4.- IMPLEMENTACIÓN

4.1.- Estructura:

En este capítulo nos vamos a centrar en el sistema que finalmente se ha implementado para el sistema de detección de caídas.

Para ello mostraremos en primer lugar los diagramas de flujo que explican su funcionamiento. A continuación detallaremos la estructura interna del sistema, explicando las tareas principales en las que se divide y una descripción de las mismas, y una vez que ya comprendamos el funcionamiento del sistema pasaremos a detallar todas las funciones y estructuras de datos que hemos empleado, tanto las de librería como las propias que hemos programado específicamente para nuestra aplicación.

En los últimos apartados mostraremos un ejemplo práctico del funcionamiento del sistema, con capturas de pantalla incluidas, y finalmente hablaremos de las pruebas que hemos realizado para testear el sistema y los resultados obtenidos.

4.2.- Diseño

4.2.1.- Diagramas de flujo

Para una mejor comprensión del funcionamiento de esta aplicación presentaremos primero sus diagramas de flujo.

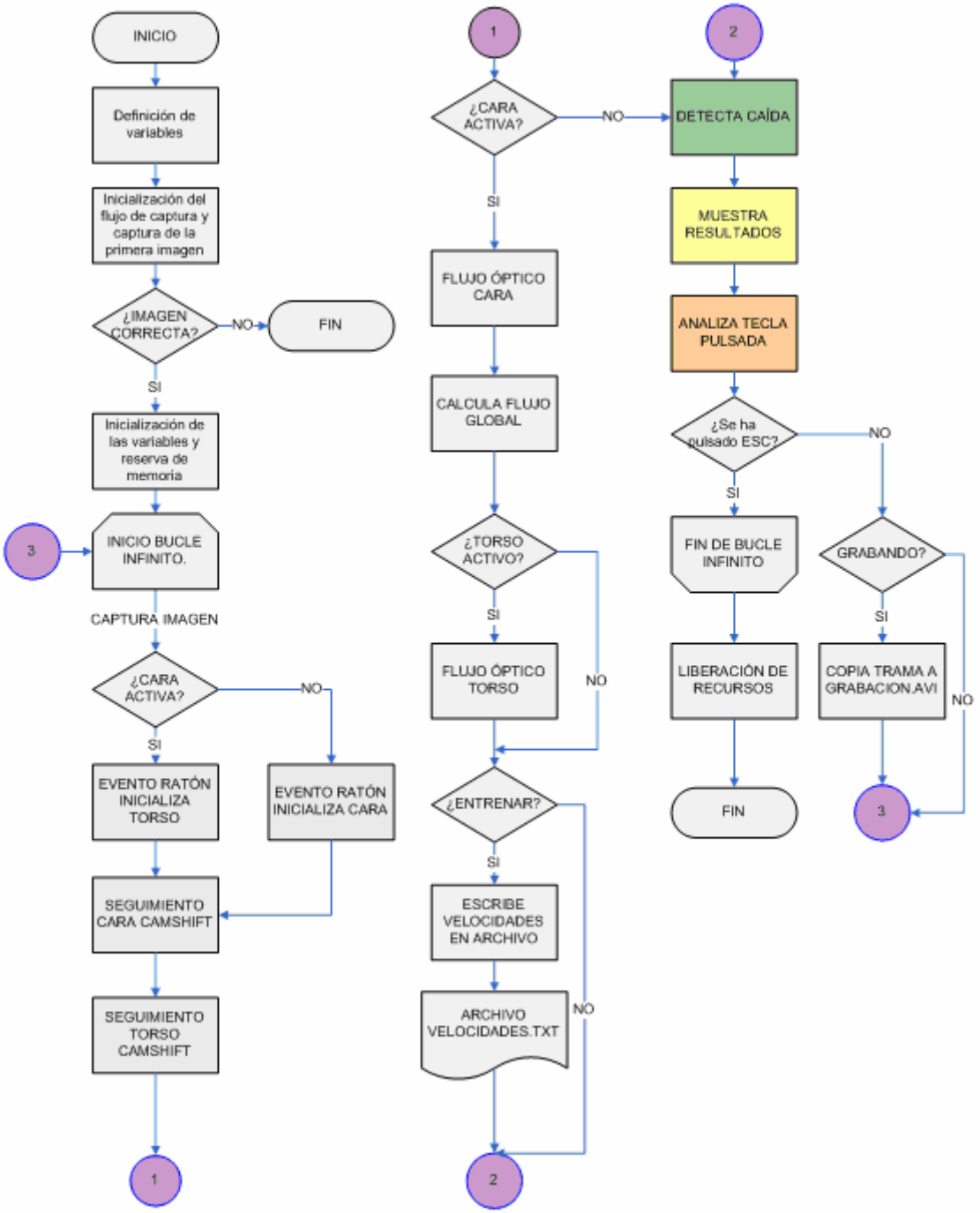


Figura 4.1.- Diagrama de flujo principal de la aplicación

Como podemos ver en el diagrama de la figura anterior, existen tres bloques de colores distintos: DETECTA CAÍDAS, MUESTRA RESULTADOS y ANALIZA TECLA PULSADA. Esos bloques se han desglosado en sus correspondientes diagramas de flujo que mostraremos a continuación y que nos ayudarán a comprender con mayor facilidad el código (anexado al final de la memoria).

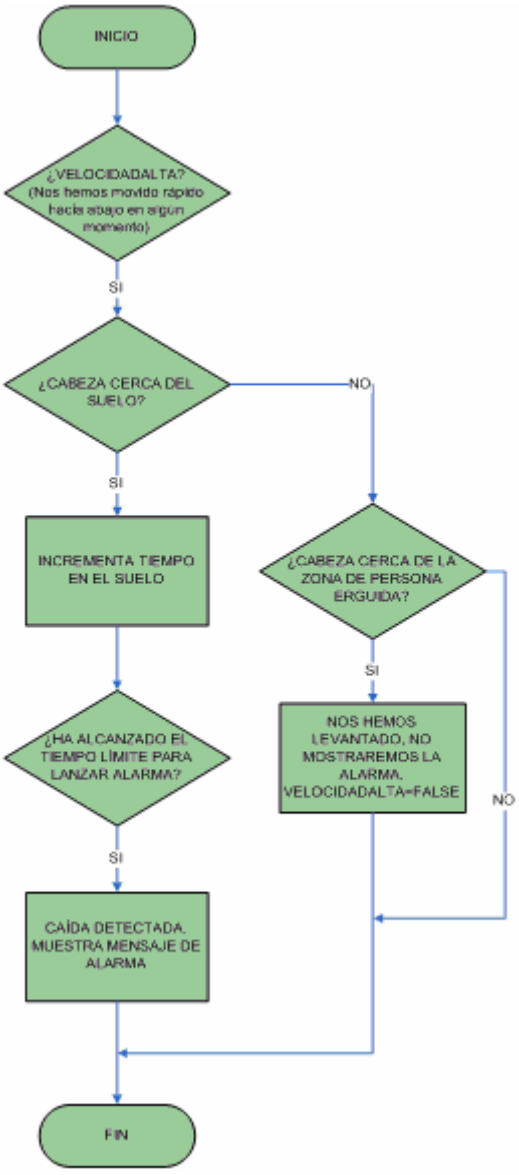


Figura 4.2.- Diagrama de flujo para la detección de caídas

Los dos siguientes diagramas son los encargados de mostrar los resultados o de gestionar qué tecla se ha pulsado.

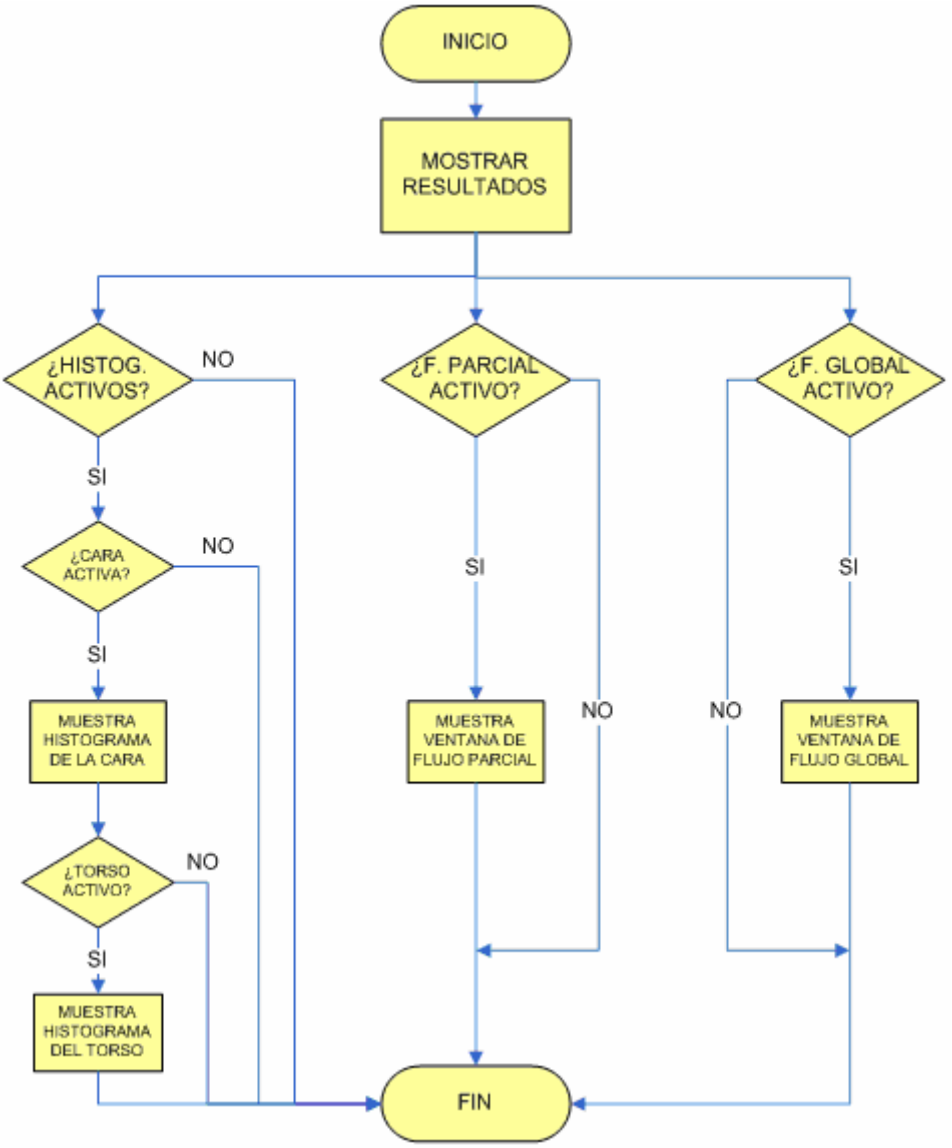


Figura 4.3.- Diagrama de flujo para el bloque MOSTRAR RESULTADOS

La forma de activar o desactivar la ventana del flujo global, flujo parcial e histogramas es mediante la pulsación de las teclas ‘g’, ‘f’ y ‘h’ respectivamente. Por defecto las ventanas del histograma están activas desde el inicio del programa, mientras que las otras dos hay que activarlas para que sean visibles.

