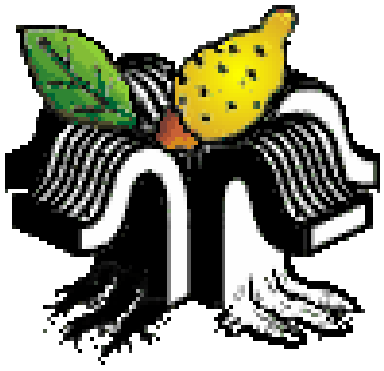


PhotoPaint

4º Grado en Ingeniería Informática
Asignatura: Informática Gráfica
Curso 2019/2020

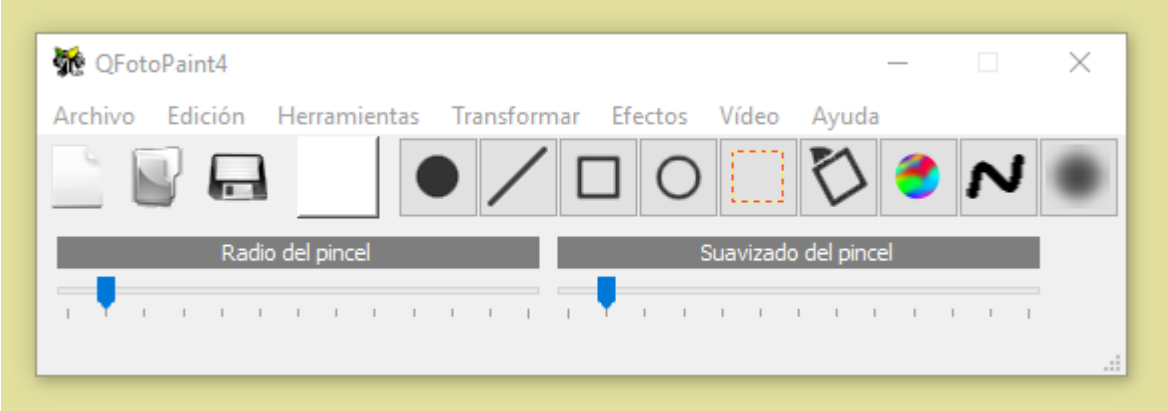
Trabajo realizado por Nicolás Enrique Linares La Barba.

Índice

PhotoPaint

La apariencia del menú principal de la aplicación PhotoPaint se ha modificado levemente, poniendo el botón para seleccionar el color del pincel a la izquierda de todas las herramientas.

Como se va a explicar más adelante, se han añadido las herramientas de rellenado, el icono de la herramienta arcoiris que quedó pendiente, la herramienta de trazo y la de suavizado, como se puede ver en la figura respectivamente.



Además de añadir en la barra superior, las distintas funciones que se van a explicar a continuación.

Menú Archivo

❖ Nueva desde el portapapeles (opcional)

Esta funcionalidad se basa en el uso de la clase QClipboard, que proporciona acceso al portapapeles de Windows. Esta clase almacena las imágenes en la representación QImage por lo que obtendremos este objeto con la imagen que se encuentre en el portapapeles y deberemos convertirla al formato Mat de OpenCV con el que trabajamos en la aplicación.

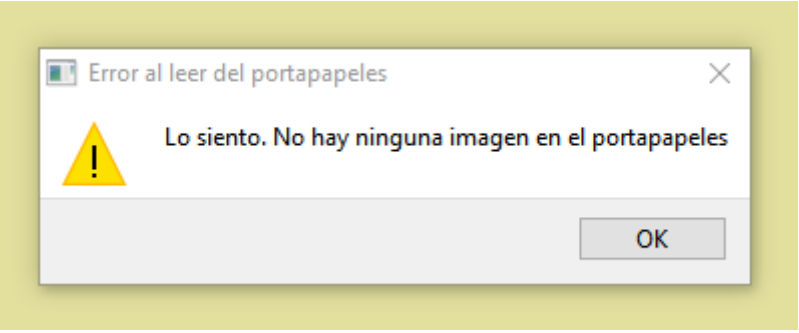
Los pasos seguidos son los siguientes:

1. Obtenemos la imagen con la función QApplication::clipboard()->image().
2. Como en el portapapeles puede haber distintos tipos de datos, nos aseguramos que contenga una imagen para poder abrirla. Comprobamos si la qimage es NULL ya que la operación anterior devuelve ese valor si no hay un objeto QImage.
3. Ahora transformamos la QImage en Mat, pero hay que tener en cuenta el formato de la imagen.

   Realizando pruebas se ha comprobado que las imágenes en el portapapeles de Windows se almacenan en el formato RGB32 (por ejemplo, cuando buscamos una imagen en Google, hacemos click derecho sobre ella y “Copiar imagen”).

   Por otro lado, si nos encontramos en el PhotoPaint y seleccionamos un ROI de la foto actual, al copiarla al portapapeles la imagen tiene un formato RGB888.

   El problema de esto es que al pasar la imagen a Mat necesitamos especificar su formato, siendo distinto para los dos. Para el primer caso se usa CV_8UC4 ya que son 32 bits repartidos en 4 canales (8 bits/canal) y para el segundo caso se usa CV_8UC3 pues son 24 bits repartidos en 3 canales (también 8 bits/canal).
4. Finalmente se emplea la función crear_nueva para mostrar la imagen y añadirla a la lista de imágenes actuales del programa. Es importante pasarle una copia de la imagen para evitar que si es un ROI de otra abierta no se encuentren referenciadas.
5. Por otro lado, si no contiene una imagen se lo informamos al usuario con una ventana de warning:



Se ha implementado en el fichero mainwindow.cpp con el siguiente código:

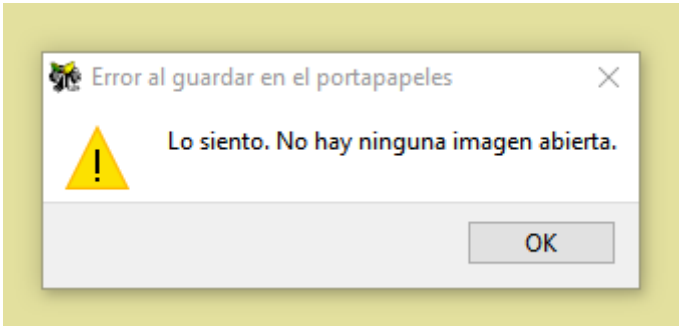
```
void MainWindow::on_actionNueva_desde_el_portapapeles_triggered()
{

    QImage qimg = QApplication::clipboard()->image();

    if (qimg.isNull())
    {
        QMessageBox::warning(NULL, "Error al leer del portapapeles", "Lo siento.
No hay ninguna imagen en el portapapeles");
        return;
    }

    Mat img;

    if (qimg.format() == QImage::Format_RGB32)
    {
        img = cv::Mat(qimg.height(), qimg.width(), CV_8UC4,
                (void*)qimg.constBits(), (uint) qimg.bytesPerLine());
        cvtColor(img, img, COLOR_BGRA2BGR);

    } else if (qimg.format() == QImage::Format_RGB888) {
        img = cv::Mat(qimg.height(), qimg.width(), CV_8UC3,
                (void*)qimg.constBits(), (uint) qimg.bytesPerLine());
        cvtColor(img, img, COLOR_RGB2BGR);

    }

    // En principio no es necesario añadir más formatos

    crear_nueva(primera_libre(), img.clone());

}
```

