PROYECTO FIN DE CARRERA

Visión por Computador en iPhone 4

Autor: Pablo Roldán Ruz
Tutor: Sergio Escalera

Resumen

Dado del importante aumento en los recursos y capacidad de computación de los dispositivos móviles, en este proyecto se propone el estudio del funcionamiento y rendimiento de algoritmos de Visión por Computador en un dispositivo móvil iPhone 4.

En concreto, se propone estudiar el funcionamiento en este dispositivo del reconocimiento de patrones de comportamiento humano a partir del testeo de detectores faciales y corporales.

También se presenta una nueva aplicación móvil que permite realizar fotografías de forma automática a partir de la detección de un determinado numero de personas en la imagen.

Abstract

Given the significant increase of the resources and computing capabilities in current mobile devices, a study of the Computer Vision algorithms performance in an iPhone 4 mobile device is proposed in this project.

Specifically, this project studies the performance and behavior of human patterns detection in this device by testing facial and corporal detector.

It is also presented a new mobile application capable of automatically taking photographs when a certain number of people are detected through the iPhone camera.

Tabla de contenido

1. Introducción ... 6
2. Análisis ... 8
2.1. Detección de patrones corporales ... 8
2.1.1. Algoritmo Viola-Jones ... 9
2.1.1.1. Representación de la imagen ... 9
2.1.1.2. Características o features ... 10
2.1.1.3. Clasificadores ... 11
2.1.1.4. Cascada de clasificadores ... 12
2.2. Entorno de desarrollo ... 13
2.2.1. Breve introducción a Objective-C ... 14
2.3. Obtención del ground truth ... 15
2.4. Estudio del rendimiento de detectores sobre una secuencia de vídeo ... 17
2.5. Análisis de los resultados ... 18
2.6. Aplicación fotografía automática ... 19
3. Diseño e implementación ... 21
3.1. Compilación de OpenCV para iPhone ... 21
3.1.1. Uso de OpenCV en un proyecto XCode ... 23
3.2. Implementación de la detección ... 24
3.2.1. Carga del detector ... 24
3.2.2. Conversión de la imagen al formato de OpenCV ... 25
3.2.3. Conversión a escala de grises ... 26
3.2.4. Detección ... 28
3.3. Análisis del rendimiento de detectores sobre una secuencia de vídeo ... 31
3.3.1. Modelo ... 31
3.3.2. Vista ... 32
3.3.3. Controlador ... 33
3.3.3.1. Extracción de imágenes de la secuencia de vídeo ... 33
3.3.3.2. Tratamiento de los resultados ... 35
3.4. Análisis de los resultados ... 37
3.4.1. Modelo ... 37
3.4.2. Vista ... 38
3.4.3. Controlador ... 39
3.5. Fotografía automática ... 40
3.5.1. Modelo ... 40

3.5.2. Vista 40
3.5.2.1. Vista principal 40
3.5.2.2. Vista de configuración 41
3.5.2.3. Vista de detección 42
3.5.3. Controlador 43
3.5.3.1. Controlador principal 43
3.5.3.2. Controlador configuración 46
3.5.3.3. Controlador de detección 47
4. Resultados 48
5. Conclusiones y trabajo futuro 51
6. Bibliografía 52
7. Anexo 53

Índice de ilustraciones

Figura 1.- Representación de imagen integral 9
Figura 2.- Ejemplo de imagen integral........ 10
Figura 3.- Cálculo en imágenes integrales........ 10
Figura 4.- Tipos de features........ 11
Figura 5.- Ejemplo de las features utilizadas en la detección de cara........ 11
Figura 6.- Cascada de clasificadores 12
Figura 7.- Pantalla de Xcode........ 14
Figura 8.- Etiquetado de patrones con imageclipper 15
Figura 9.- Recortes generados por imageclipper........ 16
Figura 10.- Archivo de texto con ground truth del detector facial 17
Figura 11.- Diagrama de casos de uso de la aplicación de análisis........ 17
Figura 12.- Diagrama de casos de uso de la aplicación de análisis de resultados 18
Figura 13.- Fórmula de la sensibilidad 19
Figura 14.- Positivos verdaderos. Unión de las áreas. 19
Figura 15.- Diagrama de casos de uso de la aplicación de fotografía automática.19
Figura 16.- Estructura del código fuente de OpenCV 21
Figura 17.- Librerías compiladas 22
Figura 18.- Framework de OpenCV dentro de un proyecto Xcode........ 23
Figura 19.- Motor NEON SIMD 27
Figura 20.- Ejemplo de todas las detecciones provocadas en la detección facial .30
Figura 21.- Diagrama de clases de la representación del ground truth 32
Figura 22.- Vista de la aplicación de análisis de detectores 32
Figura 23.- Diagrama de clases de la aplicación de análisis de vídeo 33
Figura 24.- Diagrama de clases Ground Truth 38
Figura 25.- Vista de la aplicación de análisis de resultados 38
Figura 26.- Vista principal de la aplicación de fotografía automática........ 41
Figura 27.- Vista de configuración........ 42
Figura 28.- Vista de detección 43
Figura 29.- Diagrama de clases de la aplicación de fotografía automática 43
Figura 30.- Resultados ofrecidos por la aplicación de análisis de resultados 50

1. Introducción

Debido al importante aumento de su capacidad computacional y de los recursos disponibles, los dispositivos móviles actuales son considerados en el mundo doméstico tan potentes y tan válidos como un ordenador personal. La implantación de las redes 3G, la popularización del uso del Internet móvil, la consolidación de los nuevos sistemas operativos para móviles o la aparición de numerosas aplicaciones destinadas a este tipo de dispositivo, son algunos de los múltiples factores causantes de la, cada vez más frecuente, sustitución de los ordenadores convencionales por los nuevos teléfonos móviles.

Sin embargo, a pesar de todos estos avances y de las muchas y nuevas aplicaciones que continuamente son desarrolladas para estos dispositivos, muchas áreas de la informática, como la visión por computador y la inteligencia artificial, que actualmente cuentan con estados de desarrollo similares a los de este campo, todavía no tienen una presencia notable en este sector.

Este proyecto propone la unión de estas dos áreas mediante el estudio de la viabilidad y el comportamiento de una de las aplicaciones en un dispositivo móvil de la visión por computador por excelencia, la detección de patrones. Para ello, se propone el estudio del rendimiento de uno de los algoritmos del estado del arte de la detección de patrones, el algoritmo Viola-Jones, en uno de los dispositivos más potentes de la actualidad, el *iPhone 4*.

En lugar de realizar una implementación específica del algoritmo de Viola-Jones para este dispositivo, se opta por el uso y adaptación de una de las implementaciones más extendidas de este algoritmo, incluida dentro de *OpenCV*; una librería de código abierto desarrollada por *Intel,* que recoge diversas funcionalidades de la visión por computador, entre ellas una implementación de este algoritmo. Mediante esta librería se implementan 4 detectores corporales –facial, cuerpo completo, parte superior del cuerpo y parte inferior del cuerpo—, a través de los cuales se estudia el comportamiento de este tipo de métodos en un dispositivo de recursos limitados como el propuesto.

Para realizar un análisis objetivo de su rendimiento, de forma que todos los detectores se apliquen bajo las mismas condiciones, se crea una aplicación en la que estos detectores son aplicados sobre una secuencia de vídeo que, progresivamente, muestra los diferentes patrones a detectar. Para comprobar la validez de las detecciones realizadas, la secuencia de vídeo se ha etiquetado previamente marcando las coordenadas y dimensiones de cada uno de los patrones que deberían ser detectados. Los resultados de las detecciones son comparados por una segunda aplicación que genera un informe sobre el rendimiento de los detectores analizados.

A pesar de que los recursos disponibles en estos dispositivos se han incrementado considerablemente, aún siguen siendo muy limitados. Por lo que, aunque se espera un buen rendimiento de los detectores, también se espera que estas limitaciones en los recursos impidan alcanzar los mismos niveles de rendimiento que podrían obtenerse en ordenadores convencionales. Estas suposiciones son confirmadas una vez finalizado el estudio, confirmando que, aunque con menor rendimiento y mayores dificultades que en ordenadores

convencionales, los resultados certifican la viabilidad del uso de este tipo de tecnología en dispositivos móviles.

Por último, se crea un nuevo programa que presenta una aplicación directa de la tecnología estudiada a lo largo de este proyecto. Esta aplicación es capaz de realizar una fotografía de manera automática cuando un determinado número de personas –especificado previamente por el usuario— es detectado a través de la cámara del dispositivo. Para ello, el programa analiza la imagen captada por la cámara aplicándole un detector facial. La fotografía se tomará en función del número de caras detectadas en la imagen.

2. Análisis

El proyecto tiene dos objetivos fundamentales: analizar el rendimiento de los algoritmos de detección de patrones corporales en un *iPhone* 4 e implementar una aplicación que permita realizar una fotografía cuando un determinado número de personas es detectado a través de la cámara del dispositivo. Para cumplir estos objetivos es necesaria la implementación de varias aplicaciones específicas para este teléfono móvil.

A lo largo de este capítulo se detallan los requisitos de diseño de dichas aplicaciones, así como las herramientas y tecnologías requeridas para su implementación, junto a un conjunto de conceptos y técnicas propios de los algoritmos de detección en los que se basan dichos programas.

2.1. Detección de patrones corporales

El problema de la detección de patrones, aunque trivial para la mente humana, resulta de una complejidad muy elevada cuando se pretende tratar de manera computacional. Desde el punto de vista de una máquina, una cara o un cuerpo, no es de ninguna manera diferente a cualquier otro objeto de una imagen. Es más, desde este punto de vista, no hay diferencia alguna entre un objeto y el fondo de la imagen; para una máquina una imagen no es más que un conjunto de números que definen los colores e intensidades de los píxeles que la componen. El problema, por tanto, no sólo consiste en detectar un determinado patrón corporal, sino en detectar cualquier clase de objeto.

La dificultad del problema va más allá, la detección puede volverse mucho más compleja debido a las condiciones específicas de cada imagen. En el caso concreto de la detección facial: la iluminación; el ángulo con el que fue tomada la imagen; la distancia a la que se encuentra el patrón a detectar –que influirá directamente en el tamaño de los mismos—; la ausencia o presencia de pelo u otros objetos como gafas, gorros o cualquier tipo de ropa; la expresión facial; o la oclusión de partes de la cara por otros objetos presentes en la imagen pueden convertir esta tarea en un problema realmente difícil de resolver.

Este problema ha sido abordado ampliamente por la visión por computador en los últimos años, existiendo diversas técnicas que, desde varios enfoques, consiguen buenos resultados en la resolución del problema.

Por un lado, existen técnicas basadas en el conocimiento humano de los distintos elementos que componen un patrón y en la forma en que éstos se relacionan. Por ejemplo, en el caso particular de la detección de caras, el conocimiento de la relación existente entre los ojos, la nariz o la boca podría usarse para construir un detector de cara.

Otros métodos se basan en la búsqueda de características que se mantengan constantes en el patrón, independientemente de las condiciones de la imagen (ángulo, distancia o iluminación) para posteriormente buscar esas invariantes en las imágenes e identificar los patrones a través de ellas.

Por otro lado, hay métodos que intentan aislar los posibles patrones a través del color o intensidad de ciertas regiones de la imagen para luego analizar en mayor profundidad, apoyándose en otras técnicas, esos posibles candidatos.

Cada técnica presenta sus ventajas e inconvenientes, destacando unas en determinadas condiciones y fallando ante otras circunstancias. Sin embargo, en los últimos años se han producido significativos avances en esta área, existiendo en la actualidad diversos métodos con los que conseguir resultados robustos.

2.1.1. Algoritmo Viola-Jones

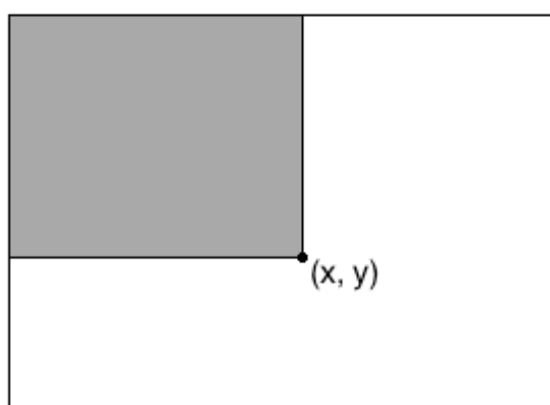
Lejos de analizar el rendimiento de las diferentes técnicas que existen en la actualidad, o tratar de averiguar qué método es el que mejor se adapta a este tipo de dispositivo móvil; el proyecto se centra en comprobar la viabilidad del uso de este tipo de técnicas en un dispositivo móvil a pesar de las limitaciones impuestas por este entorno en cuanto a capacidad de procesado y recursos disponibles.

En concreto, el proyecto se centra en el estudio de uno de los algoritmos del estado del arte de la detección de patrones, el algoritmo de Viola-Jones. Sin embargo, en lugar de realizar una implementación propia del mismo, se opta por usar una de sus implementaciones más extendidas en la actualidad, incluida dentro de *OpenCV*, una librería de código abierto desarrollada por Intel, que contiene multitud de funciones relacionadas con la visión por computador, entre ellas un método de detección de objetos basado en el algoritmo de Viola-Jones.

Este algoritmo consigue unos tiempos de detección tan bajos que puede ser usado incluso en sistemas de tiempo real. Para conseguir tal rendimiento, este algoritmo usa una representación alternativa de la imagen (*imagen integral*) que facilita los cálculos necesarios para la detección, junto a unos clasificadores aplicados en cascada y basados en *features* o *características*.

2.1.1.1. Representación de la imagen

Para obtener el alto rendimiento que el algoritmo de Viola-Jones alcanza, se utiliza una representación alternativa de la imagen conocida como *imagen integral*, que permite realizar complejos cálculos de manera muy eficiente. En este formato cada píxel de la imagen contiene el resultado de la suma del valor de los píxeles por encima y a la izquierda de dicho píxel en la imagen.



$$ii(x,y) = \sum_{x' \leq x,\, y' \leq y} i(x', y')$$

Figura 1.- Representación de imagen integral

A modo de ejemplo, en el caso trivial en el que todos los píxeles de la imagen tuvieran valor 1, la imagen integral equivalente sería la siguiente.

1	1	1
1	1	1
1	1	1

Imagen Original

1	2	3
2	4	6
3	6	9

Imagen Integral

Figura 2.- Ejemplo de imagen integral

Esta representación de la imagen permite realizar operaciones de manera sencilla, rápida y eficiente. Por ejemplo, la suma de cualquier rectángulo puede ser calculada a partir de 4 referencias a valores de la imagen.

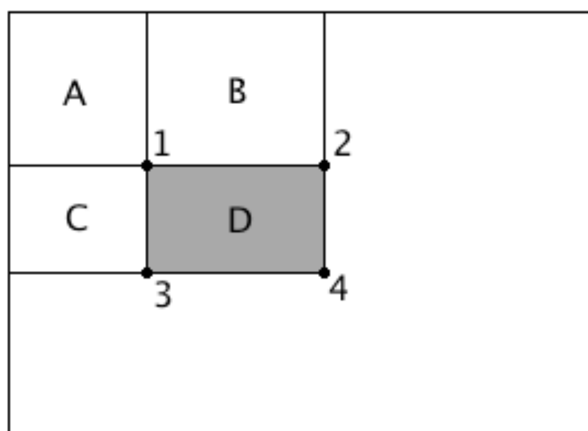


Figura 3.- Cálculo en imágenes integrales

Así, la suma de los píxeles del rectángulo D podría ser calculada de la siguiente manera.

- La suma de los píxeles del rectángulo A es el valor de la imagen integral en el punto 1.
- El valor del punto 2 se corresponderá con A + B.
- El valor del punto 3 se corresponde con la suma de las regiones A + C.
- El valor del punto 4 es A + B + C + D.
- La suma del rectángulo D será, por tanto: 4 + 1 – (2 + 3)

$$4 + 1 - (2 + 3) = (A+B+C+D)+(A)-(A+B)-(A+C)=2A+B+C+D-2A-B-C=D$$

Este tipo de cálculo es de gran utilidad para calcular el valor de las *características* o *features* en los que el algoritmo de Viola-Jones está basado.

2.1.1.2. Características o features

El algoritmo de detección clasifica las imágenes candidatas en función del valor de determinadas *features*. Se usan 3 tipos de *features*: de 2 rectángulos, de 3 rectángulos y de 4 rectángulos.

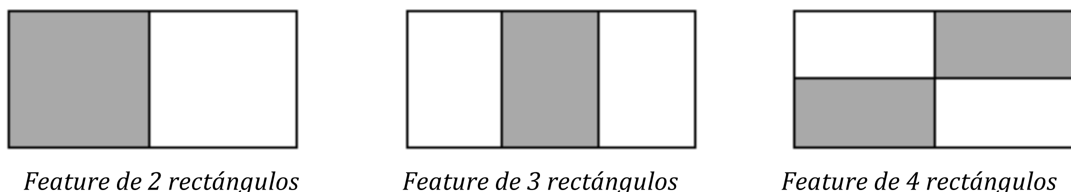


Figura 4.- Tipos de features

El valor de la *feature* se calcula como la resta de la suma de los píxeles de los rectángulos grises menos la suma de los píxeles de los rectángulos blancos. Estos cálculos, tal y como se ha visto en el apartado anterior, resultan muy efectivos y fáciles de realizar cuando la imagen se encuentra representada como una imagen integral.

