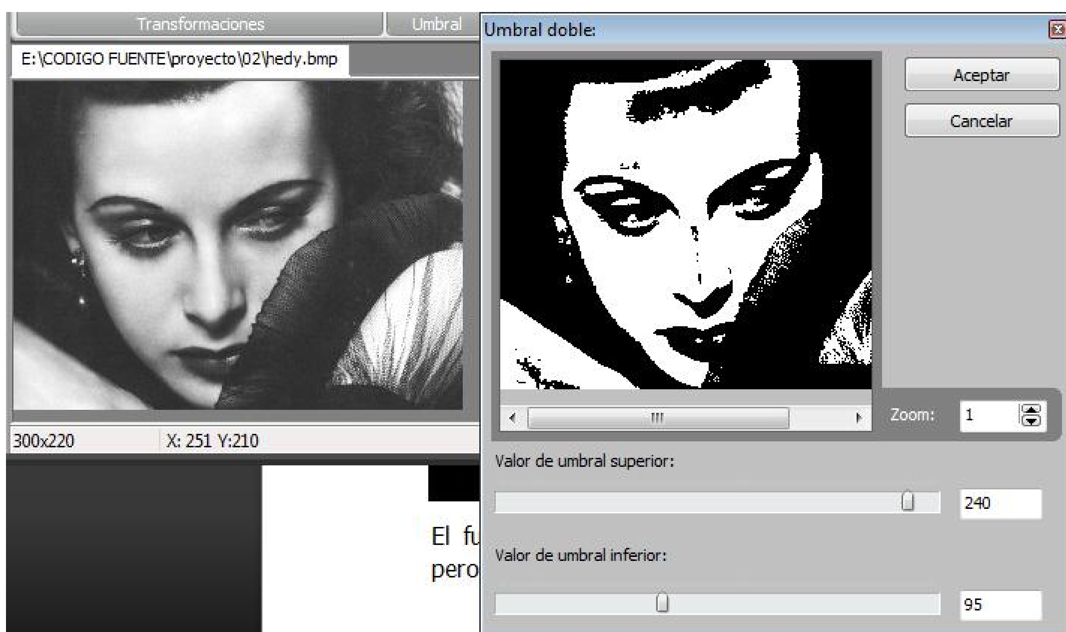
3.4.7. Umbral doble

```
void __fastcall utilidades::umbral2(intvalorinf,
                    intvalorsup);
```

El funcionamiento del umbral doble es similar al visto anteriormente [3.4.7] pero en este caso podremos extraer objetos entre dos brillos diferentes.

Por medio de la ventana dialogo se le pasarán los valores de umbral inferior y superior.

Todo lo que esté fuera de estos valores tomará un nivel de brillo cero y lo que es encuentre en la zona indicada tomará un brillo máximo.



3.4.8. Voltear horizontal

```
void __fastcall utilidades::voltearH();
```

Para voltear la imagen vamos a utilizar la función copyrect del objeto TCanvas que nos provee el API de Windows.

```
__fastcallCopyRect(constTRectDest, TCanvasCanvas, constTRectSource);
```

Esta función copia parte de una imagen especificada por un rectángulo en el TCanvas de destino dentro del rectángulo que le especifiquemos.

La forma de conseguir que la imagen se copie volteada es especificando los siguientes parámetros en el rectángulo de origen:

```
CopyRect(Rect(anchura-1,0,-1,altura),imagen->Bitmap-
>Canvas,Rect(0,0,anchura,altura));
```

Así el procesado es muy simple, primero se ajusta el fondo de la imagen según la configuración de la variable tipoFondo [3.3.1] y luego se copia la imagen completa volteada y se pega sobre sí misma.

Utilizar el API en lugar de bucles y scanLine hace el código más simple y más rápido.

3.4.9. Voltear vertical

```
void __fastcall utilidades::voltearV();
```

Como en el volteo horizontal [3.4.8] también se utilizará el API de Windows para facilitar la tarea. En este caso se le pasarán estos parámetros a la función copyRect en el rectángulo de origen:

CopyRect(Rect(0,altura-1,anchura,-1),imagen->Bitmap-
>Canvas,Rect(0,0,anchura,altura));

Este proceso también se hará en dos pasos. Se ajusta el color de fondo y luego se llama a la función copyRect.

3.4.10. Rotaciónes fijas

```
void __fastcall utilidades::rota90();

void __fastcall utilidades::rota180();

void __fastcall utilidades::rota270();
```

Esta función se hace con bucles. Solo hay que definir bien las variables del bucle para que la rotación salga correctamente. Es un proceso tan rápido que en lugar de comprobar la profundidad de bits de la imagen y luego aplicar los putpixel y getpixel necesarios se hará mediante la función Pixels de la clase TCanvas.

Tenemos un bucle doble que recorre la imagen en horizontal (variable i) y en vertical (variable j). Para rotar la imagen a 90 la línea encargada de copiar y escribir los pixeles quedará:

```
procesada->Bitmap->Canvas->Pixels[j][imagen->Bitmap->Canvas-
>ClipRect.Right-i-1]=imagen->Bitmap->Canvas->Pixels[i][j];
```

En el caso de 180 grados:

```
procesada->Bitmap->Canvas->Pixels[imagen->Bitmap->Canvas-
>ClipRect.Right-i-1][imagen->Bitmap->Canvas->ClipRect.Bottom-j-
1]=imagen->Bitmap->Canvas->Pixels[i][j];
```

Finalmente, para una rotación de 270 grados:

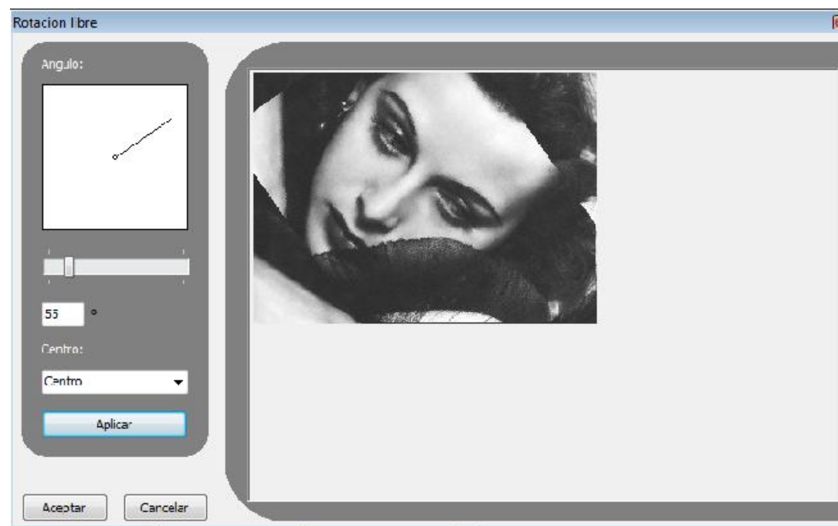
```
procesada->Bitmap->Canvas->Pixels[imagen->Bitmap->Canvas-
>ClipRect.Bottom-j-1][i]=imagen->Bitmap->Canvas->Pixels[i][j];
```

3.4.11. Rotación libre

```
void __fastcallutilidades::rotarLibre(intangulo, int cx,
int cy);
```

La rotación libre es un proceso simple también y como en el caso anterior [3.4.10] se hará por medio de bucles y la función Pixels del objeto TCanvas.