Menú Edición

❖ Copiar al portapapeles (opcional)

Para copiar al portapapeles el ROI de la imagen actual es necesario realizar los siguientes pasos:

1. Obtenemos el ROI y le cambiamos el espacio de color de BGR a RGB, ya que en OpenCV las imágenes por defecto se almacenan en el orden BGR (no en RGB).
2. Transformamos el objeto Mat a QImage para almacenarla en el portapapeles, con formato RGB888 para indicar que cada canal tiene 8 bits. Es necesario pasarlo a este formato ya que QImage solo admite BGR30, y es más cómodo pasarlo a RGB.
3. Se guarda en el portapapeles con la función QApplication::clipboard()->setImage().
4. En caso de no haber una imagen abierta en el programa, se avisará con un warning:



Se ha implementado en el fichero mainwindow.cpp con el código:

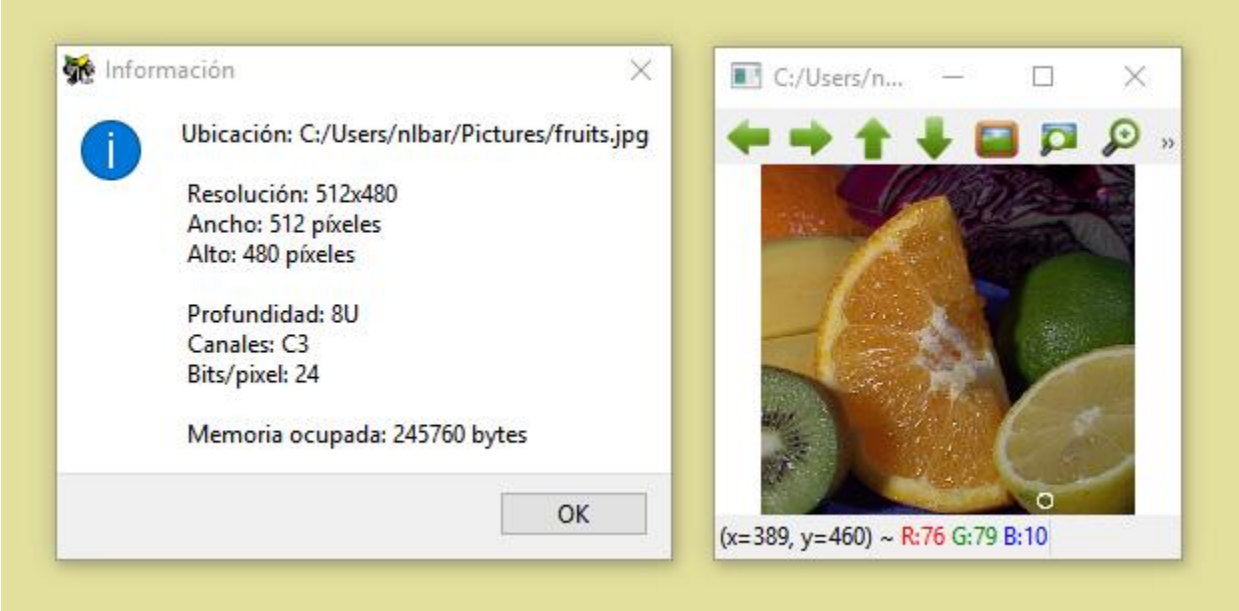
```
void MainWindow::on_actionCopiar_al_portapapeles_triggered()
{

    int fa = foto_activa();
    if (fa != -1)
    {
        Mat img = foto[fa].img;
        Mat roi = img(foto[fa].roi);
        cvtColor(roi, roi, COLOR_BGR2RGB);
        QImage qimg(roi.data, roi.cols, roi.rows, (uint) roi.step, QImage::For-
mat_RGB888);
        QApplication::clipboard()->setImage(qimg);

    } else {
        QMessageBox::warning(this, "Error al guardar en el portapapeles",
                             "Lo siento. No hay ninguna imagen abierta.");
    }
}
```

❖ Ver información (opcional)

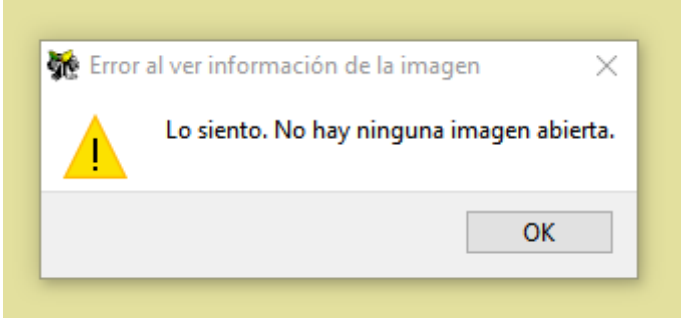
Para conocer la información de la imagen actual se van a emplear las funciones básicas de Mat: como size() para obtener el tamaño de la imagen y channels() para el número de canales. También obtendremos de la ventana la ubicación o nombre de la imagen, ya que puede ser una imagen que se encuentre almacenada en el disco o una copia que se ha creado en PhotoPaint y todavía no se ha guardado.

Con size() podemos saber cuántos píxeles tiene la imagen de ancho y alto, y sabiendo que siempre que se abre una imagen de disco se abre en 8UC3, podemos obtener el tamaño que ocupa en memoria dicha imagen. 8UC3 indica que por cada pixel hay 8 bits por 3 canales, lo que es igual a 24 bits el pixel. Los datos de la profundidad y el número de canales se mostrarán como una cadena: “8U” y “C3”, respectivamente.

Todo esto se va a mostrar en un QMessageBox::information, por ejemplo:



Pero si no hay ninguna imagen abierta se va a indicar en un QMessageBox::warning:



Se ha implementado en el fichero mainwindow.cpp con el código:

```
void MainWindow::on_actionVer_informaci_n_2_triggered()
{
    if (foto_activa() != -1) {

        ventana v = foto[foto_activa()];

        int ancho = v.img.size().width;
        int alto = v.img.size().height;

        String resolucion = to_string(ancho) + "x" + to_string(alto);

        int canales = v.img.channels(); // C3 -> 3 canales por pixel
        int prof = 8; // 8U -> 8 bits por canal
        int bit_pixel = prof * canales; //8UC3 -> 8U*C3 == 24 bits/pixel

        String tam = to_string((ancho * alto * prof)/8) + " bytes";

        QString info = QString::fromStdString(
                                "Ubicación: " + v.nombre +
                                "\n\n Resolución: " + resolucion +
                                "\n Ancho: " + to_string(ancho) + " píxeles" +
                                "\n Alto: " + to_string(alto) + " píxeles" +
                                "\n\n Profundidad: 8U" +
                                "\n Canales: C3" +
                                "\n Bits/pixel: " + to_string(bit_pixel) +
                                "\n\n Memoria ocupada: " + tam);

        QMessageBox::information (this, "Información", info);


    } else {
        QMessageBox::warning(this, "Error al ver información de la imagen",
                             "Lo siento. No hay ninguna imagen abierta.");
    }
}
```