2.1.1.3. Clasificadores

El principio sobre el que se fundamente este algoritmo consiste en que el valor de algunas de estas *features* será significativamente más alto cuando se encuentren situadas sobre una determinada región del patrón que se desea detectar.

Puesto que el número de posibles *features* presentes en una imagen llega a ser incluso superior al número de píxeles de la misma, se usa una modificación del algoritmo de aprendizaje *AdaBoost* para construir una serie de clasificadores a partir de la selección de las *features* que mayor información aportan sobre la identificación de un determinado patrón.

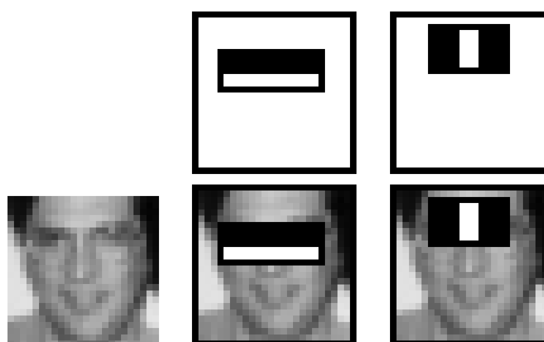


Figura 5.- Ejemplo de las features utilizadas en la detección de cara

AdaBoost considera cada una de las *features* existentes como un posible *clasificador débil*[1] y estudia los resultados que obtiene cada uno de ellos en un conjunto de imágenes de entrenamiento de las que ya se conocen, a priori, las dimensiones y coordenadas en las que aparece el patrón a detectar. De esta manera, uno de estos clasificadores débiles calculará el valor de una determinada *feature* en una determinada posición de la imagen, si su valor supera un determinado valor umbral, el clasificador considerará la imagen sobre la que ha sido aplicado como un positivo, y si no lo supera, como un negativo.

---

[1] Un clasificador débil es aquel cuya tasa de aciertos en la clasificación es tan sólo ligeramente superior a la mitad de los casos.

A partir de la información de estos resultados, *AdaBoost* construye unos *clasificadores fuertes* mediante la combinación ponderada de varios *clasificadores débiles,* previamente seleccionados en función de los resultados obtenidos por cada uno de ellos en el conjunto de imágenes de entrenamiento.

2.1.1.4. Cascada de clasificadores

La idea general del algoritmo consiste en recorrer y aplicar, a través de una subventana de la imagen, una serie de clasificadores en sucesivas iteraciones, cada una de ellas con un tamaño de imagen diferente. Sin embargo, el reescalado de la imagen es una tarea muy costosa pudiéndose convertir en un cuello de botella, por lo que Viola-Jones propone reescalar el detector en lugar de la imagen, ya que el coste de los cálculos en una imagen integral es constante, independientemente del tamaño de la misma.

No obstante, el número de clasificadores a evaluar en todas las subventanas resultantes continúa siendo muy elevado. Por ello, Viola-Jones plantea el problema de manera diferente: puesto que el número de subventanas en las que se encontrará el patrón es probablemente muy inferior al número de subventanas en las que no estará presente, es más fácil determinar cuándo el patrón no está presente en lugar de cuando lo está. De esta manera, el problema se puede transformar en descartar todas aquellas subventanas en las que no se encuentre el patrón.

Para resolver este problema, en lugar de aplicar una serie de clasificadores en paralelo que determinen dónde aparece el patrón; se construye una cascada de clasificadores compuesta por una serie de clasificadores que se aplicarán en orden. Si un clasificador descarta una determinada imagen, no se dedicará más tiempo a esa imagen y no será analizada por los siguientes clasificadores de la cascada. Por el contrario, si un clasificador determina que es posible que exista el patrón en la imagen, se aplicará el siguiente clasificador de la cascada y así sucesivamente.

La cascada está construida de forma que los clasificadores más complejos estén situados en los niveles más profundos. De esta manera, se dedica más tiempo de computación a aquellas subventanas que no han sido descartadas y cuya probabilidad de contener el patrón es más elevada.

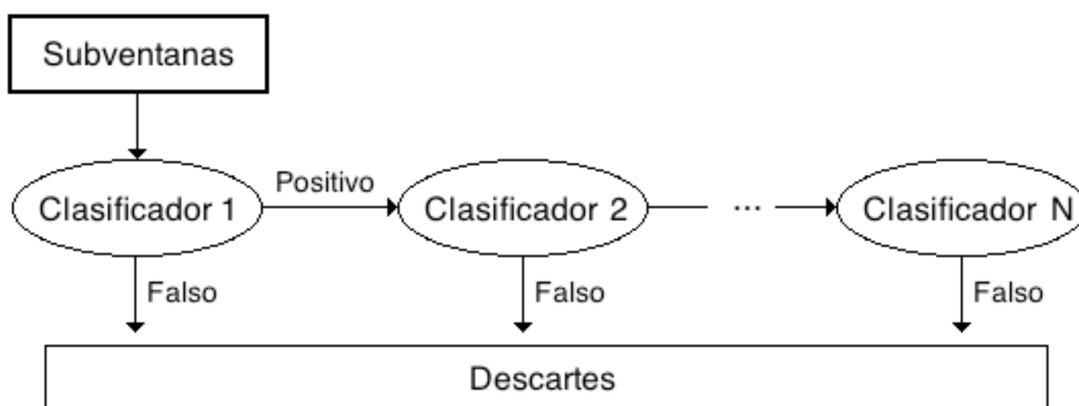


Figura 6.- Cascada de clasificadores

La imagen anterior muestra el funcionamiento general del algoritmo. La imagen es recorrida sucesivamente a través de una subventana cuyo tamaño es incrementado en cada iteración. A cada una de estas subventanas se le aplica una cascada de clasificadores, cuya complejidad es directamente proporcional a la profundidad del nivel de la cascada en el que se encuentran. Si una subventana es descartada por un clasificador, no continuará siendo analizada por los clasificadores de los siguientes niveles de la cascada. Por el contrario, aquellas subventanas que no hayan sido descartadas por ningún clasificador, serán consideradas como positivos.

2.2. Entorno de desarrollo

Existen varias herramientas y lenguajes de desarrollo para *iPhone*, sin embargo sólo una de las combinaciones posibles es la oficial y admitida por Apple: *Objective-C* y *Xcode.*

Objective-C es un lenguaje de programación orientado a objetos y reflexivo. La reflexión se define como la capacidad de un programa de observar y modificar su comportamiento y estructura en tiempo de ejecución. De este modo, en *Objective-C* se intentan posponer tantas decisiones como sea posible de tiempo de compilación a tiempo de ejecución (resoluciones de direcciones de memoria, llamadas a métodos, etc.).

Una de las principales características de este lenguaje consiste en que al igual que *SmallTalk*, uno de los lenguajes en los que está inspirado, en *Objective-C* no existen las llamadas a métodos como tales. En lugar de realizar invocaciones a métodos, se envían mensajes (o *selectores*) a objetos, que posteriormente son traducidas a punteros hacia los correspondientes métodos que los implementan. Esta es una de las características que hacen posible la reflexión en este lenguaje.

En realidad, *Objective-C* es una extensión del lenguaje C estándar y, como tal, cualquier programa para iPhone podría ser escrito usando C, C++, o incluso una mezcla de todos ellos. Esto es de vital importancia para el proyecto, puesto que se necesitará emplear C o C++ para el uso de la librería *OpenCV*, donde se encuentra la implementación del algoritmo de Viola-Jones que se usará para realizar las detecciones.

Apple ha creado una potente API en *Objective-C* para el desarrollo de software en dispositivos iOS –*iPhone*, *iPad* y *iPod Touch*—, denominada *Cocoa Touch* y que facilita en gran medida el acceso y control de todos los dispositivos y funcionalidades con los que cuenta el *iPhone* (cámara, GPS, acelerómetro, etc.). Esto convierte a *Objective-C* en el lenguaje ideal para el desarrollo de las aplicaciones que se necesitan desarrollar en el proyecto puesto que, por un lado, facilitará el control de la cámara del *iPhone* y por otro lado, permitirá intercalar fragmentos de código en C o C++ con los que realizar las llamadas oportunas a los métodos de la librería *OpenCV*.

En cuanto a herramientas de desarrollo, *Xcode* es el IDE creado por Apple para el desarrollo de aplicaciones en entornos OS (Mac). Es capaz de compilar programas escritos en *Objective-C*, *C*, *C++* y *Java*; y dispone de utilidades con las que cargar y depurar programas directamente en un *iPhone*. Este IDE, además de ser la herramienta oficial de desarrollo para *iPhone*, se presenta como la mejor opción de desarrollo para los propósitos del proyecto, puesto que permitirá usar

al mismo tiempo tanto la API *Cocoa Touch* como la librería *OpenCV*, los dos elementos necesarios para la implementación del proyecto.

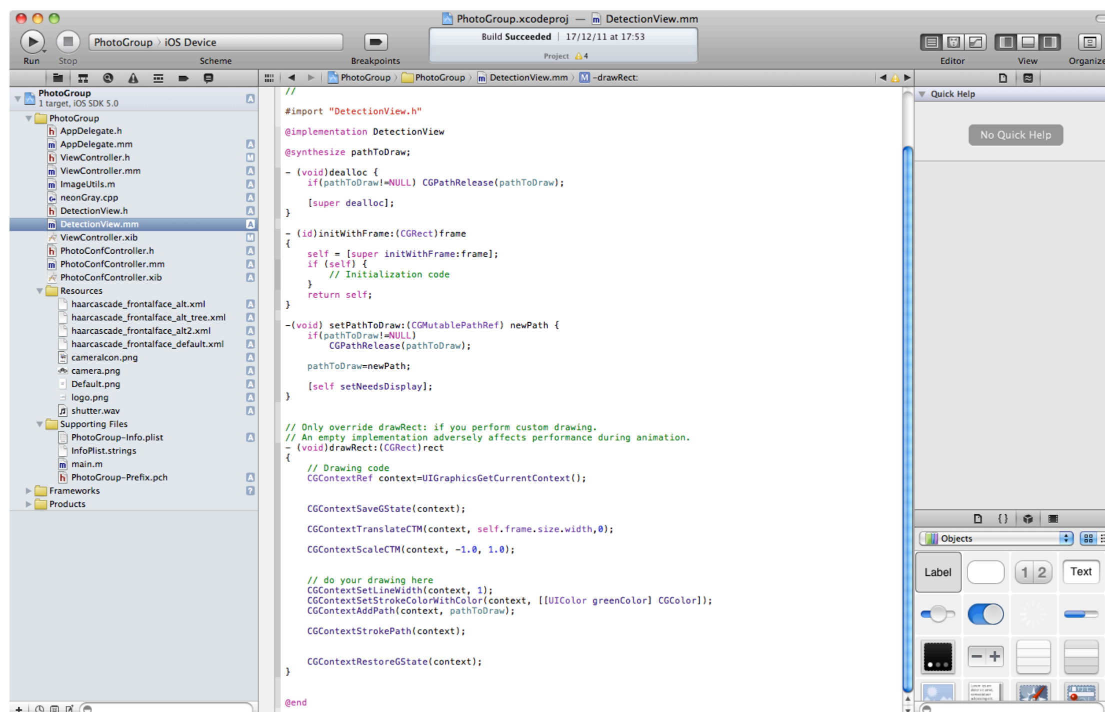


Figura 7.- Pantalla de Xcode

2.2.1. Breve introducción a Objective-C

Aunque la sintaxis y gramática de este lenguaje siguen conservando muchas de las características principales del lenguaje C, presenta importantes diferencias respecto a los lenguajes tradicionales. No obstante, a través de un sencillo ejemplo es posible conocer los fundamentos de este lenguaje y entender el código empleado a lo largo del proyecto.

```
// Definición clase
@interface NombreClase : NSObject {
    NSString *atributo;
}

- (NSString *) nombreMetodo: (NSInteger)parametro;

@end

// Implementación clase
@implementation NombreClase

- (NSString *) nombreMetodo: (NSInteger)parametro
{
    Clase *objeto = [[Clase alloc] init];

    NSString *objetoDevolver = [objeto metodo: parametro];

    return objetoDevolver;
}

@end
```

En el código anterior, se define en primer lugar una clase de nombre *NombreClase* con un atributo de la clase *NSString* y un método *nombreMetodo*,

que recibe un parámetro de la clase *NSInteger* y devuelve un objeto de la clase *NSString*. Nótese como todas las referencias a objetos son punteros tradicionales de C.

A continuación es definida la implementación de esta clase, donde se muestra la implementación del método definido en la interfaz. Este método crea, en primer un lugar, un objeto de la clase *Clase*; la llamada al método *alloc* se encarga de la reserva de memoria y la llamada a *init*, de la inicialización del objeto –nótese como se encadena la llamada a *init* al objeto devuelto por el método *alloc*—. A continuación, se realiza una llamada al método *metodo* del objeto recién creado, al que se le pasa como parámetro el objeto de la clase *NSInteger* recibido también como parámetro. El objeto devuelto por esta llamada es asignado a un nuevo objeto de la clase NSString, que al final del método es devuelto como resultado final del método a través del comando *return*.

2.3. Obtención del ground truth

Para estudiar el rendimiento de los detectores de patrones corporales se necesita crear una aplicación capaz de leer un vídeo y aplicar dichos detectores individualmente sobre todas las imágenes que componen esta secuencia de vídeo. Sin embargo, para determinar si las detecciones realizadas por los detectores son correctas o incorrectas, es necesario disponer, además, de la información real que el detector teóricamente debería detectar. Es decir, es necesario conocer la posición y tamaño de los patrones que el detector debe detectar para poder así determinar qué detecciones son correctas y qué detecciones no lo son. A esta información sobre los verdaderos patrones a detectar se le conoce como ***ground truth***.

La forma de obtener esta información consiste en etiquetar de forma manual cada uno de los patrones que aparecen en el vídeo. Es decir, se debe extraer y almacenar en un archivo el número de imagen, las coordenadas y las dimensiones de las áreas que contienen un patrón que el detector debería detectar.

Con la ayuda de *imageclipper* (http://code.google.com/p/imageclipper/) es posible recorrer la secuencia de vídeo imagen a imagen y seleccionar determinadas áreas de cada una de las imágenes, para generar a partir de las áreas seleccionadas archivos PNG.

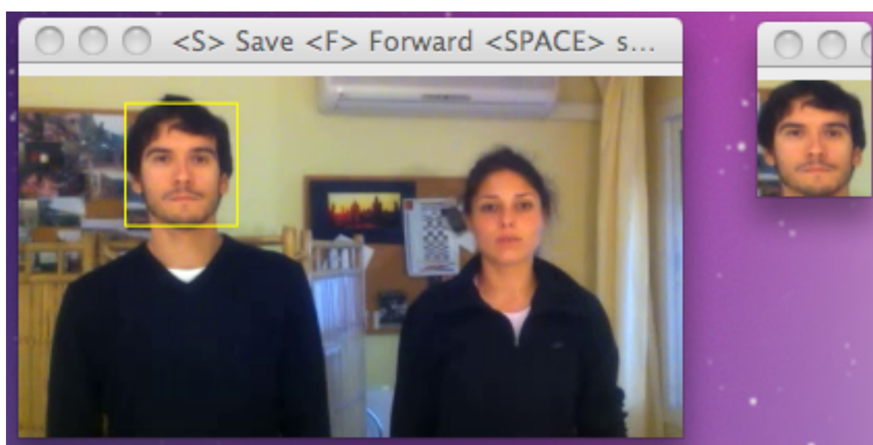


Figura 8.- Etiquetado de patrones con imageclipper

El nombre de los archivos generados a partir de *imageclipper* tiene el siguiente formato:

NombreArchivoVideo_NúmeroImagen_CoordX_CoordY_Ancho_Alto.png

Aprovechando esta circunstancia, es posible generar un archivo de texto con la información necesaria sobre los patrones a detectar a partir del nombre de los archivos de los recortes de los patrones presentes en el vídeo. Para ello, se debe generar una imagen para cada una de las detecciones que un determinado detector debería realizar.