La ventana de este tipo de procesado es propia:



En la parte izquierda tenemos todos los parámetros de configuración que se le pasan a la función y a la derecha la previsualización.

El primer parámetro es el ángulo de rotación. Para elegirlo se puede escribir directamente en la caja de texto o moviendo la barra de desplazamiento, el ángulo de rotación se puede ver en el cuadro superior.

Los otros dos parámetros que faltan son el centro de rotación, pudiendo elegir una de las cuatro esquinas de la imagen o el centro de esta.

El primer paso de este proceso es sacar el seno y el coseno del ángulo y convertirlo a radianes, haciéndolo así en vez de dentro del bucle se ahorra una gran cantidad de velocidad.

A continuación se define el fondo de la imagen, en este caso se ha dejado la imagen original pero se podría haber puesto un color plano.

Finalmente se hace el bucle. La clave de este es mapear las coordenadas de otra forma a la hora de leer el pixel para luego escribir el pixel en las coordenadas correctas. El bucle es el siguiente:

```
for (i = 0; i < anchura-1; i++) {

      for (j = 0; j < altura-1; j++) {

            nuevai=(int)(cx+(coseno*(i-cx)-seno*(j-cy)));

            nuevaj=(int)(cy+(seno*(i-cx)+coseno*(j-cy)));

            if (nuevai>0) &&(nuevai<anchura) &&(nuevaj>0)&&
(nuevaj<altura)) {

                  x=(int)(nuevai);

                  y=(int)(nuevaj);

                  if (dameModo()==8) {

                  putPixel(i,j,getPixel(x,y,imagen),procesada);

                  } else {


      putPixel24p(i,j,getPixel24(x,y,imagen),procesada);

                  }

             }

        }

  }
```

3.4.12. Cambiar tamaño del lienzo

```
void __fastcall utilidades::cambiatama(inttanchura,
inttaltura);
```

La utilidad de esta llamada es cambiar el tamaño del lienzo sin afectar a la imagen que contiene o lo que es lo mismo dar más o menos zona de dibujo en la imagen que estamos tratando. Esta utilidad también tiene su propia ventana.

El código de esta función es muy simple. Solo hay que ajustar las variables de anchura y altura [3.3.1] y las mismas variables dentro de la imagen las cuales son las propiedades Width y Height de la clase TBitmap.

3.4.13. Redimensionar imagen

```
void __fastcall utilidades::escalado(intcantidad,bool lineal);
```

Esta función cambia el tamaño del lienzo y de la imagen que contiene. Los parámetros de entrada son la escala a la que se redimensiona la imagen y la variable tipo boolean indica si queremos el escalado con filtrado lineal o no.

En esta ventana se tendrá que escoger el valor de escala en la caja de texto, cuando se introduce se calcula el nuevo tamaño de la imagen y se escribe en las cajas de texto superiores, luego se elige el tipo de filtrado y se pulsa el botón Escalar.

El primer paso de este proceso consiste en cambiar el tamaño del lienzo como vimos anteriormente [3.4.12] y después el de la imagen.

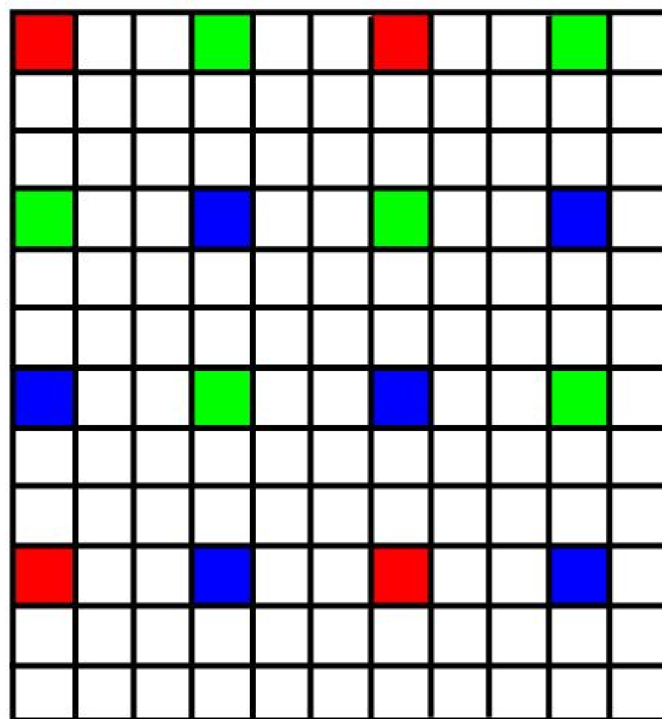
Para empezar vamos a ver como se hace sin filtrado. Se utilizará la función API stretchDraw:

```
__fastcallStretchDraw(constTRectRect, TGraphic Graphic);
```

Esta función forma parte del objeto TCanvas y su misión es copiar una imagen dentro de otra pero con el tamaño definido por el rectángulo Rect.

Entonces solo tenemos que calcular el tamaño del rectángulo y hacer la llamada a la función. Es un método rápido pero deja la imagen muy pixelada al no aplicar filtro.

El otro modo de escalado es por medio de filtrado lineal. Este filtro se basa en que tenemos la imagen dibujada con los pixeles separados entre sí con una separación dada por el factor de escala.

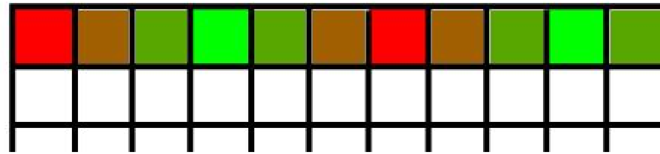


Una vez tenemos ya la imagen con sus pixeles separados se aplica interpolación lineal entre ellos. Esto lo haremos en dos pasadas horizontal y vertical.

En la horizontal primero obtenemos los colores inicial y final. En este ejemplo el color inicial será el rojo y el final el verde. Ahora se calcula el incremento que hay que sumar en los pixeles intermedios mediante esta fórmula:

$$incremento = \frac{colorFin - colorIni}{Escala}$$

El valor de incremento se suma al pixel que hay a la derecha del rojo, ahora se coge este valor, se le suma el incremento y se obtiene el valor del pixel que irá a la izquierda del verde.



Ahora que tenemos todas las líneas horizontales con sus pixeles interpolados es fácil repetir este proceso entre pixeles verticales para completar la imagen.