Por último, el diagrama ANALIZA TECLA PULSADA muestra las actividades que se gestionan desde el menú de inicio de la aplicación, pudiendo activar o desactivar el modo entrenamiento, indicar qué resultados van a ser visibles o incluso grabar un video para poder estudiar después el funcionamiento de la aplicación.

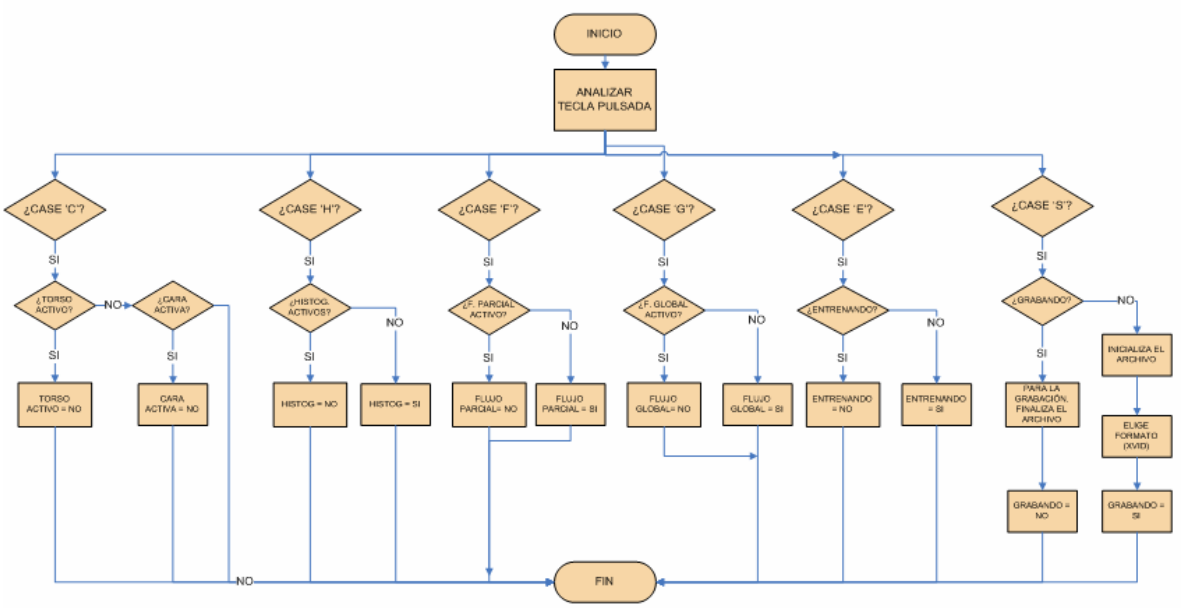


Figura 4.4.- Diagrama de flujo de ANALIZA TECLA PULSADA

Con éste quedan concluidos los diagramas de flujo, pasando a continuación a detallar la estructura interna de la aplicación, su funcionamiento, funciones, etc.

4.2.2.- Estructura interna

Como hemos podido ver en el primero de los diagramas de flujo (figura 4.1), el programa realiza varias tareas importantes durante su ejecución: Control eventos del ratón, seguimiento mediante Camshift, cálculo del flujo óptico (parcial y global), entrenamiento, detección de caídas, muestra de resultados, gestión de menú (análisis de teclas pulsadas) y grabación de vídeos. 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 en el anexo C junto con otros archivos que hemos utilizado para la implementación del sistema.

4.2.2.1.- CONSIDERACIONES GENERALES

La estructura más importante de la aplicación es un bucle infinito encargado de realizar la captura de cada imagen dentro de la secuencia de video. 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.

En cada iteración del bucle capturaremos una sola imagen o frame y la someteremos a las funciones de seguimiento, cálculo de flujo óptico (sentido y velocidad del movimiento) y detección de caídas. Al procesarse sólo una imagen en cada iteración, 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 hemos llegado a conseguir una velocidad de procesado de 25 frames por segundo aproximadamente cuando la captura de la cámara se realiza a 30 frames por segundo.

Una de las optimizaciones que hemos realizado en el código para aumentar esta velocidad de procesado es la utilización de unas estructuras de datos específicas para cada tarea. Estas estructuras son las de tipo **Track**, pensadas para las tareas de seguimiento y las de tipo **OptFlow** para las tareas de cálculo de flujo óptico. Ambas pueden verse en el archivo cabeceras.h del anexo C.

4.2.2.2.- BLOQUE DE CONTROL DE EVENTOS DEL RATÓN

En este bloque lo que se hace es monitorizar cuándo se ha producido un evento del ratón (pulsar botón izquierdo, arrastrar, levantar botón derecho, etc) y ejecutar la función **on_mouse** cuando esto ocurra.

La función on_mouse es una función propia del sistema que hemos sobrescrito para que ejecute otro código distinto al que tiene por defecto. En nuestro caso, la hemos programado para que la primera vez que se seleccione un área determinado en la pantalla se asocie a la estructura Track* trackcabeza (para el seguimiento de la cabeza del individuo) y las siguientes veces que se seleccione un nuevo área se asocie a Track* tracktorso (para seguir el torso). Ambas estructuras se podrán resetear pulsando la tecla 'c' desde el teclado.

Cuando seleccionamos una determinada zona de la imagen lo que hacemos es calcular su histograma asociado, el cual utilizaremos posteriormente para realizar el seguimiento con el algoritmo Camshift.

4.2.2.3.- BLOQUE DE SEGUIMIENTO MEDIANTE CAMSHIFT

Este bloque es el encargado de realizar el seguimiento de los elementos de interés activos en la ventana principal. Para realizar esta tarea emplearemos la función **seguimiento()**, la cual se basa en el algoritmo Camshift detallado en el capítulo 2.

Este bloque, a grandes rasgos, lo que hace es comparar el histograma de la imagen capturada con la cámara con el histograma modelo de lo que se ha seleccionado con el ratón para encontrar la nueva posición del objeto que estamos siguiendo.

Para ello este bloque primero obtiene la representación en el espacio HSV de la imagen que se acaba de capturar con la cámara (que viene dada en el espacio RGB). De la imagen multicanal HSV que acabamos de obtener, nos quedamos sólo con el plano H (hue) y le calculamos lo que se conoce como imagen **backprojection**, consistente en una imagen en escala de grises donde los píxeles con un valor H parecido al del histograma modelo tienen valores altos (blancos) y los que se parecen poco tienen valores bajos (negro). Esta imagen *backprojection* es la que se pasa por parámetro a la función cvCamshift y es la que determina la nueva posición y orientación del elemento monitorizado.

Este bloque se ejecuta dos veces: una para la cabeza y otra para el torso.

4.2.2.4.- BLOQUE DE CÁLCULO DE FLUJO ÓPTICO

En este bloque vamos a calcular el flujo óptico de los objetos que estamos siguiendo, es decir, vamos a calcular la velocidad y el sentido del desplazamiento (en caso de que lo haya habido).

Para ello vamos a emplear la función **calcula_flujo()** cuya tarea básica consiste en comparar la posición de los píxeles de la imagen actual con la posición de éstos en la imagen anterior y ver la evolución que éstos han sufrido. El método que empleamos para calcular el flujo óptico es el basado en el algoritmo iterativo de Lucas-Kanade desarrollado en el capítulo 2.

De esta forma, con la aplicación de las funciones *seguimiento* y *calcula_flujo* ya tendremos un análisis completo del movimiento que ha realizado el sujeto monitorizado, pues la primera nos proporciona la posición y orientación y la segunda nos proporciona el sentido y velocidad.

Por otro lado, en la aplicación hemos querido diferenciar entre lo que nosotros hemos llamado **flujo parcial** y **flujo global**. El flujo parcial es aquel formado por todos los vectores de desplazamiento característicos de cada uno de los píxeles de la zona de interés, mientras que el flujo global es el flujo total del elemento calculado como la media de todos los vectores de desplazamiento.

El flujo global es muy importante en nuestra aplicación, puesto que de él obtendremos algunos parámetros que luego utilizaremos en otros dos bloques: el bloque de entrenamiento y el bloque de detección de caídas.

4.2.2.5.- BLOQUE DE ENTRENAMIENTO

Este bloque se ha utilizado en nuestro sistema para poder estudiar los resultados obtenidos durante las pruebas para la implementación posterior del bloque de detección de caídas.

La forma de activar o desactivar este bloque es mediante la pulsación de la letra 'e' por teclado. Desde el momento en el que se activa este bloque, comenzaremos a escribir los valores asociados al flujo global en el archivo **velocidades.txt** para el posterior análisis de los resultados. Estos parámetros son la magnitud, la componente X y la componente Y del flujo. Veamos a continuación alguno de estos archivos.

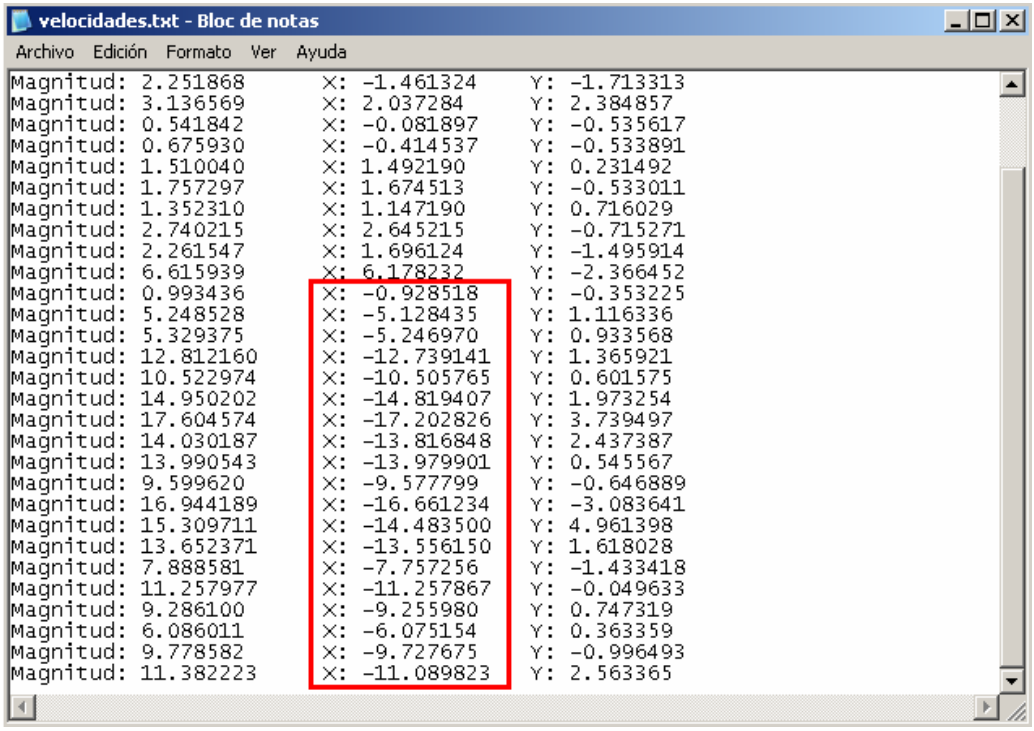


Figura 4.5.- Archivo velocidades.txt andando hacia la izquierda

La figura anterior nos muestra el archivo obtenido durante una prueba donde el individuo se movía hacia la izquierda. Como se puede ver, los valores incluidos dentro del recuadro rojo nos indican que, efectivamente, el individuo se ha movido en ese sentido. Veremos ahora uno de esos archivos cuando la prueba ha sido una caída.

