ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA DE COMPUTADORES

CONTROL Y PROCESAMIENTO DE VIDEO EN CÁMARAS IP DESDE UNA PLATAFORMA ANDROID

CONTROL AND VIDEO PROCESSING OVER IP CAMERAS RUNNING ON AN ANDROID PLATFORM

Realizado por
David Molina Alarcón

Tutorizado por
D. José María González Linares

Departamento
Arquitectura de Computadores

UNIVERSIDAD DE MÁLAGA

MÁLAGA, Diciembre de 2015

Fecha defensa:
El Secretario del Tribunal

Resumen: La aplicación Control Camera IP, desarrolla como Proyecto Fin de Carrera en la ETS. De Ingeniería Informática de la Universidad de Málaga, fue concebida como una interfaz de usuario para la monitorización y control de cámaras IP de forma remota, pudiendo ésta ejecutarse en diferentes plataformas, incluyendo dispositivos móviles con sistemas Android. En aquel momento sin embargo, las plataformas Android no disponían de una librería oficial dentro del marco de la herramienta de desarrollo utilizada (la biblioteca de desarrollo multiplataforma Qt), por lo que fue utilizada una versión alternativa no oficial denominada *Necessitas Qt for Android*. Hoy, con la versión 5 de Qt, existe la posibilidad de dar soporte a las plataformas Android de forma oficial, por lo que es posible adaptar la aplicación a esta nueva versión.

En este Trabajo Fin de Grado, se ha adaptado la aplicación *Control Camera IP* a la versión 5 de Qt, logrando así crear plataformas para dispositivos Android de forma oficial. Además, se hace uso de la biblioteca OpenCV para el desarrollo de varios métodos de procesamiento sobre la imagen recibida por la cámara IP, así como algoritmos de detección de movimiento y de caras de personas, haciendo uso de técnicas de visión por computador.

Finalmente, se introduce la posibilidad de utilizar APIs estandarizadas para la conectividad de la aplicación con cámaras IP de bajo coste, adaptando algunas de sus funciones a la aplicación *Control Camera IP*.

Palabras Clave: cámara, IP, plataforma, conexión, red, API, control, monitorización, procesamiento, imagen, técnicas, visión, artificial, algoritmos, detección, tratamiento.

Abstract: The IP Camera Control application, developed as Degree project in the *ETS. De Ingeniería Informática* at the University of Malaga, it was conceived as a user interface for monitoring and control IP cameras remotely, so it can run on different platforms, including mobile devices with Android operative systems. At that time however, the Android platform did not have an official library within the framework of the development tool used (the cross-platform Qt development library), which was used an unofficial alternative version called *Necessitas Qt for Android*. Today, with Qt on its 5th version, there is support for the Android platform officially, making it possible to adapt the application to the new version.

In this Final Project, the IP Camera Control application has been adapted to the 5th version of Qt, having succeeded indeed the creation of platforms for Android devices officially. In addition, OpenCV library has been used to develop various methods of processing on the image received by the IP camera, as well as motion and face detection algorithms by using computer vision techniques.

Finally, the possibility of using standardized APIs for the application connectivity with low cost IP cameras, adapting some of its functions to the IP Camera Control application, has been introduced.

Keywords: camera, IP, platform, connection, network, API, control, monitoring, processing, image, techniques, vision, artificial, algorithms, detection, treatment.

Contenido

Figuras

(Esta página está vacía de forma intencionada)

Capítulo 1

Introducción

En este Trabajo Fin de Grado se pretende realizar una ampliación de las funcionalidades de la aplicación C*ontrol Camera IP*, desarrollada como Proyecto Fin de Carrera (en adelante PFC) para la titulación de Ingeniería Técnica Informática de Sistemas [1]. Esta aplicación consiste en una interfaz gráfica de usuario o *GUI (Graphic User Interface*) que permite el control y monitorización de imágenes recibidas desde cámaras de red, también conocidas como cámaras IP *(Internet Protocol Cameras*), a través de diversas plataformas tipo *desktop* (Windows, Linux, etc.), *smartphones* con sistema operativo Android, así como otros tipos de dispositivos móviles.

Esta capacidad multiplataforma de la aplicación es proporcionada a través de la biblioteca de desarrollo de Qt [2].La biblioteca Qt dispone de un *framework* basado en un conjunto de librerías y módulos en el lenguaje C++, que ofrece una gran capacidad de procesamiento y portabilidad en distintas plataformas, así como una gran facilidad de realizar la compilación cruzada para cada una de ellas, permitiendo la inclusión de recursos compartidos, y la posibilidad de configurar ámbitos o *scopes* que determinen la inclusión o no de estos recursos en cada una de las plataformas configuradas.

Originalmente, la versión 4.8 de Qt no disponía de una versión oficial para la inclusión de las librerías Android entre las plataformas disponibles en su entorno de desarrollo QtCreator, por lo que fue necesario recurrir a un proyecto alternativo creado por desarrolladores de Qt y que tenía como nombre *Necessitas Qt for Android*. Este proyecto, amparado por la licencia GPL de Qt, ofrecía la posibilidad de configurar la plataforma Android dentro del proyecto, permitiendo por tanto la ejecución de la aplicación *Control Camera IP* en dispositivos con este sistema operativo. Debido a la reciente inclusión de la plataforma Android en la versión 5 de Qt, para este trabajo se ha planteado la necesidad de adaptar la aplicación *Control Camera IP* a la nueva versión de Qt, permitiendo así la creación de plataformas Android dentro del entorno de desarrollo y haciendo uso de las librerías oficiales.