Menú Herramientas

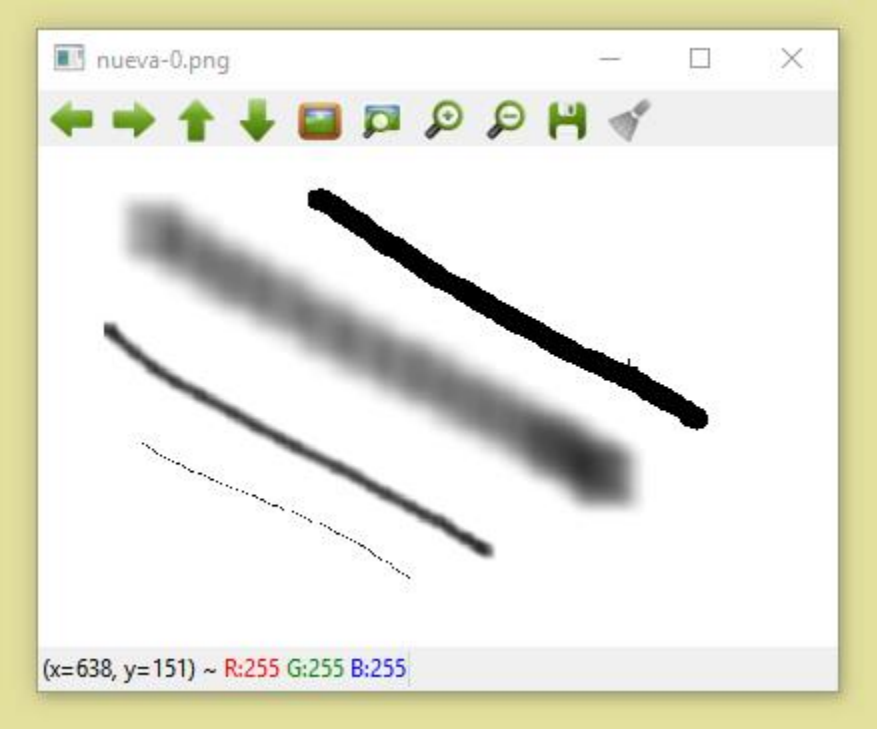
❖ Trazos (opcional) 

Esta herramienta se comportará igual que la de “punto”, pero evitará los huecos entre cada par de puntos rellenándolo con una línea recta.

Para implementar esta herramienta se incluirá el siguiente código en la función callback del fichero imagenes.cpp:

```
// 2.9. Herramienta TRAZO
case HER_TRAZO:

    if (event == EVENT_LBUTTONDOWN)
        punto_inicial = Point(x, y);
    else if (event==EVENT_MOUSEMOVE && flags==EVENT_FLAG_LBUTTON){
            cb_trazo(factual, x, y);
            punto_inicial = Point(x, y);
    } else
        ninguna_accion(factual, x, y);

    break;
}
```

La idea central consiste en establecer un punto inicial (cuando se presiona el ratón) y a partir del siguiente (cuando movemos el ratón manteniendo presionado) es cuando llamamos a la función cb_trazo para que dibuje la línea entre estos dos puntos. Y se va actualizando el valor del punto_inicial por el actual después de cada ejecución. Se puede observar su funcionamiento en la siguiente imagen:



Todas las líneas anteriores se han hecho con la herramienta trazo y se puede ver haciendo zoom en la imagen como, efectivamente, no hay hueco entre los puntos:

La función cb_trazo es idéntica a la función cb_línea pero se ha modificado levemente (marcado en amarillo lo único que se ha modificado). Al dibujar esa línea se realizará desde el punto actual (X,Y) hasta el punto inicial que se guardó previamente. El código es el siguiente:

```
void cb_trazo (int factual, int x, int y)
{
    Mat im= foto[factual].img;
    if (difum_pincel==0)
        line(im, punto_inicial, Point(x,y), color_pincel, radio_pincel*2+1);
    else {
        Mat res(im.size(), im.type(), color_pincel);
        Mat cop(im.size(), im.type(), CV_RGB(0,0,0));
        line(cop, punto_inicial, Point(x,y), CV_RGB(255,255,255), radio_pin-
cel*2+1);
        blur(cop, cop, Size(difum_pincel*2+1, difum_pincel*2+1));
        multiply(res, cop, res, 1.0/255.0);
        bitwise_not(cop, cop);
        multiply(im, cop, im, 1.0/255.0);
        im= res + im;
    }
    imshow(foto[factual].nombre, im);
    foto[factual].modificada= true;
}
```

❖ Rellenar (opcional) 

La herramienta de rellenado se debe añadir también a la función callback, comentado en el apartado anterior, del fichero imagenes.cpp.

```
    // 2.7. Herramienta RELLENAR
    case HER_RELLENAR:
        if (flags==EVENT_FLAG_LBUTTON)
            cb_rellenar(factual, x, y);
        else
            ninguna_accion(factual, x, y);
        break;
```

Se puede observar en el código anterior que cuando se presiona el botón izquierdo del ratón se realiza el rellenado, llamando a la función cb_rellenar, cuyo código es el siguiente:

```
void cb_rellenar (int factual, int x, int y)
{
    Mat img = foto[factual].img;
    Mat rellenado = img.clone();
    int flags = 4 | FLOODFILL_FIXED_RANGE | (255 << 8);

    // Establecemos los parámetros necesarios para ajustar la tolerancia de re-
llenado

    int lo = radio_pincel;
    int up = radio_pincel;

    floodFill(rellenado, Point(x, y), color_pincel, nullptr, Scalar(lo, lo, lo),
                                                Scalar(up, up, up), flags);

    // como el difuminado se estableció inicialmente hasta 120, vamos a satu-
rarlo en 100

    double transparencia;
    if (difum_pincel > 100)
        transparencia = 1.0;
    else
        transparencia = difum_pincel/100.0;

    addWeighted(img, transparencia, rellenado, 1.0-transparencia, 0, img);

    imshow(foto[factual].nombre, img);
    foto[factual].modificada= true;
}
```

La idea consiste en rellenar la región deseada en la copia de la imagen con la tolerancia indicada y aplicar una suma ponderada entre la imagen rellenada y la original para establecer la transparencia.

Los pasos para implementar el rellenado son los siguientes:

1. Primero se establecen los flags: el 4 indica el nivel de conectividad al analizar a los pixeles vecinos (puede valer 8 también) y el flag FLOODFILL_FIXED_RANGE considerará la distancia entre los vecinos.

2. Se aplica el rellenado ajustando la tolerancia, por arriba y por abajo, mediante la propia función floodFill con sus parámetros loDiff y upDiff a través del radio_pincel.