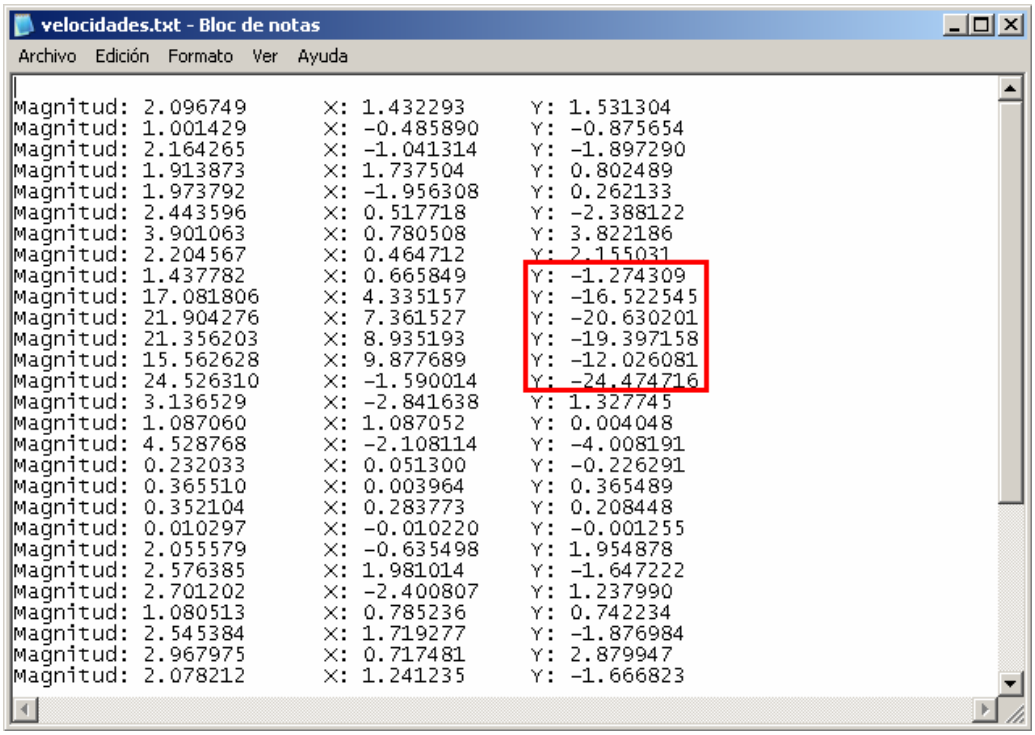


Figura 4.6.- Archivo velocidades.txt, prueba de caída

En este caso, la columna que tenemos que analizar es la correspondiente a la componente Y del flujo global. Con el análisis de varias pruebas con distintas caídas podremos determinar el valor de la componente Y que debemos tomar como límite empírico para considerar una situación de riesgo dentro del bloque de detección de caídas.

4.2.2.6.- BLOQUE DE DETECCIÓN DE CAÍDAS

En este bloque la tarea principal es detectar si se ha producido una situación de peligro. Su funcionamiento viene detallado en el diagrama de flujo de la figura 4.2.

Este bloque se ejecuta en cada iteración, por lo que estaremos constantemente controlando que no haya ocurrido ningún movimiento brusco. Si se detecta movimiento brusco y la cabeza llega cerca del suelo (caída) contaremos un determinado tiempo, de tal forma que si la persona se levanta

dentro de ese tiempo consideraremos que no hay que lanzar la alarma puesto que se ha levantado sin problemas. En caso contrario, si se llega al tiempo límite y la persona sigue en el suelo mostraremos el mensaje de alerta.

En base a los resultados obtenidos en los archivos de entrenamiento, se ha estipulado que la velocidad límite a partir de la cual se considerará que se ha producido una caída será 12 unidades (-12, pues el sentido es hacia abajo). El código asociado a la detección de caídas está dentro del archivo principal.cpp del anexo C.

4.2.2.7.- OTROS BLOQUES

En este apartado comentaremos brevemente el resto de bloques: muestra de resultados, gestión de menú y grabación de vídeo. Se han introducido en el capítulo los diagramas de flujo correspondientes al bloque de muestra de resultados y gestión de menú (figuras 4.3 y 4.4), por lo que en este apartado sólo añadiremos algunos comentarios.

En cuanto al bloque de gestión de menú, diremos que las tareas disponibles para el usuario son las siguientes:

- ESC: Salir del programa
- Tecla ‘c’: Finalizar el seguimiento del objeto seleccionado
- Tecla ‘h’: Gestiona la visualización del histograma o histogramas de los elementos que estamos siguiendo.
- Tecla ‘f’: Gestiona la visualización del flujo parcial asociado a los objetos en movimiento
- Tecla ‘g’: Gestiona la visualización del flujo global
- Tecla ‘e’: Gestiona el comienzo o finalización de la escritura en el archivo velocidades.txt asociado al bloque de entrenamiento.
- Tecla ‘s’: Se pulsa para comenzar o finalizar la grabación de un vídeo donde se muestra el contenido de la ventana principal de la aplicación.
- Por último, para comenzar con el seguimiento, seleccionar con el ratón los elementos que se van a monitorizar.

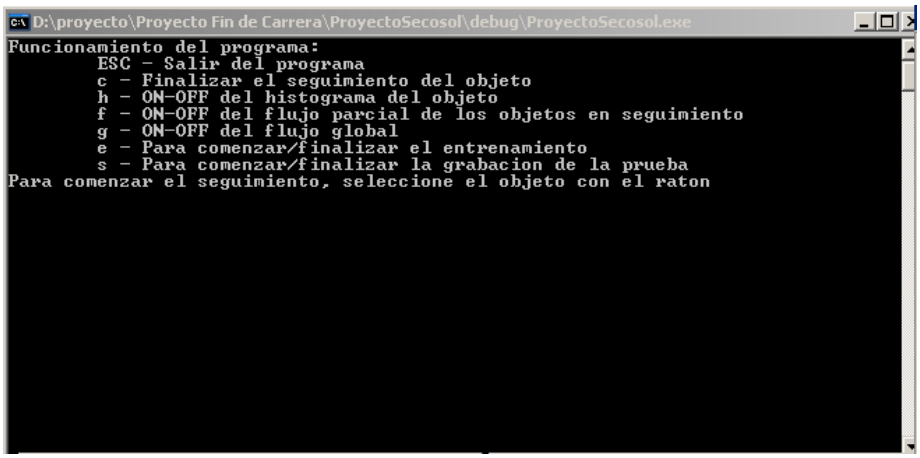


Figura 4.7.- Menú de la aplicación

En cuanto al bloque de grabación, diremos que en cuanto se selecciona la activación de esta opción mediante la tecla 's', aparecerá una ventana donde se nos dará a elegir el formato en el que queremos realizar la grabación. Además en la inicialización del archivo de vídeo indicaremos otra serie de parámetros como por ejemplo la velocidad o frames por segundo.

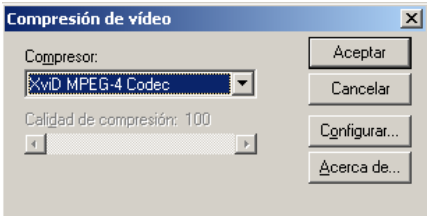


Figura 4.8.- Elección del formato del vídeo

Eligiendo el formato XviD nos aparecerá a continuación una pantalla indicándonos el estado de la grabación del video. En principio el tamaño del archivo de vídeo no está limitado.

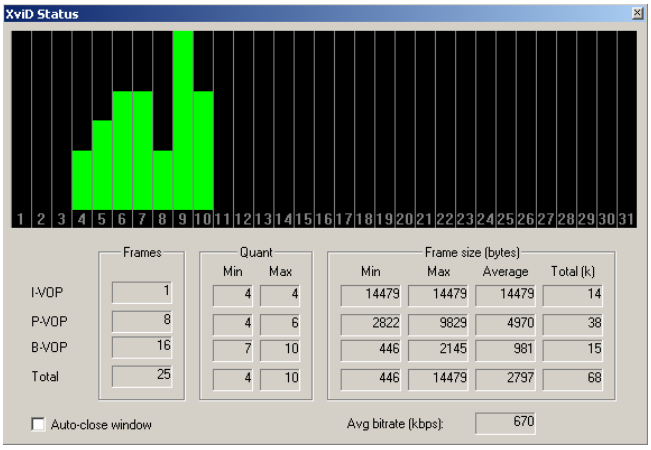


Figura 4.9.- Estado de grabación en formato XviD

4.2.3.- 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).

4.2.3.1.- FUNCIONES DE LIBRERÍA C++

Este apartado engloba a todas aquellas funciones estándar en el uso de C y C++ pertenecientes al paquete stdio.h o math.h.

Del paquete stdio.h hemos empleado la función **printf**("Cadena"), encargada de mostrar por consola (salida por defecto) la cadena de caracteres que se le indique como parámetro, la función **getchar**(), encargada de parar el flujo del programa hasta que se introduzca un carácter con el teclado (entrada por defecto) y las funciones utilizadas para la gestión de ficheros que son: **fopen**(archivo), que inicializa un archivo de texto, **fprintf**(archivo, "Cadena"), que escribe la cadena en el archivo especificado y **fclose**(archivo) que finaliza y cierra el archivo de texto.