Para la ampliación del PFC original se ha recurrido a uno de los posibles trabajos futuros planteados en el capítulo 5 del mismo, y que trata acerca de la inclusión de

procesamiento de video integrado dentro de la aplicación *Control Camera IP*. Para ello, en dicho capítulo se propone la utilización de la biblioteca de visión artificial y procesamiento de video e imagen OpenCV, que gracias a ser un *software* disponible bajo licencia BSD en la mayor parte de sus funciones, además también de su gran potencia y capacidad multiplataforma, es una herramienta óptima para ser utilizada en la aplicación original.

A continuación, realizaremos una breve introducción a las librerías más importantes de OpenCV, además de describir algunas funciones y algoritmos matemáticos interesantes aplicados a la visión artificial y el procesado de imagen. Además, en esta introducción avanzaremos algunas novedades y cambios llevados a cabo en la aplicación para adaptarla al funcionamiento en cámaras IP de bajo coste. Este aspecto será examinado con más detalle en el capítulo 3 de este Trabajo Fin de Grado. Por último, al final de este capítulo se describirán también los objetivos principales perseguidos en este trabajo.

1.1 Objetivos del Trabajo

Como ya hemos avanzado en este capítulo de introducción, los objetivos de este Trabajo Fin de Grado consistirán en adaptar el código de la aplicación *Control Camera IP* a la versión 5 de la biblioteca Qt, que permite la inclusión de plataformas Android de manera oficial a diferencia de las anteriores versiones. Para ello se realizarán las modificaciones oportunas en el código original, sustituyendo aquellas funciones que hayan podido quedarse obsoletas.

Como objetivo principal se pretende ampliar la aplicación añadiendo varias funciones y algoritmos de procesamiento de imágenes haciendo uso de la biblioteca OpenCV, para ser aplicados sobre las capturas realizadas por la cámara en diferentes dispositivos. Sobre estos métodos de manipulación y procesamiento de la imagen se podrá interactuar de forma que se puedan cambiar sus parámetros y valores umbrales a través de interfaces de diálogo creados con la biblioteca Qt.

Los métodos a realizar para el procesamiento de imágenes consistirán en:

- Un método de detección de bordes basado en el algoritmo de *Canny* [3].
- Funciones de ajuste para regulación de brillo y contraste sobre la imagen.
- Un algoritmo de sustracción de fondo de la imagen.

- Un método básico de detección de regiones (*Blobs*) de color.
- Un método de detección de caras

Por último, otro de los objetivos planteados en este proyecto trata sobre la realización de un estudio que permita comprobar la posibilidad de adaptar la aplicación *Control Camera IP* para que pueda funcionar con una variedad de cámaras IP de bajo coste, haciendo hincapié en la búsqueda de algún API estandarizado que pueda integrarse en la aplicación, logrando maximizar por tanto su conectividad de cara al futuro.

1.2 Ejemplos de uso de las cámaras IP

El ejemplo de uso más ampliamente extendido en las soluciones que incorporan cámaras IP, es el de la videovigilancia. Es notorio que a lo largo de los últimos años, la vigilancia remota a través de internet, ha sido uno de los servicios más demandados y utilizados por un grupo amplio de particulares y empresas, con el objetivo de mejorar la seguridad en las instalaciones o en los hogares, siendo sobre todo, los avances en el ámbito del desarrollo del *software* de gestión de video y en la incorporación de sistemas inteligentes basados en técnicas de visión por computador, los mejores reclamos de cara a los clientes para la adquisición de este tipo de soluciones, que en la mayoría de los casos están dotadas de una eficiente asistencia remota y presencial en caso de emergencia.

En relación a esto último, podemos destacar también el uso cada vez más frecuente de los sistemas de video IP en la detección de situaciones de peligro o de emergencia en los entornos urbanos (tráfico, zonas escolares, instituciones públicas), así como forestales (prevención de incendios), marítimos, etc., con el propósito de detectar y alertar de forma instantánea al centro de control de una situación de peligro o de alto riesgo en una localización concreta. Para ello, y al igual que comentábamos en el apartado anterior, los avances en sistemas inteligentes de detección y predicción de situaciones haciendo uso de las últimas técnicas de visión artificial, hacen que estas soluciones sean realmente efectivas y cada vez más demandas por instituciones públicas, y otros organismos de ámbito privado a nivel global.

Finalmente, como últimos avances en los sistemas de video IP nos encontramos en las nuevas generaciones de cámaras IP (*Smart IP Cameras*) con soluciones de reconocimiento facial e identificación de personas basado en parámetros biométricos

[4], así como aplicaciones para el control de calidad en procesos de producción [5], a través de la monitorización y supervisión de los procesos, y también a través de la detección de situaciones de riesgo utilizando similares soluciones *software* y técnicas inteligentes que las mencionadas en apartados anteriores para otro tipo de aplicaciones y usos.

En definitiva, son muchos los ejemplos de uso que hoy en día pueden darse para las cámaras IP, sobre todo cada vez más enfocados al análisis de información útil para la predicción y la detección de situaciones de emergencia, y para el ámbito de la seguridad en el ámbito público y privado.

