ANEXO B:
APLICACIÓN DE SEGMENTACIÓN CON EL MÉTODO "BACKGROUND SUBTRACTION"

B.1.- Introducción:

A lo largo del capítulo 2, *Introducción teórica*, pudimos ver los distintos métodos existentes para implementar nuestra aplicación para la detección de caídas. De todos aquellos métodos comenzamos a trabajar con la segmentación basada en la separación fondo-objecto (background subtraction) la cual no dio los resultados esperados en cuanto a eficiencia y velocidad de procesamiento.

A pesar de todo, hemos pensado que sería interesante introducir en la memoria el código que se utilizó para poner a prueba este método, ya que por un lado forma parte del trabajo desempeñado durante el proyecto fin de carrera y por otro lado puede ser interesante para futuras implementaciones donde la velocidad de procesamiento no sea un parámetro tan restrictivo como en nuestro caso, pues el algoritmo de segmentación mediante *background subtraction* empleado es realmente potente.

En este anexo no volveremos a introducir todo el fundamento teórico subyacente, puesto que ya fue desarrollado durante el apartado 2.3.1 del capítulo 2 de la memoria, y nos centraremos en el código y en los resultados.

B.2.- Diseño

B.2.1.- Diagramas de flujo

Para una mejor comprensión del funcionamiento de esta aplicación presentaremos primero su diagrama de flujo.

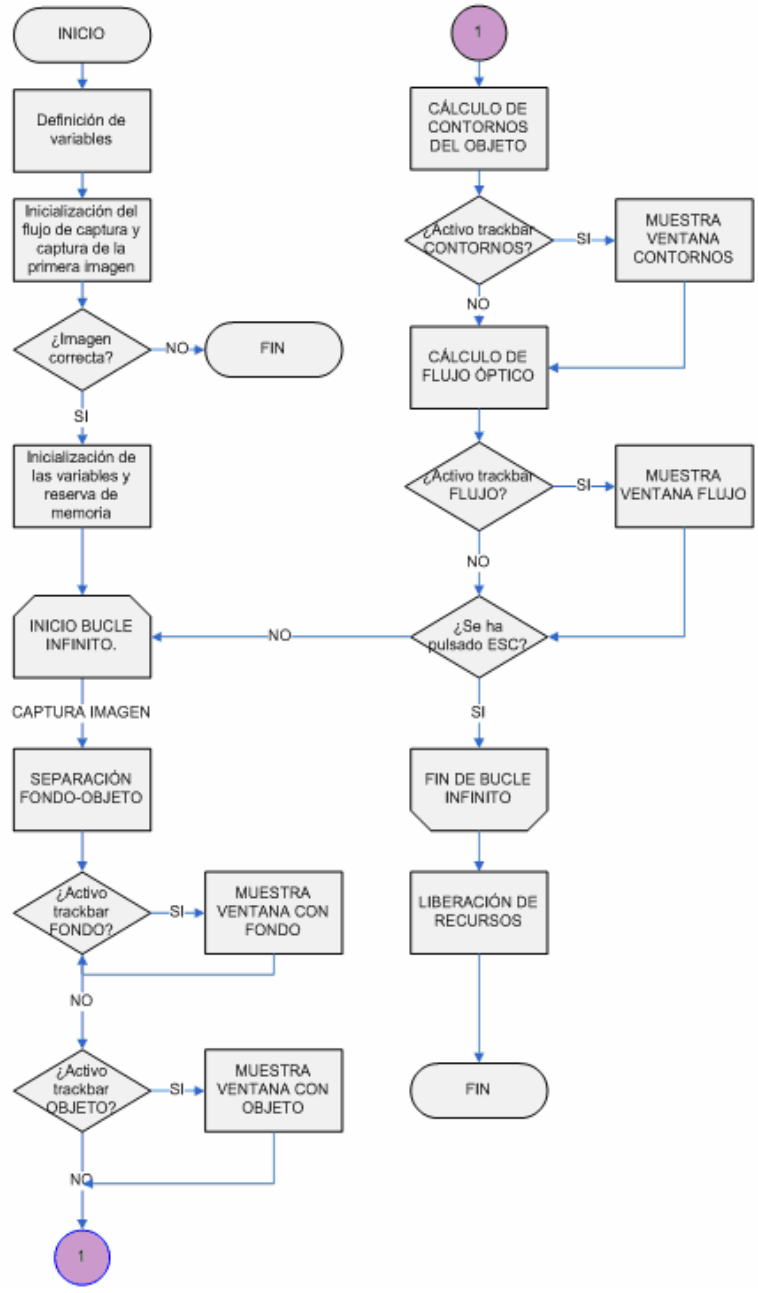

Figura B.1.- Diagrama de flujo del programa Background

B.2.2.- Estructura interna

Como hemos podido ver en el diagrama de flujo anterior, el programa se divide en tres bloques principales: Segmentación, contornos y flujo óptico. Dentro de cada uno de estos bloques existen funciones muy diversas, algunas de diseño propio y otras pertenecientes a la librería.

En este apartado vamos a describir cada uno de estos bloques y las funciones que los componen, de ese modo conseguiremos tener una visión global de la estructura del programa. En la descripción de cada parte no mostraremos el código fuente, pues éste se presentará más adelante una vez comprendido el funcionamiento de la aplicación.

B.2.2.1.- FUNCIÓN PRINCIPAL – MAIN()

Como en la mayoría de aplicaciones, la función *main()* es el esqueleto principal del programa, desde donde se inicializan al principio todos los parámetros necesarios, se llama al resto de funciones encargadas de hacer tareas específicas y desde donde finalmente se liberan los recursos del sistema al finalizar la ejecución del programa.

La estructura más importante de esta función principal es un bucle infinito encargado de realizar la captura de cada imagen dentro de la secuencia de video. En cada iteración del bucle capturaremos una sola imagen o frame, el cual será procesado obteniendo así información, como el cálculo de los píxeles pertenecientes al fondo y al objeto en movimiento, determinación de los contornos del objeto y cálculo del flujo óptico (sentido y velocidad del movimiento).

Por otro lado, al procesarse una imagen en cada iteración del bucle, intentaremos que la carga computacional dentro de éste sea lo más ligera posible, para conseguir así procesar el mayor número posible de frames por segundo. En concreto en nuestra aplicación, la función dentro de este bucle que más ralentiza el funcionamiento es la encargada de la separación fondo-objeto, consiguiendo una velocidad de procesado de 10 frames por segundo aproximadamente.

El bucle se ejecutará infinitamente hasta que pulsemos el carácter de escape “ESC” (carácter 27 en el código ASCII) o hasta que pulsemos el botón de cerrar ventana (X) de la ventana principal.

Esta función no toma como parámetro inicial ningún valor, precisando sólo que la cámara web esté conectada para poder comenzar a trabajar.

Al comenzar la ejecución del programa, la función *main()* definirá e inicializará una serie de variables, como son las imágenes que vamos a utilizar para el procesamiento de la secuencia de vídeo, el modelo estadístico para el modelado del fondo, variables de estado, variables temporales como índices de bucles, las ventanas donde se muestran los resultados del procesamiento o la variable asociada al flujo de captura de imágenes de la cámara. Por otro lado, esta función también será la encargada de reservar la memoria necesaria para aquellas variables que se inicializarán de forma dinámica durante la ejecución de la aplicación, como por ejemplo las secuencias donde almacenaremos los contornos asociados al objeto en movimiento.

Durante la ejecución de la aplicación, la función *main()* también será la encargada de decidir qué resultados son los que se van a mostrar por pantalla, gestionándolo mediante lo que se conoce como "toggle buttons", es decir, botones que presentan dos posibles estados (ON-OFF). Si el botón está en "ON" (valor 1) la imagen asociada se mostrará por pantalla, mientras que si el botón está en "OFF" (valor 0) dicha ventana se ocultará. En la siguiente figura mostramos la ventana principal de la aplicación donde se puede apreciar la secuencia de video original capturada con la webcam y los botones ON-OFF asociados a cada resultado.

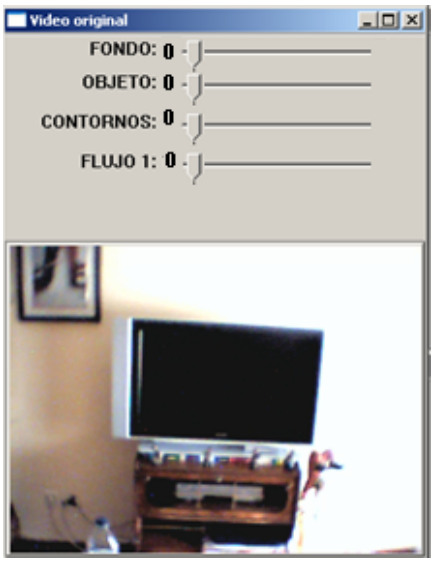

Figura B.2.- Pantalla inicial del programa

Como vemos, la interfaz gráfica es muy sencilla (pues una interfaz de usuario elaborada no es uno de los requisitos buscados en la implementación de la aplicación como ya dijimos a lo largo de la memoria).

Es importante señalar que aunque no esté activo el botón ON-OFF asociado (segmentación, contornos o cálculo del flujo óptico), la función *main()* ejecutará todas las tareas, es decir, que los botones ON-OFF sólo sirven para mostrar o no los resultados, no para evitar que una parte del código se ejecute o no. Esto es así ya que todas las tareas están interrelacionadas entre sí, por ejemplo, no es posible calcular el contorno del objeto en movimiento sin antes realizar la separación fondo-objeto pertinente.