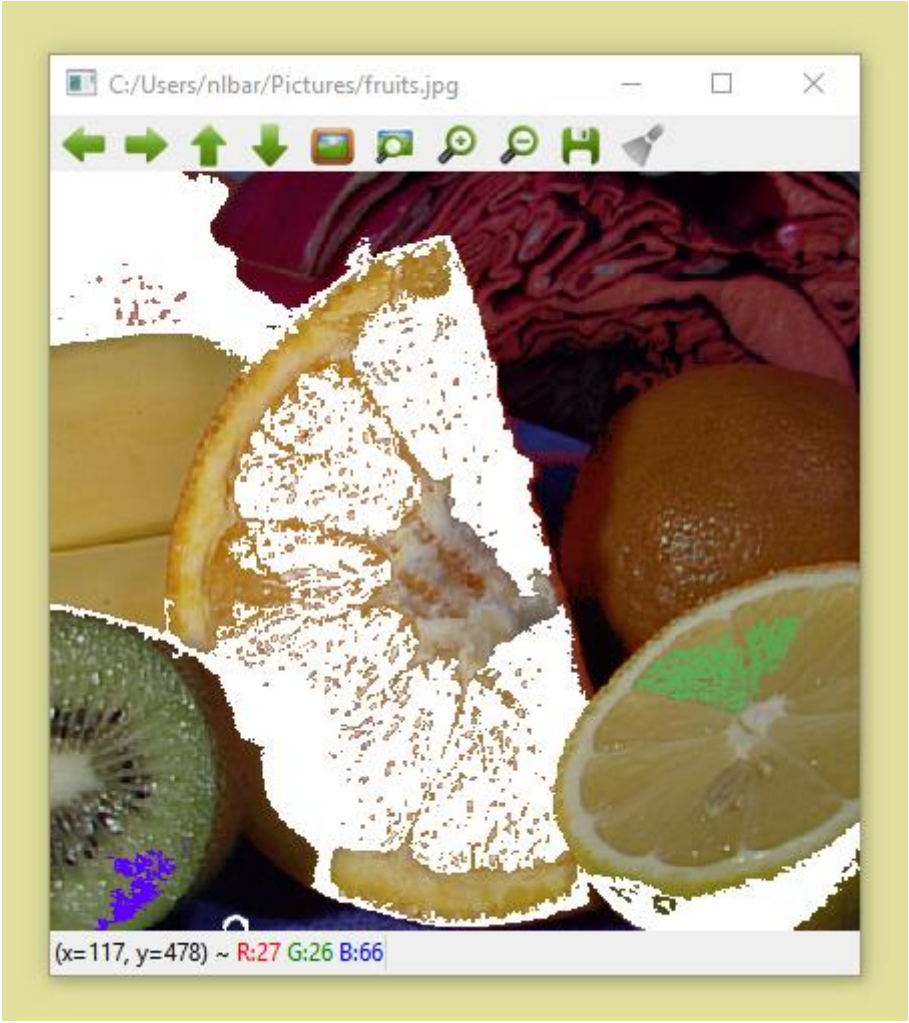
3. Una vez realizado el rellenado en una copia de la imagen, se realiza una suma ponderada para simular transparencia en la parte rellenada. La variable transparencia establece el grado de opacidad del rellenado (cuanto más alto el valor de transparencia menos visible es la región rellenada).

A continuación, se muestra un ejemplo con distintos colores y niveles de tolerancia y transparencia:

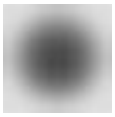
- El color blanco, hecho con un nivel alto de tolerancia y nulo de transparencia (opaco).

- El color rojo presente a partir de la esquina superior derecha se ha hecho con un nivel alto de tolerancia y transparencia.

- Los colores verde y azul se han realizado con un nivel bajo de tolerancia.



❖ Suavizado (opcional)



La herramienta de suavizado se ha implementado mediante el difuminado Gaussiano y es muy parecida a la herramienta Punto. Se comienza por añadir la herramienta a la función callback del fichero imagenes.cpp, que al igual que en Punto solo se activa la función cb_suavizar cuando se mantiene presionado el botón izquierdo del ratón:

```
    // 2.8. Herramienta SUAVIZAR
    case HER_SUAVIZAR:
        if (flags==EVENT_FLAG_LBUTTON)
            cb_suavizar(factual, x, y);
        else
            ninguna_accion(factual, x, y);
        break;
```

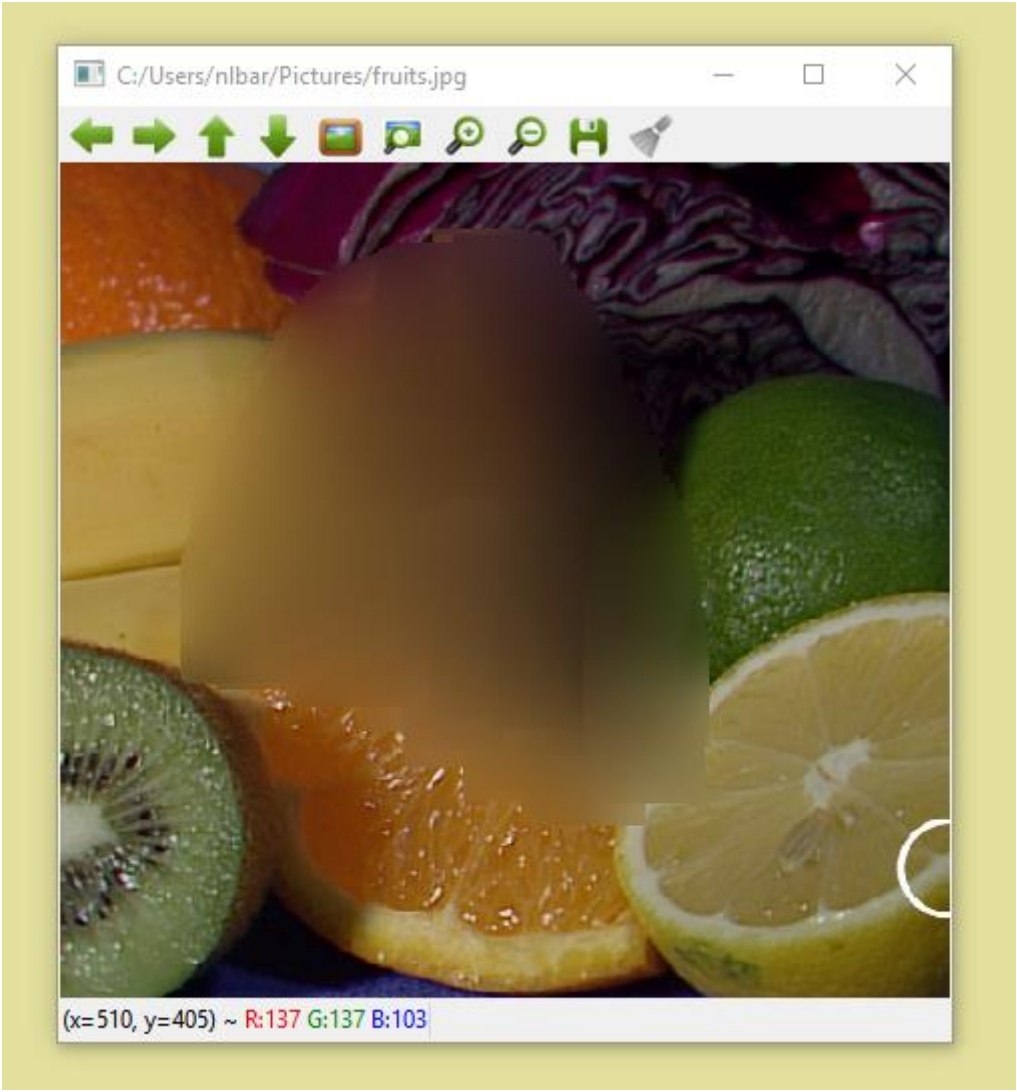
En cuanto a la implementación de cb_suavizar, se ha cogido la misma que la de Punto pero modificando que, en lugar de pintar el ROI de la región del punto, se aplique el difuminado Gaussiano. El código de su implementación es el siguiente:

```
void cb_suavizar (int factual, int x, int y)
{
    Mat im= foto[factual].img;

    int tam = radio_pincel;
    Rect rroi(x-tam, y-tam, tam*2+1, tam*2+1);

    // Si se sale por la izquierda
    if (rroi.x < 0)
    {
        rroi.width += rroi.x;
        rroi.x =0;
    }
    // Si se sale por arriba
    if (rroi.y < 0)
    {
        rroi.height += rroi.y;
        rroi.y =0;
    }
    // Si se sale por la derecha
    if (rroi.x + rroi.width > im.cols)
    {
        rroi.width = im.cols -  rroi.x;
    }
    // Si se sale por abajo
    if (rroi.y + rroi.height > im.rows)
    {
        rroi.height = im.rows -  rroi.y;
    }

    Mat roi = im(rroi);
    GaussianBlur(roi, roi, Size(difum_pincel*2+1, difum_pincel*2+1), 0);

    imshow(foto[factual].nombre, im);
    foto[factual].modificada= true;
}
```

