Procesamiento de Imágenes
Máster NTI

Guión de prácticas

Descripción
Estructura y módulos
Gestión de ventanas
El callback
Pintar un punto
Otras funciones
Cuadros de diálogo
Notas finales

Sesión 4. QFotoPaint: una aplicación de procesamiento

DESCRIPCIÓN

- La creación de una herramienta de procesamiento de imágenes debe entenderse como un **proyecto software** que, en consecuencia, hará uso de los **principios ingenieriles de desarrollo de software**: fases de análisis y diseño de la aplicación, uso de abstracciones, modularidad, ocultación de la implementación, separación interface/implementación. Estos aspectos no son el objetivo de la práctica, pero sí el medio para desarrollar buenos programas.

- En esta sesión vamos a describir la creación de una aplicación de retoque fotográfico, empezando por un **nivel de abstracción alto**: descomposición modular y tipos de datos abstractos necesarios. Básicamente, tendremos **módulos de interface** (asociados a los formularios que aparezcan) y **módulos de procesamiento** (sin formularios asociados, serán los que usen OpenCV).

- A continuación, vamos a bajar el nivel de abstracción, tratando aspectos concretos a **nivel de implementación**. Distinguimos los aspectos que están relacionados con la gestión de ventanas múltiples, los de las operaciones de procesamiento y los de la interface de usuario.

- En general, las operaciones de procesamiento de imágenes y vídeo requieren aplicar diferentes tipos de funciones. De esta forma, las técnicas elementales de procesamiento –las estudiadas en la asignatura– no aparecen aisladas sino que se combinan entre sí para conseguir los efectos buscados. Por ejemplo, una simple operación de pintar un punto en una imagen puede requerir suavizado y media ponderada, además de la propia operación de generación del punto.