Las cuatro ventanas de resultados controladas por los botones ON-OFF son las siguientes:

- FONDO: Una vez realizada la segmentación y separación fondo-objeto mediante la actualización del modelo gaussiano expuesto en el capitulo 2 (apartado. 2.3.1), obtendremos dos imágenes distintas. Una de ellas se corresponderá con el fondo de la imagen (bg_model->background) y la otra con el objeto en movimiento (bg_model->foreground). En esta ventana se mostrará la relacionada con el fondo.
- OBJETO: De las dos imágenes relacionadas con la segmentación, en esta ventana se muestra la relativa al objeto en movimiento. Además, aquellos píxeles de la imagen que no pertenezcan al objeto se pintarán de negro.
- CONTORNOS: En esta ventana se dibujarán los contornos en color rojo correspondientes al objeto en movimiento. Además, para poder identificar el objeto los pintaremos sobre una copia de la imagen original capturada por la cámara.
- FLUJO: Esta ventana se corresponde con una rejilla donde pintaremos los vectores asociados al flujo óptico de la imagen. Cada uno de esos vectores nos indicará la dirección del movimiento de cada píxel de la rejilla y la velocidad a la que éste se está moviendo.

Por último, una vez que nos salgamos del bucle infinito principal, lo que significará que ya vamos a finalizar la ejecución de la aplicación, tendremos que liberar todos los recursos del sistema que hayamos empleado. Para ello emplearemos funciones especiales como “cvReleaseImage” encargada de liberar las imágenes del tipo IPL, “cvReleaseCapture” para liberar la variable asociada al flujo de captura de la secuencia de vídeo o “cvDestroyAllWindows” para cerrar todas las ventanas abiertas.

B.2.2.2.- FUNCIONES SECUNDARIAS – SEGMENTACIÓN

Como ya hemos dicho, dentro del esqueleto principal del programa llamaremos a distintas funciones encargadas de realizar operaciones específicas para el correcto funcionamiento de la aplicación. En este apartado hablaremos de la función encargada de la segmentación o separación entre el fondo y los objetos en movimiento de la escena.

Esta función es la función **DeteccionMov()**. Como ya hemos explicado, para la segmentación vamos a emplear un modelo gaussiano para el fondo que inicializaremos en la función *main()* mediante "cvCreateGaussianBGModel". A continuación, y para cada iteración del bucle infinito principal, llamaremos a la función "DeteccionMov" dentro de la cual actualizaremos dicho modelo con la función "cvUpdateBGStatModel".

Los resultados que se obtienen con estas funciones no siempre son satisfactorios, debido a la existencia de distractores, cambios de luz, etc. Por esa razón y para minimizar en lo posible estos efectos sobre nuestros resultados realizaremos un filtrado de la imagen "bg_model->foreground" correspondiente al objeto en movimiento.

Este filtrado consiste en eliminar todos aquellos *blobs* o agrupaciones de píxeles cuyo área no supere un mínimo establecido. Para ello haremos uso de las funciones de librería cvBlobsLib, específicas para el trabajo con *blobs* en aplicaciones que hacen uso de OpenCV. Una vez realizado el filtrado ya tendremos la imagen del objeto sin ruido y la imagen del fondo separadas y listas para ser mostradas en las ventanas correspondientes. En la siguiente figura se muestran tres imágenes correspondientes a la imagen original capturada, a la imagen del fondo y a la imagen del objeto.

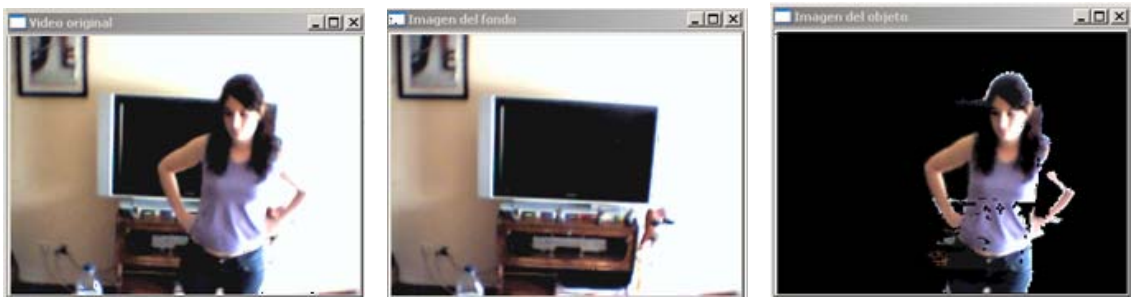

Figura B.3.- Separación fondo-objeto. De izquierda a derecha, imagen original imagen del fondo e imagen del elemento en movimiento.

B.2.2.3.- FUNCIONES SECUNDARIAS – CONTORNOS

Una vez realizada la segmentación o separación entre fondo-objeto pasaremos a calcular el contorno de la imagen correspondiente al objeto en movimiento. El dibujado de los contornos se realizará sobre una copia de la imagen original capturada desde la cámara, de forma que podremos apreciar dentro de dicha imagen qué se corresponde con el objeto y ver si la segmentación realizada está proporcionando resultados satisfactorios.

Para realizar esta operación utilizaremos la función **PintaContornos**. Su funcionamiento consiste en almacenar en una variable tipo cvSeq (secuencia) todos los contornos existentes en la imagen filtrada del objeto en movimiento empleando la función "cvFindContours". Posteriormente emplearemos la función "cvDrawContours" para pintar sobre la imagen original cada uno de esos contornos. A continuación podemos ver un ejemplo de la ventana resultante.

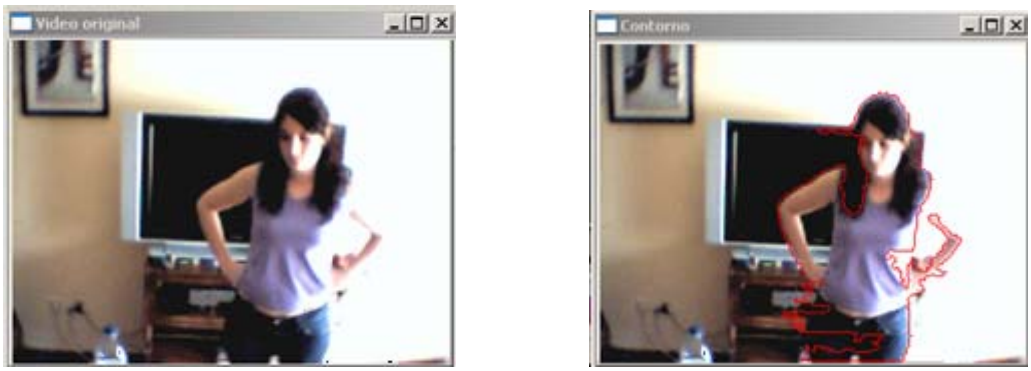

Figura B.4.- Pintado de contornos del objeto en movimiento.

B.2.2.4.- FUNCIONES SECUNDARIAS – FLUJO ÓPTICO

Para realizar un análisis del movimiento que se está produciendo en la secuencia de vídeo, será necesario conocer el flujo óptico de los píxeles, es decir, cuánto se han desplazado esos píxeles entre una imagen y otra y en qué dirección se ha realizado el desplazamiento. Teniendo en cuenta esto y sabiendo el tiempo que tardamos en procesar los datos podremos calcular la velocidad a la que se está moviendo cada uno de los píxeles o el objeto completo.

Para realizar este cometido vamos a emplear la función **tracking_flechas**. Esta función utiliza el método iterativo de Lucas&Kanade para el cálculo del flujo óptico, empleando para ello la función "cvCalcOpticalFlowPyrLK", que implementa el algoritmo iterativo piramidal

pero a la que pasaremos una serie de parámetros tales que funcionará sin emplear las pirámides, como el algoritmo general. Para ello haremos uso tanto de la imagen actual capturada como de la imagen anterior, ya que necesitamos comparar las posiciones actuales de los píxeles con respecto a sus posiciones anteriores para así poder calcular el vector desplazamiento de cada píxel.

Para calcular el flujo óptico de toda la imagen sería necesario calcular el vector de desplazamiento asociado a cada píxel de la imagen. Como eso es inviable debido a la gran carga computacional que supondría analizar todos los píxeles de la imagen, lo que haremos será calcular una rejilla, donde especificaremos a qué puntos de la imagen se les va a calcular el vector de desplazamiento. El número de píxeles a los que se les calcula ahora el flujo óptico es mucho más limitado y por lo tanto la carga computacional estará controlada. Es importante señalar que la rejilla no se calcula dentro de la función tracking_flechas, sino en la inicialización de la función *main()*.

Una vez calculados todos los vectores de desplazamiento de los puntos pertenecientes a la rejilla obtendremos un panel lleno de vectores indicando la dirección del movimiento de cada píxel y su velocidad, ya que cuanto más grande es la flecha nos indica que mayor ha sido el desplazamiento y que por lo tanto mayor ha sido su velocidad (ya que el tiempo de procesamiento entre una imagen y la siguiente es siempre el mismo).

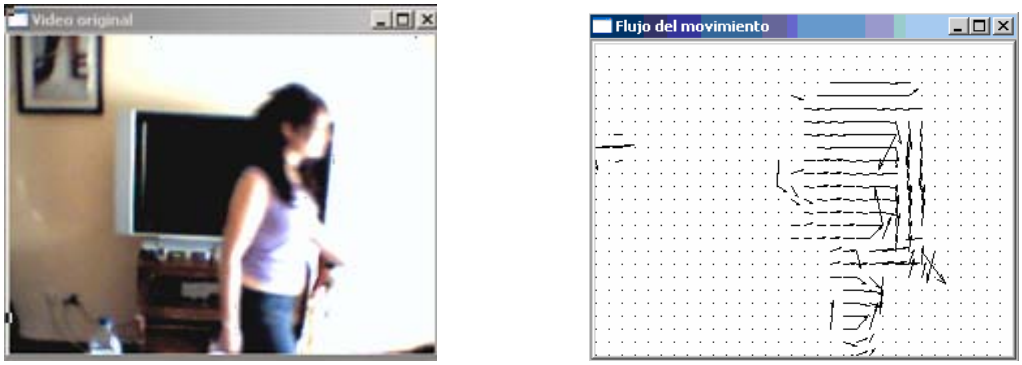

Figura B.5.- Cálculo del flujo óptico de la secuencia de video

B.2.3.- Funciones internas

Ahora que ya tenemos una visión global del funcionamiento de la aplicación pasaremos a explicar cada una de las funciones que hemos utilizado en el código fuente. De cada una de estas funciones detallaremos una breve explicación e indicaremos cuáles son los parámetros de entrada y el valor de retorno (si los hubiera).