Si la imagen es en color se tiene que hacer la interpolación por separado en cada canal de color para obtener el mismo resultado.

El código para la interpolación en escala de grises será:

```
//pasada horizontal (cantidad es el factor de escala)

      for (i = 0; i< anchura-1; i++) {

            for (j=0; j < altura-1; j++) {

            color=getPixel(i,j,imagen);

              color2=getPixel(i+1,j,imagen);

            incremento=(color2-color)/cantidad;

            for (k=0; k < cantidad; k++) {


      putPixel((i*cantidad)+k,j*cantidad,color+(k*incremento),procesad
a);

              }

            }

      }

//pasada vertical

for (i = 0; i < (anchura-1)*cantidad; i++) {

      for (j = 0; j < (altura-1); j++) {

            color=getPixel(i,j*cantidad,procesada);

            color2=getPixel(i,(j+1)*cantidad,procesada);

            incremento=(color2-color)/cantidad;

            for (k=0; k < cantidad; k++) {
```

```
            if ((j*cantidad)+k<(altura-1)*cantidad) {

            putPixel(i,(cantidad*j)+k,color+(k*incremento),procesada);

            }

            }

      }

      }
```

3.4.14. Filtros predefinidos

```
void __fastcall utilidades::LPF();

void __fastcall utilidades::HPF();

void __fastcall utilidades::media();

void __fastcall utilidades::laplaciano();

void __fastcall utilidades::horizontal();

void __fastcall utilidades::vertical();
```

Todos estos filtros se pueden ver a la vez ya que el proceso es el mismo. Ya vimos que este tipo de filtrados se hace por convolución [2.1.2.1]. Esta operación es fácil de trasladar al código fuente si se está utilizando un kernel fijo.

Aplicar un kernel a la imagen es tan fácil como recorrer la imagen pero capturando el pixel central y los que le rodean para, después de aplicar los cálculos que requiere el kernel, sumar el resultado. Así tenemos que para todos estos filtros el proceso es prácticamente el mismo, solo hay que tener en cuenta las operaciones que nos impone el kernel.

Por ejemplo, el código para un filtro paso bajo sería así:

```
for (i = 2; i< anchura-1; i++) {

      for (j=2; j < altura-1; j++) {

            color=(getPixel(i-
1,j,imagen)+getPixel(i,j,imagen)+getPixel(i+1,j,imagen)+   getPixel(i-
1,j-1,imagen)+getPixel(i,j-1,imagen)+getPixel(i+1,j-1,imagen)+
```

```
      getPixel(i-
1,j+1,imagen)+getPixel(i,j+1,imagen)+getPixel(i+1,j+1,imagen));

            putPixel(i,j,color/9,procesada);

      }

 }
```

En la variable color se recoge el pixel central y los que le rodean, como el kernel dice que hay que dividir entre 9 esta operación se hace al pintar el color.

3.4.15. Mediana

```
void __fastcall utilidades::mediana();
```

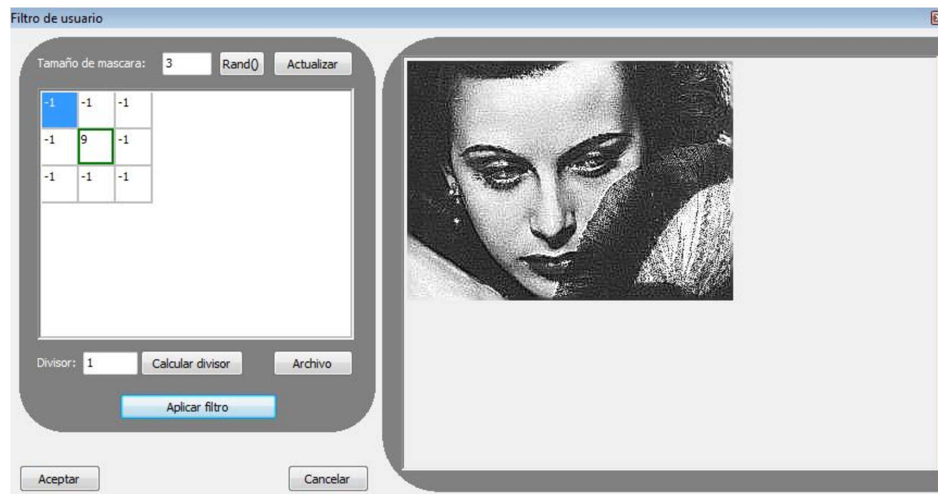
Para aplicar el filtro mediana [2.1.3.2] se recogen el pixel central y los que le rodean y se guardan dentro de un vector. Este vector se pasará a la función quicksort:

```
void __fastcall utilidades::quicksort(intarr[], intlow, inthigh);
```

A esta función se le pasará el vector y el valor de inicio y final de este. Al ejecutarla el vector ya estará ordenado y solo tendremos que escoger el valor central de este para utilizarlo como valor de color.

3.4.16. Filtro de usuario

```
void __fastcallutilidades::filtroUsr(int **mascara,int tama,
                    int divisor);
```



Esta ventana es la que va a definir la máscara de entrada y su divisor para aplicar cualquier filtro que se nos ocurra pero tiene algunas opciones extra.

El botón archivo servirá para abrir y guardar máscaras que ya se han hecho, hay un botón para calcular el divisor y en la parte superior está el botón Rand() que pondrá números aleatorios en la máscara.

El funcionamiento de esta llamada es similar a la de los filtros predefinidos con la excepción de que ahora no sabemos el tamaño de la máscara y habrá que hacer algunos cálculos extra.

Como no sabemos el tamaño de la máscara ahora se tiene que hacer un bucle un poco más complicado ya que tenemos que recorrer la máscara y la imagen a la vez. Esto se hará así:

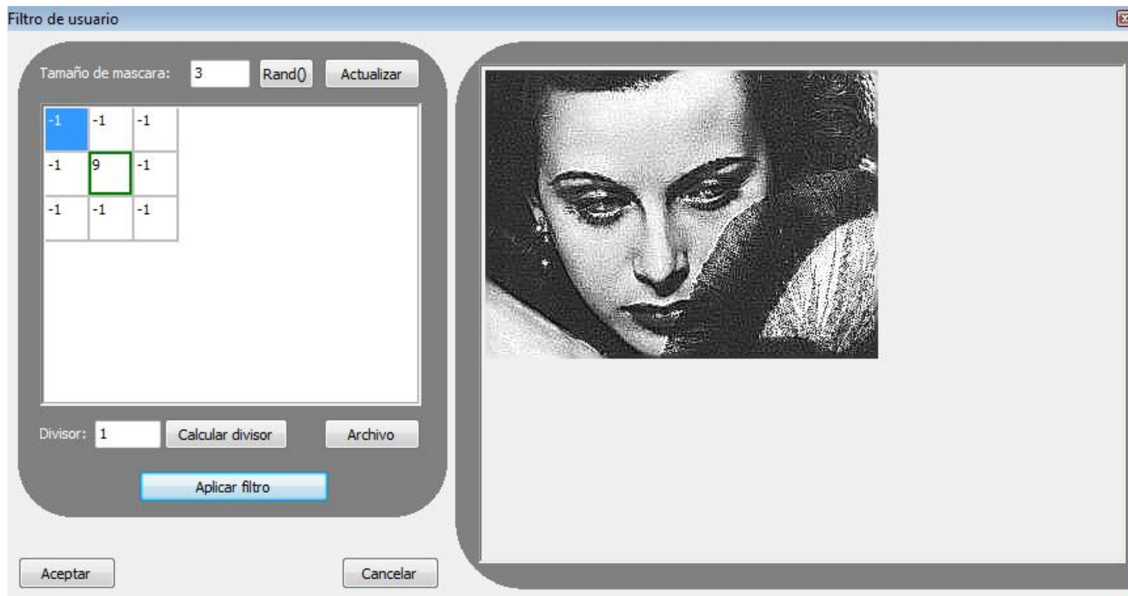
```
for (i = 0; i < anchura; i++) {
            for (j=0;j<altura;j++){
                  if ((j-tama>=0)&&(j+tama<altura)&&(i-tama>=0) &&
(i+tama<anchura)) {
                  valor=0;
                  for (x=0; x <tama; x++) {
                  for (y = 0; y<tama ; y++) {

      valor=valor+(mascara[x][y]*getPixel(i-((tama/2)-x),j-((tama/2)-
y),imagen));
                                    }
                              }


                  valor=valor/divisor;
            if ((valor>=0)&&(valor<=255)) {
                        putPixel(i,j,valor,procesada);
                   } else {

      putPixel(i,j,recortaPixel(valor,1),procesada);
                         }
                  }
             }
```

Se puede comprobar que ahora se recorre la imagen mediante el bucle que utiliza los valores i, j mientras que se la máscara se recorre con el bucle que mueve los valores x, y.

3.4.17. Gradientes

```
void __fastcallutilidades::gradientes(boolprewitt,
                    boolraiz);
```

Desde esta ventana se pueden aplicar las gradientes. Se debe elegir el tipo de máscara y el tipo de cálculo final.



Como ya vimos en la explicación de las variables [3.3.1] esta función utiliza matrices dinámicas para guardar los datos intermedios. Así que el primer paso es iniciar todas estas variables y cargarlas a cero.

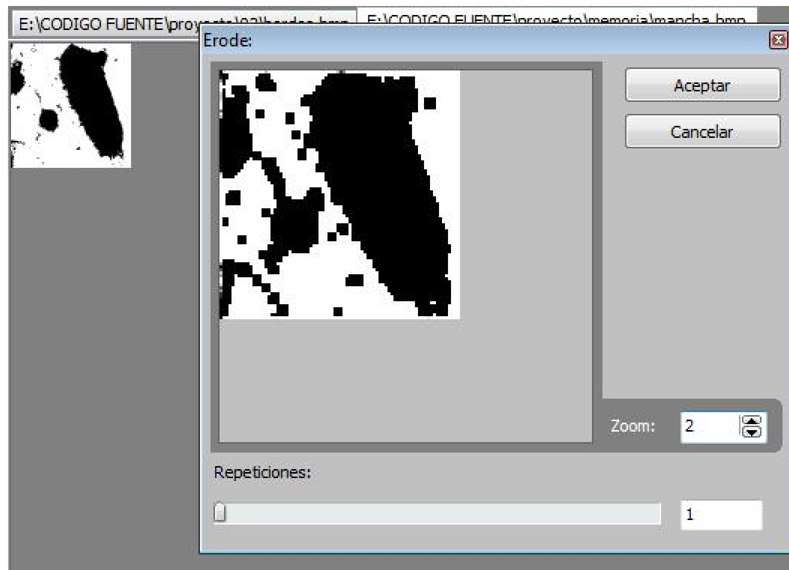
Los valores de las gradientes vertical y horizontal se guardan en las variables datos[][] para luego aplicar el tipo de cálculo seleccionado leyendo directamente desde aquí. Este será todo el proceso que se hace, no se pondrá el código fuente aquí ya que es muy extenso.

3.4.18. Erode

```
void __fastcall utilidades::erode(int repeticiones);
```

Esta operación morfológica solo necesita como parámetro de entrada el número de veces que se quiere repetir por lo que utilizará la ventana dialogo.

Esta operación solo es aplicable a imágenes en escala de grises que ya han sido umbralizadas.



Para poder repetir el proceso varias veces se utiliza una copia de la imagen alojada en un TPicture para no corromper la imagen original ni la procesada. Una vez se ha terminado el proceso se copiará esta imagen sobre procesada.

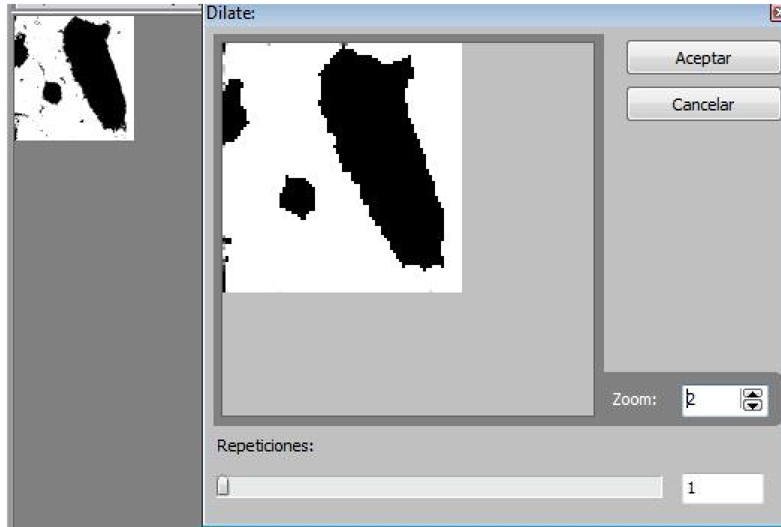
Como ya vimos en [2.1.4.1] se utiliza la operación booleana AND como base para hacer las comparaciones necesarias. Así vamos a recorrer la imagen haciendo la siguiente comprobación:

```
if ((getPixel(i-1,j-1,imgrep)==255)&&(getPixel(i,j-1,imgrep)==255)
&&(getPixel(i+1,j-1,imgrep)==255) && (getPixel(i-1,j,imgrep)==255) &&
(getPixel(i,j,imgrep)==255) && (getPixel(i+1,j,imgrep)==255) &&
(getPixel(i-1,j+1,imgrep)==255) && (getPixel(i,j+1,imgrep)==255) &&
(getPixel(i+1,j+1,imgrep)==255))

            {

                        putPixel(i,j,255,procesada);

               } else {

                  putPixel(i,j,0,procesada);

               }

            }

        }

  }
```

3.4.19. Dilate

```
void __fastcall utilidades::dilate(int repeticiones);
```

Como en la operación anterior [3.4.18] solo necesitamos un valor para pasar a la función y por esa razón se vuelve a utilizar la ventana dialogo.