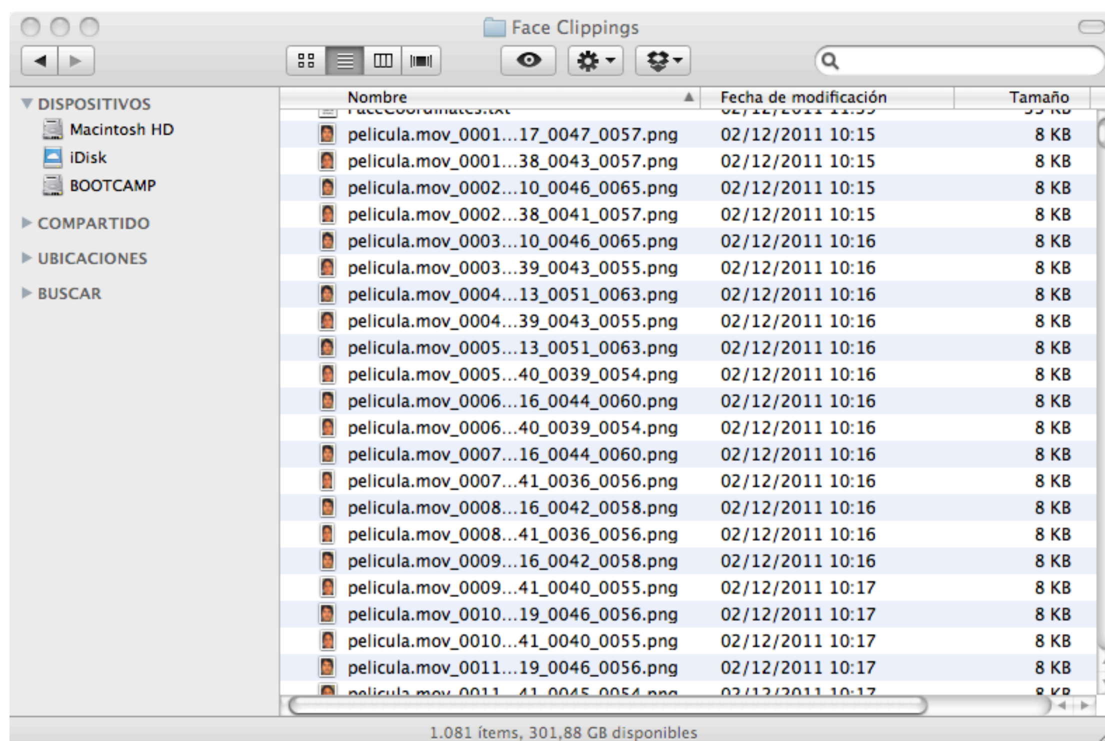


Figura 9.- Recortes generados por imageclipper

Una vez obtenidos los recortes de todos los patrones presentes en el vídeo, el archivo de ground truth con los datos de todas las áreas seleccionadas podrá ser creado a partir del nombre de todos los archivos mediante el siguiente comando.

```
$> find *_*_* -exec basename \{\} \; | perl -pe
's/([^_]*).*_0*(\d+)_0*_0*(\d+)_0*(\d+)_0*(\d+)_0*(\d+)\.[^.]*$/$2 $3 $4 $5
$6\n/g' | tee nombre_de_archivo_coordenadas.txt
```

Esto genera un archivo de texto con los siguientes campos separados por un espacio en blanco:

- Número de imagen.
- Coordenada x.
- Coordenada y.
- Anchura.
- Altura.

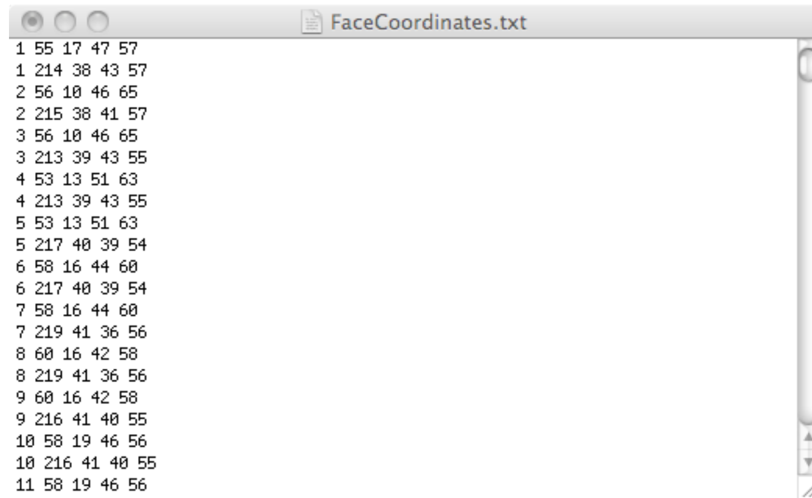

FaceCoordinates.txt
```
1 55 17 47 57
1 214 38 43 57
2 56 10 46 65
2 215 38 41 57
3 56 10 46 65
3 213 39 43 55
4 53 13 51 63
4 213 39 43 55
5 53 13 51 63
5 217 40 39 54
6 58 16 44 60
6 217 40 39 54
7 58 16 44 60
7 219 41 36 56
8 60 16 42 58
8 219 41 36 56
9 60 16 42 58
9 216 41 40 55
10 58 19 46 56
10 216 41 40 55
11 58 19 46 56
```

Figura 10.- Archivo de texto con ground truth del detector facial

Puesto que se pretenden analizar 4 detectores (de cara, de cuerpo completo, de la parte superior del cuerpo y de la parte inferior del cuerpo), se repite este proceso para cada uno de ellos. Los archivos obtenidos para cada uno de los detectores puede encontrarse en el apartado *ground truth* del anexo multimedia adjunto a esta memoria.

2.4. Estudio del rendimiento de detectores sobre una secuencia de vídeo

Para realizar el estudio del rendimiento de los detectores aplicados sobre una secuencia de vídeo se necesita crear una aplicación capaz de leer dicha secuencia y aplicar a cada una de las imágenes que la componen un determinado detector. La aplicación debe, además, generar un archivo de texto con los datos de las detecciones realizadas de manera que pueda ser comparado con el archivo *ground truth* de ese detector.

Es importante que el formato del archivo de texto generado, con los datos de las detecciones realizadas, sea el mismo que el empleado en el archivo *ground truth*, para que posteriormente puedan ser comparados y analizados.

NúmeroImagen – Coordenada X – Coordenada Y – Anchura – Altura

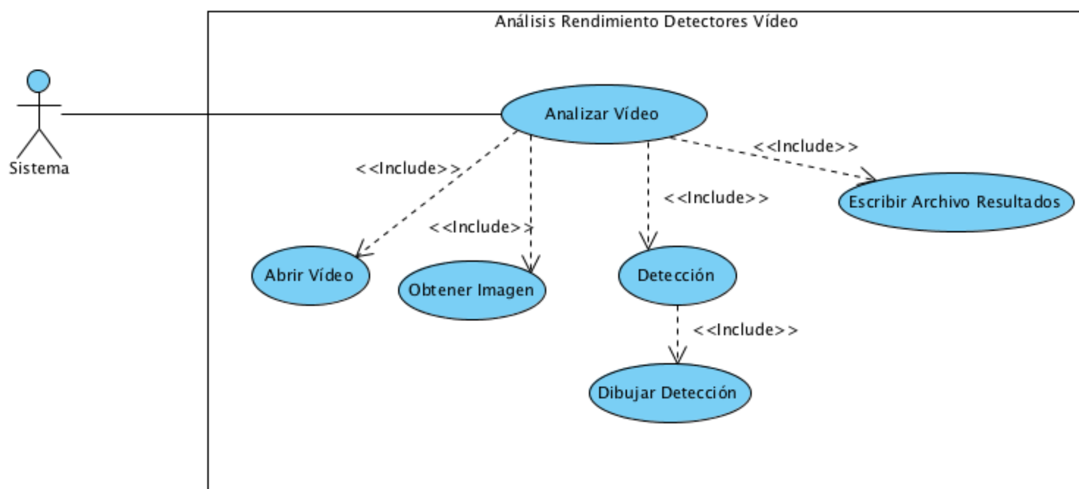


Figura 11.- Diagrama de casos de uso de la aplicación de análisis

En líneas generales, el funcionamiento de la aplicación, tal y como se deduce del diagrama de casos de uso anterior, debe ser similar a lo siguiente:

1. Abrir y realizar los tratamientos necesarios para poder leer el vídeo.
2. Obtener una a unas las distintas imágenes que componen la secuencia del vídeo.
3. Aplicar el detector a la imagen obtenida.
4. Dibujar en la imagen la detección realizada por el detector.
5. Escribir en un archivo de texto los datos de las detecciones realizadas para su posterior estudio.

2.5. Análisis de los resultados

Una vez que se hayan obtenido los resultados de los patrones detectados al aplicar el detector a la secuencia de vídeo y los patrones reales (*ground truth*), será necesario comparar estos datos para elaborar un informe sobre el rendimiento de los detectores analizados.

Para ello, se deberá crear una aplicación que, a partir de los dos archivos de texto –los resultados de la detección y *ground truth*— determine el número de detecciones correctas (positivos verdaderos), detecciones erróneas (falsos positivos) y patrones no detectados (falsos negativos).

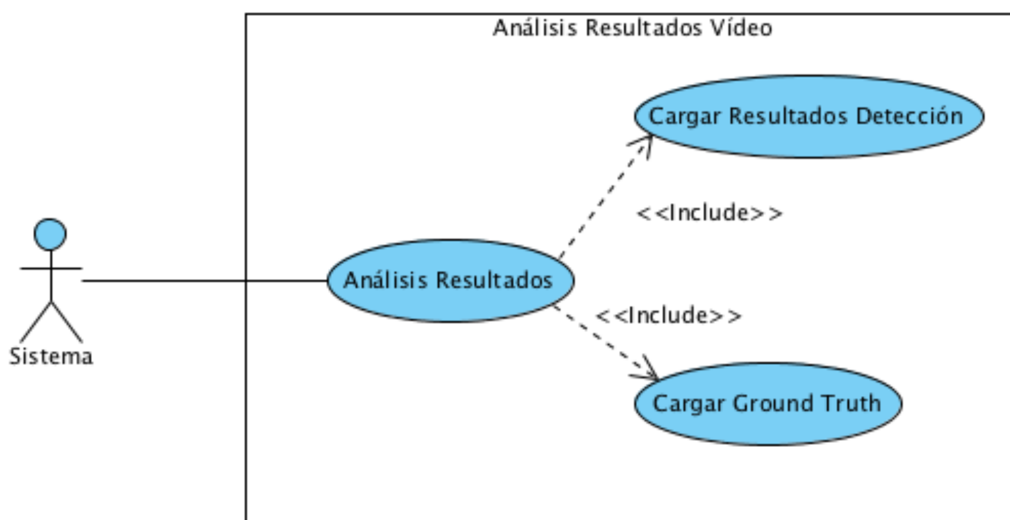


Figura 12.- Diagrama de casos de uso de la aplicación de análisis de resultados

La aplicación cargará en una estructura de datos la información contenida en los archivos con los resultados del detector aplicado a la secuencia de vídeo y del *ground truth*, y comparará los resultados de ambos archivos. Una vez analizados, presentará un informe con los siguientes datos:

- Total de positivos: número total de patrones en el vídeo.
- Positivos verdaderos: número de patrones detectados correctamente por el detector.
- Falsos positivos: número de patrones detectados por el detector incorrectamente (el patrón no existe en la realidad).
- Falsos negativos: número de patrones no detectados por el detector.

- Sensibilidad (o *sensitivity*): medida que hace referencia a la capacidad del detector de detectar positivos. De manera ideal, si un detector tuviera una sensibilidad del 100%, significaría que se podría estar completamente seguro de que los positivo detectados son verdaderos. Se define como el cociente entre el número de positivos verdaderos y la suma de los positivos verdaderos y falsos negativos.

$$\text{sensibilidad} = \frac{\text{número de positivos verdaderos}}{\text{número de positivos verdaderos} + \text{número de falsos negativos}}$$

Figura 13.- Fórmula de la sensibilidad

Una detección es considerada correcta cuando la intersección de las áreas del patrón detectado por el detector y el patrón *ground truth* es superior al 50% de la superficie de la unión de ambas superficies.

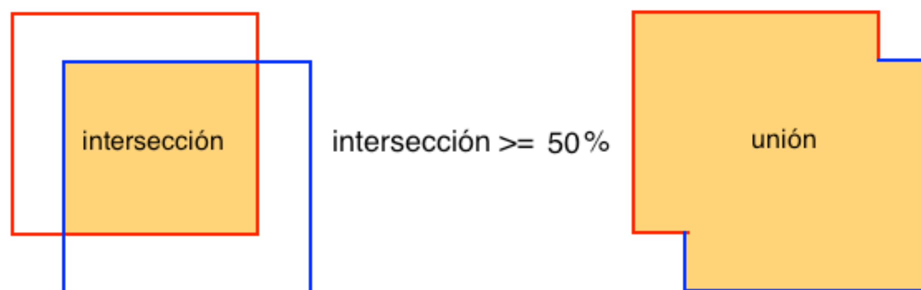


Figura 14.- Positivos verdaderos. Unión de las áreas.

2.6. Aplicación fotografía automática

Uno de los objetivos del proyecto consiste en realizar una aplicación directa de la visión por computador en los dispositivos móviles. En concreto, se realiza una aplicación que realice una fotografía cuando un determinado número de personas sea detectada a través de la cámara del dispositivo.

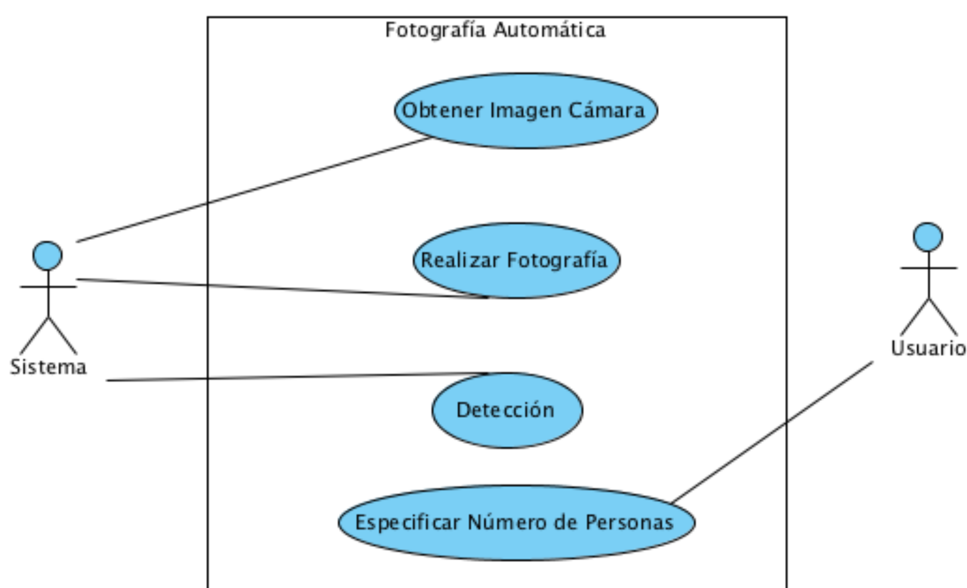


Figura 15.- Diagrama de casos de uso de la aplicación de fotografía automática

La aplicación, por tanto, deberá ser capaz de obtener la imagen obtenida por la cámara del iPhone 4, detectar el número de personas que aparecen en la imagen –calculando el número de caras presentes— y capturar la imagen si el número de personas es el que el usuario ha especificado a la aplicación. El número de personas a detectar será especificado por el usuario antes de tomar la fotografía.

3. Diseño e implementación

A lo largo de este capítulo se presenta el diseño e implementación de cada una de las aplicaciones necesarias para el desarrollo del proyecto, así como el modo de empleo de la librería *OpenCV* con la API de iPhone y su entorno de desarrollo.

3.1. Compilación de OpenCV para iPhone

Todo los detectores que se emplean en las aplicaciones definidas en el apartado de análisis son proporcionados e implementados por *OpenCV*, por tanto, es absolutamente necesario disponer de esta librería compilada y preparada para su uso en un *iPhone*.

A fecha de la realización de este proyecto no existe una versión oficial de la librería *OpenCV* para dispositivos *iOS*, tan sólo existen versiones oficiales para *Unix*, *Windows* y *Android*. Afortunadamente, las últimas versiones no estables de la librería ya ofrecen soporte para la mayoría de las funcionalidades de *OpenCV* en este entorno, por lo que actualmente es posible compilar con éxito la última versión de esta librería para este tipo de sistemas.

La compilación de esta librería se realiza a través de las siguientes 3 herramientas.

- Subversion: http://www.open.collab.net/downloads/community/
- CMake: http://www.cmake.org/cmake/resources/software.html
- Libtool: http://www.gnu.org/software/libtool/

A través de *subversion* se realiza la descarga de la última versión del código fuente de OpenCV mediante la siguiente instrucción.

```
svn co https://code.ros.org/svn/opencv/trunk .
```

Esto comando descarga y crea la estructura de archivos del código fuente de *OpenCV*, listos para ser compilados con *CMake*, el compilador usado por el equipo de *OpenCV*.

Nombre	Fecha de modificación	Tamaño	Clase
build	Hoy, 17:10	--	Carpeta
opencv	15/11/2011 17:10	--	Carpeta
3rdparty	15/11/2011 17:10	--	Carpeta
android	15/11/2011 17:04	--	Carpeta
cmake_uninstall.cmake.in	15/11/2011 17:10	4 KB	Archivo ejecutable Unix
CMakeLists.txt	15/11/2011 17:10	78 KB	Texto normal
cvconfig.h.cmake	15/11/2011 17:10	8 KB	Documento
data	15/11/2011 17:10	--	Carpeta
doc	15/11/2011 17:09	--	Carpeta
include	15/11/2011 17:05	--	Carpeta
index.rst	15/11/2011 17:10	4 KB	Documento
ios	15/11/2011 17:05	--	Carpeta
modules	15/11/2011 17:45	--	Carpeta
opencv-XXX.pc.cmake.in	15/11/2011 17:10	4 KB	Documento
OpenCV.mk.in	15/11/2011 17:10	4 KB	Documento
opencv.pc.cmake.in	15/11/2011 17:10	4 KB	Archivo ejecutable Unix
OpenCVAndroidProject.cmake	15/11/2011 17:10	8 KB	Documento
OpenCVConfig-version.cmake.in	15/11/2011 17:10	4 KB	Documento
OpenCVConfig.cmake.in	15/11/2011 17:10	12 KB	Documento
OpenCVFindIPP.cmake	15/11/2011 17:10	12 KB	Documento
OpenCVFindLATEX.cmake	15/11/2011 17:10	4 KB	Documento
OpenCVFindOpenEXR.cmake	15/11/2011 17:10	4 KB	Documento
OpenCVFindOpenNI.cmake	15/11/2011 17:10	4 KB	Documento
OpenCVFindPkgConfig.cmake	15/11/2011 17:10	16 KB	Documento
OpenCVFindXimea.cmake	15/11/2011 17:10	4 KB	Archivo ejecutable Unix
OpenCVModule.cmake	15/11/2011 17:10	12 KB	Documento
OpenCVPCHSupport.cmake	15/11/2011 17:10	12 KB	Documento
README	15/11/2011 17:10	4 KB	Documento
samples	15/11/2011 17:06	--	Carpeta
opencv_extra	15/11/2011 17:15	--	Carpeta
3d	15/11/2011 17:12	--	Carpeta
classifiers	15/11/2011 17:13	--	Carpeta
gpu_demos_pack	15/11/2011 17:11	--	Carpeta
learning_opencv_v2	15/11/2011 17:12	--	Carpeta
mini_releases	15/11/2011 17:10	--	Carpeta
testdata	15/11/2011 17:21	--	Carpeta

Figura 16.- Estructura del código fuente de OpenCV

Aunque no es estrictamente necesario, las distintas librerías generadas durante el proceso de compilación pueden ser agrupadas mediante la herramienta *libtool* en una única librería (llamada *framework por XCode, el entorno de programación para dispositivos iOS)*, que podrá ser añadida fácilmente a un proyecto de *XCode* para su posterior uso.