B.2.3.1.- FUNCIONES DE LIBRERÍA C++

Este apartado engloba a todas aquellas funciones estándar en el uso de C y C++ pertenecientes al paquete stdio.h. De este paquete sólo hemos empleado la función printf("Cadena"), encargada de mostrar por consola (salida por defecto) la cadena de caracteres que se le indique como parámetro.

B.2.3.2.- FUNCIONES DE LA LIBRERÍA OPENCV

Aquí detallaremos cada una de las funciones pertenecientes a la librería OpenCV que hemos utilizado en el código fuente ordenadas según su funcionalidad.

Muchas de estas funciones forman parte del código interno de las funciones propias que hemos programado para la aplicación y que presentaremos en el siguiente apartado.

A.- FUNCIONES GENERALES

cvCreateImage

- Declaración: `IplImage* cvCreateImage( CvSize size, int depth, int channels );`
- Descripción: Función que sirve para crear la cabecera y reservar el espacio necesario para los datos asociados a una variable de tipo IplImage.
- Parámetros:
  - size: Tamaño de la imagen (ancho y alto).
  - depth: Profundidad en bits de la imagen. Es decir, número de bits empleados para codificar cada pixel.
  - channels: Número de canales asociados a un píxel. Puede ser 1, 2, 3 o 4. En general suele valer 1 o 3, para referirse a las imágenes monocromas en escala de gris y a las imágenes en color (con sus tres planos).
  - IplImage * : Su valor de retorno es un puntero a la estructura que se acaba de reservar.

cvReleaseImage

- Declaración: `void cvReleaseImage( IplImage** image );`
- Descripción: Función que sirve para liberar los recursos que se han reservado previamente para una imagen con la función

cvCreateImage. Esta función libera tanto la cabecera como los datos que contiene la imagen.

- Parámetros:
  - image: Doble puntero a la cabecera de la imagen que queremos liberar.

cvCaptureFromCAM

- Declaración: `CvCapture* cvCaptureFromCAM( int index);`
- Descripción: Función que inicializa la captura de imágenes de una cámara determinada. Se encarga de reservar espacio e inicializar una estructura del tipo CvCapture encargada de leer el flujo de vídeo de la cámara
- Parámetros:
  - size: Variable que indica el índice de la cámara que se va a usar para la captura. Si sólo hay una cámara conectada o no importa la cámara que se vaya a utilizar (cualquiera es válida) se le puede pasar -1 como parámetro.
  - CvCapture*: Devuelve un puntero a una estructura del tipo CvCapture.

cvReleaseCapture

- Declaración: `void cvReleaseCapture(CvCapture** capture);`
- Descripción: Función que libera los recursos de una estructura del tipo CvCapture (es lo contrario a cvCaptureFromCAM).
- Parámetros:
  - capture: Doble puntero a la estructura que queremos liberar.

cvQueryFrame

- Declaración: `IplImage* cvQueryFrame(CvCapture* capture);`
- Descripción: Esta función captura un *frame* o imagen de la cámara procedente de la secuencia de vídeo, la descomprime y la devuelve.
- Parámetros:
  - capture: Puntero a una estructura CvCapture, cabecera del flujo de vídeo de la cámara.
  - IplImage: Devuelve la imagen capturada del flujo de la cámara.

cvAlloc

- Declaración: `void* cvAlloc(size_t size );`

- Descripción: Función que reserva un buffer de memoria de un tamaño determinado. Devuelve un puntero al buffer reservado. En caso de error, la función devolverá NULL.
- Parámetros:
  - size: Tamaño del buffer en bytes.

cvFree

- Declaración: `void cvFree(void** ptr );`
- Descripción: Función que libera un buffer de memoria previamente reservado con la función cvAlloc explicada anteriormente. Limpia también el puntero que apunta el buffer, es por eso por lo que se utiliza un doble puntero como parámetro.
- Parámetros:
  - ptr: Puntero al puntero que apunta al buffer que vamos a liberar.

cvCreateMemStorage

- Declaración: `CvMemStorage* cvCreateMemStorage( int block_size=0 );`
- Descripción: Función que reserva una zona de memoria y devuelve el puntero que apunta a esa zona. En un principio esa zona de memoria estará vacía.
- Parámetros:
  - block_size: Tamaño de la memoria que se va a reservar. Este tamaño se especifica en bytes. Si el valor de la variable es 0, el tamaño que se reserva es el tamaño por defecto, 64 K.
  - CvMemStorage*: Puntero a la estructura donde se reserva el espacio de memoria.

cvClearMemStorage

- Declaración: `void cvClearMemStorage( CvMemStorage* storage );`
- Descripción: Función que libera la zona de memoria reservada mediante la función cvCreateMemStorage.
- Parámetros:
  - storage: Zona de memoria que queremos liberar.

cvSmooth

- Declaración: `void cvSmooth( const CvArr* src, CvArr* dst, int smoothtype=CV_GAUSSIAN, int param1=3, int param2=0, double param3=0 );`
- Descripción: Función que suaviza la imagen.
- Parámetros:
  - src: Matriz o imagen origen que queremos suavizar.

- dst: Matriz o imagen destino donde vamos a guardar el resultado del suavizado de la imagen original.
- smoothtype: Tipo de suavizado que le vamos a aplicar a la imagen. Los posibles valores son: CV_BLUR_NO_SCALE, CV_BLUR, CV_GAUSSIAN, CV_MEDIAN, CV_BILATERAL).
- param1: Primer parámetro de la operación de suavizado.
- param2: Segundo parámetro de la operación de suavizado.
- param3: En caso de utilizar un suavizado del tipo CV_GAUSSIAN, este parámetro debe especificar el valor de la desviación estándar.

cvLine

- Declaración: `void cvLine(CvArr* img, CvPoint pt1, CvPoint pt2, CvScalar color, int thickness=1, int line_type=8, 0);`
- Descripción: Función que dibuja una línea conectando dos puntos.
- Parámetros:
  - src: Matriz o imagen donde queremos pintar la línea.
  - pt1: Primer punto del segmento de línea.
  - pt2: Segundo punto del segmento de línea.
  - color: Color del que vamos a pintar la línea
  - thickness: Grosor de la línea
  - line_type: Tipo de línea, pudiendo ser por ejemplo una línea con conectividad de 8 o línea con conectividad de 4.

cvFindContours

- Declaración: `int cvFindContours(CvArr* imgage, CvMemStorage* storage, CvSeq** first_contour, int header_size = sizeof(CvContour), int mode=CV_RETR_LIST , int method=CV_CHAIN_APPROX_SIMPLE, CvPoint offset=cvPoint(0,0));`
- Descripción: Encuentra los contornos de una imagen binaria (cuyos píxeles valen 0 o 1).
- Parámetros:
  - image: Imagen origen de un solo canal. Aquellos píxeles de la imagen que no sean cero serán tratados como si fueran unos y los que sean cero, permanecerán como tal, de ese modo la imagen será binarizada.
  - storage: Contenedor donde se van a almacenar todos los contornos encontrados en la imagen.
  - first_contour: Es un parámetro de salida, es un puntero al primer contorno.
  - header_size: Tamaño de la cabecera de la secuencia formada por los tramos del contorno.

- mode: Modo de búsqueda de los contornos en la imagen, por ejemplo, búsqueda de sólo los contornos exteriores, búsqueda de los contornos interiores o ambos.
- method: Método de aproximación utilizado para crear los contornos a partir de los segmentos de línea o de los puntos encontrados.
- int: Devuelve el número de contornos encontrados.

cvDrawContours

- Declaración: int cvDrawContours(CvArr* img, CvSeq* contour, CvScalar external_color, CvScalar hole_color, int max_level, int thickness = 1, int line_type=8);
- Descripción: Función que dibuja los contornos exteriores o interiores en una imagen
- Parámetros:
  - img: Imagen donde vamos a dibujar el contorno.
  - contour: Puntero al primer contorno que vamos a pintar
  - external_color: Color que vamos a utilizar para los contornos exteriores.
  - hole_color: Color que vamos a utilizar para los contornos interiores (los "agujeros").
  - max_level: Es el máximo nivel de profundidad para dibujar los contornos, es decir, si esta variable vale 0, sólo se dibujan los contornos, si vale 1, se dibujará este contorno y todos aquellos contornos del mismo nivel (cuando hablamos de nivel nos referimos a lo internos que son los contornos en una figura). Si vale 2, se dibujarán todos los contornos con este nivel y todos los que tengan un nivel inferior, etc.
  - thickness: Es el grosor de las líneas de los contornos que vamos a pintar. Si el valor es negativo, como por ejemplo CV_FILLED, se rellena el contorno por dentro.
  - line_type: Tipo de segmento de línea que forma el contorno. Ver descripción de cvLine.

cvCopy

- Declaración: void cvCopy(const CvArr* src, CvArr* dest, const CvArr* mask=NULL);
- Descripción: Función que se encarga de copiar un array o matriz. Las matrices también pueden ser imágenes del tipo IplImage.
- Parámetros:
  - src: Array o matriz que vamos a copiar.
  - dst: Array donde vamos a almacenar la copia del original.