Dilate solo se puede aplicar a imágenes en escala de grises que ya hayan pasado por el umbral. Como en la operación anterior [3.4.18] también se trabaja sobre una copia almacenada en un objeto TPicture.

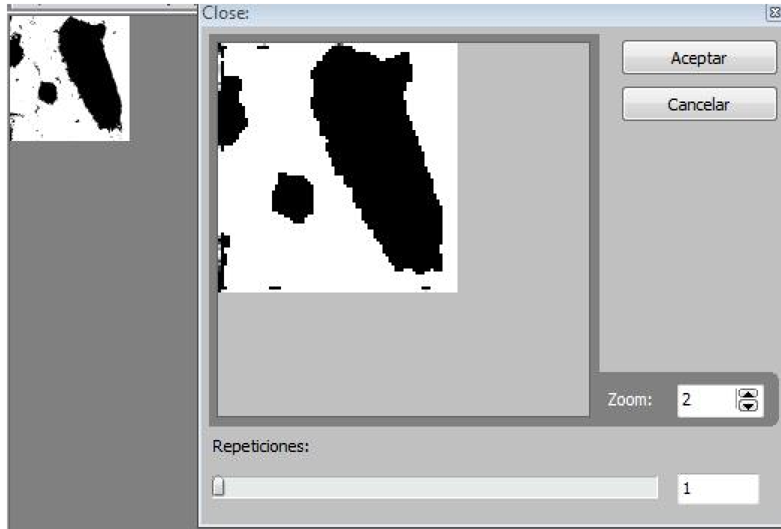
En este caso la comparación se hace mediante la operación booleana OR.

```
for (i=1 ;i< anchura-1; i++) {

            for (j=1; j < altura-1; j++) {

if ((getPixel(i-1,j-1,imgrep)==255)||(getPixel(i,j-1,imgrep)==255)
||(getPixel(i+1,j-1,imgrep)==255) || (getPixel(i-1,j,imgrep)==255) ||
(getPixel(i,j,imgrep)==255) || (getPixel(i+1,j,imgrep)==255) ||
(getPixel(i-1,j+1,imgrep)==255) || (getPixel(i,j+1,imgrep)==255) ||
(getPixel(i+1,j+1,imgrep)==255))

            {

                        putPixel(i,j,255,procesada);

               } else {

                  putPixel(i,j,0,procesada);

               }

            }

        }

  }
```

3.4.20. Close

```
void __fastcall utilidades::close(int repeticiones);
```

Una vez más la función se controla desde la ventana dialogo porque solo le vamos a pasar un dato.



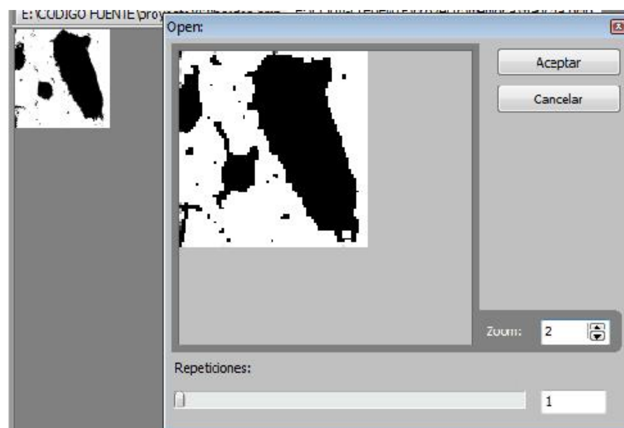
Esta función se hace en unos pocos pasos y es fácil de programar ya que hay que utilizar procesos que se han escrito. Primero se hace una copia de la imagen original, luego se aplica el proceso de dilate y se llama a la función validarImagen [3.3.2.5]. Ahora le aplicamos erode y se vuelve a reponer la imagen original teniendo el resultado final en la copia procesada.

3.4.21. Open

```
void __fastcall utilidades::open(int repeticiones);
```

Este proceso es similar a Close [3.4.20] pero en este caso vamos a hacer primero erode y después dilate.

Como solo necesitamos pasar un solo parámetro también se controlará con la ventana dialogo.



3.4.22. Cálculo de áreas

```
int __fastcallutilidades::calc_area();
```

El cálculo de áreas es un proceso sencillo una vez se ha preparado la imagen. Ya se vio antes como se hace [2.1.5.1], al llamar a esta función se contarán los pixels blancos de la imagen.

Para presentar el resultado se utilizará un cuadro de dialogo estándar que nos dará el resultado de esta llamada.

3.4.23. Cálculo de perímetro

```
int __fastcall utilidades::calc_perimetro();
```

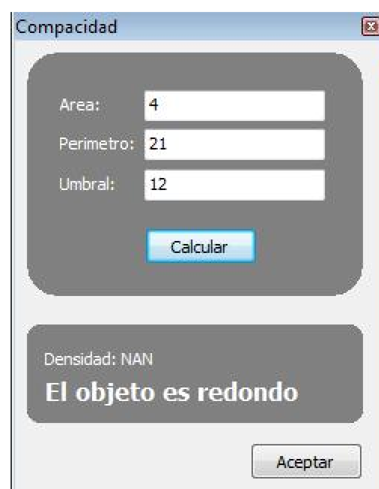
Este cálculo se hará mediante morfología [2.1.5.1]. El primer paso es aplicar un erode con una sola repetición a continuación llamaremos a esta función:

```
TPicture* __fastcallutilidades::XOR(TPicture* imgIn1, TPicture*
imgIn2);
```

Que aplica la operación booleana XOR entre dos imágenes. La aplicamos poniendo como entrada la imagen original y la procesada y guardando el resultado en la procesada. Como la imagen ya está preparada solo queda recorrerla y contar el número de pixeles blancos en la imagen procesada.

Para presentar el resultado se utiliza un cuadro de dialogo estándar.