1.3 Cámaras IP en la actualidad

Los sistemas de videovigilancia basados en cámaras IP han evolucionado a lo largo de estos años para convertirse en lo que se conoce como *Smart Solutions*, adaptándose a los nuevos tiempos que corren donde cualquier sistema de gestión de video o de cualquier otro ámbito relacionado o no con la seguridad, puede estar controlado directamente desde nuestro dispositivo *SmartPhone.*

En ese sentido, las empresas dedicadas hoy día al desarrollo de este tipo de soluciones disponen entre sus ofertas de una variedad bastante significativa de productos y de facilidades no tanto en el aspecto *Hardware*, sino también en el ámbito de las aplicaciones móviles (*smart applications*). En base a esta realidad, algunas de estas empresas han acuñado un nuevo concepto bajo la denominación de *Smart IP Cameras* (o *Smart IPC*), dotando a sus cámaras de prestaciones avanzadas en el ámbito sobre todo de la "detección inteligente", disponiendo éstas de capacidades de detección de objetos o intrusos haciendo uso de algoritmos optimizados que se ejecutan dentro del mini computador incluido en el interior de la cámara [6].

Figura 1. La nueva generación de Cámaras IP: Smart IPC

Algunas de estas cámaras disponen también de la posibilidad de almacenamiento en la nube (*cloud storage*), lo que permite una gran facilidad y flexibilidad para la disposición y almacenamiento de imágenes directamente desde la nube. El sistema que hoy en día suele estar más extendido para almacenamiento en la nube es el de *Video Surveillance as a Service* (*VSaaS*) [7] y básicamente incluye la gestión del almacenamiento de imágenes, además de la posibilidad de procesamiento de video remoto, entre otros aspectos.

Estas nuevas soluciones son ampliamente escalables, por lo que permiten la creación de sistemas de videovigilancia de gran envergadura a lo largo y ancho del planeta, gracias a la gran velocidad de las comunicaciones y a la velocidad de acceso a Internet hoy en día. Además, la cada vez mayor capacidad multiplataforma de las aplicaciones de gestión de video adaptadas a estos sistemas, hacen de la videovigilancia por red, un servicio cada vez más intuitivo y fácil de gestionar por cualquier usuario en cualquier lugar.

Es por ello, que cada vez son más demandadas por el gran público, las soluciones IP *low cost*, o de bajo coste, que lejos de lo que pudiéramos suponer, están conformadas por productos de muy buena calidad y rendimiento, ofreciendo en muchos casos prestaciones en lo referente al ámbito de la detección y la vigilancia, de tanta utilidad y eficiencia, como la de algunas de las cámaras y productos para la vigilancia más demandadas del mercado. Además de su precio reducido, las cámaras IP *low cost* tienen la ventaja de disponer en muchas ocasiones de facilidades para los desarrolladores en forma de APIs (*Application Programming Interface*), que permitan la utilización de las funciones de las cámaras en multitud de aplicaciones, independientemente de que en el momento de la compra, la cámara incluya o no con algún *software* o aplicación de gestión de video propia de la marca.

1.4 Introducción a la biblioteca OpenCV

OpenCV es una biblioteca libre de visión artificial originalmente desarrollada por Intel [8]. Desde su primera versión desarrollada en 1999 se viene utilizando para multitud de aplicaciones relacionadas con el ámbito de la seguridad (detección de movimiento, reconocimiento de objetos, etc.) además de muchos otros aplicativos en el amplio mundo de la visión artificial. Dispone de más de 500 funciones con algoritmos en el

lenguaje C, C++ y Python optimizados que aprovechan las capacidades de los procesadores multinúcleo modernos.

Figura 2. Logo de OpenCV

La amplia variedad de funciones de su API (*Application Programming Interface*) están organizadas por distintos módulos dedicados cada uno de ellos a propósitos bien diferenciados. A continuación, describimos algunos de los módulos más utilizados (y que serán usados en nuestra aplicación), así como algunos de sus métodos y funciones más importantes de cada uno de ellos:

1.4.1 Módulo *Core*

Como su propio nombre indica, está compuesto por aquellas funciones que definen el núcleo de la funcionalidad de OpenCV. Se tratan de funciones y estructuras de datos básicas que son utilizadas por el resto de módulos y que facilitan en gran medida cuestiones como la manipulación de imágenes, definición de elementos 2D sobre la imagen, creación de matrices (*Mat*), creación de estructuras dinámicas como grafos, árboles, clusterización, operaciones de persistencia, etc.

En este trabajo utilizaremos las estructuras más comunes para el procesado de imágenes tales como las estructuras *Mat* e *IplImage* que permiten almacenar fácilmente las imágenes en forma de matrices, donde cada elemento almacena un canal de color (escala de grises) o varios colores (RGB, RGBA, etc.). Además utilizaremos los métodos y funciones para trabajar con estas estructuras tales como la función *create(), size(), convertTo(), copyTo(), depth(), release(), etc.*

1.4.2 Módulo *ImgProc*

El módulo *ImgProc* (de *Image Processing*) contiene todas las funciones de OpenCV que permiten la manipulación y transformación de imágenes de cualquier tipo,

incluyendo además algoritmos de detección de características sobre la imagen, así como algoritmos para análisis estructurales, descripción de contornos y detección de objetos.