- mask: Es una matriz o imagen de un único canal y de 8 bits de profundidad. Especifica qué elementos (o píxeles) de la matriz de destino deben modificarse.

cvCvtColor

- Declaración: void cvCvtColor(const CvArr* src, CvArr* dest, int code);
- Descripción: Función que se utiliza con imágenes en color para pasar de un espacio a otro, por ejemplo, para pasar del espacio RGB al espacio HSV.
- Parámetros:
  - src: Es la imagen original, la que queremos representar en otro espacio. Puede ser de profundidad 8-bit, 16-bit o 32 bits.
  - dst: Imagen destino donde almacenaremos el resultado. Los datos de esta imagen deben ser del mismo tipo que los de la imagen original (debe tener la misma profundidad). Sin embargo, el número de canales puede ser distinto.
  - code: valor entero que nos especifica el tipo de transformación que vamos a hacer. Este cambio se puede especificar con una variable constante o macro expresada de la siguiente forma CV_<espacio_origen>2<espacio_destino>, por ejemplo CV_RGB2HSV.

cvSplit

- Declaración: void cvSplit(const CvArr* src, CvArr* dest0, CvArr* dest1, CvArr* dest2, CvArr* dest3);
- Descripción: Función que divide una matriz o imagen multicanal (lo que nosotros entendemos como imagen en color) en cada uno de sus canales. Esta función también puede extraer sólo uno de los canales, aquel donde dstX no sea NULL.
- Parámetros:
  - src: Es la imagen original en color que queremos separar en sus distintos planos.
  - dst0...dst3: Imagen monocroma donde vamos a alojar cada uno de los planos o canales que vamos a extraer de la imagen original.

cvMerge

- Declaración: void cvMerge(CvArr* src0, CvArr* src1, CvArr* src2, CvArr* src3, CvArr* dest);
- Descripción: Función que hace lo contrario que cvSplit, es decir, toma varias imágenes de un único canal y las convierte en una

imagen multicanal donde cada plano se corresponde a cada una de las imágenes que se le pasa como parámetro. También puede introducir un determinado canal en una imagen multicanal.

- Parámetros:
  - src0...src3: Son los canales de entrada.
  - dst: Imagen multicanal destino a la que vamos a introducir los canales especificados.

cvCalcOpticalFlowPyrLK

- Declaración: void cvCalcOpticalFlowPyrLK(const CvArr* prev, const CvArr* curr, CvArr* prev_pyr, CvArr* curr_pyr, const CvPoint2D32f* prev_features, CvPoint2D32f* curr_features, int count, CvSize win_size, int level, char* status, float* track_error, CvTermCriteria criteria, int flags);
- Descripción: Calcula el flujo óptico de una secuencia de imágenes utilizando para ello el método iterativo piramidal de Lucas-Kanade.
- Parámetros:
  - prev: Primer *frame* o imagen de la secuencia en el instante t.
  - curr: Segundo *frame* de la secuencia, tomado en el instante t+dt.
  - prev_pyr: Buffer para la pirámide del primer frame. Si este puntero no es NULL, el buffer debe ser capaz de almacenar la pirámide completa desde el nivel 1 hasta el nivel #level.
  - curr_prev: Esta variable es igual que *prev_pyr*, pero utilizada con el segundo frame.
  - prev_features: Matriz de puntos para los cuales queremos encontrar el flujo óptico.
  - curr_features: Matriz de puntos que contiene las nuevas posiciones calculadas para los puntos de entrada en la segunda imagen o frame.
  - count: Número de puntos a los que vamos a calcular el flujo óptico.
  - win_size: Tamaño de la ventana de búsqueda para cada nivel de la pirámide.
  - level: Valor del máximo nivel de la pirámide. Si este valor es 0, significa que en realidad no se está utilizando el método piramidal (pues sólo se está utilizando un nivel), si vale 1, estamos utilizando 2 niveles, y así sucesivamente.
  - status: Matriz donde cada uno de sus elementos tiene el valor 1 o 0 en función de si hemos encontrado el flujo óptico de su correspondiente punto o no.
  - error: Es un parámetro opcional, puede valer NULL.

- criteria: Especifica cuándo el proceso de iteración para encontrar el flujo óptico de cada punto en cada nivel de la pirámide debe parar.
- flags: Variable bandera que se utiliza para especificar tipo de funcionamiento o estado del proceso, por ejemplo, si la bandera vale CV_LKFLOW_PYR_A_READY significa que la pirámide asociada a la imagen primera se calcula antes de la llamada para calcular el flujo.

B.- FUNCIONES PARA LA SEGMENTACIÓN FONDO-OBJETO

El siguiente conjunto de funciones agrupa a las encargadas de realizar la segmentación o separación fondo-objeto en movimiento. El funcionamiento de estas funciones se basa en el artículo *"An improved adaptative background mixture model for real-time tracking with shadow detection"* de P. KadewTraKuPong y R. Borden, cuyo fundamento teórico se detalla en el capítulo 2 de la memoria.

cvCreateGaussianBGModel

- Declaración: `(CvBGStatModel*) cvCreateGaussianBGModel( IplImage* first_frame, CvGaussBGStatModelParams* parameters CV_DEFAULT(NULL));`
- Descripción: Función que a partir de la primera imagen capturada de la secuencia de video crea el modelo de gaussianas que va a modelar el fondo de la imagen.
- Parámetros:
  - first_frame: Primera imagen de la secuencia de video a partir de la cual generaremos el modelo gaussiano.
  - parameters: Puntero del tipo CvGaussBGStatModelParams * donde se incluyen parámetros para la inicialización del modelo como el número de gaussianas que formarán el modelo, valor de umbral para la segmentación, velocidad de aprendizaje, etc. Si se deja vacío se asignarán los valores por defecto.
  - CvBGStatModel *: Devuelve un puntero al modelo, el cual tendremos que ir actualizando en cada iteración del bucle.

cvUpdateBGStatModel

- Declaración: `int cvUpdateBGStatModel( IplImage* curr_frame, CvBGStatModel* bg_model);`

- Descripción: Función que ejecutaremos cada vez que capturemos una nueva imagen de la secuencia de video para actualizar el modelo de gaussianas del fondo.
- Parámetros:
  - curr_frame: Imagen actual de la secuencia de video.
  - bg_model: Puntero del tipo CvBGStatModel * que apunta a la estructura que almacena el modelo.
  - int: Devuelve el valor de una variable interna "region_count" que especifica el número de regiones o blobs que forman parte del objeto segmentado del fondo.

cvReleaseBGStatModel

- Declaración: `void cvReleaseBGStatModel(CvBGStatModel* bg_model);`
- Descripción: Función que libera los recursos reservados para el modelo gaussiano y destruye la estructura que los almacenaba.
- Parámetros:
  - bg_model: Puntero del tipo CvBGStatModel * que apunta a la estructura que almacena el modelo.

C.- GESTIÓN DE LA INTERFAZ DE USUARIO.

En este grupo de funciones introduciremos aquellas que se encargan de la creación, gestión y destrucción de ventanas y trackbars (barras de desplazamiento) y la gestión de los eventos del ratón y del teclado.

cvNamedWindow

- Declaración: `int cvNamedWindow( const char* name, int flags );`
- Descripción: Función que abre una ventana que servirá de soporte para mostrar imágenes o para contener "trackbars" (barras de desplazamiento).
- Parámetros:
  - name: Nombre o título de la venta. Servirá de identificador de ésta durante la vida de la ventana.
  - flags: Banderas para el funcionamiento de la ventana. Actualmente sólo se soporta la bandera CV_WINDOW_AUTOSIZE que ajusta automáticamente el tamaño de la ventana a la imagen que muestra. (ver cvShowImage).

cvShowImage

- Declaración: void cvShowImage(const char* name, const cvArr* image);
- Descripción: Función que muestra en la ventana la imagen que se le pasa como parámetro.
- Parámetros:
  - name: Nombre identificador de la venta.
  - image: Imagen que vamos a mostrar en la ventana.

cvMoveWindow

- Declaración: int cvMoveWindow(const char* name, int x, int y);
- Descripción: Función que desplaza la ventana con ese nombre a la posición (x, y) de la pantalla.
- Parámetros:
  - name: Nombre de la venta que lo identifica
  - x, y: Variables que indican la nueva posición de la ventana en la pantalla.

cvDestroyWindow

- Declaración: void cvDestroyWindow(const char* name);
- Descripción: Función que destruye la ventana con dicho nombre.
- Parámetros:
  - name: Nombre o título de la venta, que es lo que sirve como identificador para referenciarla.

cvDestroyAllWindows

- Declaración: void cvDestroyAllWindows();
- Descripción: Función que destruye todas las ventanas abiertas en ese momento en la aplicación.

cvCreateTrackbar

- Declaración: int cvCreateTrackbar(const char* trackbar_name, const char* window_name, int * value, int count, CvTrackbarCallback on_change);
- Descripción: Función que crea una nueva barra de desplazamiento y que la sitúa dentro de la ventana especificada como parámetro.
- Parámetros:
  - trackbar_name: Nombre de la barra de desplazamiento, que es lo que sirve como identificador para referenciarla.
  - window_name: Nombre de la ventana donde situaremos el trackbar.
  - value: Puntero a entero cuyo valor almacena la posición de la barra de desplazamiento.

- count: Máxima posición posible para el deslizador. La posición mínima siempre es 0.
- on_change: Puntero a la función que se llamará cada vez que el slider cambie de posición. Esa función debe seguir el siguiente esquema: void Funcion(int). Este parámetro lo pondremos a NULL cuando no se requiera la ejecución de ninguna función.

cvWaitKey

- Declaración: int cvWaitKey(int delay);
- Descripción: Función que espera un determinado tiempo a que se pulse cualquier tecla.
- Parámetros:
  - delay: Tiempo que se va a esperar la pulsación de la tecla expresado en milisegundos. Si el valor es negativo o 0 la función espera infinitamente.
  - **NOTA:** Esta función es el único método que posee HighGUI de OpenCV para poder manejar eventos, por lo que debe ser llamada de forma periódica para poder procesarlos a no ser que HighGUI trabaje de forma paralela con algún entorno encargado de dicha gestión.

D.- FUNCIONES DE LA LIBRERÍA CVBLOBSLIB

Dentro de una de las funciones que hemos programado, en concreto en DeteccionMov, hacemos uso de una serie de funciones específicas para el manejo de *blobs* dentro de una imagen. En este apartado resumiremos las que hemos utilizado.