Del paquete math.h hemos utilizado la función **abs**(expresion), que calcula el valor absoluto de la expresión que se le pasa por parámetro y **sqrt**(expresion) que calcula la raíz cuadrada.

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.

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.

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.

cvZero

- Declaración: void cvZero(CvArr* arr);
- Descripción: Función que limpia la matriz, es decir, que pone a cero a todos sus elementos.
- Parámetros:
  - arr: Matriz que queremos limpiar.

cvAnd

- Declaración: void cvAnd(const CvArr* src1, const CvArr* src2, CvArr* dst, const CvArr* mask=NULL);
- Descripción: Función que realiza la operación de bit AND elemento a elemento entre dos matrices. Todas las matrices involucradas deben ser del mismo tipo y del mismo tamaño.
- Parámetros:
  - src1: Primera matriz a la que vamos a aplicar la operación.
  - src2: Segunda matriz involucrada en la operación
  - dst: Matriz donde se va a almacenar el resultado.
  - mask: En caso de que no sea NULL, esta matriz indicará a qué elementos de la matriz aplicaremos la operación

cvXorS

- Declaración: void cvXorS(const CvArr* src, CvScalar value, CvArr* dst, const CvArr* mask=NULL);
- Descripción: Función que realiza la operación de bit XOR (OR exclusivo) elemento a elemento entre una matriz y un valor escalar.
- Parámetros:
  - src: Matriz origen a la que vamos a aplicar la operación.
  - value: Valor escalar utilizado en la operación.
  - dst: Matriz donde se va a almacenar el resultado.
  - mask: En caso de que no sea NULL, esta matriz indicará a qué elementos de la matriz aplicaremos la operación

cvInRangeS

- Declaración: void cvInRangeS(const CvArr* src, CvScalar lower, CvScalar upper, CvArr* dst);
- Descripción: Función que controla que el valor de cada elemento perteneciente a una matriz permanezca entre dos valores escalares.
- Parámetros:
  - src: Matriz a la que controlaremos el valor de sus elementos.
  - lower: Límite inferior permitido para los elementos de la matriz.
  - upper: Límite superior permitido para los elementos de la matriz.
  - dst: Matriz donde se va a almacenar el resultado.

cvSetImageROI

- Declaración: void cvSetImageROI(IplImage* image, CvRect rect);

- Descripción: Función que asigna el ROI (región de interés) de una imagen. El ROI vendrá dado por un rectángulo que se pasa como parámetro.
- Parámetros:
  - image: Cabecera de la imagen a la que queremos asignar el ROI.
  - rect: Rectángulo utilizado para asignar el ROI en la imagen.

cvResetImageROI

- Declaración: void cvResetImageROI(IplImage* image);
- Descripción: Función que libera el ROI (región de interés) de una imagen. Al liberar el ROI éste se queda asignado a la imagen completa.
- Parámetros:
  - image: Cabecera de la imagen de la que queremos liberar el ROI.

cvRound

- Declaración: int cvRound(double value);
- Descripción: Función que convierte un número representado en coma flotante al número entero más cercano a su argumento. Es la operación de redondeo.
- Parámetros:
  - value: Valor en coma flotante que queremos redondear.

cvFloor

- Declaración: int cvFloor(double value);
- Descripción: Función que convierte un número representado en coma flotante al número entero sin decimales (al mayor entero que no supere al argumento).
- Parámetros:
  - value: Valor en coma flotante al que aplicamos la transformación.

cvRectangle

- Declaración: void cvRectangle(CvArr* img, CvPoint pt1, CvPoint pt2, CvScalar color, int thickness=1, int line_type=8, 0);
- Descripción: Función que dibuja un rectángulo, relleno o no, de un determinado color.
- Parámetros:
  - img: Imagen donde vamos a pintar el rectángulo.
  - pt1: Uno de los vértices del rectángulo

- pt2: Vértice opuesto al anterior del rectángulo
- color: Color del rectángulo que vamos a pintar, en caso de que la imagen sea en escala de grises este valor indica el brillo
- thickness: Grosor del rectángulo que vamos a pintar. Si el valor es negativo como CV_FILLED entonces el rectángulo se rellena por dentro también.
- line_type: Tipo de línea (ver la descripción de cvLine).

cvConvertScale

- Declaración: void cvConvertScale(const CvArr* src, CvArr* dst, double scale=1, double shift=0);
- Descripción: Función que transforma una matriz en otra aplicándole una transformación lineal. Si la matriz es una imagen multicanal cada canal es procesado independientemente.
- Parámetros:
  - src: Matriz a la que queremos aplicar la transformación.
  - dst: Matriz donde vamos a almacenar el resultado de la transformación.
  - scale: Factor de escala en la transformació (se aplica elemento a elemento).
  - shift: Valor que se añade a cada elemento después de haberle aplicado la escala estipulada por scale.

cvEllipse

- Declaración: void cvEllipse(CvArr* img, CvPoint center, CvSize axes, double angle, double start_angle, double end_angle, CvScalar color, int thickness = 1, int line_type=8, 0);
- Descripción: Esta función pinta una elipse con un determinado grosor y o rellena un arco de elipse.
- Parámetros:
  - img: Imagen donde vamos a pintar la elipse
  - center: Centro de la elipse que vamos a dibujar.
  - axes: Longitud de los ejes de la elipse
  - angle: Ángulo de rotación de la elipse.
  - start_angle: Ángulo de comienzo del arco elíptico.
  - end_angle: Ángulo donde finaliza el arco elíptico. Todos los ángulos vienen dados en grados.
  - color: Color con el que vamos a pintar la elipse o arco de elipse.
  - thickness: Grosor de la línea exterior de la elipse. Si el valor es negativo la elipse estará rellena y no sólo pintada.
  - line_type: Tipo de línea que delimita la elipse. Ver descripción de la función cvLine.

cvFlip

- Declaración: void cvFlip(CvArr* src, CvArr* dst=NULL, int flip_mode);
- Descripción: Función que voltea una matriz o imagen verticalmente, horizontalmente o respecto a los dos ejes.
- Parámetros:
  - src: Matriz a la que queremos realizar la transformación, el giro.
  - dst: Matriz de destino donde almacenaremos el resultado. Si este parámetro vale NULL entonces el resultado se almacena sobre la misma imagen fuente.
  - flip_mode: Especifica qué tipo de giro se va a realizar. Si su valor es 0, el giro se realiza respecto al eje X, si su valor es positivo el giro será respecto el eje Y y si es negativo, respecto ambos ejes.

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.

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.

cvCreateHist

- Declaración: CvHistogram* cvCreateHist(int dims, int*sizes, int type, flota** ranges=NULL, int uniform=1);
- Descripción: Función que crea una variable del tipo cvHistograma (estructura "histograma").
- Parámetros:
  - dims: Numero de dimensiones o secciones en las que se divide el histograma.
  - sizes: Matriz donde se indica el valor asociado a cada sección del histograma.
  - type: Formato de representación de los datos del histograma. Como los datos siempre se organizan en matrices multidimensionales, este parámetro indica qué tipo de matriz almacena los datos: cvMatND o cvSparseMat.
  - ranges: Matriz donde se indican los rangos de valores válidos para cada sección o "bin"del histograma.
  - uniform: Es una bandera.

cvCalcHist

- Declaración: void cvCalcHist(IplImage** image, CvHistogram*hist, int accumulate=0, const CvArr* mask=NULL);
- Descripción: Calcula el histograma de una imagen o imágenes.
- Parámetros:
  - image: Imagen fuente de la que queremos calcular el histograma (hay que pasar un puntero a puntero).
  - hist: Puntero al histograma
  - accumulate: Bandera de "acumulación". Si esta bandera está activa entonces el histograma no se limpia al principio. De esa forma con esta función el usuario es capaz de procesar un único histograma proveniente de muchas imágenes o de actualizar sólo una parte del histograma durante la ejecución del programa.
  - mask: Esta matriz indica, si es distinta de NULL, qué píxeles de la imagen original consideraremos para calcular el histograma.

cvGetMinMaxHistValue

- Declaración: void cvGetMinMaxHistValue(const CvHistogram* hist, float* min_value, float* max_value, int* min_idx=NULL, int* max_idx=NULL);
- Descripción: Esta función encuentra el máximo y el mínimo de los valores de todos los sectores en los que se divide el histograma. Además también encuentra la posición de estos máximos y mínimos en el histograma.
- Parámetros:
  - hist: Histograma que vamos a analizar para encontrar los máximos y mínimos.
  - min_value: Puntero al mínimo valor del histograma
  - max_value: Puntero al máximo valor del histograma
  - min_idx: Puntero a la matriz que indique las coordenadas asociadas al mínimo del histograma.
  - max_idx: Puntero a la matriz que indique las coordenadas asociadas al máximo del histograma.

cvCalcBackProject

- Declaración: void cvCalcBackProject(IplImage** image, CvArr* back_project, const CvHistogram* hist);
- Descripción: Esta función calcula la imagen conocida como *backprojection*, consistente en una imagen en escala de grises donde los valores de los píxeles se acercan al blanco cuando se parecen mucho a un histograma dado y a cero en caso de que no se parezcan.
- Parámetros:
  - img: Imagen o imágenes originales de las que queremos calcular la imagen backprojection.
  - back_project: Imagen destino donde se almacena la imagen backproyection extraida de la imagen original. Esta imagen debe ser del mismo tipo que las imágenes originales.
  - hist: Histograma modelo que vamos a utilizar para calcular la imagen backprojection.
  - **NOTA:** El verdadero funcionamiento de la función es el siguiente. Utilizando el histograma de muestra, en cada píxel de la imagen original se pondrá el valor del histograma asociado a su color (sector). En términos estadísticos, el valor de cada píxel de la imagen de salida muestra la probabilidad de que dicho píxel pertenezca a un objeto representado por el histograma modelo.

cvCamShift

- Declaración: int cvCamShift(const CvArr* prob_image, CvRect window, CvTermCriteria criteria, CvConnectedComp* comp, CvBox2D* box=NULL);
- Descripción: Esta función encuentra el centro, tamaño y orientación de un determinado objeto cuando éste se encuentra en movimiento. Es una función pensada para tareas de seguimiento.
- Parámetros:
  - prob_image: Imagen *backprojection* del histograma del objeto al que estamos siguiendo. (Ver la descripción de la función cvCalcBackProject).
  - window: Ventana de búsqueda inicial.
  - criteria: Criterio utilizado para determinar cuándo la búsqueda del objeto debe finalizar.
  - comp: Estructura resultante de esta operación que contiene la ventana o rectángulo donde se encuentra el objeto buscado (campo comp->rect) y la suma de todos los píxeles dentro de esta ventana (campo comp->area).
  - box: Caja o ventana donde se ha encontrado el objeto. Si no es NULL contiene el tamaño del objeto y su orientación.
  - int: Esta función devuelve el número de iteraciones que ha necesitado el algoritmo MeanShift para encontrar el objeto de interés.
  - **NOTA:** Esta función está basada en el artículo de Gary R. Bradsky comentado en el capítulo 2, el cual primero encuentra el centro del objeto buscado mediante el algoritmo MeanShift y posteriormente encuentra su tamaño y orientación.