El proceso de compilación del código fuente se realiza a través de un script (incluido en el directorio *compilación* del anexo), que se encarga de compilar todos los fuentes y generar las librerías usando el compilador CMake, para a continuación, agruparlas en un *framework* mediante la herramienta *libtool*.

```
> opencvbuild.sh opencv build
```

Nombre	Fecha de modificación	Tamaño
OpenCV_iPhoneOS	Hoy, 16:44	--
include	Hoy, 16:46	--
lib	Hoy, 16:45	--
libopencv_calib3d.a	15/11/2011 17:35	750 KB
libopencv_contrib.a	15/11/2011 17:35	471 KB
libopencv_core.a	15/11/2011 17:35	2,3 MB
libopencv_features2d.a	15/11/2011 17:35	1,1 MB
libopencv_flann.a	15/11/2011 17:35	504 KB
libopencv_highgui.a	15/11/2011 17:35	254 KB
libopencv_imgproc.a	15/11/2011 17:35	2,1 MB
libopencv_legacy.a	15/11/2011 17:35	975 KB
libopencv_ml.a	15/11/2011 17:35	594 KB
libopencv_objdetect.a	15/11/2011 17:35	418 KB
libopencv_video.a	15/11/2011 17:35	254 KB
pkgconfig	15/11/2011 17:35	--
share	Hoy, 16:44	--
OpenCV_iPhoneSimulator	Hoy, 16:48	--
OpenCV_Universal	Hoy, 16:41	--
OpenCV.framework	Hoy, 16:41	--
Headers	Hoy, 16:42	--
opencv	15/11/2011 17:35	--
opencv2	Hoy, 16:42	--
OpenCV	15/11/2011 17:35	20,1 MB

Figura 17.- Librerías compiladas

Al finalizar el proceso de compilación se crean una serie de carpetas donde se encuentran las diferentes versiones de las librerías y cabeceras de *OpenCV*. En concreto:

- OpenCV_iPhoneOS	Librerías y cabeceras para iPhone.
- OpenCV_iPhoneSimulator	Librerías y cabeceras para el simulador de iPhone incluido en XCode.
- OpenCV_Universal	Librerías y cabeceras para dispositivos iOS.
- OpenCV.framework	Framework para dispositivos iOS.

3.1.1. Uso de OpenCV en un proyecto XCode

Para poder usar estas librerías en un proyecto *XCode*, tan sólo es necesario arrastrar el directorio *OpenCV.framework* dentro de la carpeta *Frameworks* del proyecto *XCode*.

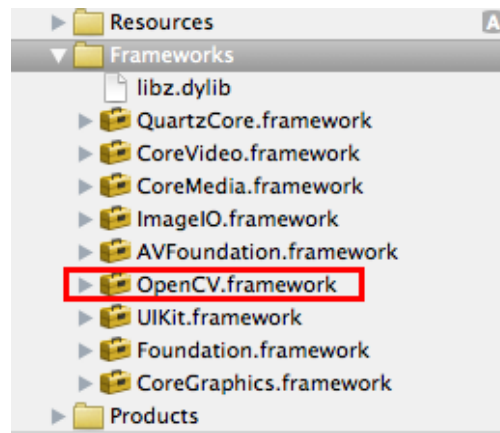


Figura 18.- Framework de OpenCV dentro de un proyecto Xcode

Además, para el correcto funcionamiento de *OpenCV*, es necesario incluir en el proyecto una serie de librerías y *frameworks* de los que depende directamente OpenCV.

- AVFoundation.framework
- ImageIO.framework
- libz.dylib
- CoreVideo.framework
- CoreMedia.framework

Por otro lado, se debe incluir la cabecera de *OpenCV* en el archivo de prefijo de cabecera del proyecto para que todos los archivos del proyecto hagan referencia a esta cabecera y puedan usar sus utilidades.

```
//
// Prefix header for all source files of the 'VideoTestCV' target in the
// 'VideoTestCV' project


#import <Availability.h>

#ifndef __IPHONE_4_0
#warning "This project uses features only available in iOS SDK 4.0 and later."
#endif

#ifdef __cplusplus
#import <OpenCV/opencv2/opencv.hpp>
#endif

#ifdef __OBJC__
    #import <UIKit/UIKit.h>
    #import <Foundation/Foundation.h>

#endif
```

Por último, puesto que *OpenCV* ha sido desarrollada en *C++*, se deben configurar todos los archivos fuente que hagan uso de *OpenCV* para que sean compilados

con el compilador de C++. Esto se consigue, simplemente, renombrando la extensión de los archivos a .mm.

3.2. Implementación de la detección

Las aplicaciones creadas en el proyecto hacen uso de la implementación del algoritmo de detección de patrones de Viola-Jones incluido en OpenCV. Aunque hay ligeras diferencias entre las aplicaciones debido a que el origen de la imagen es diferente en cada una de ellas –una de las aplicaciones usa la cámara para obtener las imágenes mientras que otra las obtiene de un archivo de vídeo—, el proceso de detección es muy similar en todas ellas y, en líneas generales, puede ser descrito en los siguientes pasos:

1. Carga del detector.
2. Obtención de la imagen.
3. Conversión de la imagen al formato de OpenCV (cv::Mat).
4. Conversión a escala de grises.
5. Detección (llamada al detector de OpenCV).
6. Tratamiento de los resultados.

Las aplicaciones del proyecto tan solo difieren en la manera en la que obtienen la imagen a procesar y en el posterior tratamiento que hacen de los resultados de la detección (puntos 2 y 6), pero coinciden en el resto de puntos del proceso. A continuación, se detallan en mayor profundidad los puntos comunes a las aplicaciones (1, 3, 4 y 5).

3.2.1. Carga del detector

Uno de los atributos de la clase encargada de la detección deberá ser el detector o clasificador que se encarga en última instancia de realizar la detección. Este detector es un objeto de la clase *CascadeClassifier* y es instanciado durante la inicialización de la aplicación.

```
@interface ClaseDetector : UIViewController {
    …
    cv::CascadeClassifier faceCascade;
    …
 }
…
@end
```

Para inicializar este objeto tan sólo se necesita llamar al método *load* de esta clase indicándole la ruta en la que se encuentra el archivo XML con el que se define el clasificador. Todos los detectores empleados en el proyecto usan las definiciones XML incluidas en la propia librería *OpenCV*.

```
// Load face cascade classifier
NSString *faceCascadePath = [[NSBundle mainBundle]
     pathForResource:kFaceCascadeFilename ofType:@"xml"];

if (!faceCascade.load([faceCascadePath UTF8String])) {
    NSLog(@"Could not load face cascade: %@", faceCascadePath);
}
```

Las aplicaciones *iPhone* son empaquetadas en un *bundle* o *paquete* junto a todos los recursos que necesitan durante su ejecución. El archivo XML con la definición del clasificador es incluido como un recurso más de la aplicación dentro de este

paquete. La primera sentencia obtiene la ruta del archivo XML dentro de este paquete, mientras que la segunda crea el clasificador a partir de este archivo de definición.

3.2.2. Conversión de la imagen al formato de OpenCV

Las limitaciones de los recursos del dispositivo no sólo afectan a la capacidad de procesamiento, sino también a la memoria disponible para el programa. Por tanto, es muy importante que todos los subprocesos de los que consta un programa optimicen el uso de memoria.

En el caso de las aplicaciones de este proyecto, el almacenamiento en memoria de las imágenes captadas por la cámara del dispositivo representan la mayor parte del consumo de memoria de la aplicación. Sin embargo, el formato en el que se obtienen las imágenes de la cámara debe ser convertido antes de poder ser procesado por el detector de *OpenCV*. Es por ello, que resulta de suma importancia que se optimice el tratamiento de estas imágenes de manera que no se dupliquen datos innecesarios y se reduzca el coste de esta conversión.

Cocoa Touch permite seleccionar el formato en el que se obtienen las imágenes de la cámara, aunque por cuestiones de rendimiento, Apple recomienda el uso de formato BGRA. Esto significa que cada píxel se define a través de 4 componentes básicos: azul, verde, rojo y alfa, cada uno de ellos representados por 1 byte. Éste es también uno de los formatos soportados por *OpenCV* y, por tanto, el formato elegido para las aplicaciones a desarrollar.

No obstante, la API *Cocoa Touch* encapsula la imagen obtenida de la cámara en uno de sus formatos propios de imagen: *CMSampleBufferRef*; mientras que la librería *OpenCV* trabaja con un formato propio: *cv::Mat*, que no es más que el formato que *OpenCV* utiliza para representar cualquier tipo de matriz. Para que la imagen pueda ser procesada por el clasificador, es necesario por tanto, convertir previamente la imagen del formato de *Cocoa Touch* al de *OpenCV*.

Puesto que la aplicación procesa las imágenes una a una, no tiene sentido crear una nueva variable para cada nueva imagen que se obtenga de la cámara, sino que se utiliza siempre la misma variable de tipo *cv::Mat* para guardar la imagen correspondiente al marco actual de la secuencia de vídeo –tanto si ésta ha sido obtenida directamente de la cámara, como si ha sido obtenida desde un archivo de vídeo—. Se define, entonces, un atributo en la clase encargada de llamar al detector de OpenCV que contenga esta imagen.

```
@interface ClaseDetector : UIViewController {
    …
    cv::Mat frameImage;
    …
 }
…
@end
```

La estructura de datos *CMSampleBufferRef* contiene la imagen encapsulada en otra estructura de datos adicional del tipo *CVImageBufferRef*. Esta última estructura contiene varios atributos a los que se accede a través diferentes funciones. Las funciones y atributos de interés para realizar la conversión son:

- *CVPixelBufferGetBaseAddress*: Obtiene la dirección de memoria que contiene los píxeles de la imagen en el formato determinado previamente (BGRA).
- *CVPixelBufferGetWidth*: Obtiene la anchura de la imagen.
- *CVPixelBufferGetHeight*: Obtiene la altura de la imagen.

Por otro lado, el atributo *data* de la clase *cv::Mat* es el que contiene la matriz de píxeles que compone la imagen. Por tanto, este atributo deberá apuntar a la dirección de memoria donde estos datos estaban almacenados en la estructura *CVPixelBufferRef*.

```
- (void)createMatImageFromSampleBuffer:(CMSampleBufferRef)sampleBuffer {

  if (sampleBuffer) {
  CVImageBufferRef imageBuffer = CMSampleBufferGetImageBuffer(sampleBuffer);
  CVPixelBufferLockBaseAddress(imageBuffer, 0);

  // get information of the image in the buffer
  void *bufferBaseAddress = CVPixelBufferGetBaseAddress(imageBuffer);
  size_t bufferWidth = CVPixelBufferGetWidth(imageBuffer);
  size_t bufferHeight = CVPixelBufferGetHeight(imageBuffer);

  // create mat image BGRA
  if (!frameImage.data) {
    frameImage = cv::Mat(bufferHeight, bufferWidth, CV_8UC4);
  }

  if (bufferBaseAddress) {
    frameImage.data = (uchar*)bufferBaseAddress;
  }

  // release memory
  CVPixelBufferUnlockBaseAddress(imageBuffer, 0);
 }
}
```

De esta manera, la conversión es realizada sin duplicar los datos de la estructura de origen en la estructura de destino, reduciéndose al máximo el consumo de memoria y el coste de procesamiento en el proceso de conversión.

3.2.3. Conversión a escala de grises

La imagen que se obtiene de la cámara está en formato BGRA. Esto significa que el color de cada uno de los píxeles de la imagen se representa como una combinación de 3 colores básicos: azul (B), verde (G) y rojo (R). Junto a estos 3 colores se define, además, un cuarto componente llamado alfa (A), que define la transparencia del color. Tanto en el caso en que la imagen se capture desde la cámara como desde un archivo de vídeo, alfa se define siempre como no transparente.

Por tanto, cada píxel de la imagen está representado por 4 bytes, cada uno de los cuales se corresponde con uno de los componentes del formato BGRA. En el formato de escala de grises, sin embargo, cada píxel es representado por un único valor, el de su luminosidad. Este valor se puede obtener a partir del valor de sus componentes RGB. Existen diversos métodos y fórmulas para realizar la conversión a escala de grises. En este proyecto se ha aplicado la siguiente fórmula a cada uno de los píxeles que contiene la imagen.

$$\text{Píxel gris} = 0.2126\ \text{Rojo} + 0.7152\ \text{Verde} + 0.0722\ \text{Azul}$$

Esta conversión puede hacerse de manera trivial recorriendo todos los píxeles de la imagen y aplicando de manera sucesiva esta fórmula a cada uno de ellos. Sin embargo, aprovechando la capacidad de procesado del motor SIMD NEON[2] del que dispone el iPhone, se puede realizar una implementación mucho más eficiente para este dispositivo.

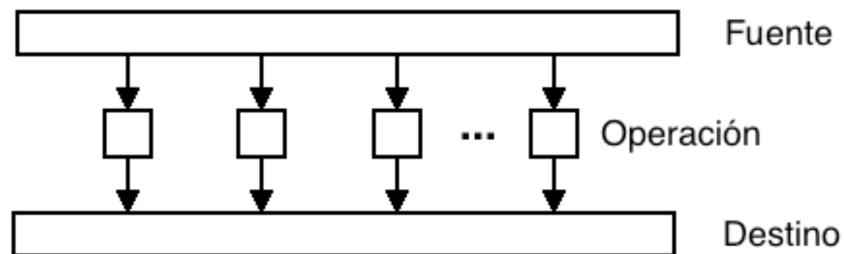


Figura 19.- Motor NEON SIMD

Este motor permite procesar múltiples bytes con una sola instrucción, lo que permitirá acelerar en gran medida el proceso de conversión a escala de grises. En concreto, es posible procesar 8 píxeles de la imagen en cada iteración. De esta manera se recorre la imagen de 8 en 8 píxeles, aplicando la formula de conversión directamente a todos estos píxeles simultáneamente. Esto supone una importante aceleración en la conversión.

```
int n = numero_pixeles / 8;
for (Int i = 0; i < n; ++i)
{
        buffer_8_pixeles = obtener_8_pixeles(puntero_imagen_fuente);
        multiplicar(buffer_8_pixeles[0] * factor_conversion_azul) ;
        multiplicar(buffer_8_pixeles[1] * factor_conversion_verde);
        multiplicar(buffer_8_pixeles[2] * factor_conversion_rojo);
        puntero_imagen_fuente = puntero_imagen_fuente + 8 * 4;
}
```

Puesto que el algoritmo de conversión es muy sencillo, se ha realizado una implementación del mismo en ensamblador para conseguir una máxima optimización del proceso.

```
static void neon_asm_convert(uint8_t * __restrict dest, uint8_t * __restrict
src, int numPixels)
{
__asm__ volatile("lsr          %2, %2, #3      \n"
 "# build the three constants: \n"
 "mov          r4, #28         \n" // Blue channel multiplier
 "mov          r5, #151        \n" // Green channel multiplier
 "mov          r6, #77         \n" // Red channel multiplier
 "vdup.8       d4, r4          \n"
 "vdup.8       d5, r5          \n"
 "vdup.8       d6, r6          \n"
 "0:                           \n"
 "# load 8 pixels:             \n"
 "vld4.8       {d0-d3}, [%1]!  \n"
 "# do the weight average:     \n"
 "vmull.u8     q7, d0, d4      \n"
 "vmlal.u8     q7, d1, d5      \n"
```

[2] SIMD son las siglas de *Single Instruction Multiple Data*, que hace referencia a la capacidad del procesador de aplicar una misma operación o instrucción sobre múltiples datos de manera simultanea.

```
  "vmlal.u8     q7, d2, d6        \n"
  "# shift and store:             \n"
  "vshrn.u16    d7, q7, #8        \n" // Divide q3 by 256 and store in the d7
  "vst1.8       {d7}, [%0]!       \n"
  "subs         %2, %2, #1        \n" // Decrement iteration count
  "bne          0b             \n" // Repeat unil iteration count is not zero
  :
  : "r"(dest), "r"(src), "r"(numPixels)
  : "r4", "r5", "r6"
  );
}
```

Ya que esta conversión se debe realizar para todas las imágenes a procesar, se crea un atributo en la clase encargada de la detección que contendrá la versión en escala de grises de la imagen obtenida de la secuencia.

```
@interface ClaseDetector : UIViewController {
    …
    cv::Mat grayImage;
    …
 }
…
@end
```

Esta imagen tendrá el mismo tamaño que la imagen almacenada en el atributo *frameImage*. Sin embargo, en lugar de necesitar 4 bytes para cada píxel, necesitará solamente uno, esto se indica definiendo el parámetro correspondiente con la constante CV_8UC1.

```
…
grayImage = cv::Mat(frameImage.size().height,
                    frameImage.size().width,
                    CV_8UC1);
…
```

La conversión es realizada invocando a la función definida anteriormente, *neon_to_gray*.

```
neon_to_gray(frameImage,grayImage);
```

3.2.4. Detección

Como se ha visto en el apartado 2.1.1, el algoritmo de Viola-Jones va recorriendo la imagen a través de una subventana, cuyo tamaño se incrementa en cada iteración. Esto significa que el número de iteraciones depende directamente del tamaño de la imagen y del tamaño de la subventana. Para conseguir un buen rendimiento del detector será necesario, por tanto, encontrar un buen equilibrio entre el tamaño de la imagen y el tamaño de la subventana.