Utilizaremos el módulo *ImgProc* a un nivel muy básico sobre todo para tareas de conversión de imágenes. Tal y como comentamos anteriormente, las imágenes de varios canales de color que podamos obtener de cualquier fuente vendrán dadas normalmente en la configuración RGB. Hay que tener en cuenta sin embargo que OpenCV trabaja usualmente con los canales de color invertidos. Es decir, en lugar de la típica secuencia RGB (*Red Green Blue*), los canales de color de las imágenes en OpenCV suelen tener la configuración BGR (*Blue Green Red*). Es por esta razón que uno de los métodos más utilizados del módulo *ImgProc* suele ser el método de conversión de color *cvtColor()*, el cual usaremos frecuentemente para hacer algunas conversiones sobre la imagen en este trabajo.

1.4.3 Módulo *HighGui*

Independientemente de que OpenCV esté diseñado para el uso en aplicaciones que estén creadas bajo *frameworks* con gran riqueza de funcionalidades para la creación de interfaces de usuario (tal y como lo es Qt), o incluso para aplicaciones sin interfaz de usuario, en ocasiones es necesario disponer de herramientas que permitan realizar pruebas rápidas para visualizar resultados sin tener que recurrir a componentes con los que está diseñada la propia aplicación. Es para ello para lo que se ha diseñado el módulo *HighGui*.

Funciones como *namedWindow*() y *imshow()*, ofrecen la posibilidad de mostrar los resultados de las operaciones realizadas sobre imágenes en una ventana de forma independiente. Además, este módulo dispone también de funciones para la lectura y escritura de imagen (*imread*, *imwrite*) y de video (*VideoCapture*).

1.4.4 Módulo *Features2D*

Es un módulo también muy utilizado sobre todo en el ámbito de la visión por computador. Dispone de funciones y algoritmos de gran eficiencia que permiten realizar una descripción y detección de una serie de características sobre la imagen.

Entre los algoritmos más conocidos de descripción de características destacan el *BRIEF (Binary Robust Independent Elementary Features)*, el *FAST*, un detector de esquinas de gran efectividad, y el *FREAK (Fast REtinA Keypoint)*, que se inspira en el sistema de

visión humano. Los detectores y extractores de características más conocidos son el *SIFT* y el *SURF*, y son verdaderamente efectivos. El problema es que estos algoritmos pertenecen al módulo *NonFree* de OpenCV [9], que contiene algoritmos que pueden estar patentados y que por ello su uso está limitado. Sin embargo, existe una alternativa gratuita con el nombre *ORB* (*Oriented-FAST and Rotated BRIEF*) que está basada en los dos anteriores, y que también es bastante eficiente. Este algoritmo también se encuentra en el módulo *Features2D.*

1.4.6 Módulo ObjDetect

Por último, el módulo *ObjDetect* nos permite definir clasificadores en cascada (*Cascade Classifiers*), los cuales pueden ser entrenados con muestras de objetos concretos de una escala y tamaño determinados, para ser aplicado a una región de interés dentro de una imagen de entrada y de esta forma poder detectar si esta zona de interés es parecida a dichos objetos similares con los que se ha entrenado el clasificador.

Más adelante explicaremos la funcionalidad para detección de caras que se ha implementado en este Trabajo Fin de Grado utilizando funciones del módulo *ObjDetect.*

1.5 Contenido de la memoria

En el capítulo 2, "Control y Procesamiento de Video en Cámaras IP desde una plataforma Android", mostraremos las modificaciones que se han realizado en la aplicación *Control Cámara IP* desarrollada en el PFC original, con el propósito de crear una nueva plataforma para dispositivos Android haciendo uso de las librerías oficiales que vienen incorporadas en la última versión de Qt.

En el capítulo 3, "Funcionamiento de la Aplicación en Cámaras IP de Bajo Coste", se realizará un estudio acerca de la posibilidad de integración de la aplicación con cámaras IP de bajo coste.

En el capítulo 4, "Integración de Procesamiento de Vídeo con la Librería OpenCV", realizaremos una descripción de los nuevos métodos y funciones de procesamiento de vídeo añadidos a la aplicación *Control Cámara IP*, mostrando capturas y ejemplos de las pruebas realizadas.

En el capítulo 5, "Conclusiones y Trabajos Futuros" se mencionan de forma breve las conclusiones y trabajos que en un futuro se pueden desarrollar para la ampliación de este proyecto.

Capítulo 2

Control y Procesamiento de Video en Cámaras IP desde una plataforma Android

En este capítulo trataremos de recordar de forma resumida los aspectos más importantes en la realización de la aplicación *Control Cámara IP* englobada dentro del PFC de nombre *Interfaz Gráfica Multiplataforma para control de Cámaras IP*, la cual ha sido modificada para incorporar las mejoras y ampliaciones realizadas en el presente Trabajo Fin de Grado. Uno de estos cambios ha consistido en adaptar la aplicación a la última versión de la biblioteca de desarrollo Qt, la cual incorpora ya de manera oficial, la posibilidad de crear plataformas Android dentro del entorno QtCreator. En la última sección de este capítulo explicaremos como hemos llevado a cabo esta adaptación.