B.- 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.

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.

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.

cvSetMouseCallBack

- Declaración: void cvSetMouseCallBack(const char* window_name, CvMouseCallback on_mouse, void* param=NULL);

- Descripción: Indica qué función se va a realizar cuando se produzca un evento de ratón. Los posibles eventos y banderas asociadas a esta función se muestran a continuación:

```
#define CV_EVENT_MOUSEMOVE      0
#define CV_EVENT_LBUTTONDOWN    1
#define CV_EVENT_RBUTTONDOWN    2
#define CV_EVENT_MBUTTONDOWN    3
#define CV_EVENT_LBUTTONUP      4
#define CV_EVENT_RBUTTONUP      5
#define CV_EVENT_MBUTTONUP      6
#define CV_EVENT_LBUTTONDBLCLK  7
#define CV_EVENT_RBUTTONDBLCLK  8
#define CV_EVENT_MBUTTONDBLCLK  9

#define CV_EVENT_FLAG_LBUTTON   1
#define CV_EVENT_FLAG_RBUTTON   2
#define CV_EVENT_FLAG_MBUTTON   4
#define CV_EVENT_FLAG_CTRLKEY   8
#define CV_EVENT_FLAG_SHIFTKEY  16
#define CV_EVENT_FLAG_ALTKEY    32
```

- Parámetros:
  - window_name: Nombre de la ventana sobre la que se producirá el evento de ratón.
  - on_mouse: Puntero a la función que será llamada cada vez que ocurra un evento de ratón en la ventana especificada. Esta función debe tener una declaración similar a la siguiente: `void Foo(int event, int x, int y, int flags, void* param)`, donde *event* es uno de los CV_EVENT_XXX que hemos indicado antes, *x* e *y* se corresponden con las coordenadas de la posición donde se ha producido el evento de ratón, *flags* es una combinación de CV_EVENT_FLAG y *param* es un parámetro definido por el usuario y que coincide con el parámetro que se le pasa a la función **cvSetMouseCallBack**.
  - param: Parámetro definido por el usuario que se le pasará a la función on_mouse.

cvCreateVideoWriter

- Declaración: `CvVideoWrite* cvCreateVideoWriter( const char* filename, int fourcc, double fps, CvSize frame_size, int is_color=1);`
- Descripción: Función que crea e inicializar una variable del tipo CvVideoWriter consistente en un "escritor" de archivos de video
- Parámetros:
  - filename: Nombre del archivo de vídeo de salida

- fourcc: Código de cuatro caracteres para indicar la codificación que se va a utilizar para comprimir las imágenes del video. Por ejemplo: CV_FOURCC('P','I','M','1') se corresponde con MPEG-1 y CV_FOURCC('M','J','P','G') es *motion-jpeg*. Si trabajamos con plataformas Win32 es posible pasar -1 como parámetro, consiguiendo que aparezca un diálogo desde donde se elegirá en tiempo real el tipo de compresión del vídeo.
- fps: Frames por segundo del flujo del vídeo de salida
- frame_size: Tamaño del vídeo expresado en frames o imágenes.
- is_color: Si este parámetro no vale 0 el codificador de vídeo considerará y codificará los frames como si fueran en color, en caso cntrario, trabajará sólo con imágenes en escala de grises. Esta opción sólo está disponible en Windows.
- CvVideoWriter* : Devuelve un puntero a una estructura donde se almacenarán los parámetros para la grabación

cvReleaseVideoWriter

- Declaración: void cvReleaseVideoWriter(CvVideoWriter** writer);
- Descripción: Esta función finaliza la escritura del archivo de vídeo y libera la estructura asociada al mismo.
- Parámetros:
  - writer: Puntero a una estructura del tipo CvVideoWriter donde se almacenan los datos del "escritor" de vídeo que se quiere liberar.

cvGetCaptureProperty

- Declaración: double cvGetCaptureProperty(CvCapture* capture, int property_id);
- Descripción: Esta función accede al valor de un parámetro determinado de la captura de vídeo
- Parámetros:
  - capture: Estructura asociada a la captura de vídeo.
  - property_id: Identificador de la propiedad o parámetro del que queremos obtener su valor. Puede ser uno de los siguientes: CV_CAP_PROP_POS_MSEC (posición actual en milisegundos de la captura de vídeo), CV_CAP_PROP_POS_AVI_RATIO (posición relativa del archivo de vídeo, si vale 1 es el inicio del archivo, si vale 0 es el final), CV_CAP_PROP_POS_FRAMES (índice correspondiente a la siguiente imagen o frame que se va a capturar), CV_CAP_PROP_FRAME_WIDTH (ancho de las imágenes de vídeo), CV_CAP_PROP_FRAME_HEIGHT (alto

de las imágenes de vídeo), CV_CAP_PROP_FPS (velocidad en frames por segundo durante la captura), CV_CAP_PROP_FOURCC (código de cuatro caracteres que indica la compresión utilizada) y CV_CAP_PROP_FRAME_COUNT (número total de imágenes en el vídeo capturado).

- o double: El valor de retorno se corresponde con el valor del parámetro indicado en la variable "property_id".

cvWriteFrame

- Declaración: int cvWriteFrame(CvVideoWriter* writer, const IplImage* image);
- Descripción: Esta función añade un nuevo frame o imagen al archivo de video que se está creando.
- Parámetros:
  - o writer: Estructura asociada al "escritor" del archivo de vídeo.
  - o image: Nueva imagen que vamos a incluir en el archivo

4.2.3.3.- FUNCIONES PROPIAS

En este último apartado incluiremos las funciones de programación propia que hemos utilizado para el programa de detección de caídas

CvCreateTrack

- Declaración: Track* CvCreateTrack(CvCapture* flujo);
- Descripción: Esta función crea e inicializa una estructura del tipo Track* especializada en el seguimiento de objetos.
- Parámetros:
  - o flujo: Flujo de imágenes capturadas de la cámara web y cuya primera imagen utilizaremos para inicializar todas las variables del tipo IplImage existentes en la estructura.
  - o Track*: Devuelve una estructura del tipo Track* con todas las variables que la componen inicializadas.

CvCreateOptFlow

- Declaración: OptFlow* CvCreateOptFlow(CvCapture* flujo);
- Descripción: Función que crea e inicializa una estructura del tipo OptFlow* especializada en el cálculo del flujo óptico de objetos en movimiento.
- Parámetros:

- flujo: Flujo de imágenes capturadas de la cámara web y cuya primera imagen utilizaremos para inicializar todas las variables del tipo IplImage existentes en la estructura.
- OptFlow*: Devuelve una estructura del tipo OptFlow* con todas las variables que la componen inicializadas.

Calcula_flujo

- Declaración: void calcula_flujo(OptFlow* paramflow, IplImage* mascara, int entrenar);
- 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:
  - paramflow: Estructura del tipo OptFlow* donde se incluyen todos los parámetros necesarios para realizar la tarea del cálculo de flujo óptico (ver capítulo 2 - Introducción teórica). Esta variable se crea e inicializa con la función CvCreateOptFlow.
  - mascara: Imagen o matriz que indica qué píxeles deben ser computados para el cálculo del flujo óptico. Los elementos de la matriz que valgan distinto de cero serán procesados.
  - entrenar: Variable bandera que nos indica si estamos en mitad de una prueba de entrenamiento. En caso de que estemos durante un entrenamiento tendremos que escribir los resultados del flujo global del movimiento dentro del archivo **velocidades.txt** para su posterior análisis (magnitud, componente X y componente Y).

Seguimiento

- Declaración: void seguimiento(Track* paramtrack);
- Descripción: Es la función encargada de realizar el seguimiento de un objeto previamente seleccionado con el ratón en la ventana principal de la aplicación. Para realizar el seguimiento del mismo esta función se basa en el método diseñado por Gary R. Bradsky e implementado en la librería OpenCV mediante la función cvCamshift entre otras.
- Parámetros:
  - paramtrack: Estructura del tipo Track* donde se incluyen todos los parámetros necesarios para realizar la tarea del seguimiento (ver capítulo 2 - Introducción teórica). Esta estructura se inicializa con la función CvCreateTrack.

hsv2rgb

- Declaración: CvScalar hsv2rgb(float hue);
- Descripción: Esta función pasa del espacio HSV al espacio RGB a partir del valor Hue (tono) exclusivamente. Con este tipo de funciones podremos pasar de una representación monocanal basada en el plano Hue a una representación multicanal RGB si la aplicamos a cada uno de los píxeles de la imagen.
- Parámetros:
  - hue: Valor hue de un determinado píxel, el cual queremos que sea representado en el espacio RGB

on_mouse

- Declaración: void on_mouse(int event, int x, int y, int flags, void* param);
- Descripción: Función que se va a ejecutar cuando ocurra algún evento de ratón. Es una función propia del sistema que nosotros hemos sobrescrito y así hemos conseguido que haga otras tareas específicas de la aplicación, como la selección de la ventana de búsqueda inicial en la tarea de seguimiento y la selección de la porción de imagen a la que calcular el histograma que nos servirá de patrón.
- Parámetros:
  - event: Tipo de evento de ratón que ha disparado la llamada a esta función.
  - x: Posición X del ratón donde se ha producido el evento.
  - y: Posición Y del ratón donde se ha producido el evento.
  - flags: Bandera que indican qué tipo de botón o tecla se ha activado en este evento. Puede tener los siguientes valores:

```
#define CV_EVENT_FLAG_LBUTTON  1
#define CV_EVENT_FLAG_RBUTTON  2
#define CV_EVENT_FLAG_MBUTTON  4
#define CV_EVENT_FLAG_CTRLKEY  8
#define CV_EVENT_FLAG_SHIFTKEY 16
#define CV_EVENT_FLAG_ALTKEY   32
```

  - param: Variable definida por el usuario que se pasa a esta función a través de la llamada cvSetMouseCallback (ver descripción de esta función).

creaMascara

- Declaración: void creaMascara(CvRect rect1, IplImage* mascara);
- Descripción: Función que sirve para pintar en la imagen *mascara* un rectángulo blanco que se le pasa como parámetro. Esta función la utilizaremos en el cálculo de flujo óptico para determinar qué zonas de la imagen general se van a procesar.
- Parámetros:
  - rect1: Rectángulo que pintaremos de blanco.