En el caso de que la imagen haya sido obtenida directamente desde la cámara del dispositivo, su resolución es excesiva teniendo en cuenta las limitaciones de memoria existentes. Por ello, se decide reducir el tamaño de la imagen antes de procesar la imagen con el detector.

Para ello, se crea un atributo en la clase encargada de la detección, que contendrá la versión reducida de la imagen obtenida de la cámara.

```
@interface ClaseDetector : UIViewController {
    …
    double scale;
```

```
    cv::Mat smallImage;
    …
 }
…
@end
```

El tamaño de la imagen reescalada se define a través del atributo *scale* que es inicializado con un valor de 3.25. Puesto que la conversión se realiza a partir de la imagen en escala de grises, sólo se necesita un byte por píxel (CV_8UC1).

```
…
smallImage = cv::Mat(frameImage.size().height/scale,
                     frameImage.size().width/scale,
                     CV_8UC1);
…
```

El reescalado de la imagen se realiza a través del método *resize* proporcionado también por *OpenCV*.

```
…
cv::resize(grayImage,
           smallImage,
           cv::Size(smallImage.size().width,
           smallImage.size().height));

…
```

En el caso de que la imagen haya sido obtenida a partir de una secuencia de vídeo, la aplicación no tiene que preocuparse del tamaño de la imagen, puesto que ésta ya ha sido reducida en el vídeo a una resolución óptima.

```
…
cv::resize(grayImage,
           smallImage,
           cv::Size(smallImage.size().width,
           smallImage.size().height));

…
```

Una vez que la imagen ha sido reducida y convertida a escala de grises estará lista para ser procesada por el clasificador. El método *detectMultiScale* de la clase *CascadeClassifier* es el encargado de implementar el algoritmo de Viola-Jones.

```
faceCascade.detectMultiScale(smallImage,
                             faces,
                             1.2f,
                             2,
                             0 | CV_HAAR_DO_CANNY_PRUNING,
                             cv::Size(20, 20));
```

A continuación se detallan los distintos parámetros de este método y los valores que se le han pasado a cada uno de ellos.

- **image**: Parámetro de entrada con la imagen a procesar. Se le pasa como parámetro la imagen reducida y convertida a escala de grises almacenada en la variable *smallImage*.

- **objects**: Parámetro de salida con el vector de objetos detectados. Se crea una vector de tipo *cv::Rect* (rectángulo) que al término de la detección contendrá los objetos detectados por el clasificador.

  ```
  std::vector<cv::Rect> faces;
  ```

- **scaleFactor**: Parámetro de entrada que define el factor con el que en cada iteración se incrementará la subventana con la que se recorre la imagen. Tras las pruebas realizadas se ha encontrado que 1.2 es un factor óptimo.

- **minNeighbors**: Parámetro de entrada que define el umbral de detecciones "vecinas" a partir del cual se considera una detección como positiva.

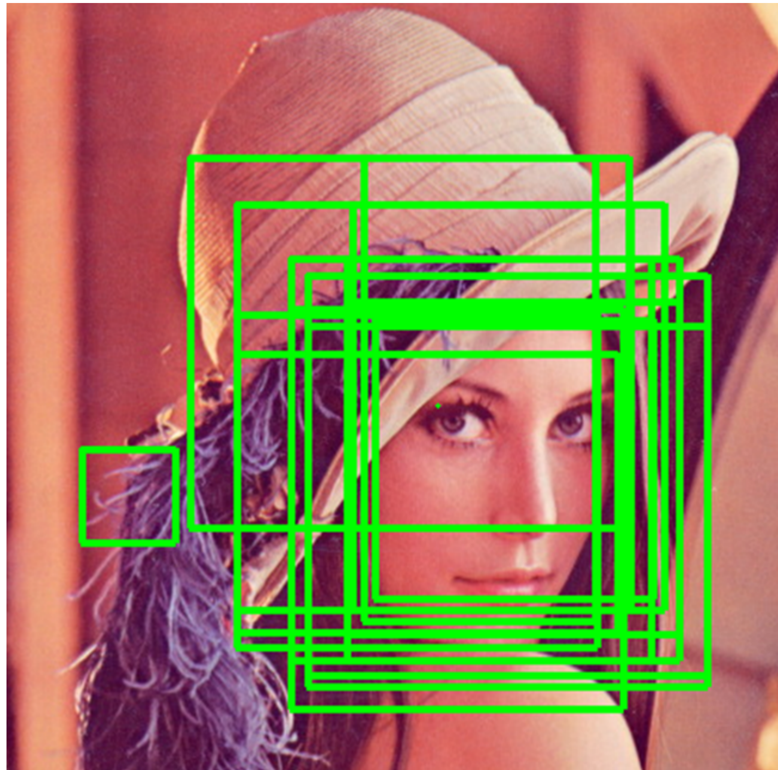


Figura 20.- Ejemplo de todas las detecciones provocadas en la detección facial

Normalmente, la presencia del patrón a detectar no genera un único positivo, sino que genera muchas detecciones. Por este motivo, las detecciones aisladas suelen considerarse como errores y son descartadas. Este parámetro define el número mínimo de detecciones que un patrón debe provocar para que sea considerado como una detección.

- **flags**: Parámetro de entrada que define el modo de operación del detector. Existen 4 modos de operación que pueden ser combinados o usados de manera independiente.

    - **CV_HAAR_DO_CANNY_PRUNING**: El clasificador usa un detector *Canny Edge* (detector de bordes o aristas) para descartar aquellas zonas que contengan muy pocas o demasiadas aristas como para que el patrón a detectar esté presente en ellas.
    - **CV_HAAR_SCALE_IMAGE**: En lugar de aumentar el tamaño del detector según la escala definida en *scaleFactor* en cada iteración, se reduce el tamaño de la imagen según este factor de escala.

- **CV_HAAR_FIND_BIGGEST_OBJECT**: El detector sólo tiene en cuenta el mayor objeto presente en la imagen. Esto quiere decir que sólo encontrará un objeto como máximo.
- **CV_HAAR_DO_ROUGH_SEARCH**: Sólo se puede usar cuando está activo el modo CV_HAAR_FIND_BIGGEST_OBJECT y el número mínimo de vecinos es superior a cero. En este caso, el detector no buscará otros objetos de menor tamaño, en cuanto encuentre uno con el suficiente número de vecinos dejará de analizar la imagen.

Tras realizar pruebas con todos los modos de operación, se encuentra que los resultados óptimos, en términos de velocidad y número de detecciones correctas, se consiguen utilizando únicamente la opción de CV_HAAR_DO_CANNY_PRUNNING.

- **minSize**: Parámetro que indica el tamaño mínimo de la subventana con la que se recorrerá la imagen. El tamaño de esta subventana se incrementará en cada iteración aplicándole el valor definido en *scaleFactor*.

- **maxSize**: Parámetro que indica el tamaño máximo de la subventana. comenzando con el tamaño definido en *minSize*, el tamaño de la subventana se incrementa en cada iteración hasta alcanzar el tamaño definido en este parámetro. Se ha optado por dejar el valor por defecto que se corresponde con el tamaño de la imagen.

3.3. Análisis del rendimiento de detectores sobre una secuencia de vídeo

Los resultados de la detección de patrones corporales dependen en gran medida de las imágenes sobre las que se aplican. Si las condiciones de estas imágenes cambian continuamente, será difícil conocer de manera objetiva el verdadero rendimiento de estos detectores y compararlos entre sí. Por ello, se ha desarrollado una aplicación que aplica estos detectores sobre una misma secuencia de vídeo, de manera que los resultados de todos los detectores se obtengan bajo unas mismas condiciones objetivas.

La API (*Cocoa Touch*) de *iPhone* ha sido diseñada para que todas las aplicaciones desarrolladas con ella sigan el patrón MVC (modelo-vista-controlador), por lo que todas las aplicaciones desarrolladas en este proyecto siguen este patrón. Así, la aplicación de análisis de vídeo está compuesta principalmente por una única vista encargada de mostrar el vídeo con las detecciones realizadas por el detector junto a la detección *ground truth*; y por un controlador que, a través de las clases proporcionadas por *Cocoa Touch* para el tratamiento de vídeo, se encarga de obtener individualmente todas las imágenes que componen la secuencia de vídeo y aplicar sobre ellas los detectores de patrones corporales, dibujando al mismo tiempo el resultado de las detecciones y el *ground truth* del detector correspondiente.

3.3.1. Modelo

Además de representar las detecciones realizadas por el detector, la aplicación debe ser capaz de representar la información *ground truth* con la que se

compararán posteriormente los resultados. Para ello, la aplicación dispone de un archivo de texto con la información *ground truth* del vídeo. Este archivo tiene el siguiente formato.

Nº de marco en la secuencia	Coordenada X	Coordenada Y	Anchura	Altura

El archivo *ground truth* es cargado en una estructura de datos denominada *ResultSet*, que no es más que una colección de *Frames*, cada uno de los cuales dispone de un listado de detecciones, representadas por la clase *Detection*.

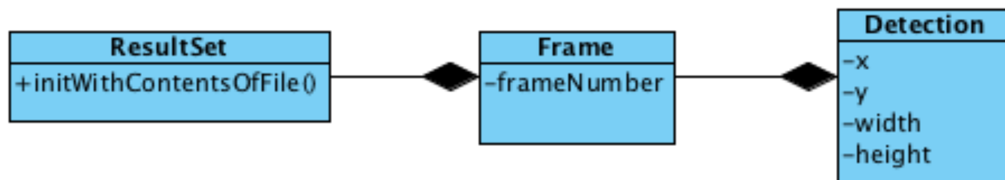


Figura 21.- Diagrama de clases de la representación del ground truth

Con esta información, la aplicación será capaz de dibujar en cada imagen del vídeo la detección *ground truth* correspondiente.

3.3.2. Vista

La única vista de la aplicación muestra la imagen del vídeo una vez que ha sido procesada por el detector. Esta imagen ya ha sido modificada para incluir la representación del *ground truth* y las detecciones realizadas.

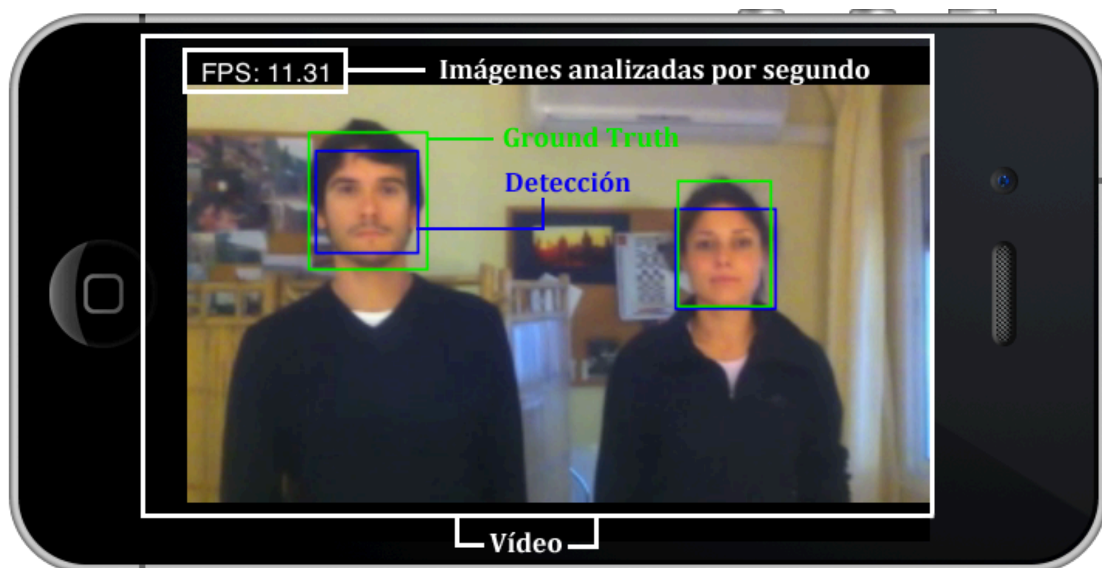


Figura 22.- Vista de la aplicación de análisis de detectores

En la parte superior de la vista se encuentra la velocidad de procesamiento del detector, indicada en marcos procesados por segundo (*FPS o frames por segundo*); es decir, el número de imágenes procesadas por segundo. Esta medida puede ser de utilidad a la hora de medir la eficiencia del detector.

3.3.3. Controlador

El único controlador de la aplicación (*VideoPlayerCV*) se encarga tanto de extraer las imágenes de la secuencia de vídeo, como de procesarlas y dibujar las detecciones realizadas por el detector de OpenCV. Tal y como se ha explicado en el apartado 3.2 y en sus respectivos subapartados, el proceso de detección es común en todas las aplicaciones. Las únicas diferencias se presentan en el origen de las imágenes y en el tratamiento de los resultados. En el caso de esta aplicación, las imágenes tienen como origen una secuencia de vídeo almacenada en un archivo, mientras que los resultados son dibujados en la imagen de origen que se mostrará por pantalla una vez analizada. Se detallan, a continuación, estos dos procedimientos.

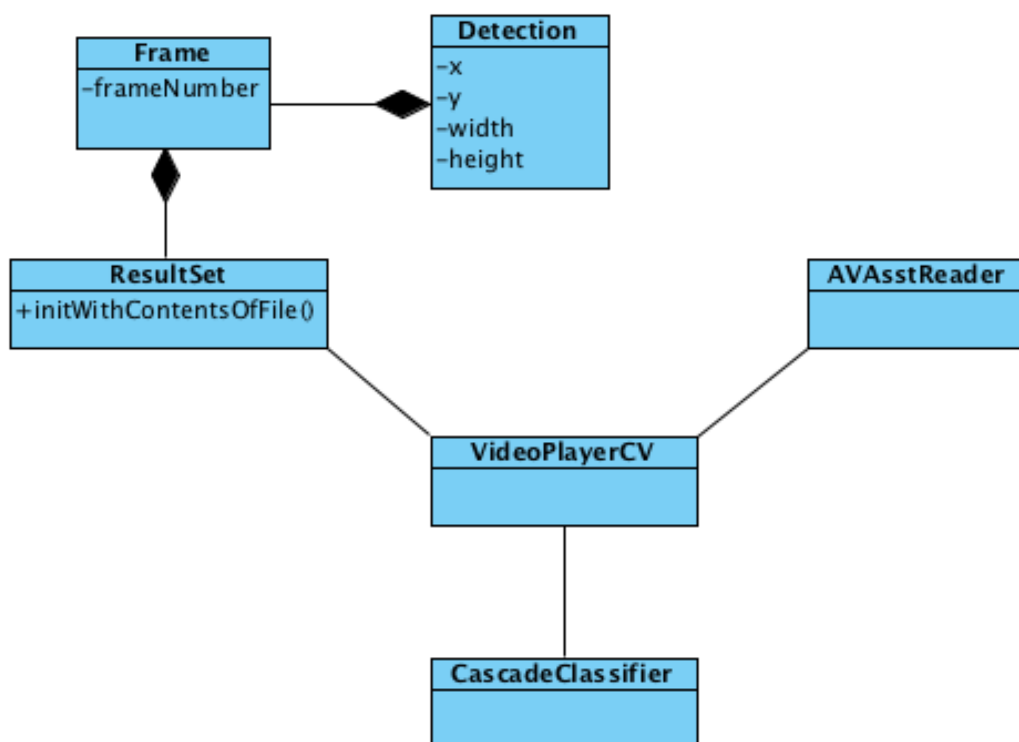


Figura 23.- Diagrama de clases de la aplicación de análisis de vídeo

3.3.3.1. Extracción de imágenes de la secuencia de vídeo

La API *Cocoa Touch* dispone de una serie de clases encapsuladas en el *framework AVFoundation* con las que es posible acceder y extraer las imágenes que componen una secuencia de vídeo.