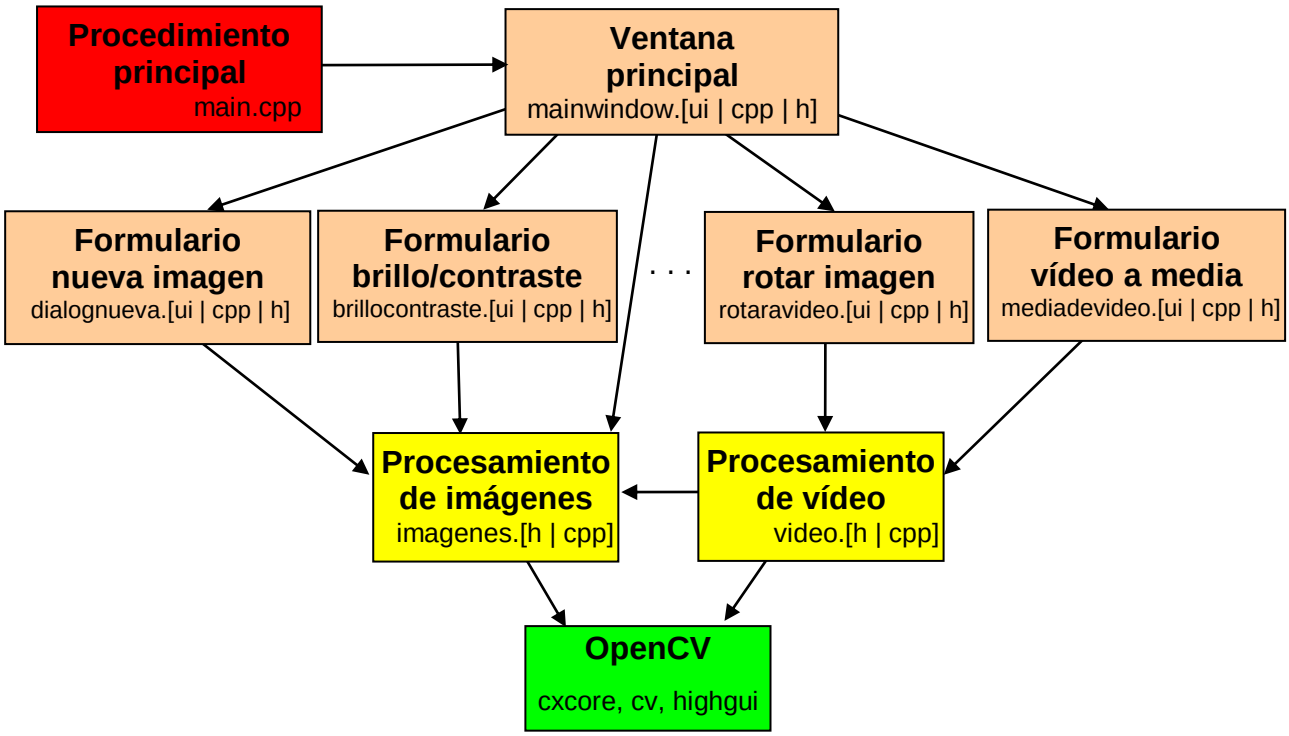
ESTRUCTURA Y MÓDULOS DE LA APLICACIÓN

1. Como hemos visto, diferenciamos la interface de usuario y la parte de procesamiento de imágenes y vídeo.
   - La interface constará de una **ventana principal** (de tipo QMainWindow) desde la que se podrán ejecutar todos los comandos y seleccionar las herramientas, y de varios **cuadros de diálogo** (de tipo QDialog) relacionados con cada una de las operaciones que necesiten información adicional.

- La parte de procesamiento constará de dos módulos, uno para el procesamiento de **imágenes** y otro para las operaciones de **vídeo**. Estos dos últimos módulos serán los que manejen las librerías OpenCV. Los módulos de interface utilizan las operaciones aquí implementadas.

2. A medida que el proyecto vaya creciendo, será necesario añadir otros módulos de interface y también podría ser conveniente realizar otra descomposición modular de la parte de procesamiento.

3. Gráficamente, la **estructura de módulos de la aplicación** es la siguiente.



4. Recordar que cada **módulo** está compuesto de:
   - Módulos de interface de usuario:
     - **Fichero UI**: definición del aspecto visual de la interface de usuario; lo editamos con el diseñador de formularios.
     - **Fichero H**: fichero de cabecera donde viene la definición de la clase ventana (subclase de QMainWindow o QDialog).
     - **Fichero CPP**: fichero de implementación de los métodos asociados a los eventos de las ventanas, y otros que se quieran añadir.
   - Módulos de procesamiento de imágenes y vídeo:
     - **Fichero H**: fichero de cabecera con la parte pública del módulo, las operaciones, tipos y variables que ofrece a sus usuarios.
     - **Fichero CPP**: fichero de implementación de las operaciones definidas en el fichero de cabecera.

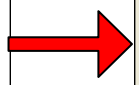
5. Para mostrar imágenes IplImage se usarán **ventanas de HighGUI**. Por lo tanto, estas ventanas serán controladas desde los módulos de procesamiento (imagenes.cpp y video.cpp), también los *callbacks* necesarios. Entre ellas, las imágenes que se están editando actualmente aparecerán en ventanas de HighGUI.

6. El **estado interno** de la aplicación (imágenes abiertas, herramienta, color, tamaño y suavizado actual, capturador de cámara, etc.), se almacenará también en los módulos de procesamiento, que ofrecerán mecanismos para cambiar el estado y acceder a él.

7. En concreto, el módulo imagenes.cpp define el TAD **Ventana de Imagen**, que almacena toda la información referida a una imagen que está siendo editada actualmente (nombre del fichero, IplImage asociado, ROI seleccionado, modificada o no, etc.). Tendremos un array de elementos de este tipo, según el número máximo de imágenes que se pueden abrir en el programa.