- mascara: Imagen donde almacenaremos el resultado. La imagen resultante servirá de filtro para otras funciones. Aquellos píxeles que valgan cero no se procesarán y los que valgan distinto de cero (que serán los correspondientes al rectángulo) se procesarán.

pintaFlecha

- Declaración: void pintaFlecha(CvPoint centro, CvPoint extremo, CvScalar color, int grosor, IplImage* imagen);
- Descripción: Función que sirve para pintar una flecha dentro de una imagen.
- Parámetros:
  - centro: Punto que indica el comienzo de la flecha.
  - extremo: Punto que indica el extremo de la flecha, hacia dónde ésta apunta.
  - color: Color de la flecha que vamos a pintar
  - grosor: Grosor de la flecha que pintaremos.
  - imagen: Imagen donde pintaremos la flecha especificada por los anteriores parámetros.

4.3.- 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 resultados. (Las imágenes que se muestran a continuación se corresponden con pruebas diferentes y con individuos monitorizados distintos).

En primer lugar, al abrir el ejecutable de esta aplicación aparecerán cuatro pantallas por defecto correspondientes al menú principal con todas las opciones de operación, la ventana principal donde podemos ver la secuencia de vídeo capturada por la cámara web y dos ventanas en principio vacías correspondientes a los histogramas de la cabeza y del torso del individuo que vamos a monitorizar. Estas dos últimas ventanas aparecen vacías inicialmente puesto que aún no hemos seleccionado la cabeza o el torso del individuo con el ratón para comenzar el seguimiento.

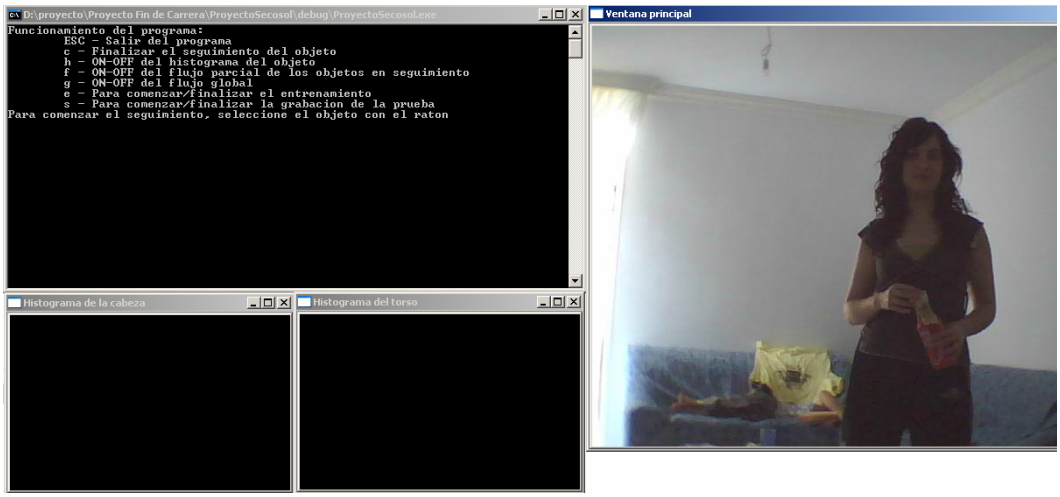


Figura 4.10.- Ventana inicial de la aplicación

Lo primero que tendremos que hacer para comenzar el seguimiento y monitorización del individuo es seleccionar la cabeza y el torso del mismo con el ratón (como se nos indica en el menú) y el programa comenzará automáticamente a funcionar.

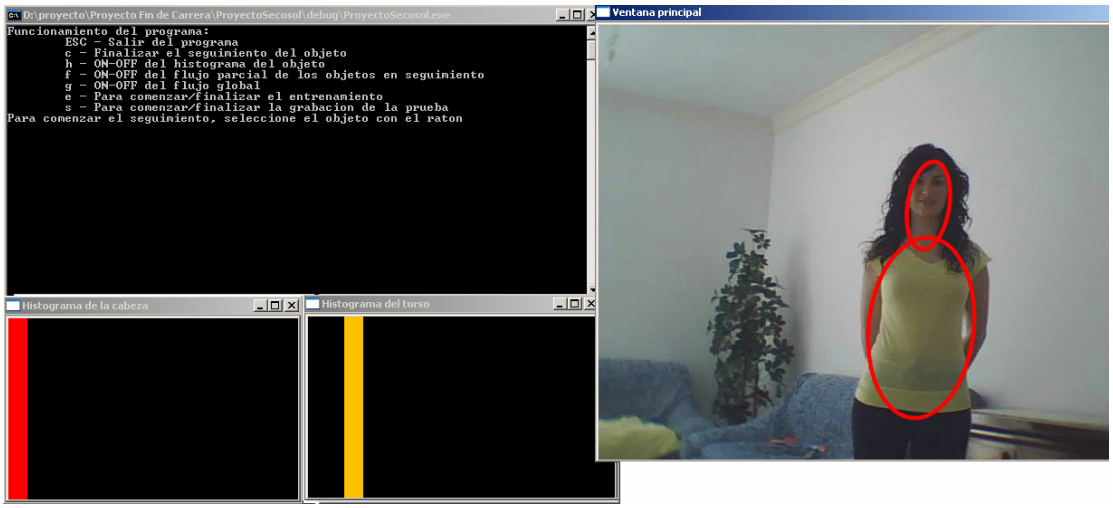


Figura 4.11.- Selección de los elementos a seguir. Cabeza y torso

Como podemos ver en la figura anterior, una vez seleccionados los elementos de interés aparecen los histogramas correspondientes a cada uno de ellos. Estos histogramas serán los que utilizaremos dentro de la función **seguimiento** para calcular la imagen *backprojection* a partir de la imagen capturada por la cámara.

A partir de ese momento, el individuo estará continuamente monitorizado, de tal forma que aunque se mueva la función sabrá las posiciones

de su cabeza y de su torso y por lo tanto podrá determinar si se ha producido una situación de riesgo o caída.

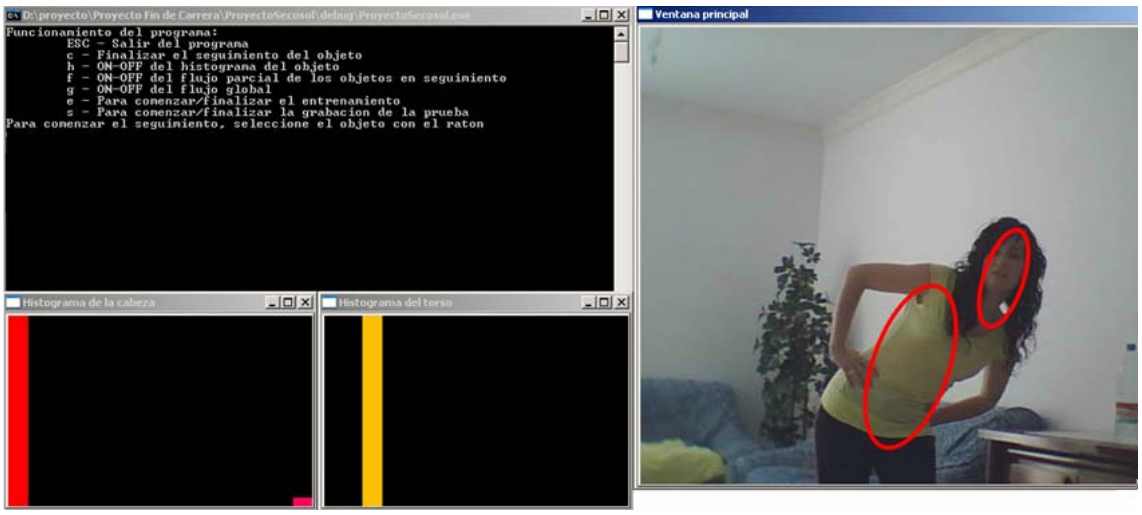


Figura 4.12.- Individuo monitorizado por la aplicación

Atendiendo a las distintas opciones existentes en el menú de la aplicación veremos todas las posibilidades que se nos ofrecen con la aplicación. En primer lugar es posible ver el flujo óptico del individuo, tanto el parcial como el global, donde podemos ver los vectores de desplazamiento individuales asociados a cada píxel de los elementos de interés o el vector global que nos indica, en media, hacia dónde se está moviendo el individuo. A continuación mostramos dos figuras donde podemos ver el flujo parcial y el global en dos situaciones distintas.