Los pasos se resumen en los siguientes puntos:

1. En primer lugar, se obtiene el ROI de la imagen donde se va a aplicar el efecto.
2. Se comprueba que no se salga de la ventana, en su caso satura las dimensiones del ROI para ajustarlo y que no de error.
3. Aplica el efecto de difuminado con la función GaussianBlur (se podría cambiar por otro suavizado, como Media o Mediana).

Se muestra en la siguiente figura un ejemplo de difuminado con esta herramienta:



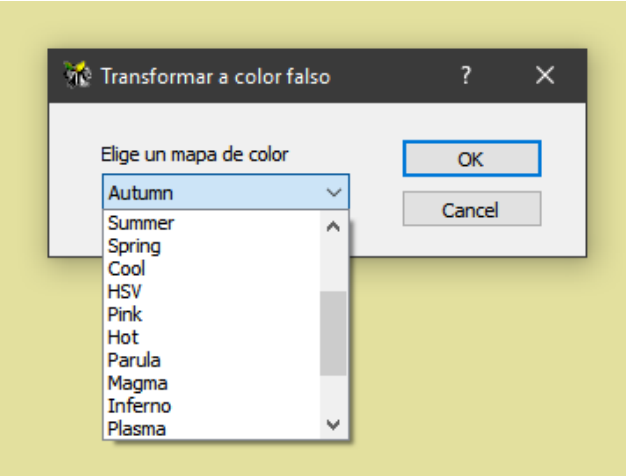
Menú Transformar

❖ Convertir a color falso (opcional)

Esta función va a transformar una imagen en escala de grises a una paleta de colores con el objetivo de hacer más visibles las pequeñas variaciones del nivel de gris.

Se ha implementado de forma que coja la imagen actual y, sin importar su escala de color, la convierta a escala de grises y aplique la transformación. También se podría restringir solo a imágenes abiertas que ya estén en escala de grises, pero para facilitar su uso y que sea más cómodo se ha elegido hacerlo de esta manera.

Su implementación, a diferencia de los casos anteriores, comienza por definir una clase nueva llamada “colorfalso.*” y se creará una pequeña interfaz que permita elegir la paleta de colores falsos que se desea aplicar.



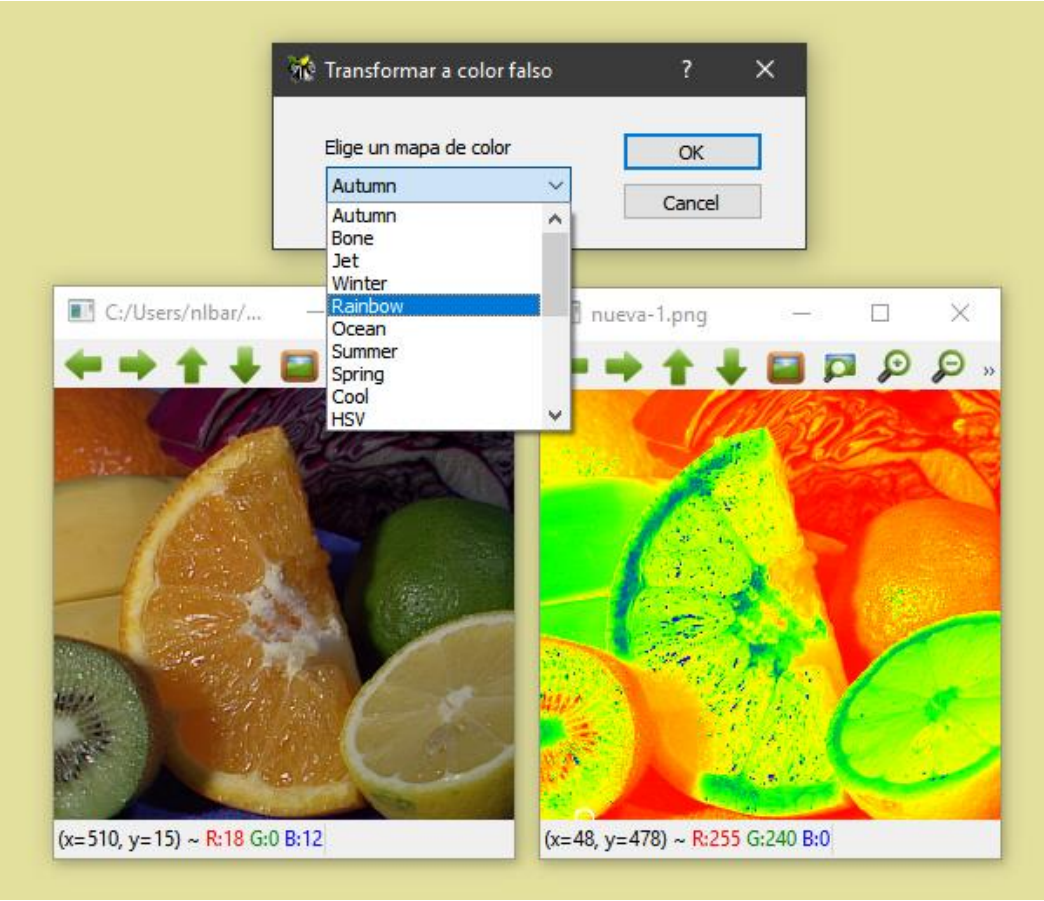
Después de elegir la paleta de colores, cuando se presiona el botón OK, se traduce el nombre de la paleta por su correspondiente identificador (se ha importado el fichero imgproc.hpp para tener acceso a ellos) y se llama a la función ver_colorfalso implementado en el fichero imagenes.cpp:

```
void ver_colorfalso(int nfoto, int nres, int map)
{
    assert(nfoto>=0 && nfoto<MAX_VENTANAS && foto[nfoto].usada);

    // Primero convertimos la imagen a escala de grises
    Mat img_gray, img_res;
    cvtColor(foto[nfoto].img, img_gray, COLOR_BGR2GRAY);
    // Aplicamos el mapa de color
    applyColorMap(img_gray, img_res, map);
    // Mostramos la transformación
    crear_nueva(nres, img_res);

}
```