3.4.24. Compacidad

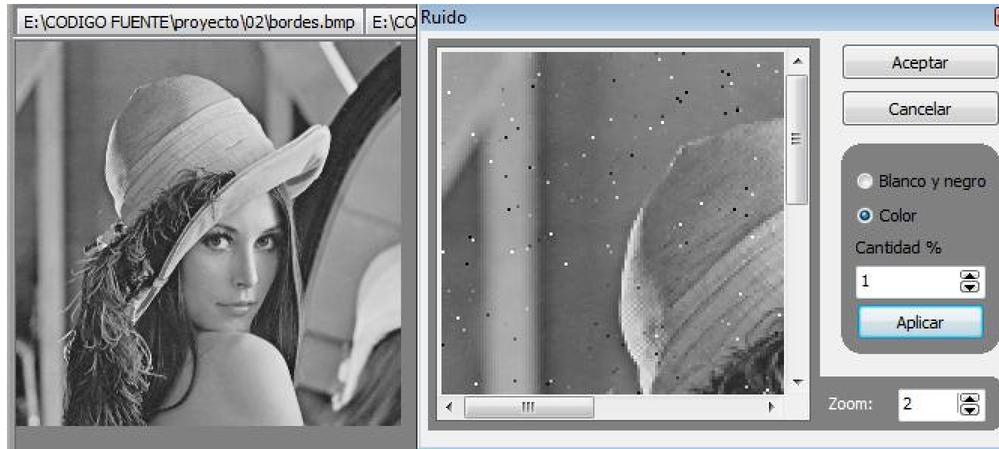


Esta función no es parte de la clase utilidades pero para hacer los cálculos se necesita el área [3.4.22] y el perímetro [3.4.23]. Al pulsar en calcular aplica la fórmula que ya vimos [2.1.5.1]

3.4.25. Ruido

```
void __fastcall utilidades::ruido(intcantidad,int colores);
```

Con esta función podremos añadir ruido aleatorio a la imagen. Se puede poner ruido en blanco y negro o en color (para imágenes en 24 bits).



Primero se calcula el número de pixeles que se van a sustituir por ruido teniendo en cuenta el valor de cantidad y la anchura de la línea. Solo nos interesa la anchura porque se aplicará a través de la función scanLine que trabaja sobre líneas.

El código para la versión en blanco y negro sería:

```
numpixels=(cantidad*anchura)/100;

if (dameModo()==8){

      for (i = 0; i<altura; i++) {

            p=(byte *) procesada->Bitmap->ScanLine[i];

            for ( j=0; j < numpixels+1; j++) {

                  p[rand()%(anchura+1)]=rand()%colores;

            }

 }
```

La diferencia entre trabajar en blanco y negro o no es que se pondrá un valor 0 o 255 o que se pondrá un color aleatorio con un valor entre 0 y 255.

3.5. Dibujando

Para dibujar se utilizarán tanto las funciones proporcionadas por la clase TCanvas como las del API pero se llamarán desde dentro de la clase utilidades.

3.5.1. Paleta de colores

Dependiendo de la profundidad de bits se debe utilizar una paleta u otra. El caso de las imágenes de 24 bits es el más fácil ya que utiliza el cuadro de dialogo de color de Windows.

Si la imagen es en escala de grises se desplegará un panel en el que tenemos una imagen con un degrado de 256 grises. Al pulsar sobre la imagen se extrae el color con la función pixels que nos da el objeto TCanvas.

3.5.2. Herramientas de dibujo

Estas herramientas y las de portapapeles funcionan en dos capas. Tenemos una capa superior que es un TPaintBox que nos da información diferente según la herramienta seleccionada, por ejemplo para dibujar un rectángulo en esta capa se dibujará la pre visualización de esta o a la hora de dibujar con pincel se verá el tamaño de este.

La capa inferior es la que presenta la imagen mediante un objeto del tipo TImage. En esta capa lo único que hacemos es dibujar la imagen final y actualizarla cuando haga falta, las formas que se pintan con estas herramientas se hacen sobre la imagen final que contiene el objeto utilidades.

3.5.3. Portapapeles

Las operaciones para cortar, copiar y pegar vienen del objeto ClipBoard.

Para copiar y cortar se utiliza la función copyRect que ya vimos antes.

El pegado puede ser simple o "especial". Este último puede pegar la imagen aplicando las operaciones booleanas AND o XOR. Para el pegado simple tenemos la función del objeto utilidades pintaImagen. Si queremos utilizar pegado especial tenemos la siguiente función API:

```
BOOL BitBlt(  _In_  HDC hdcDest,  _In_  intnXDest,  _In_  intnYDest,
_In_  intnWidth,  _In_  intnHeight,  _In_  HDC hdcSrc,  _In_  intnXSrc,
_In_  intnYSrc,  _In_  DWORD dwRop);
```

Entre otros parámetros se le pasan las imágenes origen y destino, sus puntos para recortar y lo más importante el modo de pegado. Si queremos pegar aplicando la operación AND se indicará en la variable dwRop poniéndole SRCERASE, si la función elegida es XOR el valor será SRCINVERT.

3.6. Formatos de imagen

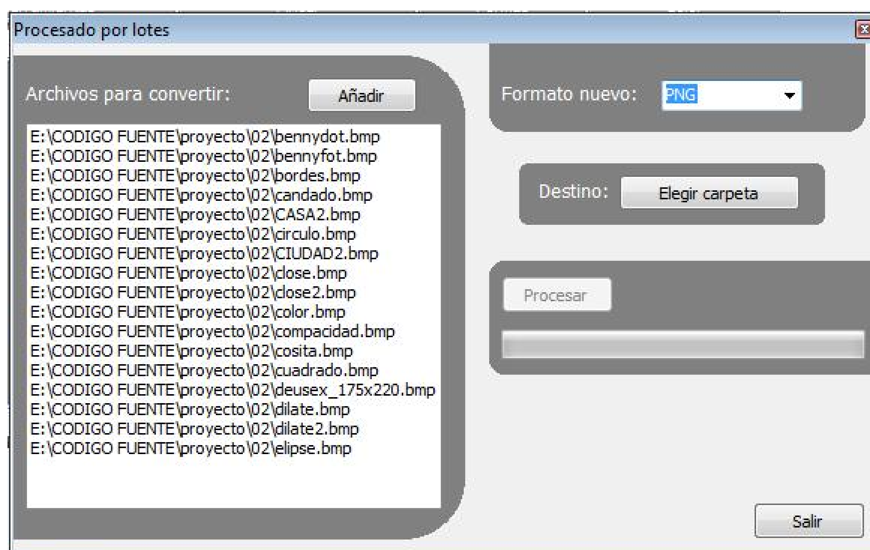
TekPaint es capaz de trabajar con imágenes en formato bmp, jpg, png y gif. En los formatos gif y png no se tiene en cuenta la transparencia ni la animación.

Para trabajar con estos formatos el compilador nos proporciona las librerías jpeg.hpp, GIFImg.hpp y pngimage.hpp. Con estas librerías podemos crear los objetos correspondientes para cargar los datos de imagen dentro.

A la hora de cargar una imagen primero se lee la extensión de esta en el nombre de archivo para luego cargarla dentro del objeto del tipo necesario. Una vez está cargada se copia dentro del objeto TPicture contenido en la clase utilidades y ya se puede destruir el objeto que haya hecho falta para cargar.

3.6.1. Procesado por lotes

Al ser tan fácil pasar de un formato de imagen a otro también es posible hacer una utilidad de procesado por lotes. Esta utilidad, accesible desde el menú del programa, puede convertir rápidamente imágenes guardadas en diferentes tipos de archivo a un solo formato de archivo.