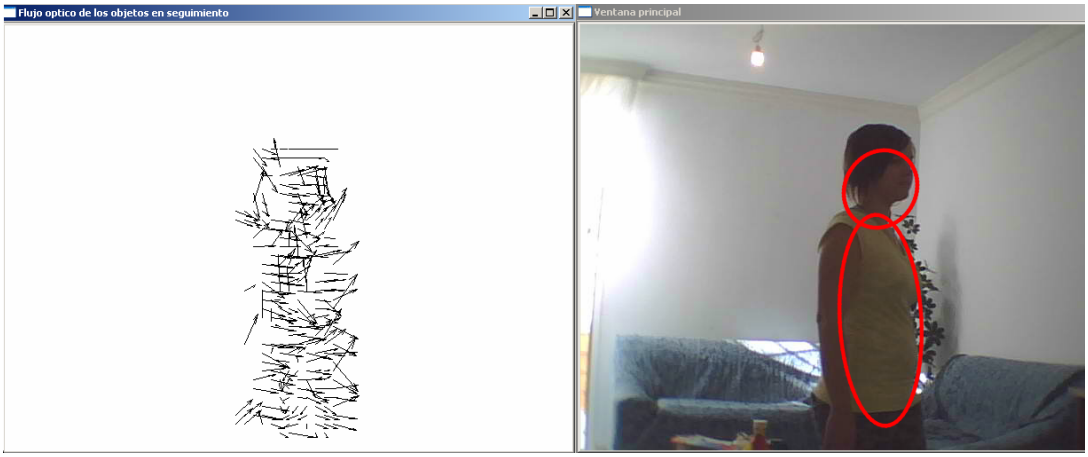


Figura 4.13.- Flujo óptico parcial. Caminando hacia la derecha

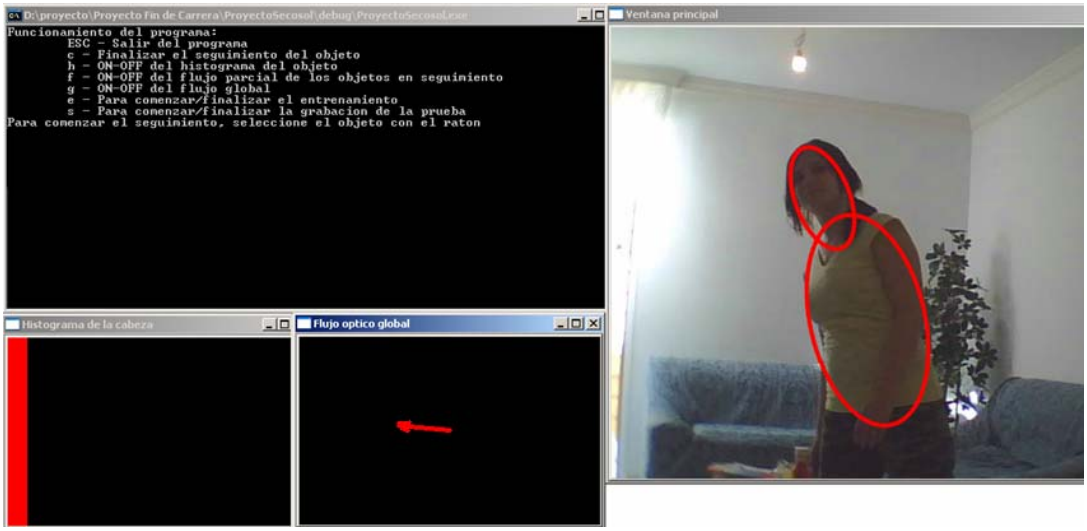


Figura 4.14.- Flujo óptico global. Inclinándose hacia la izquierda

Otra opción que se nos facilita en el sistema es la grabación de un vídeo para poder almacenar los resultados de distintas pruebas. Para comenzar la grabación pulsaremos la tecla 's', apareciendo una ventana dónde se nos pide que indiquemos el tipo de compresión que vamos a utiliza en nuestro video.

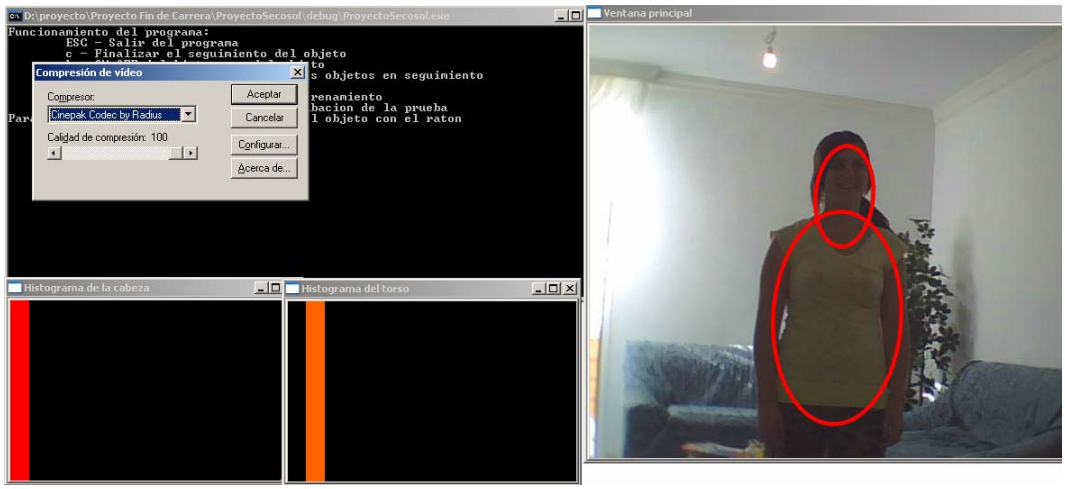


Figura 4.15.- Pantalla de inicialización de la grabación

Una vez seleccionado el tipo de compresión y tras pulsar 'Aceptar', comenzará la grabación. Esto se indicará por pantalla con el mensaje 'Grabando' y se pintará un punto por cada frame incluido en el archivo de vídeo. En nuestro caso hemos elegido siempre un tipo de compresión XviD, el cual muestra una ventana de estado mientras se está grabando como se puede observar en la imagen 4.9 de este capítulo.

Para finalizar la grabación pulsaremos de nuevo la tecla 's', mostrándose por pantalla el mensaje 'Grabación parada'. En la misma carpeta donde se

encuentra el ejecutable aparecerá entonces un archivo llamado **grabacion.avi** donde se podrá ver la secuencia de vídeo grabada, correspondiente a las imágenes mostradas en la ventana principal durante el tiempo de grabación.

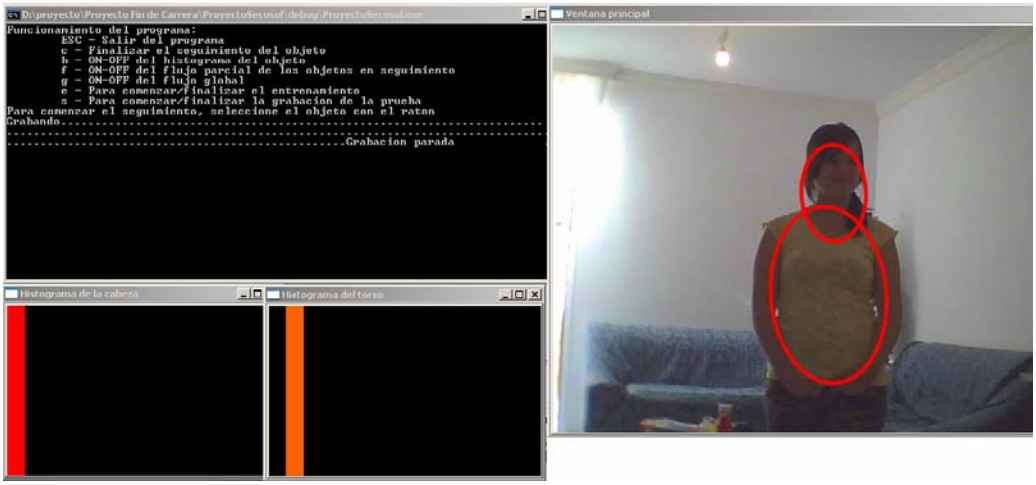


Figura 4.16.- Captura de pantalla tras la grabación

Por otro lado, y como ya hemos detallado anteriormente en el capítulo, la aplicación ofrece lo que hemos llamado 'Bloque de entrenamiento', encargado de escribir en un fichero de salida los parámetros relacionados con el flujo global del individuo. Estos ficheros se incluyen en el anexo C de la memoria y son los que se han utilizado para determinar de forma empírica la velocidad límite de la cabeza del individuo a partir de la cual se considerará que es posible que se haya producido una caída. Para comenzar o finalizar la fase de entrenamiento debemos pulsar la tecla 'e', mostrándose por pantalla los mensajes 'Comienzo del entrenamiento' y 'Fin del entrenamiento'.

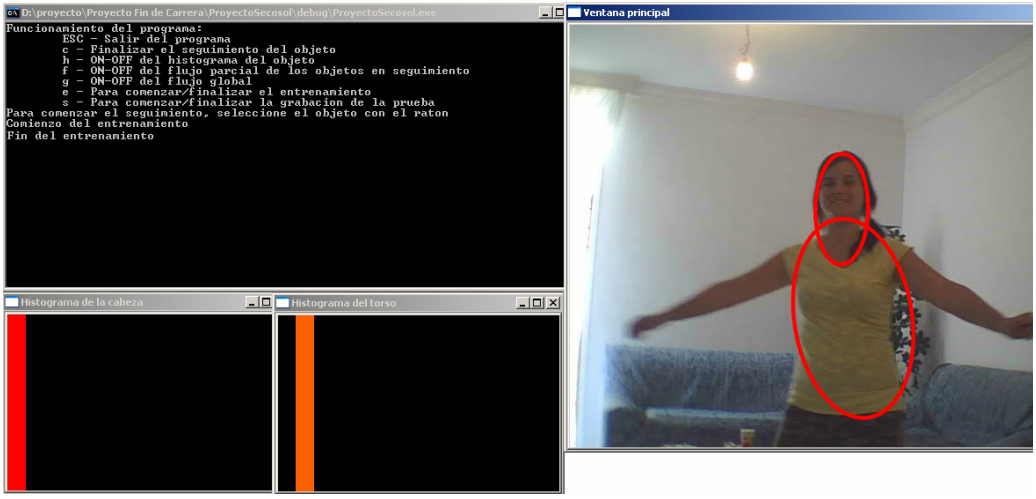


Figura 4.17.- Inicio y finalización de la fase de entrenamiento

El archivo donde se escriben los parámetros relacionados con el flujo óptico global (magnitud, componente X y componente Y) se llama **velocidades.txt** y se guardará en el mismo directorio donde se encuentre el ejecutable de la aplicación.

En cuanto a la fase de detección de caídas, cuyo funcionamiento se explica en el apartado 4.2.2.6, mostraremos a continuación algunas imágenes mostrando en primer lugar una situación en la que la caída se ha detectado (pues el individuo ha permanecido en el suelo durante un determinado tiempo) y en segundo lugar una imagen en la que nos hemos recuperado de la caída (donde nos levantamos antes de que se cumpla el tiempo límite).

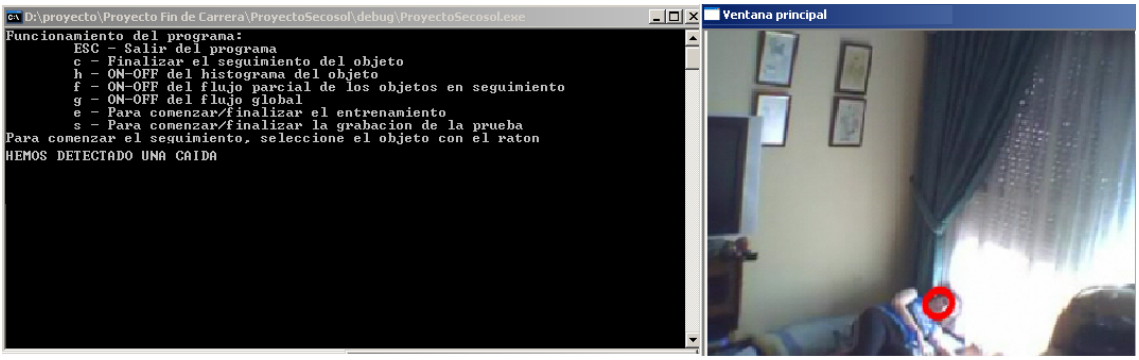


Figura 4.18.- Detección de una caída con la aplicación

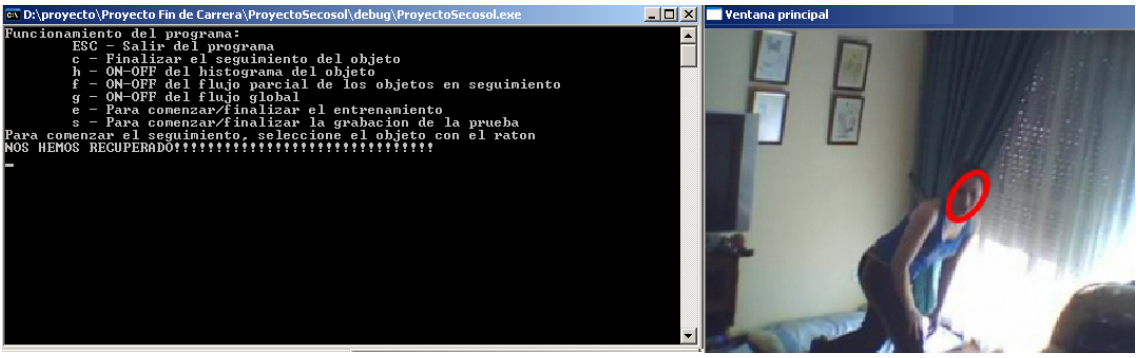


Figura 4.19.- Recuperación tras una caída. Incorporación del individuo

Por último incluiremos algunas capturas de pantalla más donde se muestra el funcionamiento de la aplicación en otras situaciones. En estas figuras podremos observar principalmente flujos ópticos parciales y globales.