2.1 La aplicación original

La idea de la aplicación *Control Cámara IP* surgió de la dificultad presente en aquel momento de encontrar soluciones software de videovigilancia por red, que fueran capaces de funcionar en las diversas plataformas con las que nos encontramos hoy día (*smartphones, tablets, desktop*, etc.). En definitiva, se trataba de crear una aplicación de control y monitorización de videovigilancia en red fácilmente distribuible que fuera capaz de ejecutarse tanto en entornos de escritorio como en dispositivos móviles, y que además este cambio de entorno no afectara a su funcionalidad.

Desde el principio se pensó en la biblioteca de desarrollo Qt para la elaboración del proyecto, ya que dispone de una alta capacidad multiplataforma y además se compone de una *framework* en C++ orientada al desarrollo de interfaces gráficas, que dispone a su vez de una serie de módulos con funciones que ofrecen un alto rendimiento y flexibilidad.

Sin embargo, en aquel momento, la versión de Qt (que por aquel entonces pertenecía a la compañía Nokia [10]) no ofrecía soporte para las plataformas Android, lo cual era fundamental para los requerimientos iniciales que nos habíamos marcado para el proyecto. Es por esto que hubo que hacer uso de una herramienta no oficial creada por un grupo de desarrolladores de Qt, que ofrecía la posibilidad de compilar las

aplicaciones desarrolladas con la biblioteca Qt de forma que éstas pudieran ser ejecutadas en dispositivos Android. A este proyecto se le llamó *Necessitas Project*, y a día de hoy todavía sigue estando presente en la web, ofreciendo soporte a través de la propia página oficial de Qt [11].

2.1.1 Descripción general de la aplicación

La aplicación *Control Camera IP* muestra en su ejecución una interfaz gráfica de usuario sencilla, caracterizada por disponer de una zona principal de monitorización y control sobre las imágenes recibidas a través de la conexión con una cámara de red (cámara IP). También dispone de una barra de menús desde la que se pueden comandar diferentes aspectos de conexión y control, edición de máscaras, carga de controles preestablecidos, etc.

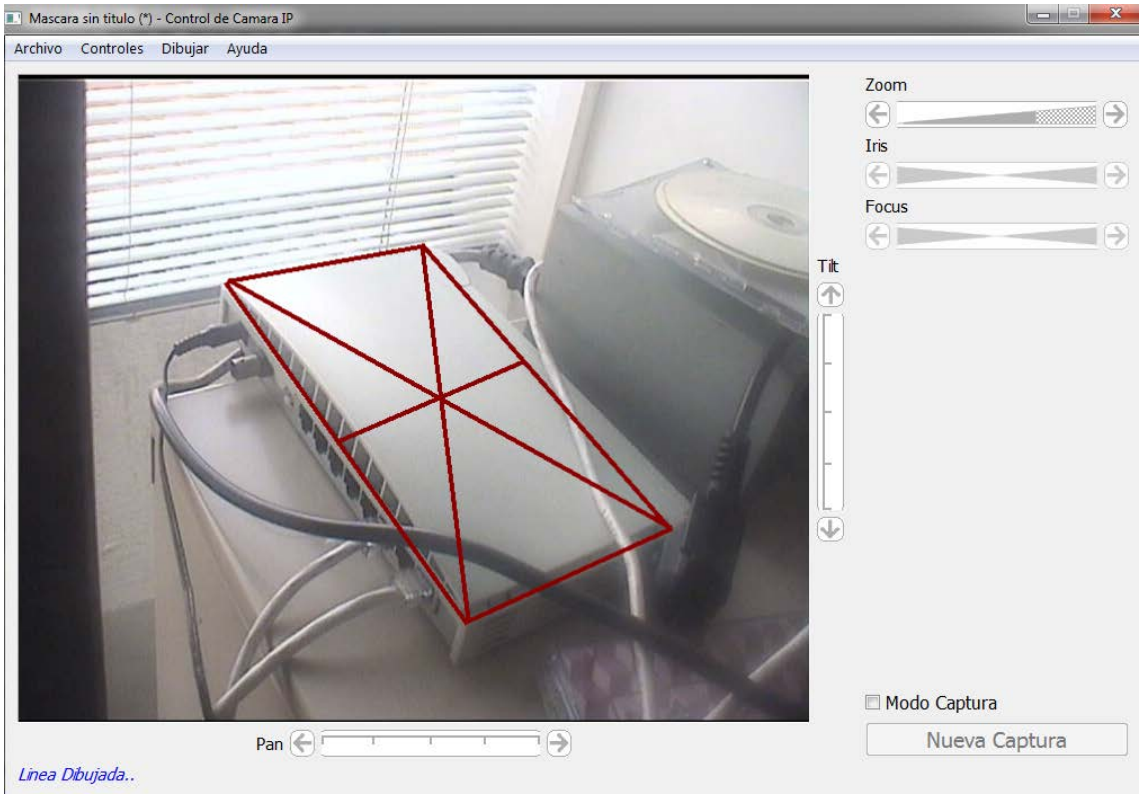

Figura 3. Ejecución de la aplicación *Control Camera IP* en el entorno *Microsoft Windows 7*

Para el proceso de conexión con la cámara utilizada en la realización del PFC (Axis Domo modelo 232D+) se creó una pasarela de comunicación haciendo uso de un túnel SSH para encapsular las peticiones HTTP realizadas por la aplicación a la cámara, estableciéndose por tanto una comunicación cifrada que garantizara la seguridad y evitara el acceso de terceras personas.