GetBlob

- Declaración: CBlob CBlobResult::GetBlob(int indexBlob);
- Descripción:Método de la clase CBlobResult que devuelve el blob especificado.
- Parámetros:
  - indexBlob: Índice del blob que vamos a devolver, siempre que el índice no sea -1.
  - CBlob: Devuelve el blob con dicho índice

GetNumBlobs

- Declaración: int CBlobResult::GetNumBlobs();

- Descripción: Método de la clase CBlobResult que devuelve el número de blobs dentro de la variable CBlobResult que lo haya llamado.
- Parámetros:
  - int: Devuelve el número de blobs.

Filter

- Declaración: `void CBlobResult::Filter( CBlobResult &dst, int filterAction, funcio_calculBlob *evaluador, int condition, double lowLimit, double highLimit);`
- Descripción:Es un método dentro de la clase CBlobResult. Este método filtra los blobs de la clase, dejando sólo aquellos que cumplan (B_INCLUDE) o que no cumplan (B_EXCLUDE) una determinada condición, basándose en características como el área, perímetro, convexidad, elongación, etc.
- Parámetros:
  - dst: Variable tipo CBlobResult donde dejaremos los blobs filtrados.
  - filterAction: Nos indica si vamos a incluir (B_INCLUDE) los blobs que cumplan la condición o si los vamos a excluir (B_EXCLUDE) del resultado.
  - evaluador: Función para evaluar los blobs, por ejemplo, si la condición depende del área el evaluador será CBlobGetArea.
  - condition: Variable que nos indica la condición que debe cumplir cada blob. Puede ser: B_EQUAL, B_NOT_EQUAL, B_GREATER, B_LESS, B_GREATER_OR_EQUAL, B_LESS_OR_EQUAL, B_INSIDE, B_OUTSIDE.
  - lowLimit: Valor numérico para la comparación, límite para el filtrado.
  - highLimit: Valor numérico para la comparación. Sólo se utilizará este parámetro en caso en que la condición tenga dos valores (B_INSIDE por ejemplo). En caso contrario lo pondremos a 0.

FillBlob

- Declaración: `void CBlob::FillBlob( IplImage* image, CvScalar color );`
- Descripción: Es un método dentro de la clase CBlob. Este método pinta el blob que la ha llamado del color especificado y devuelve el resultado en la imagen introducida como parámetro.
- Parámetros:
  - image:Imagen donde se podrá ver el resultado después de pintar el blob que llamó al método.

- color: Color que utilizaremos para pintar el blob.

B.2.3.3.- FUNCIONES PROPIAS

En este último apartado incluiremos las funciones de programación propia que hemos utilizado para el programa de segmentación con *Background subtraction*.

DeteccionMov

- Declaración: void DeteccionMov(CvBGStatModel* bg_model, IplImage* original, IplImage* objeto, IplImage* fondo, IplImage* sinRuido);
- Descripción: Es la función encargada de hacer la segmentación entre el fondo y el objeto en movimiento existente en la escena mediante la actualización de su modelo y de filtrar el resultado obtenido eliminando aquellos *blobs* que no son lo suficientemente grandes y que por lo tanto consideramos ruido.
- Parámetros:
  - bg_model: Variable del tipo CvBGStatModel donde se almacena el modelo del fondo de la imagen basado un conjunto de gaussianas.
  - original: Imagen original capturada desde la cámara y que utilizaremos para actualizar el modelo gaussiano.
  - objeto: Imagen donde almacenaremos el resultado de la segmentación correspondiente al objeto una vez realizado el filtrado de ruido.
  - fondo: Imagen donde almacenaremos el resultado de la segmentación correspondiente al fondo. Es decir, guardamos el valor de bg_model->background.
  - sinRuido: Imagen en escala de grises (monocanal) que utilizaremos para realizar el filtrado del ruido.

PintaContornos

- Declaración: void PintaContornos(IplImage* gris, IplImage* resultado);
- Descripción: Es la función encargada de pintar en una copia de la imagen original los contornos pertenecientes a la imagen objeto obtenida después de la segmentación.
- Parámetros:

- gris: Imagen en escala de grises (monocanal) donde se encuentra el objeto del que queremos calcular los contornos
- resultado: Imagen multicanal sobre la que queremos pintar los contornos y donde se almacenará el resultado final.

Tracking_flechas

- Declaración: int tracking_flechas(IplImage* imagenGris,IplImage* actual, IplImage* anterior, IplImage* piramidal, IplImage* ant_piramidal, char* estado, int inicio_seg, int cont, int salto, CvPoint2D32f* puntos0, CvPoint2D32f* puntos1, IplImage* flujo_mov);
- Descripción: Es la función encargada de calcular el flujo óptico de una secuencia de vídeo comparando dos imágenes consecutivas de dicha secuencia. Para ello se utiliza el método iterativo de Lucas-Kanade explicado en el capítulo 2.
- Parámetros:
  - imagenGris: Imagen en escala de grises de la imagen original capturada por la cámara.
  - actual: Imagen correspondiente al primer frame para calcular el flujo óptico. Al inicio de la función esta variable almacena la "imagen actual" de la iteración anterior, es decir, lo que en esta iteración se correspondería con la "imagen anterior".
  - anterior:Imagen correspondiente al segundo frame para calcular el flujo óptico. Dentro de la función, se realiza una actualización de las imágenes anterior y actual tal que: anterior = actual; actual = imagenGris.
  - piramidal:Imagen utilizada para el método piramidal relacionada con la imagen actual que acabamos de capturar. (No se utiliza en nuestra aplicación pues el parámetro "level" de cvCalcOpticalPyrLK vale 0).
  - ant_piramidal: Imagen utilizada para el método piramidal relacionada con la imagen capturada en el instante anterior. (No se utiliza en nuestra aplicación pues el parámetro "level" de cvCalcOpticalPyrLK vale 0).
  - estado: Matriz donde cada uno de sus elementos tiene el valor 1 o 0 en función de si hemos encontrado el flujo óptico de su correspondiente punto o no.
  - inicio_seg: Número de puntos totales a los que les queremos calcular el flujo óptico (todos los puntos de la rejilla)
  - cont: Número de píxeles que realmente han sufrido un desplazamiento y a los que les hemos calculado el flujo óptico.
  - salto: Separación entre cada uno de los píxeles de la imagen a los que les vamos a calcular el flujo óptico. (Separación entre los puntos de la rejilla)

- puntos0: Conjunto de puntos que indican la posición inicial de los píxeles a los que les vamos a calcular el flujo óptico.
- puntos1: Conjunto de puntos que indican las posición final de dichos píxeles.
- flujo_mov: Imagen donde vamos a guardar el resultado del cálculo del flujo óptico. En esta imagen se puede ver la rejilla de puntos y sus vectores de desplazamiento asociados.

B.3.- Ejemplo del funcionamiento

Una vez vistos el diagrama de flujo y cada una de las funciones que forman el programa, con sus declaraciones, definiciones y parámetros, pasaremos a ver un ejemplo de la aplicación funcionando, con sus correspondientes pantallas y una breve explicación referente a los resultados obtenidos y a las razones que nos llevaron a desechar este algoritmo para nuestro sistema de detección de caídas.

En primer lugar, y tras abrir el ejecutable de esta aplicación, seleccionaremos todas las ventanas visibles (FONDO, OBJETO, CONTORNOS y FLUJO) para que podamos ver en todo momento los resultados que se van obteniendo. Tras la activación, y sin introducir ningún elemento en movimiento dentro de la imagen, el resultado que obtenemos es el siguiente:

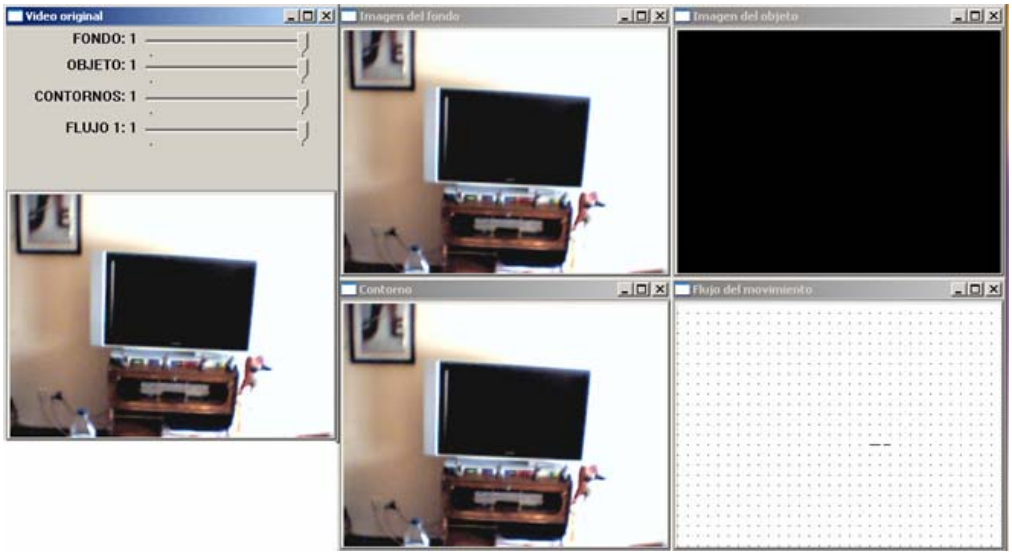

Figura B.6.- Imagen inicial tras la activación de la ventana

Como podemos ver en la figura anterior, la ventana “Flujo del movimiento” nos muestra que existe un leve movimiento en la secuencia de

vídeo (vemos dos pequeños vectores de desplazamiento) y sin embargo vemos que la ventana “Imagen del objeto” aparece vacía. Esto es así debido al filtrado del ruido que se le aplica a la imagen-objeto.