8. Se ha realizado una modificación dentro del fichero **main.cpp** para permitir que los objetos de tipo QApplication y MainWindow se puedan acceder desde fuera. Originalmente, son creados como variables locales estáticas del procedimiento main. Ahora son creados como variables dinámicas.

```
1   #include <QtGui/QApplication>
2   #include "mainwindow.h"
3
4   int main(int argc, char *argv[])
5   {
6       QApplication a(argc, argv);
7       MainWindow w;
8       w.show();
9       return a.exec();
10  }
```



```
1   #include <QtGui/QApplication>
2   #include "mainwindow.h"
3
4   MainWindow *w= NULL;
5   QApplication *a= NULL;
6
7   int main(int argc, char *argv[])
8   {
9       a= new QApplication(argc, argv);
10      w= new MainWindow();
11      w->show();
12      int res= a->exec();
13      delete w;
14      delete a;
15      return res;
16  }
```

En los sitios donde se quieran usar, se deben poner las declaraciones *extern* necesarias.

```
60  extern MainWindow *w;
61  extern QApplication *a;
```

GESTIÓN DE VENTANAS MÚLTIPLES

9. Una cuestión importante en la aplicación es que se pueden abrir muchas imágenes simultáneamente para editarlas. Surgen cuestiones relacionadas con su gestión: conocer las ventanas abiertas en cada momento, saber cuál es la ventana activa actualmente. Todas las ventanas compartirán el mismo *callback*.

10. El tipo **struct ventana** almacena la información necesaria para cada imagen abierta en una ventana de HighGUI. La definición del tipo (en imagenes.h) es la siguiente:

```
struct ventana {
    bool usada;         // Indica si se está usando actualmente esta posición
    bool modificada;    // Imagen modificada después de abrirla o crearla
    char *nombre;       // Nombre del fichero que contiene la imagen
    HWND handle;        // Manejador de Windows de la ventana de HighGUI
    CvRect roi;         // Región de interés seleccionada (-1 si no hay ninguna)
    IplImage *img;      // La imagen en sí, siempre de 3 canales
};
```

11. El array **ventana foto[MAX_VENTANAS];** (dentro de imagenes.cpp) almacena la información de todas las ventanas. Por lo tanto, las ventanas se referencian con un número, desde 0 hasta MAX_VENTANAS-1. Por ejemplo, en las funciones: void **invertir** (int nfoto), void **rotar_90** (int nfoto), void **rotar_m90** (int nfoto); el número **nfoto** se refiere a una de las entradas del array (en estos casos, al sitio donde se almacenará el resultado).

12. Operaciones sobre las ventanas del array **foto**:
    - void **inic_fotos** (void): inicializa todo el array **foto** con los valores iniciales.
    - void **fin_fotos** (void): libera todas las imágenes existentes en el array **foto**.
    - int **primera_libre** (void): busca la primera posición libre donde colocar una nueva imagen. Si no hay ninguna, devuelve -1.
    - void **crear_nueva** (int nfoto, ...): crea una nueva imagen en la posición nfoto, que debe ser una posición vacía. Existen diferentes formas de crearlas: pasándole un tamaño y color de fondo; pasándole un nombre de archivo; pasándole un IplImage (que queda asociado a la ventana, por lo que no hay que liberarlo fuera).
    - void **guardar_foto** (int nfoto, const char *nombre): guarda en disco una imagen del array **foto**.
    - void **cerrar_foto** (int nfoto, bool destruir_ventana): cierra una imagen, liberando la entrada del array **foto**.
    - void **mostrar** (int nfoto): actualiza el contenido de la ventada, es decir, muestra la imagen en la ventana asociada. Es necesario aplicar esta operación siempre que se modifique la imagen.
    - int **nombre_a_numero** (const char *nombre): dada una cadena, comprueba si corresponde a una imagen abierta. Si no se encuentra, devuelve -1.
    - int **foto_activa** (void): de todas las ventanas abiertas actualmente en el programa, indica la que está activa (la que está encima de todas). Si no hay ninguna, devuelve -1.

Nota sobre la última operación. ¿Cómo conocer la ventana activa actualmente? HighGUI no resuelve esta cuestión. Para hacerlo, debemos usar funciones de las API de Windows (buscar en Google Windows SDK). En concreto, usamos las funciones **GetActiveWindow** y **GetNextWindow**. Estas funciones usan el tipo **HWND**, que indica un manejador de una ventana. Por esta razón se ha añadido el atributo **HWND handle**, que se inicializa en la creación de las ventanas del array **foto**:

```
foto[nfoto].handle= GetActiveWindow();
```

EL CALLBACK DE LAS VENTANAS

13. El ***callback*** asociado a las ventanas es una función **muy importante** en nuestra aplicación, ya que deberá hacer diferentes cosas según la herramienta actual (dibujar puntos, líneas, seleccionar, etc.), según se estén pulsando los botones o no, y todo ello sobre la ventana activa en cada momento.