Este túnel permitía redirigir la información hacia el destino especificado. En nuestro caso, utilizábamos el cliente PuTTY para conectar con la dirección del host del Departamento de Arquitectura de Computadores de la E.T.S. de Ingeniería Informática de la Universidad de Málaga (*ssh.ac.uma.es*). Una vez establecida la comunicación la información se redirigía a través de la técnica *port_forwarding* a la dirección privada y el puerto de conexión configurados para la cámara. Finalmente, el túnel quedaba establecido redirigiendo la información enviada por la cámara a la dirección local de nuestro dispositivo (*localhost*) en el puerto 8080.

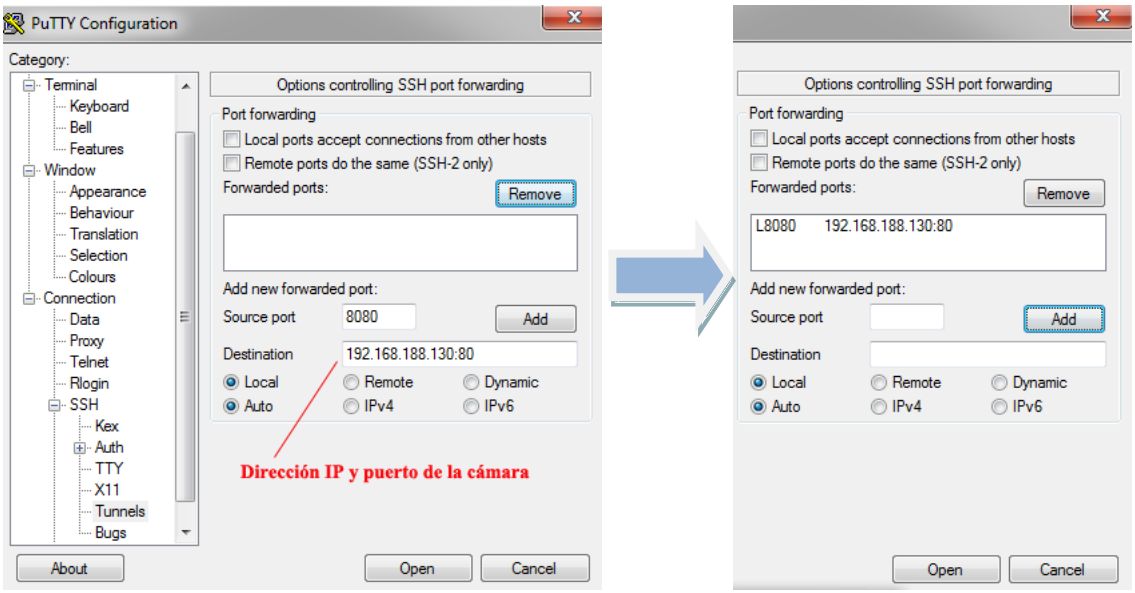

Figura 4. Creación del túnel de comunicación con la cámara con PuTTY

Una vez establecida la conexión directa con la cámara, la aplicación *Control Camera IP* dispone de una opción de inicio de sesión en el menú *Archivo* que muestra un diálogo de *login* o autenticación.

A través de este diálogo la aplicación aplicará un segundo nivel de seguridad sobre el usuario que intenta identificarse, en el que se determinará si figura en el registro de usuarios de la cámara, así como el nivel de privilegios en función del grupo al que pertenezca. Una vez aceptado el diálogo, la aplicación se inicia mostrando las imágenes enviadas por la cámara en tiempo real, y permitiendo al usuario interactuar con los distintos controles disponibles.

2.1.2 Funciones más importantes

En este apartado explicaremos algunas de las funciones principales de la aplicación original incluyendo algunos detalles de implementación. Esto nos permitirá entender

algunas de las claves más destacadas sobre las modificaciones que se han realizado para adaptar la aplicación al ámbito del presente Trabajo Fin de Grado.

El Módulo de Control

Uno de los pilares fundamentales para el funcionamiento de la aplicación está constituido por el módulo de control (*controlCamara*), que es el encargado de realizar todas aquellas peticiones a la cámara a través de comandos CGI (*Common Gateway Interface*) que devuelven la diferente información aportada por la cámara bajo la forma de objetos MIME (*Multiporpose Internet Mail Extensions*), haciendo uso del protocolo HTTP para ello.

Esto quiere decir que cada vez que la aplicación necesita hacer una petición de una imagen o de un control de la cámara determinado (*pan, tilt, focus,* etc.), ésta se realizara a través de la creación de una nueva sesión HTTP que recibirá como parámetro una URL, que está formada por el comando CGI que ejecuta el programa en la cámara para la obtención del recurso solicitado. La petición podrá ser lanzada a través de uno de los dos métodos de HTTP más utilizados, GET y POST. A diferencia del método GET, las peticiones POST permiten enviar junto con la URL información que posteriormente será procesada en el servidor HTTP. En una petición GET, esta información solo puede enviarse encapsulada dentro de la propia URL.