Para implementar esta función se han seguido los siguientes pasos:

1. Primero convertimos la imagen a escala de grises, como se ha comentado antes, ya que así permitimos que se pueda aplicar la transformación con cualquier imagen abierta. Se podría modificar quitando esta conversión para aceptar solo imágenes que ya se encuentren en la escala de grises y mostrando un warning si no lo estuviese.
2. Se aplica el mapa de color con la función applyColorMap indicando la imagen origen, destino y el mapa de color a aplicar (que ha sido elegido desde la ventana emergente).
3. Se muestra la transformación realizada creando una nueva ventana.

Por ejemplo, se aplica el mapa de color Rainbow a la imagen abierta, aunque se han añadido todos los mapas de color existentes en imgproc.hpp:



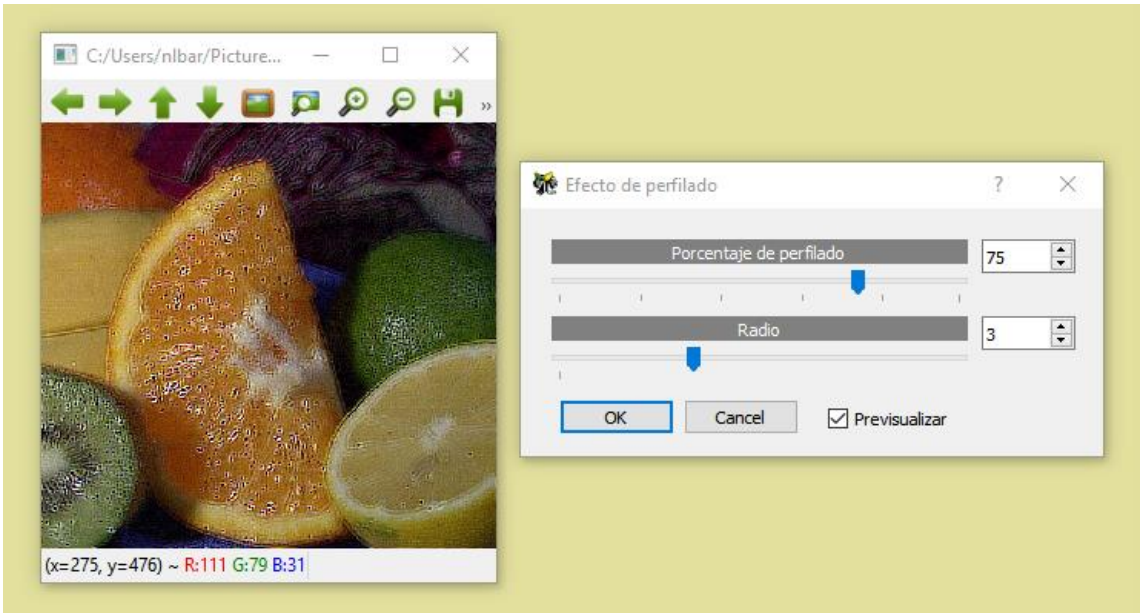
Menú Efectos

❖ Perfilado (opcional)

Mediante el efecto de perfilado se van a resaltar los contrastes de la imagen, es la operación opuesta al suavizado. Al aplicar este efecto se harán más visibles las variaciones y bordes de la imagen, lo que permitirá mejorar las apariencias borrosas en una imagen que se haya hecho mediante una lente con imperfecciones. Pero si realizamos un perfilado muy pronunciado estropeará la imagen ya que aumenta el ruido de la misma.

Se ha comenzado creando una nueva clase llamada “perfilado.*” y se ha diseñado una ventana para ajustar el porcentaje y el radio del efecto. Los componentes que permiten modificar el radio de la Laplaciana (necesaria para obtener el efecto de perfilado) solo tomarán los valores 1, 3, 5 y 7.

Se muestra el siguiente ejemplo con un radio igual a 3 y un porcentaje del 75%:



Cada vez que modifiquemos estos valores se reflejará en la imagen y podremos previsualizar el resultado. La función que realiza el perfilado y permite visualizarlo se encuentra implementada en el fichero imagenes.cpp:

```
void ver_perfilado (int nfoto, int radio, double porcentaje, bool guardar)
{
    assert(nfoto>=0 && nfoto<MAX_VENTANAS && foto[nfoto].usada);

    Mat img_original, img_original_16S, laplace, resultado;

    img_original = foto[nfoto].img.clone();
```

```
    img_original = img_original(foto[nfoto].roi);

    // Convertimos a 16S
    img_original.convertTo(img_original_16S, CV_16S);
    // Aplicamos la laplaciana a la imagen original
    Laplacian(img_original, laplace, CV_16S, radio);
    // Multiplicamos la laplaciana por el porcentaje
    porcentaje = porcentaje/100.0;
    laplace *= porcentaje;
    // Se suma la laplaciana a la imagen original de 16S
    resultado = img_original_16S - laplace;
    // Convertimos a 8U
    resultado.convertTo(resultado, CV_8U);

    imshow(foto[nfoto].nombre, resultado);

    if (guardar) {
        resultado.copyTo(foto[nfoto].img);
        foto[nfoto].modificada= true;
    }
}
```

Para implementarlo se han seguido estos pasos:

1. La imagen original se convierte a 16 bits con signo (16S).
2. Se aplica la función Laplacian a la imagen original con el tamaño ksize especificado (radio), lo que permitirá aumentar el ruido y se almacena en 16 bits con signo (16S).
3. Multiplicamos la laplaciana por un peso (el porcentaje de perfilado).
4. Finalmente se suma el resultado anterior con la imagen original de 16S y se convertirá a 8U.
5. Además, se puede indicar que se guarde la imagen para conservar los cambios.