14. Todas las ventanas compartirán el mismo procedimiento *callback*. Por ello, el último parámetro de la llamada nos permitirá identificar la posición de la ventana (dentro del array **foto**) sobre la que se ha producido el evento. Aunque sea un (void *), realmente almacenará un entero; por eso empezamos el *callback* con:

```
int factual= (int) _nfoto;  // Casting de (void *) a (int)
```

15. El valor con el que será llamado se indica al crear la ventana (dentro de las funciones **crear_nueva**):

```
                                        // Casting de (int) a (void *)
cvSetMouseCallback(foto[nfoto].nombre, callback, (void*) nfoto);
```

16. ¿Qué hace el *callback*? (1) Tratar eventos y casos especiales. (2) Según la herramienta actual (puntos, líneas, etc.): según el evento producido (pinchar, arrastrar o moverse por encima), actuar en consecuencia.

```
void callback (int event, int x, int y, int flags, void *_nfoto)
{
  int factual= (int) _nfoto;

  // 1. Eventos y casos especiales
  // 1.1. Evento cerrar ventana
  if (event==CV_EVENT_CLOSE) {
    cb_close(factual);
    return;
  }

  // 1.2. El ratón se sale de la ventana
  if (x>=foto[factual].img->width || y>=foto[factual].img->height)
    return;

  // 1.3. Se inicia la pulsación del ratón (almacenar punto inicial)
  if (event==CV_EVENT_LBUTTONDOWN) {
    downx= x;
    downy= y;
  }

  // 2. Según la herramienta actual
  switch (herr_actual) {

    // 2.1. Herramienta PUNTOS
    case puntos:
       if (flags==CV_EVENT_FLAG_LBUTTON)
         cb_punto(factual, x, y);
       else
         ninguna_accion(factual, x, y);
       break;

    // 2.2. Herramienta LINEAS
    case lineas:
       if (event==CV_EVENT_LBUTTONUP)
         cb_linea(factual, x, y);
       else if (event==CV_EVENT_MOUSEMOVE && flags==CV_EVENT_FLAG_LBUTTON)
         cb_ver_linea(factual, x, y);
       else
         ninguna_accion(factual, x, y);
       break;

    // 2.3. Herramienta SELECCIONAR
    case seleccionar:
       if (event==CV_EVENT_LBUTTONUP)
         cb_seleccionar(factual, x, y);
       else if (event==CV_EVENT_MOUSEMOVE)
         cb_ver_seleccion(factual, x, y, flags!=CV_EVENT_FLAG_LBUTTON);
       break;
  }
}
```

¿COMO PINTAR UN PUNTO?

17. Una vez que sabemos que se quiere pintar un punto, el *callback* llama a la operación: int **cb_punto** (int nfoto, int x, int y); ¿Qué debe hacer esta operación?

18. Si el punto no está difuminado, la implementación es inmediata:

```
cvCircle(im, cvPoint(x, y), radio_pincel, color_pincel, -1);
```

19. Pero si el punto está difuminado, entonces hay que hacer más cosas…

1) Supongamos que la imagen donde queremos dibujar el punto es **im**

2) Crear una imagen **res**, del mismo tamaño que **im**, inicializada con el color del pincel

3) Crear una imagen **cop** del mismo tamaño que **im**, inicializada a negro

4) Dibujar un círculo en **cop** de color blanco

5) Suavizar el círculo de la imagen **cop** (esta imagen se usará para hacer una media ponderada entre **im** y **res**, según **cop**)

6) Multiplicar **res** por **cop**, guardando el resultado en **res**

7) Multiplicar **im** por la inversa de **cop**, guardando el resultado en **im**