Desde el punto de vista de *Cocoa Touch*, cualquier archivo multimedia puede ser considerado un *Asset* y ser tratado a través de la clase *AVAsset* y todas sus clases relacionadas. El método *playVideo*, invocado cuando la vista es cargada en pantalla, hace uso de estas clases para cargar el vídeo y prepararlo para su análisis.

```
- (void) playVideo
{
  if (self.videoFile != nil) {
    // input asset options
```

```
    NSDictionary *inputOptions = [NSDictionary dictionaryWithObject:
         [NSNumber numberWithBool:YES]
         forKey:AVURLAssetPreferPreciseDurationAndTimingKey];

    // init the input asset with that options and url
    self.inputAsset = [[AVURLAsset alloc] initWithURL:
           [NSURL fileURLWithPath:self.videoFile] options:inputOptions];

    // when the asset is loaded the following code is executed in another
    // thread
    [self.inputAsset loadValuesAsynchronouslyForKeys:
       [NSArray arrayWithObject:@"tracks"]
       completionHandler:^(void) {

        …

    }];
  }
}
```

Este método crea, en primer lugar, un *asset de entrada* a partir de la ruta que contiene el nombre del archivo de vídeo y lo almacena en el atributo *inputAsset* del controlador. A continuación, carga de forma asíncrona, mediante el método *loadValuesAsynchrouslyForKeys,* lo que *Cocoa Touch* conoce como los *tracks* del archivo multimedia, es decir, las pistas que forman el vídeo: la pista de sonido y la pista de vídeo o de imagen. Este método es algo peculiar puesto que como segundo parámetro tiene el código que se muestra a continuación y, que se ejecutará cuando el proceso de carga se haya completado.

```
// init the asset reader
self.reader = [[AVAssetReader alloc] initWithAsset:inputAsset error:NULL];

// get the video track
NSArray* video_tracks = [inputAsset tracksWithMediaType:AVMediaTypeVideo];
AVAssetTrack* video_track = [video_tracks objectAtIndex:0];

// image format
NSMutableDictionary* dictionary = [[NSMutableDictionary alloc] init];

[dictionary setObject:[NSNumber numberWithInt:kCVPixelFormatType_32BGRA]
               forKey:(NSString*)kCVPixelBufferPixelFormatTypeKey];

// image output asset
self.assetReaderOutput = [[AVAssetReaderTrackOutput alloc]
      initWithTrack:video_track outputSettings:dictionary];

[self.reader addOutput:self.assetReaderOutput];

// start reading frames
[self.reader startReading];

frameNumber = 0;

// display frames
[self displayProcessedVideo];
```

Este código se encarga de inicializar, a partir del *asset de entrada* definido anteriormente, el atributo *reader* del controlador, un objeto de la clase *AVAssetReader* a través del cual se extraerán las imágenes que forman el vídeo. A continuación, obtiene una referencia al objeto que representa la pista de vídeo del archivo multimedia y define BGRA como el formato en el que las imágenes serán extraídas del vídeo. Por último, define la pista de vídeo como la salida del

lector de assets e inicia la lectura del vídeo mediante una llamada al método *startReading*. Las imágenes del vídeo se extraen y procesan desde el método displayProcessedVideo, invocado en último lugar.

```
- (void) displayProcessedVideo
{
  CMSampleBufferRef sample;

  dispatch_queue_t playingThread = dispatch_queue_create("playingThread",
                                                  DISPATCH_QUEUE_SERIAL);

  … Cálculo FPS …

  // while video is not finished
  while ( [self.reader status]==AVAssetReaderStatusReading ) {

      // get the next frame
      sample = [self.assetReaderOutput copyNextSampleBuffer];

      // displaye the frame in another thread
      dispatch_sync(playingThread, ^(void){
          [self showFrame:sample];
      });

}

  … Cálculo FPS …

  dispatch_release(playingThread);
}
```

Este método obtiene (a través del método *copyNextSampleBuffer* del objeto lector de *assets*) una a una las imágenes del vídeo en uno de los formatos propios de Cocoa, *CMSampleBufferRef*, que como se ha visto anteriormente deberá ser previamente convertido antes de que pueda ser analizado por OpenCV. Una vez que la imagen ha sido extraída se procesa y se muestra en el método *showFrame*. Este método es invocado a través de *dispatch_sync* para que sea ejecutado en un hilo paralelo y acelerar así la velocidad de procesado del vídeo.

3.3.3.2. Tratamiento de los resultados

El procesado final de la imagen es llevado a cabo por el método *showFrame*, que se encarga de invocar el clasificador de *OpenCV* siguiendo los pasos detallados en el apartado 3.2 y representar los resultados obtenidos tras la detección.

```
- (void)showFrame: (CMSampleBufferRef) sample {

  [self createMatImageFromSampleBuffer:sample];

  neon_to_gray(frame, gray);

  std::vector<cv::Rect> faces;

  faceCascade.detectMultiScale(gray,
                               faces,
                               1.2f,
                               2,
                               0 | CV_HAAR_DO_CANNY_PRUNING,
                               cv::Size(20, 20));
  frameNumber++;

  …
```

```
}
```

Los resultados de la detección son devueltos en un vector de objetos *cv::Rect*, una clase de OpenCV que representa un rectángulo. La aplicación recorre este vector y los dibuja en la imagen de origen a través de otro método de *OpenCV*, *cv::rectangle*. Además, guarda un registro de las coordenadas y dimensiones de las detecciones para que los resultados puedan ser analizados posteriormente.

```
- (void)showFrame: (CMSampleBufferRef) sample {

 …
 for (int i = 0; i < faces.size(); i++) {
   cv::rectangle(frame,
                 cv::Point(faces[i].x,faces[i].y),
                 cv::Point(faces[i].x+faces[i].width,
                           faces[i].y+faces[i].height),
                 cv::Scalar(255,0,0));

   NSLog(@"%d %d %d %d %d",frameNumber,faces[i].x,faces[i].y,
                faces[i].width,faces[i].height);

 }
 …
}
```

Además de representar los objetos detectados, el controlador debe representar también las detecciones *ground truth*, de forma que en el propio vídeo pueda ser comprobado si la detección es correcta o no. Para ello, previamente habrá cargado esta información en el atributo *groundTruth*, del que se extraen los distintos objetos que existen en ese marco de la secuencia de vídeo y se representan de igual manera que las detecciones realizadas por el detector.

```
- (void)showFrame: (CMSampleBufferRef) sample {

  …

  NSString *clave = [NSString stringWithFormat:@"%d",frameNumber];

  Frame *f = [self.groundTruth.frames objectForKey:clave];

  for (Detection *d in f.detections) {
    cv::rectangle(frame,
                  cv::Point(d.x,d.y),
                  cv::Point(d.x+d.width,d.y+d.height),
                  cv::Scalar(0,255,0));
}

  [f release];

  CGImageRef imageRef = [self createCGImageFromMatImage:frame];

  [self.view.layer performSelectorOnMainThread:@selector(setContents:)
          withObject:(id)imageRef waitUntilDone:YES];

  CGImageRelease(imageRef);
  CFRelease(sample);
  [self calculateFPS];
}
```

Por último, la imagen procesada debe ser transformada de nuevo a un formato propio de *Cocoa Touch* para que pueda ser mostrada en la pantalla del dispositivo. De esta tarea se encarga el método *createCGImageFromMatImage*,

que a partir de una imagen en *cv::Mat* devuelve uno de los formatos propios de iPhone, *CGImageRef*.

```
- (CGImageRef)createCGImageFromMatImage:(cv::Mat&)image {

  CGColorSpaceRef colorSpace;
  CGImageRef imageRef;

  if (image.channels() > 1) {
      colorSpace = CGColorSpaceCreateDeviceRGB();
      CGDataProviderRef provider = CGDataProviderCreateWithData(NULL,
           image.data, image.elemSize()*image.total(), NULL);

      imageRef = CGImageCreate(image.cols, image.rows, 8,
                 image.elemSize()*8, image.step[0],colorSpace,
                 kCGImageAlphaNoneSkipFirst | kCGBitmapByteOrder32Little,
                 provider, NULL, false, kCGRenderingIntentDefault);

      CGDataProviderRelease(provider);
  } else {
      colorSpace = CGColorSpaceCreateDeviceGray();

      CGContextRef newContextGray = CGBitmapContextCreate(image.data,
                    image.size().width, image.size().height, 8,
                    image.size().width, colorSpace, kCGImageAlphaNone);

      // Create CgImageRef from cgcontext
      imageRef = CGBitmapContextCreateImage(newContextGray);
  }

  CGColorSpaceRelease(colorSpace);

  return imageRef;
}
```

Este procedimiento simplemente crea una nueva variable *CGImageRef* a partir de los datos almacenados en el atributo data de la variable *cv::Mat* pasada como parámetro.

3.4. Análisis de los resultados

La aplicación de análisis de resultados compara las coordenadas y dimensiones de las detecciones realizadas por la aplicación de análisis de vídeo con las coordenadas y dimensiones recogidas en los archivos *ground truth* de los distintos detectores.

Al igual que todas las aplicaciones desarrolladas con *Cocoa Touch*, esta aplicación sigue el patrón de diseño MVC.

3.4.1. Modelo

La aplicación dispone tanto de la información de las detecciones realizadas por todos los detectores como de sus correspondientes *ground truth* en unos archivos de texto, todos ellos con el mismo formato.

Nº de marco en la secuencia	Coordenada X	Coordenada Y	Anchura	Altura

Cada uno de estos archivos es cargado en una estructura de datos denominada *ResultSet*, que no es más que una colección de *Frames*, clase que representa las distintas imágenes que componen la secuencia de vídeo. Cada uno de estos

frames los cuales dispone de un listado de detecciones, representadas por la clase *Detection*.

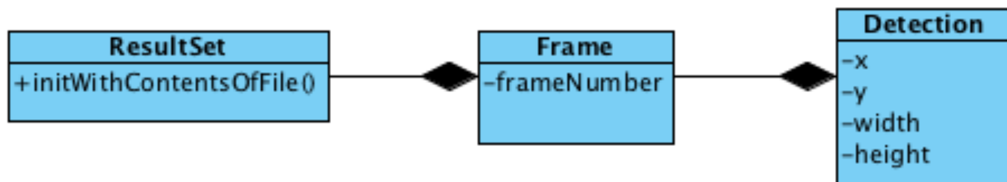


Figura 24.- Diagrama de clases Ground Truth

Una vez que toda esta información haya sido cargada en estas estructuras de datos, la aplicación estará en condiciones de compararlas y generar un informe con los resultados de las detecciones realizadas por los detectores.

3.4.2. Vista

La aplicación consta de una única vista que muestra los resultados de las detecciones realizadas. Para cada uno de los detectores aplicados sobre la secuencia de vídeo se muestran los siguientes datos.

- Total de positivos.
- Positivos verdaderos.
- Falsos positivos.
- Falsos negativos.
- Sensibilidad.

La vista tendrá, por tanto, el aspecto que se muestra a continuación.

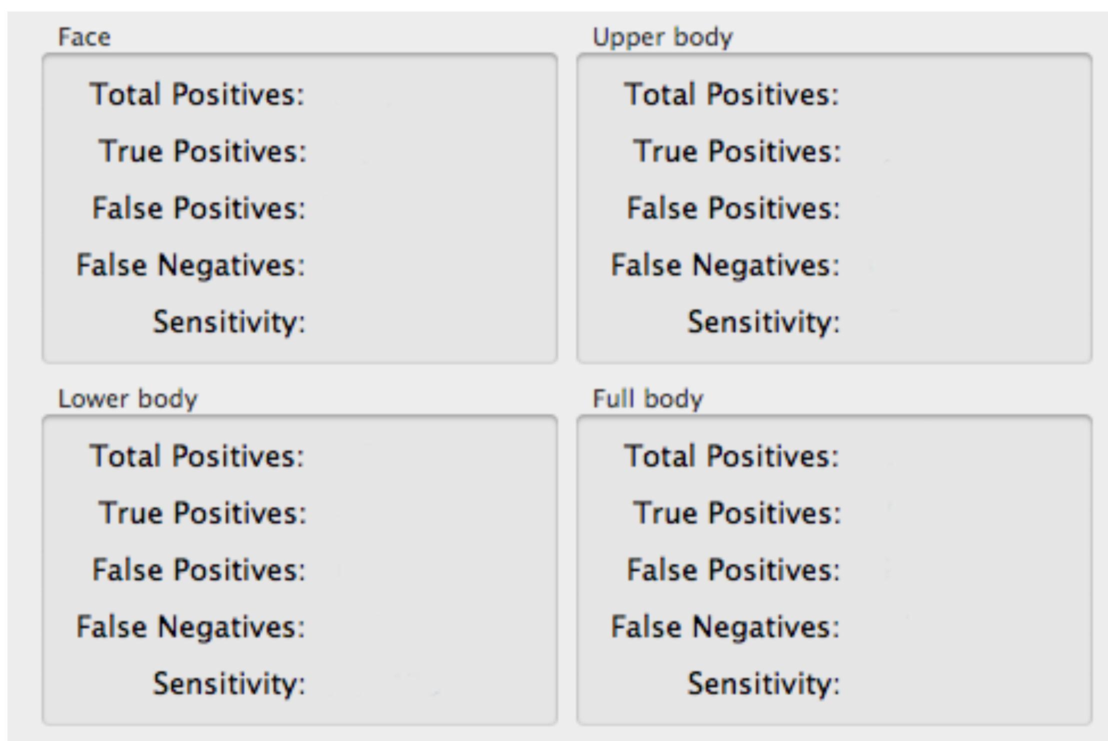


Figura 25.- Vista de la aplicación de análisis de resultados

3.4.3. Controlador

El único controlador de la aplicación, *AppDelegate*, está compuesto por 8 atributos de la clase *ResultSet*, que se cargan con la información contenida en los registros de detecciones y *ground truth* –2 objetos para cada detector, uno contiene las detecciones y otro el ground truth—.

```
@interface AppDelegate : NSObject <NSApplicationDelegate> {
    ResultSet *faceResults;
    ResultSet *faceGroundTruth;
    ResultSet *upperBodyResults;
    ResultSet *upperBodyGroundTruth;
    ResultSet *lowerBodyResults;
    ResultSet *lowerBodyGroundTruth;
    ResultSet *fullBodyResults;
    ResultSet *fullBodyGroundTruth;
    …
}
…
@end
```

En cada uno de estos atributos se carga la información correspondiente a través del método *initWithContentsOfFile*.

```
…
faceResults = [[ResultSet alloc] initWithContentsOfFile:
[[NSBundle mainBundle] pathForResource:@"FaceDetectionResults"
ofType:@"txt"]];

faceGroundTruth = [[ResultSet alloc] initWithContentsOfFile:
[[NSBundle mainBundle] pathForResource:@"FaceGroundTruth" ofType:@"txt"]];
…
```

Una vez que la información ha sido cargada en las estructuras de datos correspondientes, se comparan los datos de las detecciones realizadas por los detectores con sus respectivos *ground truth* mediante el método *compareTo*.

```
…
[faceResults compareTo:faceGroundTruth];
[upperBodyResults compareTo:upperBodyGroundTruth];
[lowerBodyResults compareTo:lowerBodyGroundTruth];
[fullBodyResults compareTo:fullBodyGroundTruth];
…
```

Una detección se considera correcta cuando la superficie de la intersección de las áreas de la detección y *ground truth* sea superior al 50% de la superficie de la unión de ambas áreas. Este método se encarga de comparar entre sí las detecciones de ambas colecciones, marcando como encontradas aquellas para las que se encuentre una detección equivalente en el *ground truth*.

Al final del proceso, las detecciones de los resultados del detector marcadas como encontradas se corresponderán con positivos verdaderos, mientras que las detecciones que no hayan sido marcadas como encontradas se corresponderán con falsos positivos (detecciones realizadas por el detector de las que no se ha encontrado equivalente en el *ground truth*). Estos valores pueden ser obtenidos mediante los métodos *truePositives* y *falsePositives* de la clase *ResultSet*, que devuelven el número de detecciones marcadas como encontradas y el número de detecciones marcadas como no encontradas, respectivamente.

```
truePositives = [lowerBodyResults truePositives];
falsePositives = [lowerBodyResults falsePositives];
falseNegatives = [lowerBodyGroundTruth falsePositives];
```

```
total = [lowerBodyGroundTruth numberOfDetections];
sensitivity = (float)truePositives/(float)(truePositives + falseNegatives);
```

Las detecciones del objeto ground truth que no hayan sido marcadas como encontradas se corresponden con detecciones que el detector debería haber realizado pero que no ha realizado, es decir, falsos negativos. Este valor se obtiene invocando al método *falsePositives* del objeto *ground truth* correspondiente. El método numberOfDetections devuelve el número total de detecciones que contiene el objeto *ResultSet*, invocado sobre el objeto ground truth devolverá el número total de patrones presentes en el vídeo.

3.5. Fotografía automática

La aplicación de fotografía automática realiza y almacena de manera automática una fotografía cuando detecta en la imagen un número de personas determinado previamente por el usuario. Al igual que el resto de aplicaciones, su diseño sigue el patrón MVC. A continuación, se muestran los principales puntos de cada uno de los componentes que forman la aplicación.

3.5.1. Modelo

Las imágenes son capturadas directamente desde la cámara y posteriormente analizadas en busca de un determinado número de personas. Este número es determinado por el usuario directamente en la pantalla diseñada para tal efecto y almacenado dentro del atributo *numberOfPeople* de tipo *NSNumber* del controlador *PhotoConfController*, cuya única misión es la configuración del número de personas a detectar en la imagen. *NSNumber* es una clase de *Cocoa Touch* que modela un valor numérico, es capaz de almacenar tanto números enteros como números en coma flotante.

El controlador principal de la aplicación, *ViewController,* obtiene las imágenes de la cámara en el formato propio de *CocoaTocuh*, *CMSampleBufferRef* –visto con detalle en las anteriores aplicaciones—, las analiza y si encuentra al menos el número de personas especificado en el atributo *numberOfPeople* del controlador de configuración, *PhotoConfController*, almacena automáticamente la fotografía que ha obtenido en formato *CMSampleBuffer* en el álbum de fotografías del iPhone.

3.5.2. Vista

La aplicación consta de las siguientes 3 vistas que se detallan a continuación.

3.5.2.1. Vista principal

La vista principal de la aplicación visualiza en la parte central de la pantalla las imágenes que están siendo capturadas por la cámara. Puesto que el *iPhone* dispone de 2 cámaras, una situada en parte delantera y otra en la trasera, se debe poder seleccionar la cámara deseada a través de un botón situado en la parte superior derecha de la pantalla.