❖ Texto (opcional)

Esta función se ha llevado a un nivel más, es decir, se ha ampliado para hacer más versátil su uso. La funcionalidad, además de escribir el texto con un tamaño, posición, color y sombreado ajustados por el usuario, permite quitar la sombra o modificar sus parámetros para cambiar el color y la transparencia. Además, se implementado de forma que se actualice en tiempo real cada cambio realizado en cualquiera de los ajustes.

A continuación, se muestra la interfaz para añadir texto y varios ejemplos de texto:

Primero se ha creado una nueva clase llamada “texto.*” y se ha diseñado la interfaz que podemos ver en la figura anterior. En la clase texto.cpp se ha añadido una función actualizar_imagen que se va a encargar de llamar a la función que dibuja el texto actual con las características que se han indicado.

Por ello, cada vez que el usuario interacciona con esta ventana, se refleja inmediatamente el cambio en la imagen, pues se llama en cada componente a esta función de actualizar, que contiene el siguiente código:

```
void texto::actualizar_imagen(bool guardar)
{
    cadena = ui->plainTextEdit->toPlainText();
    posicion = Point(ui->spinBox_2->value(), ui->spinBox_3->value());
    tam = ui->spinBox->value();
    transparencia = ui->spinBox_4->value();
    ver_texto(nfoto, cadena, posicion, tam,  color_texto, aplicar_sombreado, co-
lor_sombra, (transparencia-1)/10.0, guardar);
}
```

Solo cuando se pulse el botón Ok el booleano “guardar” se pasará como true.

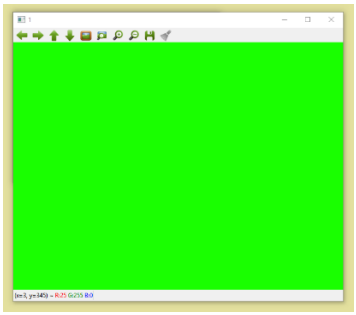
Además, es necesario establecer un callback al comenzar para que no se modifique la imagen si pulsamos o pasamos el ratón sobre ella. Que se anula cuando se termina de interactuar con el efecto.

Se muestra a continuación la función ver_texto que se encarga de dibujar la cadena y su sombreado:

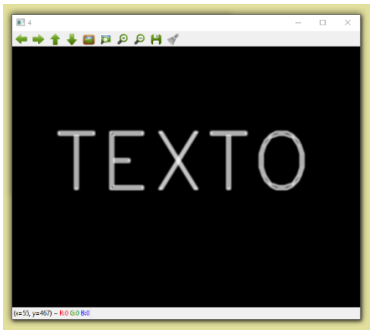
```
void ver_texto(int nfoto, QString cadena, Point posicion, int tam, Scalar co-
lor_texto, bool sombrear, Scalar color_sombra, double transparencia, bool guar-
dar)
{
    Mat im = foto[nfoto].img.clone();

    if (sombrear)
    {
        Mat res(im.size(), im.type(), color_sombra);
        Mat cop(im.size(), im.type(), CV_RGB(0,0,0));
        putText(cop, cadena.toLatin1().data(), Point(posicion.x, posicion.y +
2), FONT_HERSHEY_DUPLEX, tam , CV_RGB(255,255,255), 2, LINE_AA);

        blur(cop, cop, Size(5, 5));
        addWeighted(cop, transparencia, res, 1.0 - transparencia, 0, res);

        multiply(res, cop, res, 1.0/255.0);
        bitwise_not(cop, cop);
        multiply(im, cop, im, 1.0/255.0);
        im= res + im;
    }

    putText(im, cadena.toLatin1().data(), posicion, FONT_HERSHEY_DUPLEX, tam ,
color_texto, 2, LINE_AA);

    imshow(foto[nfoto].nombre, im);
    if (guardar) {
        im.copyTo(foto[nfoto].img);
        foto[nfoto].modificada= true;
    }
}
```

Si se ha decidido sombrear el texto, se va a seguir la misma estrategia que con las herramientas de Punto, Línea, Rectángulo, etc, pero dibujando el texto con la función putText. Se van a utilizar imágenes para acompañar la explicación:

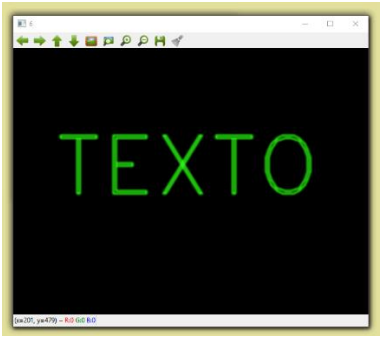
1. Se crea un Mat (res) y se pinta con el color de la sombra.



2. Se crea otro Mat (cop) en color negro, se escribe el texto en color blanco sobre él y se aplica un suavizado para dar el efecto de la sombra.



3. Se suman estas dos imágenes (res + cop) aplicando un peso para la transparencia de la imagen y se multiplica el resultado (res * cop) para aumentar la intensidad y se obtiene una imagen negra con las letras del color seleccionado.



4. Se invierte la imagen negra del paso 4 (cop) para multiplicarla con la imagen original (im) con lo que se obtiene la imagen original con las letras sombreadas y de color negro.

5. Ahora solo queda sumar la imagen anterior (im) con la imagen resultante del paso 6 para obtener la imagen final con la sombra de color verde, en este caso.

Todo esto para obtener el sombreado, solo queda añadir el texto real y guardar la imagen si así se indica con la variable “guardar”, dando como resultado la siguiente imagen.

Menú Vídeo

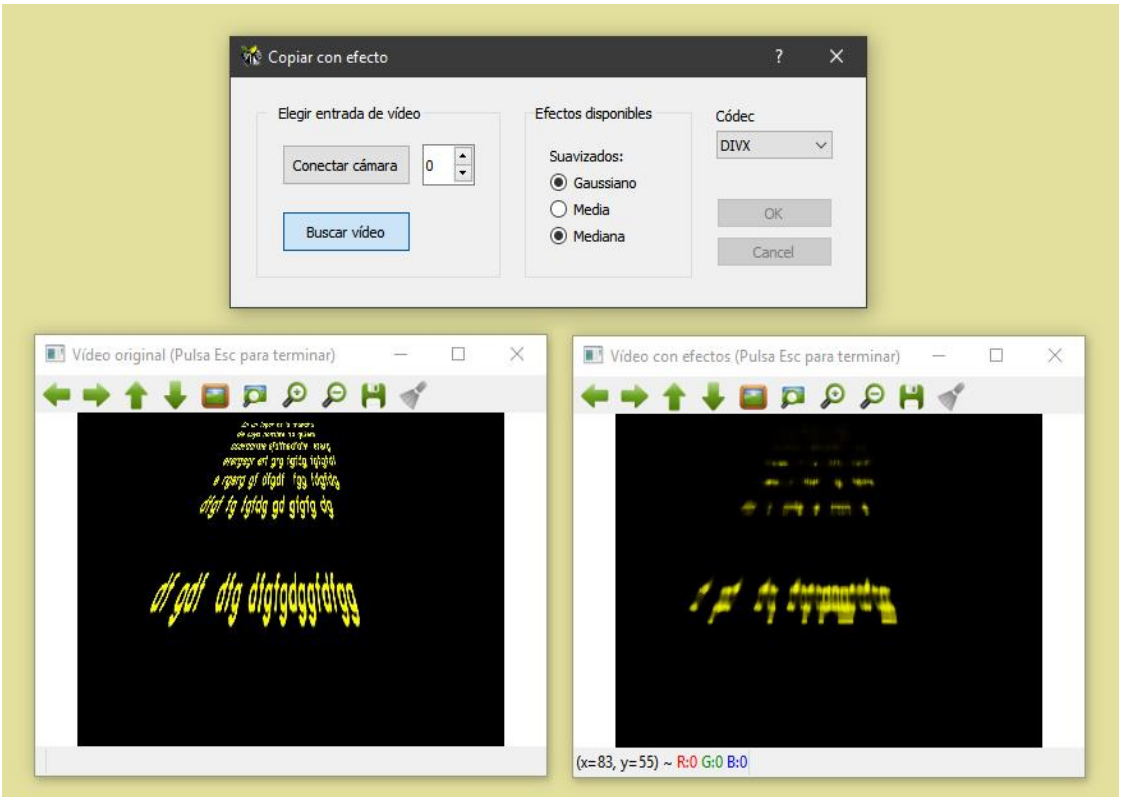
❖ Copiar con efectos (obligatoria) y otros efectos (Opcional)