8) Sumar **im** y **res**, guardando el resultado en **im**

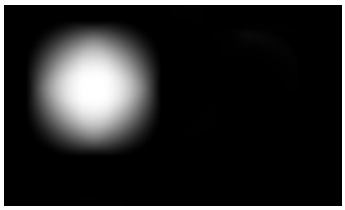
1) im

2) res

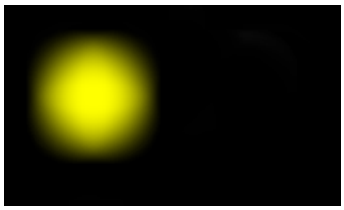
3) cop

4) cvCircle(cop)

5) cvSmooth(cop)



6) res= res*cop



7) im= im*(255-cop)

8) im= im+res

```
IplImage *res= cvCreateImage(cvGetSize(im), im->depth, im->nChannels);
cvSet(res, color_pincel);
IplImage *cop= cvCreateImage(cvGetSize(im), im->depth, im->nChannels);
cvZero(cop);
cvCircle(cop, cvPoint(x, y), radio_pincel, CV_RGB(255,255,255), -1);
cvSmooth(cop, cop, CV_BLUR, difum_pincel*2+1, difum_pincel*2+1);
cvMul(res, cop, res, 1.0/255.0);
cvNot(cop, cop);
cvMul(im, cop, im, 1.0/255.0);
cvAdd(res, im, im);
cvReleaseImage(&cop);
cvReleaseImage(&res);
```

20. Algunas indicaciones:

- La misma idea se puede usar para pintar otros tipos de elementos difuminados, como líneas, rectángulos, elipses, etc. El único cambio sería en el paso 4.
- Observar que para conseguir que las operaciones sean **eficientes** se evita el acceso píxel a píxel (a no ser que no haya otra alternativa), usando las operaciones globales de OpenCV. Este modo de trabajar puede requerir crear muchas imágenes temporales, que luego se eliminan.
- No obstante, esta implementación tampoco es muy eficiente, puesto que las operaciones se aplican sobre toda la imagen, cuando el punto pintado ocupará

solo un trozo pequeño. **Ejercicio:** modificar la operación para que seleccione un ROI y se aplique solo sobre esa parte.

OTRAS FUNCIONES DEL MODULO IMAGENES

21. Observar dentro del *callback* el comportamiento de otras herramientas. Por ejemplo, en el caso de la herramienta de **líneas**, al pulsar el botón (LBUTTONDOWN) simplemente almacenamos la posición de inicio de la línea. Al soltar el botón (LBUTTONUP) ya sabemos la posición final, y es cuando pintamos la línea. Cada herramienta tendrá su propia forma de funcionar.

22. Algunas funciones usan **asertos** para garantizar que se cumplen ciertas precondiciones. El usuario que hace la llamada debe garantizar que siempre se cumplan esas condiciones.

23. En cuanto al uso de regiones de interés (ROI) observar que las imágenes (dentro del array **foto**) no tienen ROI establecido. En su lugar, la ROI aparece como un campo en la estructura **ventana**. Solo cuando se vaya a aplicar una operación que use ROI se debe poner la ROI (cvSetImageROI) y después de aplicarla se debe quitar (cvResetImageROI).

24. A modo de ejemplo, se han incluido en imagenes.cpp algunas funciones de procesamiento relacionadas con operaciones del enunciado:
    - void **invertir** (int nfoto, int nres): crea una imagen nueva en nres, invirtiendo los colores de nfoto.
    - void **rotar_angulo** (int nfoto, IplImage *imgRes, double grado): rota la imagen nfoto en cantidad grado (ángulo en grados), almacenando el resultado en imgRes (que debe estar creada).
    - void **rotar_exacto** (int nfoto, int nres, int grado): rotación exacta de la imagen nfoto, guardando el resultado en nres, de 0, 90º, 180º, 270º, según grado valga 0, 1, 2, 3, respectivamente.
    - void **ver_brillo_contraste** (int nfoto, int suma, double prod, bool guardar= false): realiza una suma y multiplicación global sobre nfoto; si guardar es false, solo se visualiza el resultado en la ventana pero no se cambia nfoto.
    - void **ver_suavizado** (int nfoto, int ntipo, int tamx, int tamy, bool guardar= false): realiza un suavizado sobre nfoto; si guardar es false, solo se visualiza el resultado en la ventana pero no se cambia nfoto.
    - void **media_ponderada** (int nf1, int nf2, int nueva, double peso): calcula la media ponderada de dos imágenes nf1, nf2, guardando el resultado en nueva.