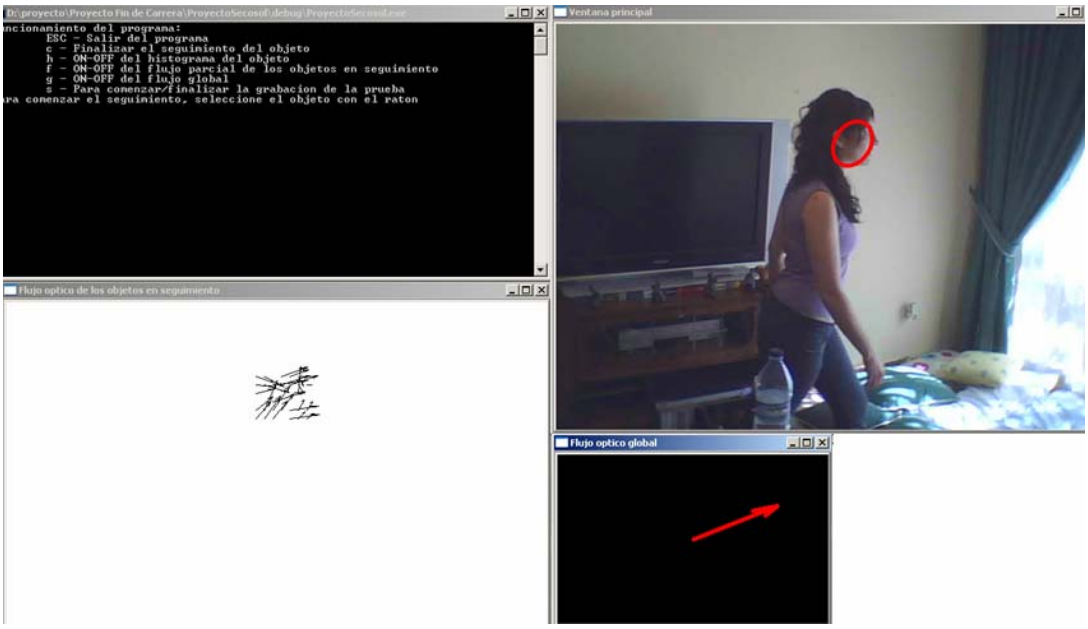
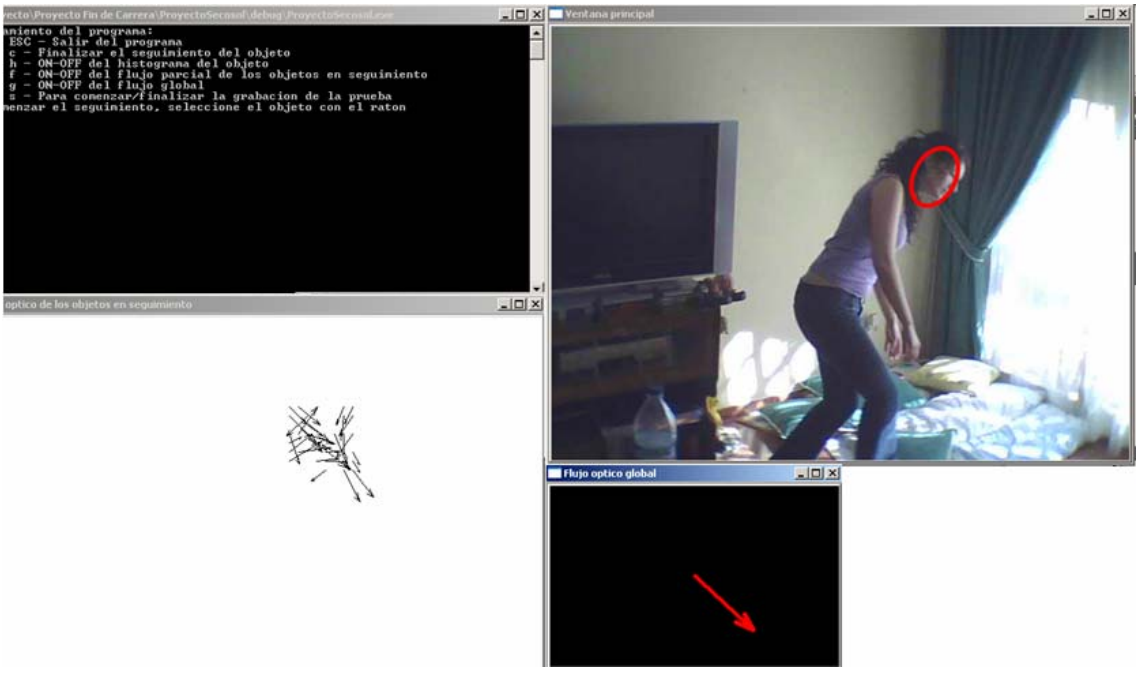


Figura 4.20.- Otras capturas de pantalla (I)



4.21.- Otras capturas de pantalla (II)

4.4.- Resultados

En este apartado vamos a detallar las pruebas que hemos realizado sobre la aplicación para comprobar su funcionamiento y encontrar las posibles

deficiencias que presente. En base a los resultados expondremos una serie de posibles mejoras en el siguiente capítulo de la memoria.

- Funcionamiento general: En este caso sometemos al sistema a la prueba de cada una de las tareas que se le han programado, desde la muestra de resultados (activando y desactivando ventanas) hasta la grabación de vídeos o escritura de archivos de texto.

  Una de las primeras ventajas que se pueden observar en comparación con el comportamiento del algoritmo basado en *background subtraction* es que aunque la persona permanezca inmóvil durante mucho tiempo el sistema sigue monitorizándola y el seguimiento se realiza sin complicaciones.

- Batería de caídas: Este test consiste en tratar de detectar una serie de caídas y comprobar que el bloque de detección de caídas funciona correctamente. Esta prueba se realizó con la cámara web capturando a 15 frames por segundo y después capturando a 30 frames por segundo, obteniendo resultados muy distintos.

  Para la captura a 15 frames por segundo los resultados de detección fueron realmente malos. De 16 caídas realizadas por personas distintas sólo se detectaron 3 correctamente. El resto de caídas que no fueron detectadas se debió a que el sistema, al trabajar con un "frame rate" muy bajo, no era capaz de seguir la cabeza del individuo, perdiéndose en algún distractor cercano al elemento de interés (en la pared, en un cuadro, etc).

  Sin embargo, para la captura con 30 frames por segundo los resultados fueron bastante satisfactorios, ya que de 20 caídas realizadas se llegaron a detectar 14 de ellas correctamente, alcanzando así un porcentaje de éxito mucho mayor.

Velocidad captura	Batería de caídas	Caídas detectadas	Porcentaje éxito
15 fps	16	3	18,75 %
30 fps	20	14	70 %

  Con esta prueba dejamos patente lo importante que es en nuestra aplicación un procesamiento de imágenes veloz y eficiente, ya que es una condición que marca la diferencia entre un sistema de detección de caídas no fiable y uno mucho más fiable.

- Cambios en la iluminación: Para comprobar lo robusto que es nuestro sistema ante cambios de luminosidad se han realizado pruebas modificando la luz presente en la escena.

  En un principio se realizaron pruebas modificando paulatinamente la iluminación y viendo que el sistema funcionaba correctamente sin problemas se pasó a realizar pruebas con un cambio de iluminación más brusco, como el encendido o apagado de la lámpara principal de la escena. Hemos de decir que en rasgos generales la aplicación es bastante robusta ante este tipo de situaciones.

- Oclusiones: Por último hemos procedido a realizar dos tests distintos para comprobar la robustez del sistema cuando se producen oclusiones de los elementos de interés.

  El primero de los tests se realizó utilizando como distractor un objeto cuyo histograma fuese distinto al del elemento que estamos siguiendo. En este caso el programa tiende a situar la ventana de búsqueda al lado del distractor (no sobre él) y por lo tanto se pierde la posición del último sitio donde se detectó el elemento de interés. Cuando el objeto que oculta nuestro elemento desaparece, si la ventana de búsqueda está cercana, en general el sistema suele recuperarse, en caso contrario se produce una pérdida definitiva del elemento que estamos siguiendo.

  El segundo test se realizó utilizando como distractor uno con un histograma similar al del objeto de interés. En nuestro caso hemos empleado la mano, por presentar el mismo tono que la cara. En este caso, si el área del distractor es mayor que la del elemento seguido, el sistema terminará siguiendo al distractor; si el área es menor el sistema no se ve afectado por la oclusión y sigue funcionando sin problemas.

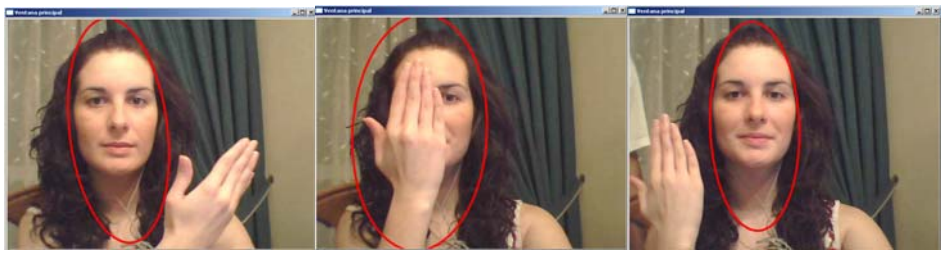


Figura 4.22.- Secuencia de imágenes. Test de oclusión

En el siguiente capítulo introduciremos una serie de mejoras interesantes para el sistema relacionadas con todos estos resultados.