Para esta funcionalidad se ha creado una clase “copiarconefecto.*” y se ha diseñado la ventana que se muestra en la figura para controlar la entrada de vídeo, los efectos que se desean aplicar a todo el vídeo y el códec con el que se guardará.

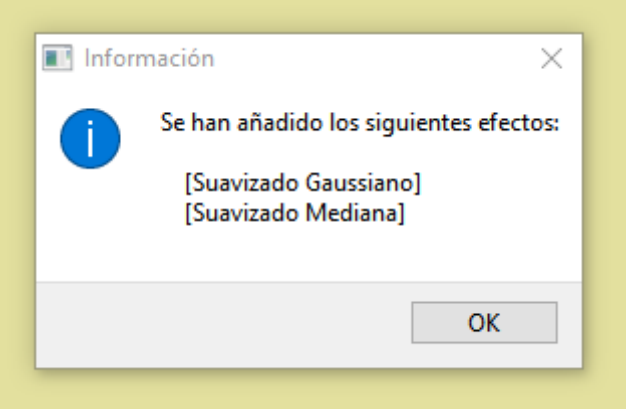
Se han añadido tres efectos sencillos de suavizado para no complicar demasiado la explicación y la implementación del código relevante, pero se podría ampliar con todos los efectos y transformaciones que se encuentran ya implementados.

El funcionamiento para un vídeo desde la cámara es el siguiente: elegimos que queremos conectar la cámara (pero no se activará en ese momento), luego marcamos los efectos que deseamos y el códec. Entonces cuando confirmemos con el botón OK nos pedirá la ruta de guardado para el video resultante y después comenzará la grabación y a su vez la transformación, mostrándose dos vídeos a la vez el original y el que se está modificando.

Para el caso de elegir un vídeo de entrada, al pulsar OK nos pedirá que seleccionemos el vídeo y luego procederá como el caso anterior.



Para terminar la grabación o el vídeo solo debemos pulsar la tecla ESC como indica el nombre de las ventanas. Y se mostrará mediante un mensaje emergente los efectos que se han aplicado al vídeo.



Se ha incluido una función en la clase video.cpp llamada ver_copiar_con_efectos cuyo código es el siguiente:

```
void ver_copiar_con_efectos(int modo, String nombre_in, int camara, String nom-
bre_out, list<int> efectos, String aplicar,int codec)
{
    VideoCapture vc;
    QString nombre_ventana;

    if (modo == 0)
    {
        vc.open(camara);
    }
    else
    {
        if (!vc.open(nombre_in))
        {
            QMessageBox::warning(NULL, "Error al abrir el vídeo",
QString::fromStdString("No se ha podido abrir el archivo " + nombre_in) );
            return;
        }
    }

    Mat frame, res;
    if (!vc.read(frame))
    {
        QMessageBox::warning(NULL, "Error en el vídeo", QString::fromStdS-
tring("No se ha podido leer el archivo ") );
        return;
    }

    VideoWriter writer(nombre_out, codec, 30, frame.size());
    if (writer.isOpened())
    {
        int tecla = 27;
```

```
        namedWindow("Vídeo original (Pulsa Esc para terminar)", 0);
        moveWindow("Vídeo original (Pulsa Esc para terminar)", 200, 200);
        namedWindow("Vídeo con efectos (Pulsa Esc para terminar)", 0);
        moveWindow("Vídeo con efectos (Pulsa Esc para terminar)", 700, 200);

        while((tecla = waitKey(1)) != 27)
        {
            list<int>::iterator it;
            for (it = efectos.begin(); it != efectos.end(); it++) {
                // Se transforma el frame aplicando los efectos deseados
                switch (*it) {
                case SUAVIZADO_GAUSSIANO:
                    GaussianBlur(frame, res, Size(15, 15), 0);
                    break;
                case SUAVIZADO_MEDIA:
                    blur(frame, res, Size(15, 15));
                    break;
                case SUAVIZADO_MEDIANA:
                    medianBlur(frame, res, 15);
                    break;
                }
            }
            imshow("Vídeo original (Pulsa Esc para terminar)", frame);
            imshow("Vídeo con efectos (Pulsa Esc para terminar)", res);

            writer << res;
            if (!vc.read(frame))
                break;
        }

        vc.release();
        writer.release();
        destroyWindow("Vídeo original (Pulsa Esc para terminar)");
        destroyWindow("Vídeo con efectos (Pulsa Esc para terminar)");

        QMessageBox::information(NULL, "Información", "Se han añadido los si-
guientes efectos: \n" + QString::fromStdString(aplicar));

    }
}
```

Se ha implementado siguiendo estos pasos:

1. Se inicializa un VideoCapture para leer el vídeo, ya sea un fichero “nombre_in” o de una cámara “camara” pasados como parámetro. Se elige la entrada de vídeo mediante el parámetro “modo”. Y se muestran mensajes de error en caso de que no se pueda abrir la cámara o no se pueda leer el vídeo almacenado.
2. Se inicializa ahora el VideoWriter que se encargará de almacenar el vídeo en la dirección indicada y con el códec elegido.
3. Con el VideoWriter en funcionamiento se va a leer el vídeo original frame a frame aplicándoles la lista de efectos marcados. Y se almacena el frame modificado.
4. Se sigue ese procedimiento hasta que se pulse la tecla ESC o hasta que se acabe de reproducir el vídeo seleccionado.
5. Se cierra el VideoCapture y el VideoWriter, se destruyen las ventanas y se muestra un mensaje emergente con los efectos aplicados.

Referencias

- Documentación online de OpenCV: https://docs.opencv.org/
- Apuntes proporcionados por la asignatura.
- Ejemplos de la carpeta “samples” de OpenCV.