Junto al botón de selección de cámara, la aplicación muestra el número de personas detectadas en la imagen actual y el número de personas necesarias para que se tome la fotografía.

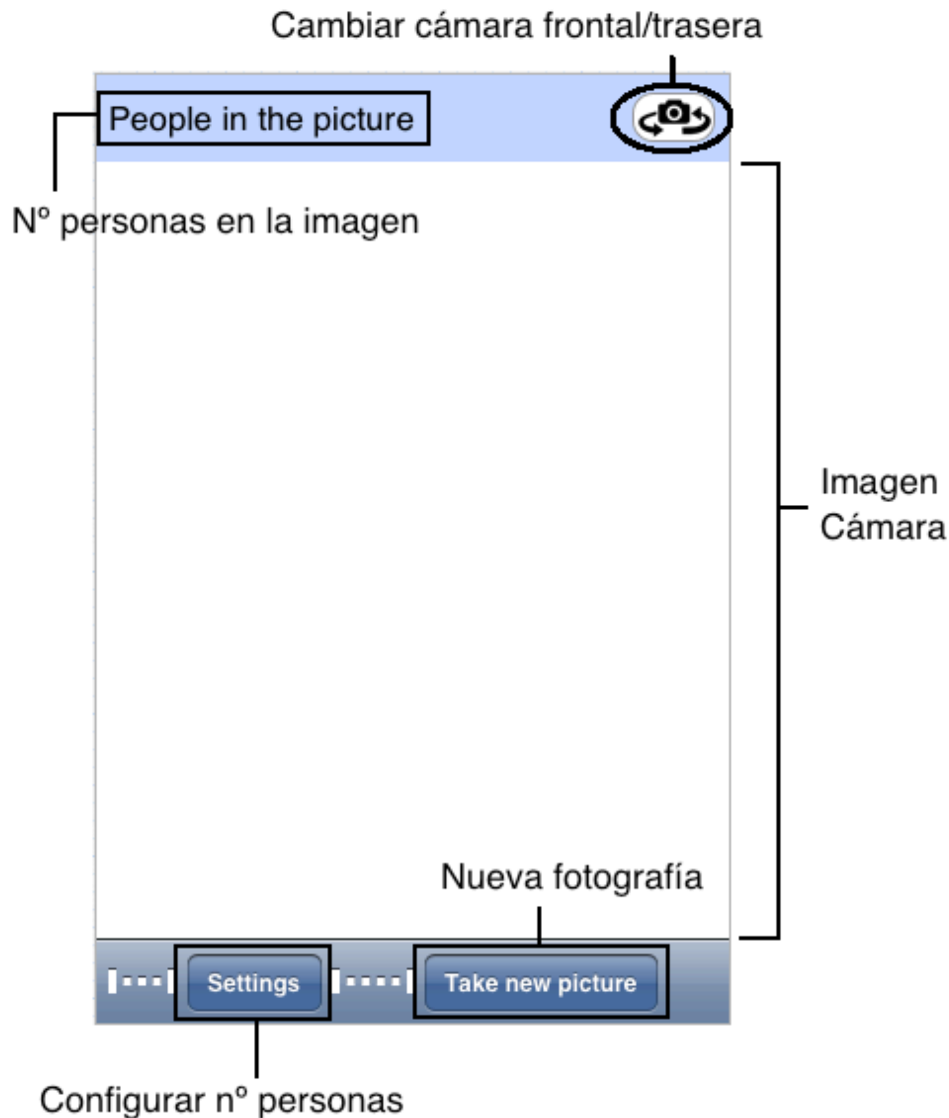


Figura 26.- Vista principal de la aplicación de fotografía automática

En la parte inferior de la pantalla se muestra una barra con 2 botones. El primero de ellos debe llevar al usuario a la vista de configuración donde se puede definir el número de personas que deben aparecer en la imagen para que se haga la fotografía. El segundo de ellos le indica a la aplicación que se desea tomar una nueva fotografía. Al ser pulsado, el título del botón cambiará de "*Take new picture*" a "*Taking new picture*"; será en este momento cuando la aplicación comience a analizar las imágenes de la cámara hasta que se detecte el número de personas necesario para que se dispare la fotografía.

3.5.2.2. Vista de configuración

Al pulsar el botón de configuración de la pantalla principal (*settings*), la aplicación navega hasta la vista de configuración. En esta vista el usuario puede aumentar o disminuir, a través de una barra deslizadora, el número de personas que deben ser detectadas en la imagen para que la fotografía sea disparada automáticamente.

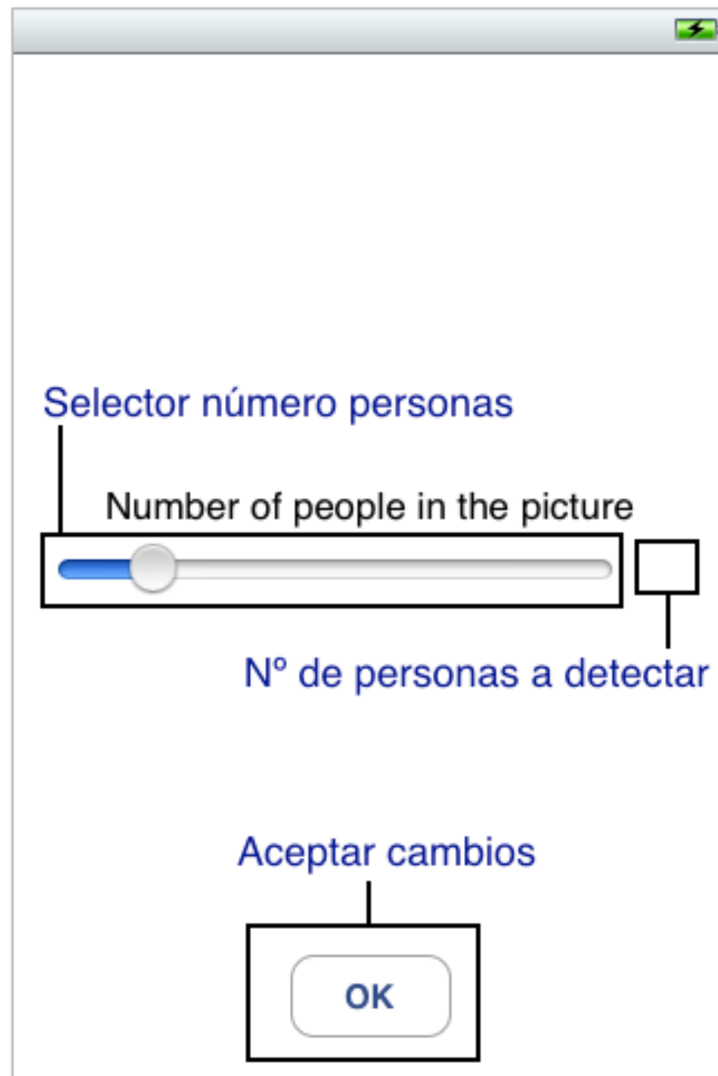


Figura 27.- Vista de configuración

La pulsación del botón OK provocará que la aplicación vuelva de nuevo a la vista principal y asuma el número de personas indicado en el control de selección, como el número de personas a detectar para realizar la fotografía.

3.5.2.3. Vista de detección

Esta vista está situada sobre la región de la vista principal donde se visualizan las imágenes captadas por la cámara. Su objetivo es dibujar un cuadrado con las coordenadas y dimensiones de las detecciones realizadas por el detector. De esta manera el usuario puede visualizar las personas que el detector está reconociendo.

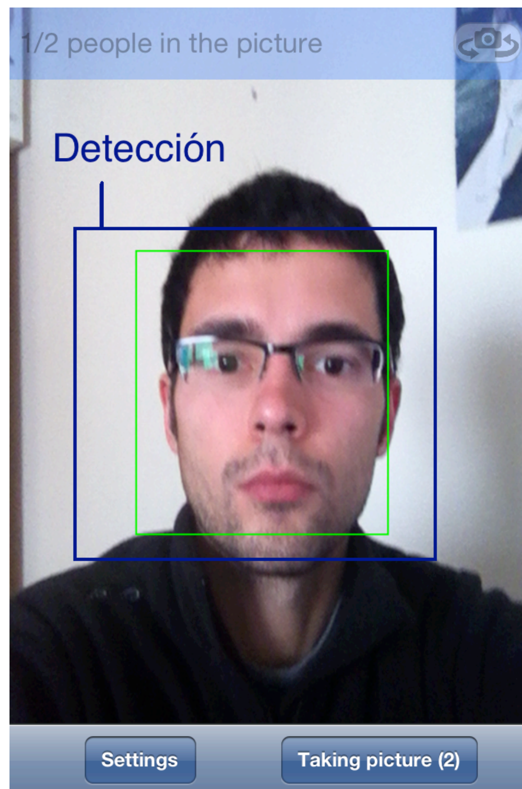


Figura 28.- Vista de detección

3.5.3. Controlador

La aplicación está compuesta por 3 controladores, uno para cada una de las vistas definidas anteriormente, cuyo funcionamiento es explicado a continuación.

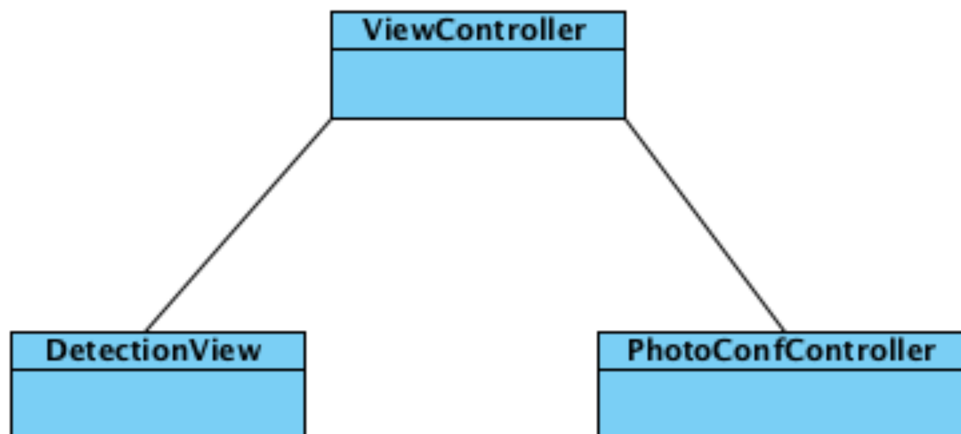


Figura 29.- Diagrama de clases de la aplicación de fotografía automática

3.5.3.1. Controlador principal

El objetivo fundamental del controlador principal, implementado en la clase *ViewController*, consiste en realizar una fotografía cuando un determinado número de personas ha sido detectado en la imagen captada por el dispositivo. Para ello, analiza cada una de las imágenes que obtiene de la cámara siguiendo

los pasos explicados en el apartado 3.2, con la excepción de que en este caso el origen de las imágenes analizadas es la cámara del dispositivo.

Captura de la imagen desde la cámara

A través del framework *AVFoundation* de *Cocoa Touch* es posible adquirir fácilmente imágenes de la cámara del dispositivo. Para ello es necesario configurar adecuadamente una sesión de captura a través de la clase *AVCaptureSession*. Una vez que el objeto ha sido configurado correctamente, se creará un nuevo hilo de ejecución que recibirá y procesará las imágenes capturadas por la cámara a razón de 30 imágenes por segundo.

La sesión de captura es modelada dentro del atributo *captureSession* del controlador, se configura de la siguiente manera mediante el método *setupCaptureSession*, invocado al inicio de la aplicación.

```
- (void)setupCaptureSession {

  // Create the session
  self.captureSession = [[AVCaptureSession alloc] init];

  // Session resolution
  self.captureSession.sessionPreset = AVCaptureSessionPreset640x480;

  // Find a suitable AVCaptureDevice
  AVCaptureDevice *device = [AVCaptureDevice
                  defaultDeviceWithMediaType:AVMediaTypeVideo];

  // Create a device input with the device and add it to the session
  NSError *error;
  self.videoInput = [AVCaptureDeviceInput deviceInputWithDevice:device
                                                          error:&error];

  [self.captureSession addInput:videoInput];

  …
}
```

En primer lugar, instancia un nuevo objeto de la clase *AVCaptureSession* y, a continuación, establece que la resolución con la que se obtendrán las imágenes será de 640x480 píxeles. Se ha elegido esta resolución porque ofrece un buen equilibrio entre calidad de imagen y velocidad de detección. Después, crea e instancia un objeto de la clase *AVCaptureDevice* que representa el dispositivo que físicamente realiza la captura de imágenes, en este caso la cámara. A partir de este objeto, crea un dispositivo de entrada de tipo *AVCaptureDeviceInput* y lo añade a la sesión. De esta manera, se define que la sesión obtendrá las imágenes de la cámara.

```
- (void)setupCaptureSession {

  …

  // Make a still image output
  stillImageOutput = [AVCaptureStillImageOutput new];

  if ([captureSession canAddOutput:stillImageOutput])
    [self.captureSession addOutput:stillImageOutput];

  …
}
```

A continuación, se crea y añade a la sesión una salida de tipo *AVCaptureStillImageOutput* que permite capturar una única imagen de la cámara. Esta salida servirá para hacer la fotografía cuando el número de personas necesario haya sido detectado en la imagen.

```
- (void)setupCaptureSession {

  …

  // Create video data output and add it to the session
  videoOutput = [[AVCaptureVideoDataOutput alloc] init];


  // Specify the pixel format (BGRA is supposed to be faster)
  videoOutput.videoSettings = [NSDictionary dictionaryWithObject:
          [NSNumber numberWithInt:kCVPixelFormatType_32BGRA]
   forKey:(id)kCVPixelBufferPixelFormatTypeKey];

  videoOutput.alwaysDiscardsLateVideoFrames = YES;


  // Configure output
  cameraQueue = dispatch_queue_create("cameraQueue", DISPATCH_QUEUE_SERIAL);

  [videoOutput setSampleBufferDelegate:self queue:cameraQueue];


  if([captureSession canAddOutput:videoOutput])
      [self.captureSession addOutput:videoOutput];

  …
}
```

Para finalizar la configuración de la sesión, se crea otra salida de tipo *AVCaptureVideoDataOutput* que proporcionará la secuencia de vídeo capturada por la cámara, de donde se obtendrán las imágenes a analizar para detectar el número de personas presentes en la imagen. Puesto que el proceso de detección es muy lento en comparación con la velocidad a la que se obtienen las imágenes (30 imágenes por segundo), se configura la salida de video para que descarte las imágenes capturadas mientras se estaba procesando una imagen. Esto se consigue estableciendo el atributo *alwaysDiscardsLateVideoFrames* del objeto *videoOutput* a cierto.

Este tipo de salida necesita un método *delegate o delegado* en el que se delega el procesado de las imágenes obtenidas. Cuando la sesión de captura comience, se usará un hilo de ejecución paralelo (especificado como parámetro, *cameraQueue*) y las imágenes capturadas serán enviadas al objeto delegado para su procesado. A través del método *setSampleBufferDelegate* se establece el objeto controlador (*self*) como el encargado de esta tarea y el hilo de ejecución que se empleará para ello. Por último, se añade la salida a la sesión.

Una vez configurada la sesión, puede iniciarse la captura llamando simplemente al método *startRunning* del objeto sesión, *captureSession*.

```
[self.captureSession startRunning];
```

En el momento en que se inicia la captura, las imágenes son capturadas y enviadas al método *captureOutput* del objeto delegado. Es en este método, por tanto, donde se procesan las imágenes y se envían al detector de *OpenCV*.

```
- (void)captureOutput:(AVCaptureOutput *)captureOutput
didOutputSampleBuffer:(CMSampleBufferRef)sampleBuffer
       fromConnection:(AVCaptureConnection *)connection {

  if (!pictureTaken) {
    std::vector<cv::Rect> faces;

    // Process frame image
    [self createMatImageFromSampleBuffer:sampleBuffer];

    neon_to_gray(frameImage,grayImage);

    cv::resize(grayImage,
               smallImage,
               cv::Size(smallImage.size().width,smallImage.size().height));

    faceCascade.detectMultiScale(smallImage,
                                 faces,
                                 1.2f,
                                 2,
                                 0 | CV_HAAR_DO_CANNY_PRUNING,
                                 cv::Size(20, 20));


    numberOfPeopleDetected = faces.size();

    if (numberOfPeopleDetected >= [self.settingsController.numberOfPeople
                                      intValue] && pictureTaken == NO)
    {
       pictureTaken = YES;
       numberOfPeopleDetected = 0;
       dispatch_async(dispatch_get_main_queue(), ^(void) {
                  [self takePicture:nil];
       });
    }

  }
}
```

La detección se realiza siguiendo los mismos pasos explicados en el apartado 3.2. Los resultados son devueltos por el detector de OpenCV en el vector *faces*, cuyo número de objetos determinará el número de personas presentes en la imagen. Si éste es superior al número de personas establecido, se disparará la fotografía mediante una invocación al método *takePicture* que se encarga de capturar la imagen, a través de la salida creada con tal finalidad en la sesión, y almacenarla en el álbum de fotografías del iPhone.