25. En el módulo video.h/video.cpp se incluyen algunos ejemplos de procesamiento de vídeo (crear un vídeo rotando una imagen y calcular la imagen media obtenida de una cámara).

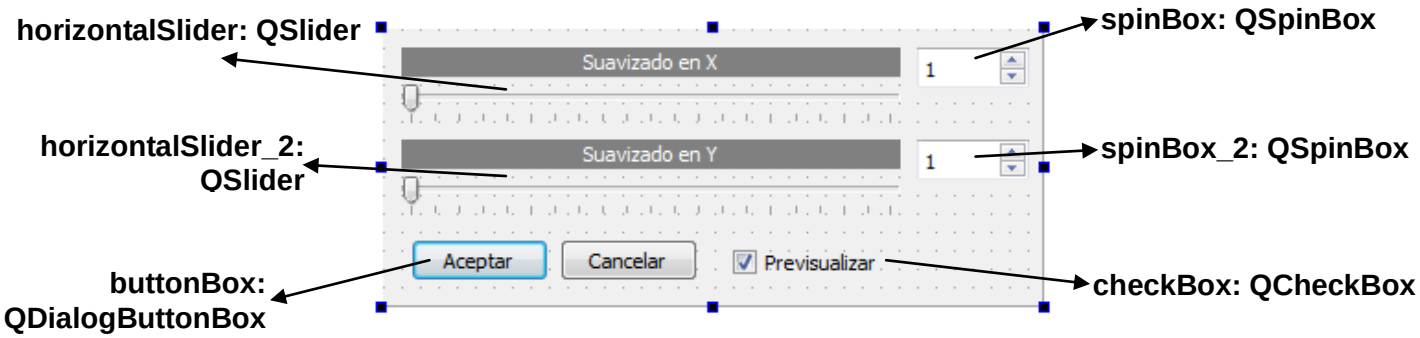
LOS CUADROS DE DIALOGO DE LA APLICACION

26. El programa base contiene varios ejemplos de cuadros de diálogo creados para la aplicación (acercade, brillocontraste, dialognueva, mediadevideo, mediaponderada,

rotaravideo y suavizados). Estos cuadros de diálogo corresponden a operaciones que requieren información adicional (aparte del radio, color y suavizado del pincel). Todos ellos son subclases de **QDialog**.

27. Observemos, por ejemplo, el cuadro de diálogo "**suavizados**".

Aspecto del formulario (suavizados.ui)



Declaración de la clase **suavizados** (en suavizados.h)

```
10  class suavizados : public QDialog {
11      Q_OBJECT
12  public:
13      suavizados(int tipo, QWidget *parent = 0);
14      ~suavizados();
15
16  protected:
17      void changeEvent(QEvent *e);
18
19  private:
20      Ui::suavizados *ui;
21      int nfoto;
22      int num_tipo;   // Tipo de suavizado: CV_GAUSSIAN, CV_BLUR
23
24  private slots:
25      void on_buttonBox_rejected();
26      void on_buttonBox_accepted();
27      void on_horizontalSlider_2_valueChanged(int value);
28      void on_horizontalSlider_valueChanged(int value);
29      void on_spinBox_2_valueChanged(int );
30      void on_spinBox_valueChanged(int );
31  };
```

28. Dentro de la parte **private:** se han añadido dos atributos (**nfoto** y **num_tipo**) que almacenan el estado interno del cuadro de diálogo. Ambos se inicializan en el constructor del tipo (suavizados(int tipo, QWidget *parent)), en suavizados.cpp:

```
nfoto= foto_activa();
num_tipo= tipo;
```