En la aplicación original todas las peticiones se realizan con el método GET, ya que es en la propia URL donde se incluyen los parámetros necesarios para hacer las modificaciones oportunas, y para que la cámara nos responda con los nuevos recursos modificados. La aplicación se compone principalmente de tres tipos de peticiones: petición de imagen, petición de lectura de controles y petición de escritura de controles. A continuación podemos ver un esquema de cómo estaba organizado el módulo de control dentro de la aplicación *Control Camera IP* original:

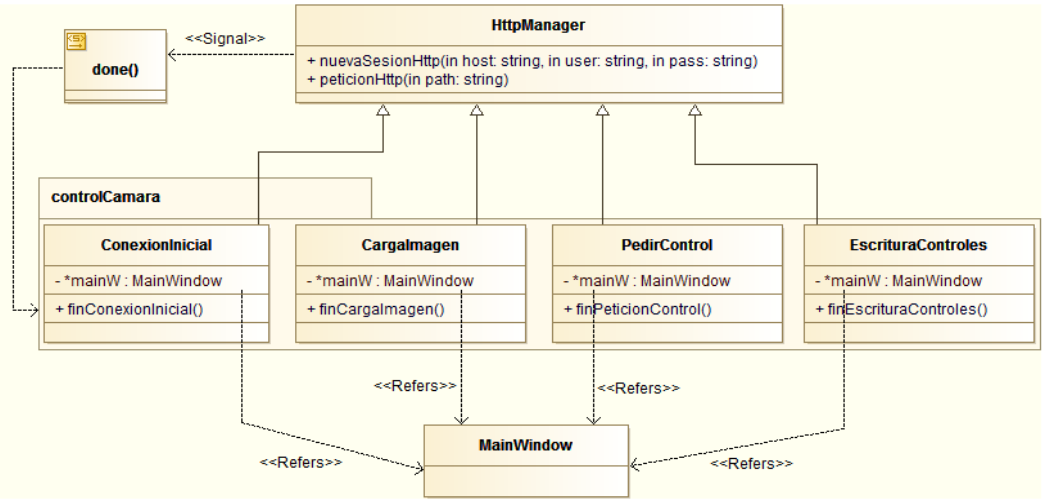

Figura 5. Estructura general de las clases del paquete '*controlCamara*'

La parte referente a la conexión inicial con la cámara se lleva a cabo a través de la combinación de dos peticiones. Por un lado una petición de los controles actuales de la cámara para poder almacenarlos en memoria, y una petición de la imagen actual que, tras su carga, dará luz verde para que se empiece a ejecutar el temporizador (*timer*) que controla la carga de las imágenes en la aplicación.

Temporización

Como acabamos de comentar, en la aplicación original la carga de las imágenes está comandada por un temporizador (esto ya no sucede así, tal y como explicaremos en la próxima sección).

El funcionamiento es sencillo: cada vez que finaliza el *Timer* de refresco se realiza una petición de carga de imagen. A continuación, el temporizador se para y el programa espera a que se reciba la señal *done()* comandada por la instancia de la clase *HttpManager*, tal y como se aprecia en la imagen anterior. Una vez recibida la señal, esto quiere decir que hemos obtenido respuesta de la cámara y que la información está almacenada en memoria. Acto seguido, la imagen es cargada en el monitor de la interfaz principal, y es en ese momento cuando se vuelve a lanzar el *Timer* de carga de la nueva imagen.

En el caso de las peticiones de lectura y escritura de controles, el proceso se realiza de forma similar, habiéndose implementado diferentes temporizadores para cada tipo de petición.

La razón de la utilización de temporizadores para el manejo de las peticiones dentro de la aplicación fue debida a la falta de mecanismos de control de la clase de QHttp (disponible en la versión 4.8 de la biblioteca Qt pero no en la versión 5) para manejar las sucesivas peticiones asíncronas que pueden darse en una sesión HTTP, de forma que en muchos casos la aplicación no era capaz de recibir correctamente la información ya que ésta se solapaba con la información recibida de otras peticiones que se ejecutaban en un momento anterior o posterior, pudiendo por tanto llegar a producir errores de red y en algunos casos la salida del programa. Añadiendo por tanto el mecanismo de temporización comentado anteriormente se solucionaron estos problemas.

En la nueva versión 5 de Qt, la clase QHttp utilizada en la aplicación original ha quedado obsoleta y, por tanto, se hace necesario sustituirla por otras funciones del módulo de red que se han mantenido en la versión 5, y que han resultado ser incluso más eficientes en relación a los inconvenientes que hemos comentado. Es por ello que, al no seguir existiendo la clase QHttp en la nueva versión de Qt, el módulo de control de la aplicación se ha modificado casi por completo incluyendo un nuevo mecanismo de peticiones HTTP para obtener los recursos de la cámara. Aunque por un lado nos hemos visto en la obligación de tener que modificar gran parte del código de la aplicación, las nuevas funciones de red de Qt utilizadas en esta nueva versión del proyecto nos permiten prescindir de las temporizaciones, ya que ahora las respuestas a las peticiones HTTP sí son controladas por las propias clases de Qt, por lo que se simplifica el control sobre la recepción de información de la cámara por parte de la aplicación.

Este proceso de adaptación de la aplicación original a la nueva versión de Qt será explicado con más detalle en la próxima sección.