Aquí se recorre la lista de imágenes cargadas, se lee la extensión, se carga en el objeto de su tipo y se pasa al objeto requerido según la extensión.

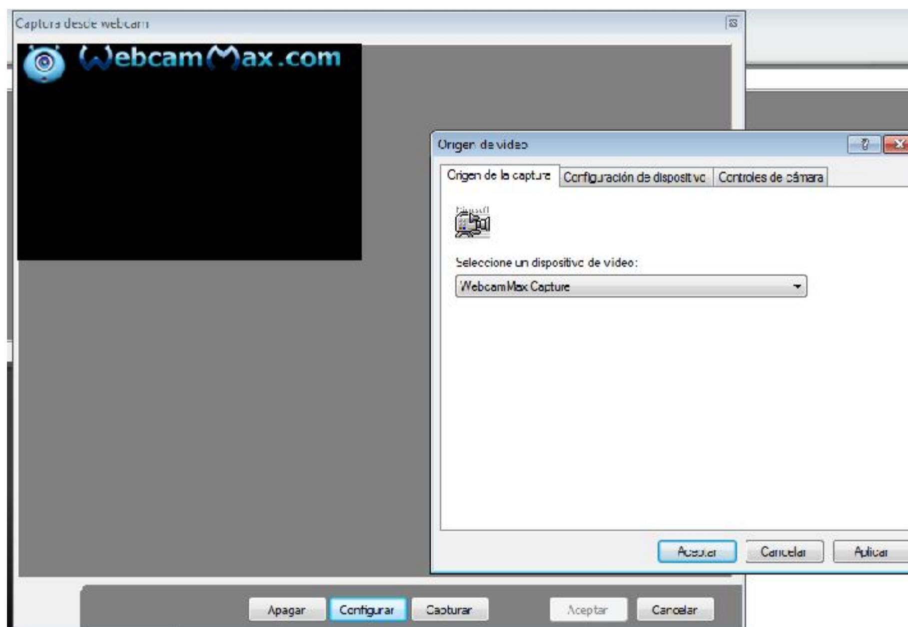
3.7. Captura

3.7.1. Captura de pantalla

Para capturar la pantalla se utiliza la llamada API BitBlt. A esta llamada hay que pasarle la imagen de origen y la de destino. Hasta este momento como imagen de origen se le pasa el handle un TCanvas que se convierte en DC (DeviceContext).

Si lo que se quiere copiar es toda la pantalla se le pasará el DC con handle cero. Éste es el que corresponde a la pantalla.

3.7.2. Captura por webcam



La captura por webcam se hace con la librería Video For Windows que nos da todas las funciones API de control multimedia necesarias.

Antes de empezar a capturar se tiene que encender la webcam si esta existe. Para esto se guardará en la variable visor del tipo HWND (HWindow) la superficie en la que se dibujará la captura de cámara. Cuando ya tenemos configurado el visor se manda el mensaje de encendido indicándole la superficie de visión y se configura. Las llamadas API son:

```
visor=capCreateCaptureWindowA("ventana",WS_VISIBLE|WS_CHILD,0,0,320,20
0, Panel1->Handle,0);

if (visor!=0) {

fOK = SendMessage (visor, WM_CAP_DRIVER_CONNECT, 0, 0L);
```

```
capPreviewRate(visor, 30);

capPreview(visor, TRUE);

//para ver si el render va con overlay

capDriverGetCaps(visor, &CapDrvCaps, sizeof (CAPDRIVERCAPS));

if (CapDrvCaps.fHasOverlay)

      capOverlay(visor, TRUE);
```

Una vez se ha iniciado la cámara el sistema se encargará de dibujar lo que esté capturando en nuestra superficie de visualización. Para capturar se llamará a la función BitBlt.

La webcam tiene una ventana de configuración, esta varía según el modelo. Para poder mostrarla tenemos esta llamada:

```
capDlgVideoSource(visor);
```

Cuando se ha terminado de capturar hay que apagar la cámara para lo que tenemos esta función:

```
fOK = SendMessage (visor, WM_CAP_DRIVER_DISCONNECT, 0, 0L);
```

3.8. Transformada de Fourier

Para calcular la FFT se ha utilizado una librería externa llamada FFTW.

Lo primero que hay que hacer es pasar los datos de la imagen a un vector. Ahora se calcula el tamaño de la imagen en la que se guardará el resultado de la transformada con esta línea:

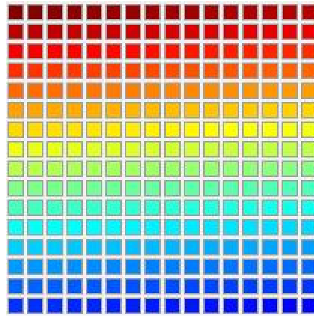
```
tamaY=(altura/2)+1;
```

Ahora se crea un array dinámico para guardar el resultado y se llama a la librería para que lo llene mediante esta función:

```
fftw_plan fftw_plan_dft_r2c_2d(int n0, int n1, double *in,
fftw_complex *out, unsigned flags);
```

Como ya se ha dicho el resultado se almacena en un array. En la posición 0 de este van los valores reales y en la posición 1 los valores imaginarios. Para hacer la representación se calculará el módulo y el logaritmo.

La representación gráfica se hace en una imagen de 256 colores con paleta (color indexado). La paleta que utilizamos es esta:



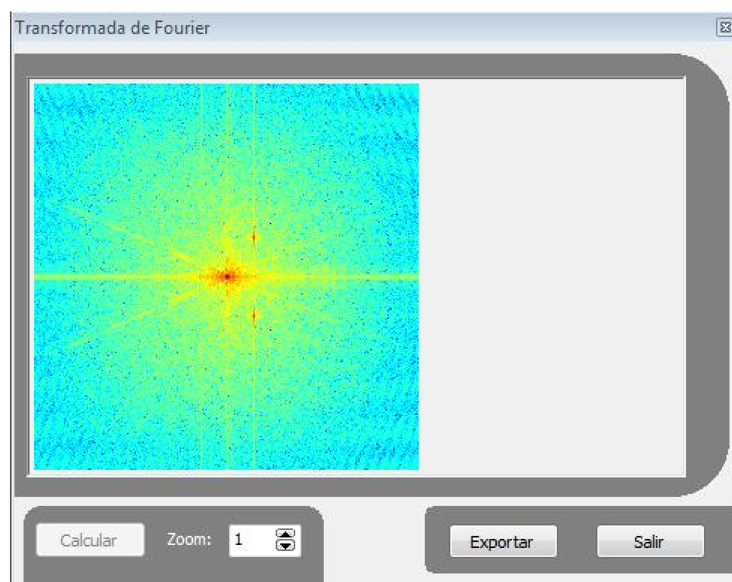
Para ajustar el resultado del módulo a los valores necesarios para utilizar una paleta de color indexado se hacen dos conversiones:

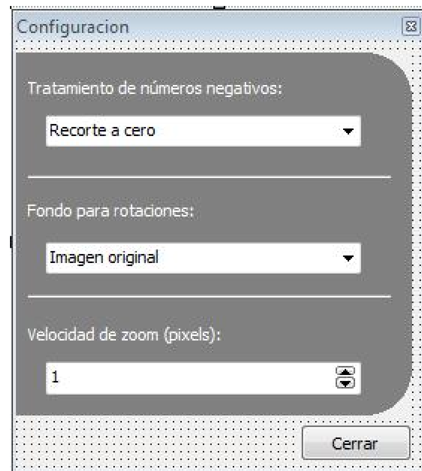
```
scale=(float)(255.0/(max-min));

pixel[k]=doc->recortaSimple((byte)(((amp[i*tamaY+j]-
min)*scale+0.5)+1));
```

La primera conversión se hace en un bucle y la segunda en el bucle que se encarga de dibujar la imagen. Esta imagen solo nos da la parte inferior de la representación, para obtener la parte superior se pegará esta volteada verticalmente sobre la imagen final para luego pegar la original debajo.

Los valores calculados se pueden exportar en un archivo html para su posterior consulta o en csv para importar en Excel.

El archivo html se genera escribiendo tanto los datos y los tags en un campo de texto multilínea que luego es fácilmente exportable a un archivo. El archivo csv se escribe igual pero sin añadir tags html.





La configuración del programa se guardará en el registro de sistema. Esta tarea se hace con la librería Registry.hpp.

(Predeterminado)	REG_SZ	(valor no establecido)
archivo0	REG_SZ	E:\CODIGO FUENTE\proyecto\02\Lena.bmp
archivo1	REG_SZ	E:\CODIGO FUENTE\proyecto\memoria\mancha.b...
archivo2	REG_SZ	E:\CODIGO FUENTE\proyecto\02\bordes.bmp
archivo3	REG_SZ	E:\CODIGO FUENTE\proyecto\memoria\mancha.b...
archivo4	REG_SZ	E:\CODIGO FUENTE\proyecto\memoria\erode.bmp
archivo5	REG_SZ	E:\CODIGO FUENTE\proyecto\memoria\dilate.bmp
archivo6	REG_SZ	E:\CODIGO FUENTE\proyecto\memoria\open.bmp
archivo7	REG_SZ	E:\CODIGO FUENTE\proyecto\memoria\close.bmp
archivo8	REG_SZ	E:\CODIGO FUENTE\proyecto\memoria\close.bmp
archivo9	REG_SZ	E:\CODIGO FUENTE\proyecto\memoria\dilate.bmp
filtrado	REG_DWORD	0x00000000 (0)
fondo	REG_DWORD	0x00000000 (0)
max	REG_DWORD	0x0000000a (10)
zoom	REG_DWORD	0x00000001 (1)

Aquí se pueden ver los datos que se guardan. Además de la configuración del programa se guardan los documentos recientes.

Al iniciar T3k Paint se cargará la configuración y los documentos recientes. Estos documentos se almacenan en una pila mientras el programa está en ejecución para guardarlos al salir de él.

4. CONCLUSIONES

En un principio el programa fue implementado utilizando el lenguaje de programación Delphi pero debido a que el acceso a punteros es más sencillo en lenguaje C++ fue necesario convertirlo a este lenguaje. Así la versión final ha sido programada en el entorno C++ Builder.

La primera versión del programa era funcional pero mal estructurada de forma que se ha vuelto a diseñar introduciendo todo los cálculos dentro de la clase utilidades para ahorrar recursos y mejorar la estabilidad.

Una vez terminada la nueva versión se adaptó para utilizar los controles tipo ribbon y añadir la posibilidad de abrir varios documentos a la vez. Para lograr este objetivo se hubo de consultar una gran cantidad de documentación referente a acceso de punteros y el API de Windows.

Para acceder a los diferentes documentos se ha optado por la opción de utilizar pestañas.Debido a un error de diseño de la clase utilidades original se hicieron pruebas convirtiendo el programa a MDI (Multiple Document Interface). Este nuevo sistema no provoca errores pero se pierde el diseño original así que para volver a poder utilizar pestañas se vuelve a reescribir la clase utilidades.

El API de Windows se utilizará para gestionar las herramientas de dibujo y los contenedores de imágenes.

Finalmente ha sido necesario también conseguir documentación referente al sistema de programación de ayuda utilizado en las nuevas versiones de Windows.

5. LÍNEAS FUTURAS

El principal cuello de botella que se ha encontrado a la hora de realizar los cálculos y procesos de las imágenes se localiza en las funciones de acceso a los pixeles. Las soluciones pasarían por volver a implementarlas utilizando las librerías DirectX, OpenGL o incluso el sistema de aceleración por hardware CUDA.

Existe la posibilidad de compilar el objeto utilidades como una librería externa para que pueda ser reutilizable por otros programas.Esta opción haría muy sencilla la tarea de añadir nuevos filtros y procesos.

Un añadido que se hubiera podido hacer sería emplear con imágenes de 32 bits con transparencia para ofrecer la posibilidad de trabajar por capas con las imágenes. Para ello también sería necesario crear un formato gráfico nuevo que pudiera almacenar esa información y que incluyera algún tipo de compresión sin perdidas.

6. BIBLIOGRAFÍA

1) Apuntes de Tratamiento digital de la imagen.
2) Ignacio Bosch Roig. Pablo SanchisKilders. Jorge Gosálbez Castillo. José Javier López Monfort. Prácticas de tratamiento digital de la imagen. Editorial UPV (2009). Ref: 2009.1018
3) Apuntes de Tratamiento de imagen digital.
4) Marco Cantú. Delphi 2009 Handbook. Embarcadero Technologies (2009).
5) Marco Cantú. Building User Interfaces with Delphi 2009. Embarcadero Technologies (2008).
6) Juana Sánchez Rey. Fundamentos de Programación (Ciclo superior A.S.I.). BorlandC++ Builder (2002/2003)
7) Foley, van Dam, Feiner, Hughes. Computer Graphics: Principles and Practice. Addison-Wesley Publishing Company (1990). ISBN 0-201-12110-7
8) Ginés García Mateos. Procesamiento audiovisual. Dept. de informática y sistemas. Universidad de Murcia.
9) William K. Pratt. Digital Image Processing. Wiley (2007).
10) Dr. Boris Escalante Ramírez. Procesamiento Digital de Imágenes. Agosto 2006.
11) Michel E. Mortenson. Computer graphics handbook. Industrial Press (1990)
12) Michael Abrash. Graphics Programming Black Book. (2001)
13) RAD Studio XE documentation Wiki <docwiki.embarcadero.com>
14) Win 32 API <msdn.microsoft.com>
15) Torry's Delphi Pages. <www.torry.net>
16) efg'sComputerLab. <www.efg2.com>
17) FFTW library documentation <http://www.fftw.org/>