3.5.3.2. Controlador configuración

El controlador de configuración, *PhotoConfController*, tan sólo consta de un atributo, *numberOfPeople*, cuyo valor está asociado al control de selección del número de personas.

```
@interface PhotoConfController : UIViewController {
    NSNumber *numberOfPeople;
    UILabel *numberOfPeopleLabel;
    UISlider *numberOfPeopleSlider;
}
```

```
@property (nonatomic,retain) IBOutlet UILabel *numberOfPeopleLabel;
@property (nonatomic,retain) IBOutlet UISlider *numberOfPeopleSlider;
@property (nonatomic,retain) NSNumber *numberOfPeople;

- (IBAction)numberOfPeopleDidChange:(id)sender;
- (IBAction)acceptChanges:(id)sender;

@end
```

Cada vez que el usuario mueve la barra deslizadora del control de selección de número de personas, el método *numberOfPeopleDidChange* es invocado y se actualiza el valor del atributo *numberOfPeople*.

```
- (IBAction)numberOfPeopleDidChange:(id)sender {
  self.numberOfPeople = [NSNumber numberWithFloat:
                                [self.numberOfPeopleSlider value]];
  self.numberOfPeopleLabel.text = [NSString stringWithFormat:@"%d",
                                [self.numberOfPeople intValue]];

}
```

Al pulsar el botón "OK", se invoca el método *acceptChanges* que simplemente elimina la vista actual y vuelve a la principal.

```
- (IBAction)acceptChanges:(id)sender {
    [self dismissModalViewControllerAnimated:YES];
}
```

3.5.3.3. Controlador de detección

El controlador *DetectionView* dibuja por pantalla los cuadros correspondientes a las detecciones realizadas por el detector recogidos en el atributo *pathToDraw*.

```
@interface DetectionView : UIView {
    CGMutablePathRef pathToDraw;
}

@property (nonatomic, assign) CGMutablePathRef pathToDraw;

@end
```

El controlador principal se encarga de añadir en este atributo los cuadros correspondientes a cada una de las detecciones realizadas. El método *drawRect* es el responsable final de visualizar en pantalla la vista controlada por esta clase y, por tanto, de representar esta información.

```
- (void)drawRect:(CGRect)rect
{
  // Drawing code
  CGContextRef context=UIGraphicsGetCurrentContext();

  CGContextSaveGState(context);

  CGContextTranslateCTM(context, self.frame.size.width,0);

  CGContextScaleCTM(context, -1.0, 1.0);

  CGContextSetLineWidth(context, 1);
  CGContextSetStrokeColorWithColor(context, [[UIColor greenColor] CGColor]);
  CGContextAddPath(context, pathToDraw);

  CGContextStrokePath(context);
```

```
  CGContextRestoreGState(context);
}
```

4. Resultados

Se ha analizado el comportamiento de 4 detectores corporales: cara, cuerpo completo, parte superior del cuerpo y parte inferior del cuerpo. Estos detectores han sido aplicados sobre una misma secuencia de vídeo en formato MPEG-4 y con una resolución de 320x180 en la que, progresivamente, aparecen 3 personas en la imagen. De esta manera, se obtienen unos resultados objetivos para cada uno de los detectores, al haber sido aplicados todos ellos bajo las mismas condiciones y sobre las mismas imágenes.

El dispositivo móvil en el que se han implementado las aplicaciones del proyecto es un iPhone 4. Este dispositivo dispone de 512 MB de memoria RAM y un procesador *ARM Cortex-A8* de un único núcleo y 1 GHz de velocidad, integrado junto a procesador gráfico *PowerVR* en un chip Apple A4.

Este tipo de procesadores, usados en dispositivos móviles, son diseñados específicamente para conseguir tasas de consumo de energía muy bajas, por lo que su rendimiento está por debajo del ofrecido por los procesadores usados en ordenadores convencionales, que no tienen por qué preocuparse de su consumo. Así, las cotas de velocidad de procesado alcanzadas en los resultados obtenidos por Viola y Jones en su detector, serán difícilmente alcanzables en este caso, puesto que dicho detector fue implementado en un procesador *Intel Pentium III*, capaz de computar 3,4 DMIPS/MHz (test de *Dhrystone*), mientras que el procesador *ARM Cortex-A8,* sobre el que se ejecuta el detector de este proyecto, tan sólo es capaz de alcanzar los 2.0 DMIPS/MHz y dispone de mayores limitaciones de memoria que en un ordenador personal convencional.

Además, se unen dos circunstancias adicionales que deben ser tenidas en cuenta a la hora de analizar la velocidad de procesamiento de los detectores. Por un lado, la API ofrecida por *Cocoa Touch* para el tratamiento de secuencias de vídeo no fue diseñada para procesar las imágenes del vídeo en tiempo real. Esto hace que al tiempo de procesado de la detección haya que sumar el tiempo empleado en la descompresión y tratamiento previo del vídeo, necesario para obtener la imagen en el formato adecuado para su análisis con *OpenCV*. Por otro lado, se encuentra el hecho de que *OpenCV* obtiene su mayor rendimiento cuando es ejecutada en un procesador *Intel* y que el iPhone dispone de un procesador *ARM Cortex-A8*, completamente diferente a este tipo de procesador.

A pesar de las limitaciones hardware impuestas por este tipo de dispositivo, la velocidad de procesado de las imágenes puede incrementarse considerablemente si se encuentra un equilibrio óptimo entre el tamaño de la subventana que recorre la imagen y la precisión de la detección. Cuanto mayor sea el tamaño de esta subventana, menor será el número de iteraciones necesarias para recorrer la imagen y, por tanto, se necesitará menor tiempo para procesarla. Sin embargo, un tamaño demasiado grande puede provocar que no se detecte el patrón en muchos casos, mientras que un tamaño demasiado pequeño

puede obligar al detector a realizar iteraciones innecesarias para el tipo de patrón que se está buscando.

A continuación, se muestra la velocidad media de procesado para cada uno de los detectores, expresada en FPS (*frames per second* o imágenes por segundo), así como el tamaño de subventana empleado para cada uno de ellos. El tamaño de la subventana ha sido elegido a través de numerosas pruebas, intentando encontrar el equilibrio óptimo entre velocidad de procesado y precisión en la detección.

Detector	FPS	Subventana
Cara	0,95	20x20
Cuerpo Completo	13,82	60x60
Parte Superior	0,80	30x30
Parte Inferior	2,18	30x30

El mayor tamaño de subventana empleado por el detector de cuerpo completo hace que su velocidad de procesado sea muy superior a la del resto. Resalta también, como dos detectores, los de la parte superior e inferior del cuerpo, que comparten el mismo tamaño de subventana presentan velocidades de procesado muy diferentes, siendo el procesado del detector de parte inferior mucho más rápido. Esto puede ser debido a que este detector descarta muchas de las subventanas en los primeros niveles de la cascada de clasificadores, al contrario que el detector de parte superior que descartaría muchas subventanas en niveles de la cascada de clasificadores más profundos.

En cuanto a la precisión en las detecciones, mediante la aplicación de análisis de resultados, se cargan los archivos con los resultados de la detección y *ground truth* de los detectores y se analizan estos datos, obteniéndose los siguientes resultados.

Detector	Total Positivos	Positivos Verdaderos	Positivos Falsos	Negativos Falsos	Sensibilidad
Cara	1080	647	155	433	0,599
Cuerpo Completo	281	97	15	184	0,345
Parte Superior	1050	593	164	457	0,565
Parte Inferior	281	4	0	277	0,014

Destaca por un lado el alto grado de sensibilidad del detector de cara y de la parte superior del cuerpo, y por otro, la escasa sensibilidad del detector de la parte inferior del cuerpo. Es posible que la condiciones de la secuencia de vídeo sobre la que se ha aplicado no sean las más propicias para este detector, puesto que en las primeras pruebas del proyecto, en las que se probaron todos los detectores sobre la secuencia de vídeo obtenida directamente de la cámara de

vídeo del dispositivo, ofrecían resultados no tan lejanos al del resto de detectores corporales, aunque éstos no se acercaban a la precisión del resto de detectores. Queda claro, no obstante, que el detector facial y el de la parte superior del cuerpo son los que mejores resultados obtienen en cualquier circunstancia.

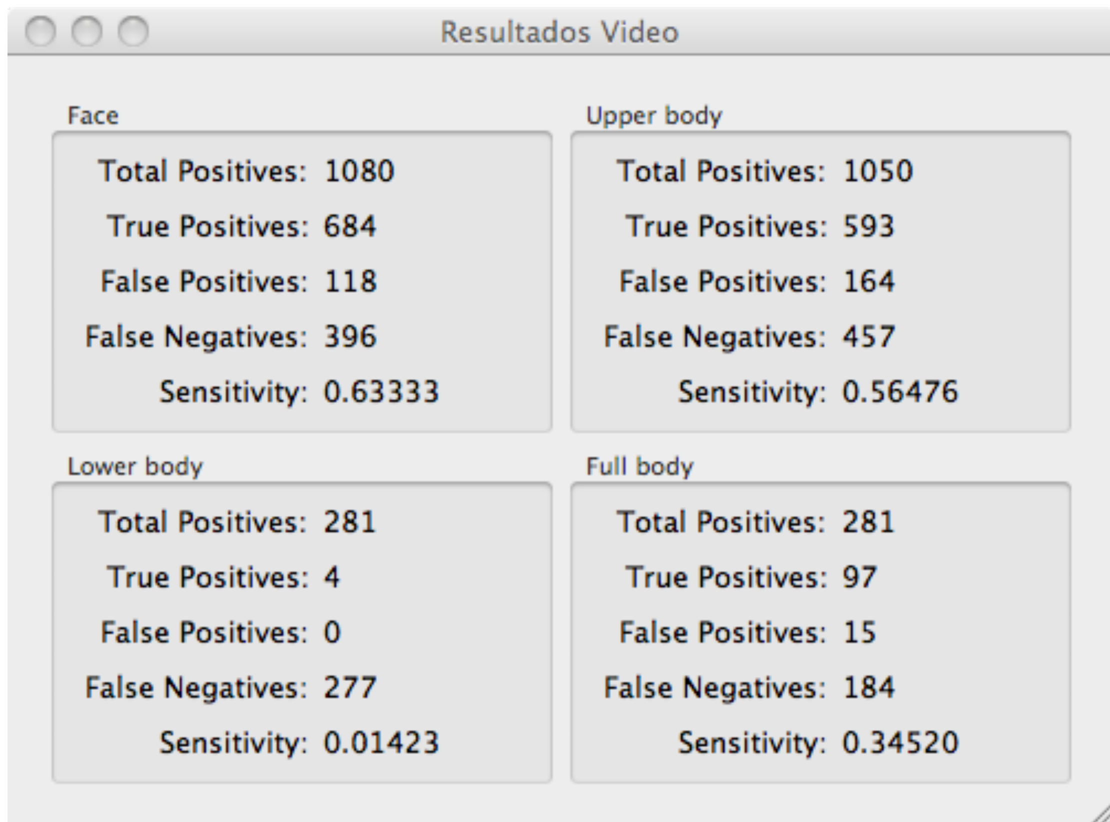


Figura 30.- Resultados ofrecidos por la aplicación de análisis de resultados

5. Conclusiones y trabajo futuro

El análisis de los resultados de la aplicación de los detectores sobre la secuencia de vídeo y de la aplicación de fotografía automática muestra que el uso de este tipo de técnicas en dispositivos móviles es completamente viable. Si bien, el comportamiento de alguno de los detectores analizados no alcanza los niveles de sensibilidad deseados, su precisión podría ser mejorada notablemente con un mejor entrenamiento del mismo. No obstante, el resultado general de los mismos es cuanto menos aceptable.

En líneas generales, a pesar de las limitaciones en los recursos disponibles en los dispositivos móviles y de que *OpenCV* no ha sido diseñada para este tipo de dispositivo, los detectores ofrecen un buen rendimiento y unos resultados robustos que hacen viable su uso en aplicaciones en tiempo real. Además, la versatilidad de este tipo de dispositivos y la posibilidad de combinar fácilmente las tecnologías disponibles en este tipo de dispositivos con las técnicas de visión por computador es, sin lugar a dudas, un gran punto a su favor que abre un sinfín de posibilidades.

Sin embargo, aunque la velocidad de procesado de los detectores no es muy elevada, es suficiente para muchas de las posibles aplicaciones que este tipo de técnicas podrían tener en este área. No obstante, se obtendría un mejor rendimiento si los detectores hubieran sido implementados específicamente para este dispositivo. Así, las principales líneas de trabajo futuras incluirían implementaciones de las técnicas de detección de patrones específicas para este dispositivo. Estas implementaciones deberían cumplir los siguientes requisitos.

- No usar cálculos en coma flotante cuyo procesado es más lento en un *iPhone*.
- Usar las posibilidades ofrecidas por el motor *NEON SIMD*, capaz de aplicar una misma operación a múltiples datos simultáneamente, en ciertas partes de la implementación del detector.
- Los detectores deberían poder ser aplicados sobre un formato de imagen nativo que no necesitara de una conversión previa. De esta manera no se consumiría tiempo de procesado adicional; concentrándose todo el esfuerzo computacional en la detección.

Todas estas mejoras podrían incrementar drásticamente el rendimiento de estos detectores en dispositivos móviles.

6. Bibliografía

GRAMACY R. (2002). *Information Theoretic Face Detection: A Survey*. University of California at Santa Cruz.

JENSEN O. (2008). *Implementing the Viola-Jones Face Detecetion Algorithm*. Technical University of Denmark: Kongens Lingby.

LAGANIÈRE R. (2011). *OpenCV 2 Computer Vision Application Programming Cookbook*. Packt Publishing.

BRADSKI G.; KAEHLER A. (2008). *Learning OpenCV*. O'Reilly Media.

VIOLA P.; JONES M. (2001). *Robust Real-time Object Detection*. Second International workshop on statistical and computational theories of vision – modeling, learning, computing, and sampling.

ARM. *Cortex-A8 processor*. [en línea]. http://www.arm.com/products/processors/cortex-a/cortex-a8.php [fecha de consulta: noviembre 2011].

KHVEDCHENYA E. (2011). *A very Fast BGRA to grayscale conversion on Iphone*. [en línea]. http://computer-vision-talks.com/2011/02/a-very-fast-bgra-to-grayscale-conversion-on-iphone [fecha de consulta: noviembre 2011].

SUMMERHILL R. (2011). *Computer vision with iOS Part 1: Building an OpenCV framework*. [en línea]. http://aptogo.co.uk/2011/09/opencv-framework-for-ios/#rebuilding [fecha de consulta: noviembre 2011]

PIPENBRICK N. (2009). *ARM NEON Optimization. An example*. [en línea]. http://hilbert-space.de/?p=22 [fecha de consulta: noviembre 2011].

KHVEDCHENYA E. (2011). *Building OpenCV for iPhone in one click*. [en línea]. http://computer-vision-talks.com/2011/02/building-opencv-for-iphone-in-one-click/ [fecha de consulta: noviembre 2011].

WIKIPEDIA. *Instructions per second*. [en línea]. http://en.wikipedia.org/wiki/Instructions_per_second [fecha de consulta: diciembre 2011].

WIKIPEDIA. *iPhone 4*. [en línea]. http://en.wikipedia.org/wiki/IPhone_4 [fecha de consulta: diciembre 2011].

7. Anexo

Adjunto a esta memoria se entrega anexo multimedia con el siguiente contenido.

- **apps**: directorio con los archivos binarios de las aplicaciones desarrolladas para la consecución de los objetivos del proyecto. Contiene las siguientes aplicaciones.

    - *FaceDetectionVideo*. Análisis del detector facial sobre la secuencia de vídeo.
    - *FullBodyDetectionVideo*. Análisis del detector de cuerpo completo sobre la secuencia de vídeo.
    - *LowerBodyDetectionVideo*. Análisis del detector de la parte inferior del cuerpo sobre la secuencia de vídeo.
    - *UpperBodyDetectionVideo*. Análisis del detector de la parte superior del cuerpo sobre la secuencia de vídeo.
    - *PhotoGroup*. Aplicación que realiza de manera automática una fotografía cuando un determinado número de personas son detectadas en la imagen.

    Para instalarlas en un dispositivo, deben ser primero cargadas en iTunes y posteriormente cargadas en el dispositivo deseado.

    - *Resultados Video*. Aplicación para Mac OS X que analiza los resultados de las detecciones realizadas por los detectores anteriores y presenta un informe sobre el rendimiento de los mismos.

- **compilación**: contiene el script de compilación de la librería *OpenCV*.

- **ground truth**: directorio que contiene los archivos de texto con las dimensiones y coordenadas de las detecciones que deberían ser realizadas en la secuencia de vídeo.

- **results**: contiene los archivos de texto con las dimensiones y coordenadas de las detecciones realizadas por los diferentes detectores aplicados sobre la secuencia de vídeo.

- **src**: directorio con los proyectos Xcode de las aplicaciones desarrolladas en el proyecto.

    - *PhotoGroup*. Código de la aplicación que realiza una fotografía automáticamente al detectar un determinado número de personas en la imagen.
    - *Resultados Video*. Aplicación para Mac OS X que analiza los resultados obtenidos por los detectores corporales.

- *VideoTestCV*. A partir de este proyecto se generan las aplicaciones que aplican sobre la secuencia de vídeo los diferentes detectores analizados.

- **video**: archivos de vídeo con la demostración del funcionamiento de las aplicaciones.

  - *ResultadosDetección*: Video demostrativo de las aplicaciones que analizan los distintos detectores corporales sobre una secuencia de vídeo.
  - *FotografíaAutomática*: Video demostrativo de la aplicación que realiza automáticamente una fotografía al detectar un determinado número de personas en la imagen.
  - *SecuenciaOrigen*: Video analizado por los detectores en la aplicación de análisis del rendimiento.

- **presentación**: contiene el archivo PowerPoint con la presentación del proyecto final de carrera.