29. Dentro de un QDialog, siempre que el usuario pincha en "Aceptar" o "Cancelar", se generan los eventos **accepted()** y **rejected()** del **buttonBox**. También se generan esos eventos cuando se pulsan las teclas Intro, Esc, cuando se pincha en el aspa, etc. Por lo tanto, nuestro código asociado a la aceptación del cuadro de diálogo debe ir en el slot accepted(), y el código asociado a la cancelación debe ir en el slot rejected().

```
void suavizados::on_buttonBox_accepted()
{
    ver_suavizado(nfoto, num_tipo,
                  ui->horizontalSlider->value()*2-1,
                  ui->horizontalSlider_2->value()*2-1, true);
}

void suavizados::on_buttonBox_rejected()
{
    mostrar(nfoto);
}
```

- **ver_suavizado** es una función definida dentro de imagenes.cpp para suavizar una imagen, modificando o no la imagen original. El tamaño del suavizado debe ser impar, por eso se hace: valor*2-1.
- **mostrar** hace que se muestre la imagen original (se hace cuando se cancela el suavizado).
- Después de ejecutarse los eventos accepted() o rejected(), la ventana se cierra sola (no es necesario llamar a close() para cerrarla). No obstante, el objeto asociado al cuadro de diálogo sigue existiendo, por lo que el cliente del mismo puede seguir usándolo.

30. Los eventos asociados a los QSlider y QSpinBox tienen dos propósitos: **previsualizar** el resultado actual (si está activado el checkBox) y **sincronizar** cada *slider* con su *spinBox*.

31. Todos los QDialog tienen un método: int **exec()** que sirve para:
    - Mostrar el cuadro de diálogo (previamente creado) de forma modal (la ventana padre no puede usarse).
    - Esperar (en el que llama a exec) hasta que se cierre el cuadro de diálogo.
    - Si el usuario ha terminado pulsando en “Aceptar”, la llamada devuelve 1. Si ha terminado con “Cancelar” devuelve 0.

32. Para usar un cuadro de diálogo previamente definido necesitamos: (i) hacer el *include* correspondiente del fichero de cabecera; (ii) crear un objeto del tipo del cuadro de diálogo; (iii) llamar al método exec() del objeto; (iv) liberar el objeto. Observar, por ejemplo, el uso de la clase **suavizado** desde el formulario principal (mainwindow.cpp):

```
#include "suavizados.h"

...

void MainWindow::on_actionGausiano_triggered()
{
    if (foto_activa()!=-1) {     // Comprueba que hay alguna imagen abierta
        suavizados sg(CV_GAUSSIAN); // Crea un objeto estático de la clase
        sg.exec();               // Llama a exec y queda esperando
    }   // Como el objeto sg es una variable estática local, no hace falta
}     // liberarla. Se liberará cuando se termine el bloque en el que está
```

33. **Ejercicio**: incluir el suavizado de mediana dentro de Efectos|Suavizado.

34. **Ejercicio**: añadir un nuevo cuadro de diálogo para rotar una imagen en un ángulo cualquiera.

ALGUNAS NOTAS FINALES

35. El **programa base** aquí explicado se entrega para facilitar la realización de la práctica. Pero si los alumnos no se aclaran con su funcionamiento, no les gusta cómo está estructurado, o simplemente prefieren empezar a desarrollar su aplicación desde cero, tendrán absoluta libertad para no usar el programa base en su práctica.

36. Los posibles fallos o defectos existentes en el programa base no serán excusa para no cumplir los requisitos del enunciado de la práctica.

37. Recordar que se debe entregar con la práctica el programa ejecutable y, también, todo el código y ficheros necesarios para compilarlo. No es necesario incluir las DLL.

38. Llevar cuidado con los **errores frecuentes**:
    - Hay que liberar la memoria de todas las imágenes cuando dejen de usarse (ver el administrador de tareas de Windows).
    - Usar imágenes que tengan las profundidades adecuadas. Por ejemplo, si una operación puede tomar valores negativos no usar 8U, sino 16S o 32F.
    - Para ver si se está obteniendo un resultado correcto, probar las operaciones con los ejemplos de las transparencias de clase.
    - ¡Probar y depurar el programa antes de entregarlo!