A continuación, introducimos un elemento en movimiento delante de la cámara. Los resultados que se obtienen los podemos observar en la siguiente imagen:

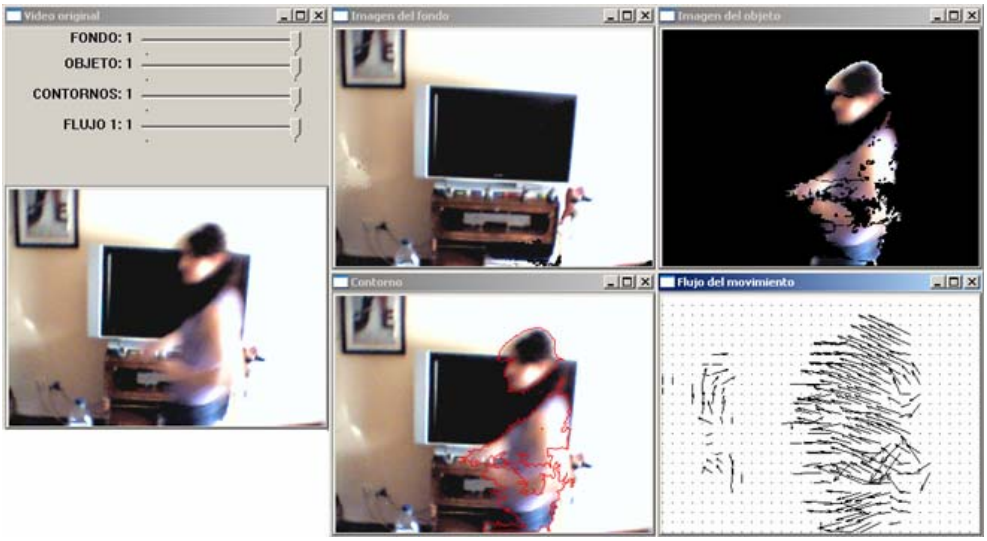

Figura B.7.- Introducción de un elemento en movimiento

Una vez establecido el modelo de fondo, si el elemento de interés deja de moverse el sistema lo sigue considerando como objeto (aunque no esté en movimiento).

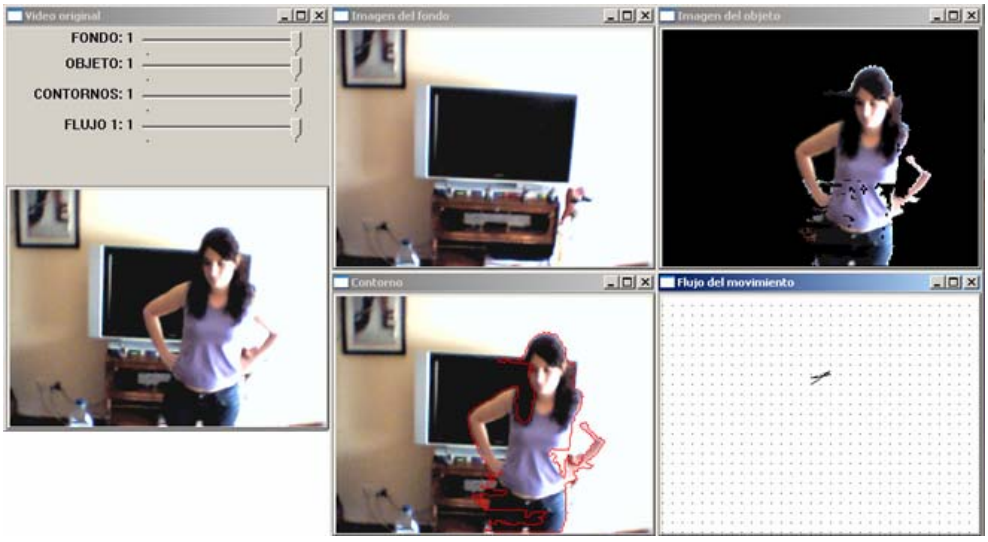

Figura B.8.- Elemento de interés sin moverse

Pero sin embargo, debido a que el modelo se va actualizando dinámicamente, si el sujeto al que estamos siguiendo permanece quieto en el mismo sitio durante un tiempo determinado (alrededor de unos 30 segundos) entonces éste comienza a considerarse como parte del fondo, como podemos ver en la siguiente imagen.

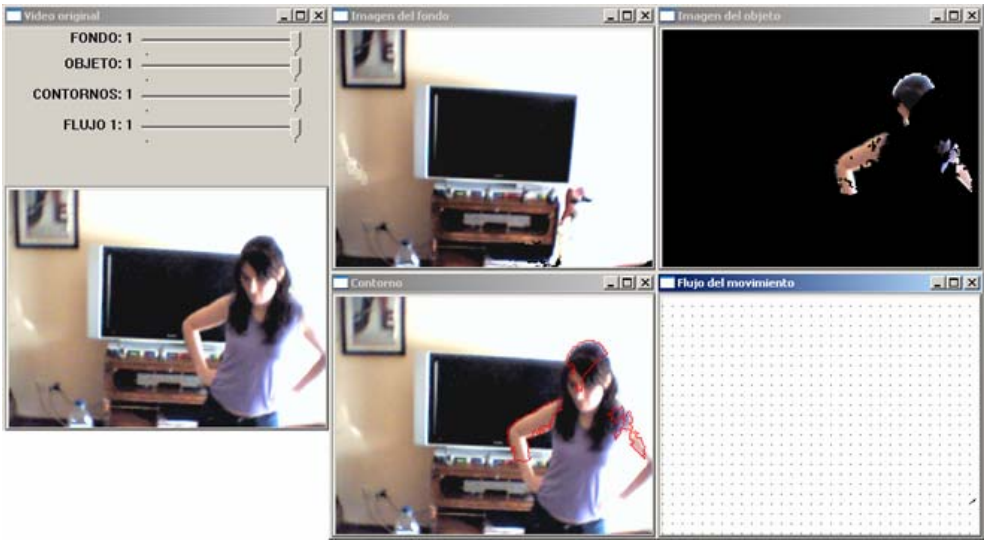

Figura B.9.- Sujeto quieto que comienza a confundirse con el fondo

Esto es un inconveniente bastante serio para nuestro detector de caídas, ya que es bastante probable que el individuo monitorizado permanezca inmóvil en la pantalla durante largos periodos de tiempo (si está leyendo, viendo el televisor, sentado, etc), de tal forma que dejaríamos de verlo, y no sólo eso, sino que en esa zona de la pantalla donde ha permanecido quieto se produciría una **zona de oclusión**, pues cada vez que el individuo vuelva a pasar por ahí, aunque se esté moviendo, se confundirá con el fondo.

Otro inconveniente de este algoritmo, como ya vimos en el capítulo 2, es que no es robusto ante cambios de iluminación o sombras como podemos observar en la figura B.10, haciendo que ni el filtrado de ruido sea capaz de solventar este problema.

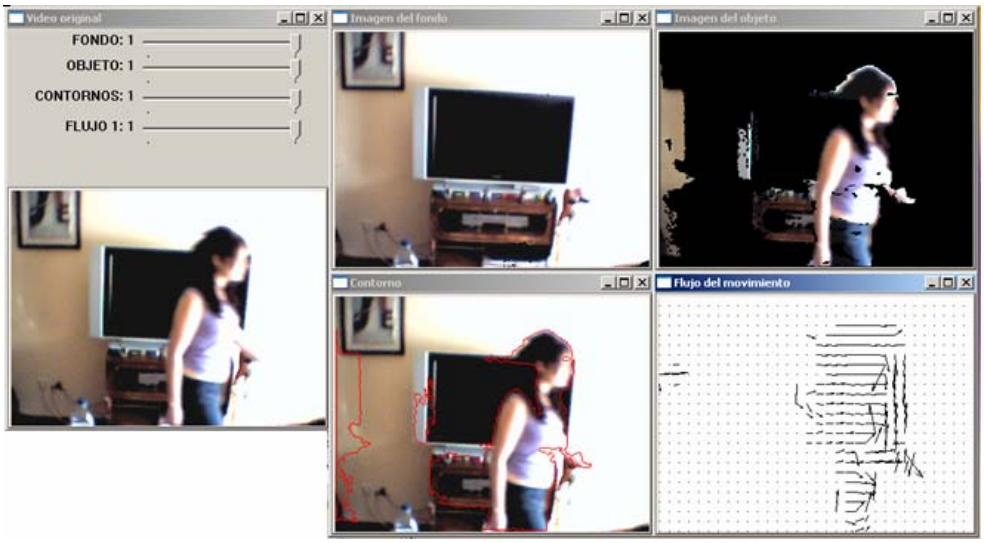

Figura B.10.- Problemas con los cambios de iluminación y sombras

Por estas razones y debido a que el procesamiento de imágenes no es lo suficientemente veloz (15 frames por segundo aproximadamente) decidimos desechar este algoritmo e intentarlo con el método Camshift.

B.4.- Código fuente

En este apartado detallaremos y comentaremos el código utilizado para la implementación de una aplicación centrada en la segmentación mediante separación fondo-objeto.

El código está formado por tres archivos: deteccion.cpp, principal.cpp y cabeceras.h, que presentaremos a continuación.

B.4.1.- Archivo cabeceras.h

Este archivo contiene la declaración de todas las funciones que luego emplearemos en el resto del código.

```
////////////////////////////////////
//  ARCHIVO CABECERAS.H  //
////////////////////////////////////
#include "cvaux.h"
#include "cv.h"
#include "highgui.h"
#include <stdio.h>
#include <math.h>
#include "BlobResult.h"
```

```
#include "BlobExtraction.h"
#include "blob.h"

#define CONTADOR_MAX 2000
#define BARRA_HORIZ 10
#define BARRA_VERTIC 30

//DECLARACIÓN DE LAS FUNCIONES
void DeteccionMov(CvBGStatModel* bg_model, IplImage* original, IplImage* objeto, IplImage*
fondo, IplImage* sinRuido);
void PintaContornos(IplImage* gris, IplImage* resultado);
int tracking_flechas(IplImage* imagenGris,IplImage* actual, IplImage* anterior, IplImage*
piramidal, IplImage* ant_piramidal,char* estado, int inicio_seg,int cont, int salto, CvPoint2D32f*
puntos0, CvPoint2D32f* puntos1, IplImage* flujo_mov);
```

B.4.2.- Archivo principal.cpp

Este archivo contiene la definición de la función main(), a partir de la cual llamaremos al resto de funciones que hacen todas las tareas de segmentación, dibujado de contornos y cálculo del flujo de movimiento.

```
#include "cvaux.h"
#include "cv.h"
#include "highgui.h"
#include <stdio.h>
#include <math.h>
#include "BlobResult.h"
#include "BlobExtraction.h"
#include "blob.h"
#include "cabeceras.h"
#include <windows.h>

void main(int argc, char** argv)
{
        //Declaramos todas las imágenes que vamos a utilizar a lo largo del código
        IplImage*  original = NULL;
        IplImage*  fondo = NULL;
        IplImage*  objeto = NULL;
        IplImage*  mascara = NULL;
        IplImage*  imgcontorno = NULL;
        IplImage*  gris= NULL;
        IplImage*  flechas=NULL;
        IplImage*  anterior=NULL;
        IplImage*  actual=NULL;
        IplImage*  piramidal=NULL;
        IplImage*  ant_piramidal=NULL;

        //Variable para realizar la captura de imágenes de la cámara
        CvCapture*     flujocamara = NULL;

        int trama;
        //La variable posicion_barra servirá para saber qué ventanas están activas y por lo tanto
        deben mostrarse
        int posicion_barra[4]={0,0,0,0};
        int contador=0;
        int inicio_seg=0;
```

```
int salto=10;
int i,j,k;
char* estado = 0;


//Nos servirá para realizar el tracking del individuo en movimiento.
CvPoint2D32f* puntos[2] = {0,0};

puntos[0] = (CvPoint2D32f*)cvAlloc(CONTADOR_MAX*sizeof(puntos[0][0]));
puntos[1] = (CvPoint2D32f*)cvAlloc(CONTADOR_MAX*sizeof(puntos[0][0]));


estado = (char*)cvAlloc(CONTADOR_MAX);

//Inicializamos flujocamara y comenzamos la captura de las imágenes con
//cvQueryFrame
flujocamara = cvCaptureFromCAM(-1);
original = cvQueryFrame(flujocamara);

if(!original)
{
        printf("Revise por favor la cámara. La conexión no es correcta \n");
        exit(0);
}

//PARA EL TRACKING:
//Para poder estudiar el movimiento de la imagen no podemos estudiar todos los
//puntos, lo que haremos será centrarnos en puntos de la imagen separados entre sí por
//una distancia o "salto" de 10 píxeles. Es decir, estamos inicializando la rejilla que vamos
//a utilizar para calcular el flujo óptico.

for(i = 0; i < original->height; i+=salto)
{
        for(j = 0; j < original->width; j+=salto)
        {
                puntos[0][contador] = cvPoint2D32f(j, i);
                puntos[1][contador] = cvPoint2D32f(j, i);
                contador++;
                // Contaremos el número total de puntos que vamos a considerar al
                //final.(Dependerá del tamaño de la imagen)
        }
}

//Inicialimos todas las imágenes que vamos a utilizar durante el código utilizando la
//imagen original como modelo para el tamaño.
fondo = cvCreateImage(cvSize(original->width,original->height),IPL_DEPTH_8U, 3);
objeto = cvCreateImage(cvSize(original->width,original->height),IPL_DEPTH_8U, 3);
mascara = cvCreateImage(cvSize(original->width,original->height),IPL_DEPTH_8U, 1);
imgcontorno = cvCreateImage(cvSize(original->width,original->height),IPL_DEPTH_8U, 3);
gris = cvCreateImage(cvSize(original->width,original->height),IPL_DEPTH_8U, 1);
flechas=cvCreateImage(cvSize(original->width,original->height),IPL_DEPTH_8U, 1);
anterior = cvCreateImage(cvSize(original->width,original->height),IPL_DEPTH_8U, 1);
actual = cvCreateImage(cvSize(original->width,original->height),IPL_DEPTH_8U, 1);
piramidal = cvCreateImage(cvSize(original->width,original->height),IPL_DEPTH_8U, 1);
ant_piramidal=cvCreateImage(cvSize(original->width,original->height),IPL_DEPTH_8U, 1);


fondo->origin = original->origin;
objeto->origin = original->origin;
mascara->origin = original->origin;
imgcontorno->origin = original->origin;
gris->origin = original->origin;
flechas->origin = original->origin;
```

```
anterior->origin = original->origin;
actual->origin = original->origin;
piramidal->origin = original->origin;
ant_piramidal->origin = original->origin;


//Creamos el pequeño interfaz gráfico de la aplicación.
cvNamedWindow("Video original",1);
cvMoveWindow( "Video original", 0, 0 );

cvCreateTrackbar("FONDO","Video original",&posicion_barra[0],1,NULL);
cvCreateTrackbar("OBJETO","Video original",&posicion_barra[1],1,NULL);
cvCreateTrackbar("CONTORNOS","Video original",&posicion_barra[2],1,NULL);
cvCreateTrackbar("FLUJO 1","Video original",&posicion_barra[3],1,NULL);

//Creamos el modelo de Fondo/Objeto utilizando las funciones de la librería
CvBGStatModel* bg_model = cvCreateGaussianBGModel( original );

inicio_seg=contador;

//Este bucle se ejecutará para capturar una nueva imagen de la secuencia de video y
//procesarla
for( trama = 1;original; original = cvQueryFrame(flujocamara), trama++ )
{
        //Con esta función en cada imagen separaremos el fondo del objeto en
        //movimiento
        DeteccionMov(bg_model, original, objeto, fondo, mascara);

        //Determinamos si mostramos o no los resultados en función de la posición de las
        //barras
        if(posicion_barra[0]==1)  //Barra para el fondo
        {
                cvNamedWindow("Imagen del fondo", 1);
                cvMoveWindow( "Imagen del fondo",(original->width)+BARRA_HORIZ, 0 );
                cvShowImage("Imagen del fondo", fondo);
        }
        else
        {
                cvDestroyWindow("Imagen del fondo");
        }

        if(posicion_barra[1]==1)  //Barra para el objeto en movimiento
        {
                cvNamedWindow("Imagen del objeto", 1);
                cvMoveWindow("Imagen del objeto", 2*(original->width+BARRA_HORIZ),
                0);
                cvShowImage("Imagen del objeto", objeto);
        }
        else
        {
                cvDestroyWindow("Imagen del objeto");
        }

        cvCopy(original,imgcontorno,0);

        if(posicion_barra[2]==1)
        //Barra para visualizar el contorno del objeto en movimiento
        {
                cvNamedWindow("Contorno",1);
                cvMoveWindow("Contorno",(original->width)+BARRA_HORIZ,     original->
                height +BARRA_VERTIC);
```

```
                    //Con esta función pintamos en una copia de la imagen original
                    //capturada los contornos de la imagen objeto después de la
                    //segmentación
                    PintaContornos(mascara, imgcontorno);
            }
            else
            {
                    cvDestroyWindow("Contorno");
            }

            cvCvtColor( original, gris, CV_BGR2GRAY );

            //Con esta función calculamos el flujo óptico de la imagen original capturada, y
            //así podremos saber hacia dónde se está moviendo el objeto y la velocidad
            //que lleva.
            contador=tracking_flechas(gris, actual, anterior, piramidal, ant_piramidal,
            estado, inicio_seg, contador, salto, puntos[0], puntos[1], flechas);

            if(posicion_barra[3]==1) //Barra para visualizar el flujo óptico de la imagen
            {
                    cvNamedWindow("Flujo del movimiento",1);
                    cvMoveWindow("Flujo del movimiento", 2*(original->width +
                    BARRA_HORIZ), (original->height)+BARRA_VERTIC);
                    cvShowImage("Flujo del movimiento", flechas);
            }
            else
            {
                    cvDestroyWindow("Flujo del movimiento");
            }

            //Si pulsamos la tecla "ESC" finalizamos la aplicación
            k = cvWaitKey(5);
            if( k == 27 )
                    break;
    }

    //Liberamos todos los recursos reservados, ventanas, imágenes y el flujo de captura de la
    //cámara
    cvReleaseBGStatModel( &bg_model );
    cvReleaseCapture(&flujocamara);
    cvReleaseImage(&fondo);
    cvReleaseImage(&objeto);
    cvReleaseImage(&original);
    cvReleaseImage(&mascara);
    cvReleaseImage(&gris);
    cvReleaseImage(&flechas);
    cvReleaseImage(&imgcontorno);
    cvReleaseImage(&anterior);
    cvReleaseImage(&piramidal);
    cvReleaseImage(&ant_piramidal);
    cvDestroyAllWindows();

}
```

B.4.3.- Archivo funciones.cpp

Este archivo contiene todas las funciones que se llaman desde main() y que se encargan de realizar cada una de las tareas básicas de la aplicación: segmentación (separación fondo-objeto), reconocimiento y dibujado de los contornos del objeto en movimiento y por último cálculo del flujo óptico.

```
////////////////////////////////////
//   ARCHIVO FUNCIONES.CPP   //
////////////////////////////////////

#include "cvaux.h"
#include "cv.h"
#include "highgui.h"
#include <time.h>
#include <stdio.h>
#include <ctype.h>
#include <math.h>
#include "BlobResult.h"
#include "BlobExtraction.h"
#include "blob.h"
#include "cabeceras.h"

//Función para la detección de objetos en movimiento. Detectaremos con estas funciones si el
//individuo se está moviendo, separando en dos imágenes distintas el fondo de la imagen
//(objetos que no se mueven) y el objeto de interés (elemento que se está moviendo)

void DeteccionMov( CvBGStatModel* bg_model, IplImage* original, IplImage* objeto, IplImage*
fondo, IplImage* sinRuido)

{
        int i,j;

        //Como estamos trabajando con imágenes en color, para aplicar el filtrado del ruido
        //posterior necesitaremos aplicarlo a cada plano por separado, trabajando así con tres
        //imágenes monocromáticas correspondientes al plano R, al plano G y al plano B.

        IplImage*       temp1 = NULL;
        IplImage*       temp2 = NULL;
        IplImage*       temp3 = NULL;

        temp1 = cvCreateImage(cvSize(original->width,original->height),IPL_DEPTH_8U, 1);
        temp2 = cvCreateImage(cvSize(original->width,original->height),IPL_DEPTH_8U, 1);
        temp3 = cvCreateImage(cvSize(original->width,original->height),IPL_DEPTH_8U, 1);

        temp1->origin = original->origin;
        temp2->origin = original->origin;
        temp3->origin = original->origin;

        cvShowImage("Video original", original);

        //Con esta función actualizamos el modelo gaussiano para la separación fondo-objeto
        cvUpdateBGStatModel( original, bg_model );
        cvCopy(bg_model->background, fondo,0);
        cvCopy(bg_model->foreground, sinRuido,0);

        //El resultado que se obtiene en bg_model->foreground suele ser muy ruidoso,
        //presentando como objeto pequeñas agrupaciones de píxeles que han sufrido algún
        //tipo de modificación (p.ej. en la iluminación) y que en realidad pertenecen al fondo.
        //Tendremos que filtrar ese ruido.

        //CÓDIGO PARA ELIMINAR EL POSIBLE RUIDO DE LA IMAGEN
        //Ahora introducimos el código para separar la imagen en blobs.

        CBlobResult blobs;
        CBlobResult filtrado_blobs;
        CBlobGetArea evaluador;
        CBlob Blob;

        // Extraemos los blobs utilizando un threshold de valor 100
```

```
		blobs = CBlobResult( sinRuido, NULL, 100, true );
		evaluador=CBlobGetArea();

		//Vamos a filtrar los blobs de la imagen. Vamos a eliminar aquellos que son demasiado
		//pequeños (ruido)
		blobs.Filter(filtrado_blobs, B_INCLUDE, evaluador, B_LESS_OR_EQUAL, 500, 0);
		blobs.Filter(blobs, B_EXCLUDE, evaluador, B_LESS_OR_EQUAL, 500, 0);

		//Rellenamos de negro aquellos blobs que consideramos ruido
		for (i=0;i<filtrado_blobs.GetNumBlobs();++i)
		{
			Blob = filtrado_blobs.GetBlob(i);
			Blob.FillBlob(sinRuido, CV_RGB(0,0,0));
		}

		//Tenemos que tener en cuenta que el negro es el (0,0,0) en el plano RGB. Separamos la
		//imagen en sus tres planos
		cvSplit( original, temp1, temp2, temp3, NULL);

		//Pintamos de negro
		for(i = 0; i < original->height; i++)
		{
			for(j = 0; j < original->width; j++)
			{
				if(((sinRuido->imageData + i*original->width)[j])==0)
				{
					//Pertenece al fondo, pintamos de negro
					(temp1->imageData + i*original->width)[j] = 0;
					(temp2->imageData + i*original->width)[j] = 0;
					(temp3->imageData + i*original->width)[j] = 0;
				}

			}
		}

		//Volvemos a unir los tres planos RGB con el filtrado hecho
		cvMerge(temp1,temp2,temp3, NULL,objeto);

		//Liberamos los recursos
		cvReleaseImage(&temp1);
		cvReleaseImage(&temp2);
		cvReleaseImage(&temp3);

}

//Función encargada de encontrar los contornos de la imagen objeto y pintarlos sobre una
//réplica de la imagen original capturada.
void PintaContornos(IplImage* gris, IplImage* resultado)
{
		CvMemStorage* reserva = cvCreateMemStorage(0);

		//Secuencia donde vamos a introducir cada uno de los contornos de la imagen
		//pertenecientes al objeto en movimiento.
		CvSeq* contornos=0;

		cvFindContours(gris, reserva, &contornos, sizeof(CvContour),CV_RETR_CCOMP,
		CV_CHAIN_APPROX_SIMPLE,cvPoint(0,0));

		for( ; contornos != 0; contornos = contornos->h_next )
		{
			//Dibujamos cada contorno
			cvDrawContours(resultado, contornos, CV_RGB(255,0,0), CV_RGB(0,255,0),0, 1,
			8,cvPoint(0,0));
```

```
		}

		free(contornos);
		cvClearMemStorage(reserva);
}

//Función tracking_flechas: Encargada de calcular el flujo óptico dentro de una imagen en
//función de la imagen actual y un conjunto de imágenes anteriores.
int tracking_flechas(IplImage* imagenGris, IplImage* actual, IplImage* anterior, IplImage*
piramidal, IplImage* ant_piramidal, char * estado, int inicio_seg, int cont, int salto, CvPoint2D32f*
puntos0, CvPoint2D32f* puntos1, IplImage* flujo_mov, bool* mov)
{
		int i,j;
		int bandera=0;
		CvPoint p1, p2, p3, p4, p5;

		//Suavizamos la imagen para mejorar la precisión
		cvSmooth(imagenGris, imagenGris, CV_GAUSSIAN, 3, 0, 0);
		cvSmooth(imagenGris, imagenGris, CV_GAUSSIAN, 3, 0, 0);
		cvSmooth(imagenGris, imagenGris, CV_GAUSSIAN, 3, 0, 0);

		//La imagen que en el bucle anterior era la actual, ahora es la imagen anterior.
		cvCopy(actual, anterior, 0);
		cvCopy(imagenGris, actual, 0);

		//Actualizamos también las imágenes que vamos a utilizar para el método iterativo
		//piramidal LK
		cvCopy(piramidal, ant_piramidal, 0);

		for(i = inicio_seg; i < cont; i++)
		{
			puntos0[i] = puntos1[i];
		}

		//Calculamos el flujo del movimiento utilizando el método iterativo piramidal LK con el
		//parámetro level=0, para que se comporte como el algoritmo Lucas-Kanade general
		cvCalcOpticalFlowPyrLK(anterior, actual, ant_piramidal, piramidal, puntos0, puntos1,
		cont, cvSize(salto, salto), 3, estado, 0, cvTermCriteria(CV_TERMCRIT_ITER |
		CV_TERMCRIT_EPS, 20, 0.03), bandera);

		//Eliminamos el posible ruido que se produzca en la imagen.

		for(i = 0; i <inicio_seg; i++)
		{
			if(abs((int)(puntos0[i].x - puntos1[i].x)) > 50 | |
				abs((int)(puntos0[i].y - puntos1[i].y)) > 50)
			{
				puntos1[i] = puntos0[i];
			}
		}

		//Actualizamos el valor de las banderas

		bandera |= CV_LKFLOW_PYR_A_READY;

		//Refrescamos la imagen del flujo del movimiento que estamos representando

		for (i = 0; i < flujo_mov->height; i++)
		{
			for (j = 0; j < flujo_mov->width; j++)
			{
				(flujo_mov->imageData+flujo_mov->widthStep*i)[j] = (char)255;
				//Ponemos toda la imagen en blanco
```

```
                }
        }

        //Ahora dibujamos el flujo del movimiento. Lo dibujaremos con flechas. A mayor
        //velocidad del movimiento, mayor será el tamaño de la flecha.

        for (i = 0; i < inicio_seg; i++)
        {
                //Dibujaremos sólo velocidades intermedias, ni muy lentas ni muy rápidas.

                if((abs((int)(puntos0[i].x - puntos1[i].x)) >= 5 || abs((int)(puntos0[i].y -
                puntos1[i].y)) >= 5) &&(abs((int)(puntos0[i].x - puntos1[i].x)) <= 30 &&
                abs((int)(puntos0[i].y - puntos1[i].y)) <= 30))
                {
                        //Dibujamos la flecha
                        //Primero dibujamos el palo de la flecha
                        cvLine(flujo_mov, cvPointFrom32f(puntos0[i]),cvPointFrom32f(puntos1[i]),
                        cvScalar(0, 0, 0,0), 1, 8, 0);

                        //Ahora dibujamos la cabeza de la flecha

                        p1 = cvPoint((int)(cvPointFrom32f(puntos0[i]).x + 0.7*(cvPointFrom32f
                        (puntos1[i]).x - cvPointFrom32f(puntos0[i]).x)), (int)(cvPointFrom32f
                        (puntos0[i]).y + 0.7*(cvPointFrom32f(puntos1[i]).y -cvPointFrom32f(
                        puntos0[i]).y)));

                        p2 = cvPoint((int)(cvPointFrom32f(puntos0[i]).x +0.7*(cvPointFrom32f
                        (puntos1[i]).x - cvPointFrom32f(puntos0[i]).x)),(int)cvPointFrom32f
                        (puntos1[i]).y);

                        p3 = cvPoint((int)cvPointFrom32f(puntos1[i]).x,(int)(cvPointFrom32f(
                        puntos0[i]).y + 0.7*(cvPointFrom32f(puntos1[i]).y - cvPointFrom32f(
                        puntos0[i]).y)));

                        p4 = cvPoint((int)(p1.x + 0.5*(p2.x-p1.x)),(int)(p1.y + 0.5*(p2.y-p1.y)));

                        p5 = cvPoint((int)(p1.x + 0.5*(p3.x-p1.x)),(int)(p1.y + 0.5*(p3.y-p1.y)));

                        cvLine(flujo_mov, cvPointFrom32f(puntos1[i]),p4, cvScalar(0, 0, 0,0), 1, 8,
                        0);
                        cvLine(flujo_mov, cvPointFrom32f(puntos1[i]),p5, cvScalar(0, 0, 0,0), 1, 8,
                        0);

                }

                else
                {
                        //Dibujamos un punto
                        cvLine(flujo_mov, cvPointFrom32f(puntos0[i]),cvPointFrom32f(puntos0[i]),
                        cvScalar(0, 0, 0,0), 1, 8, 0);

                }

        }
        return(cont);

}
```