Lectura y Escritura de Controles

Tal y como ya hemos visto, a parte de la carga de las imágenes de la cámara en el monitor, la aplicación original consta de funcionalidad básica para el manejo de los controles principales de la cámara de red. Concretamente, los controles con los que podemos interactuar son el *pan* (movimiento horizontal de 360º), el *tilt* (movimiento vertical de 90º), el *zoom*, el *focus* (enfoque de la cámara) y el *iris* (exposición lumínica). Además, también podremos modificar la resolución de la imagen atendiendo a tres tipos de resolución: *QCIF* (baja), *CIF* (media) *y 4CIF* (alta).

Cada vez que iniciamos la aplicación obtenemos de la cámara los valores de sus controles actuales y los guardamos en un fichero de texto llamado *Inicial.txt*. De esta forma si en el transcurso del tiempo en el que estamos usando la aplicación hemos modificado los controles de posición, zoom o enfoque de la cámara y queremos volver a los controles iniciales, simplemente leeremos los controles de ese fichero inicial.

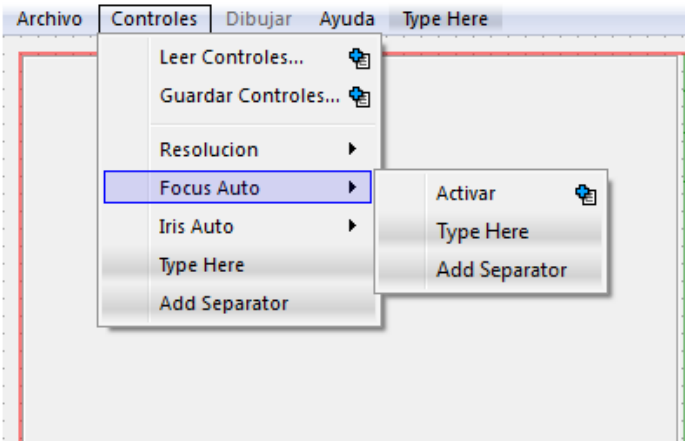

Figura 6. Grupo de acciones del menú Controles

Para la lectura de controles en la aplicación se creó un mecanismo que ejecutaba un proceso de carga escalonada a través de temporizadores que permitía solventar los problemas relativos a los solapamientos de las diferentes sesiones HTTP creadas con la clase QHttp de la versión 4.8 de Qt. Debido a los cambios realizados para adaptar la aplicación a la versión 5, este mecanismo ya no será necesario.

De forma similar, la parte relativa al proceso de escritura de controles en un fichero plano se verá modificada para ser adaptada a la nueva versión de Qt.

Edición de Máscaras

Otra funcionalidad que se desarrolló como uno de los objetivos para el PFC fue la posibilidad de crear y editar máscaras que pudieran superponerse sobre la imagen, utilizando para ello las funciones de dibujo en 2D que incorpora la biblioteca Qt.

El módulo de edición y gestión de máscaras se compone por un lado de opciones de edición que nos permiten dibujar líneas y polígonos sobre el monitor de la aplicación (acciones del menú Dibujo), y por otro, de varias opciones de creación, almacenamiento y carga de máscaras a partir de un fichero de texto (acciones del menú Archivo).

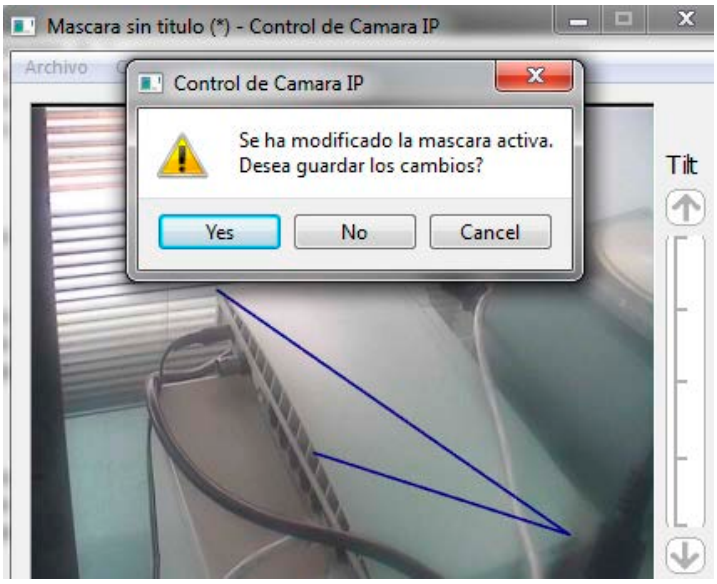

Figura 7. Edición de máscaras con la aplicación *Control Camera IP*

Al guardar una máscara, ésta se almacena de dos formas. En primer lugar, tal y como hemos comentado, la máscara se almacena en un fichero de texto en forma de vector de coordenadas. Este vector guarda una relación de coordenadas relativas a la imagen cargada en el monitor, en base a las coordenadas del mismo en la aplicación. Por lo tanto, este fichero únicamente es utilizado por la aplicación como información para dibujar de nuevo la máscara sobre el monitor en las coordenadas exactas donde se dibujó inicialmente.

En segundo lugar, al guardar una máscara, ésta también se almacena como una imagen en formato PBM (*Portable BitMap*). Este tipo de formato, caracterizado por ser mapas de bits monocromáticos fácilmente portables, permite el intercambio de imágenes entre distintas plataformas de forma sencilla.

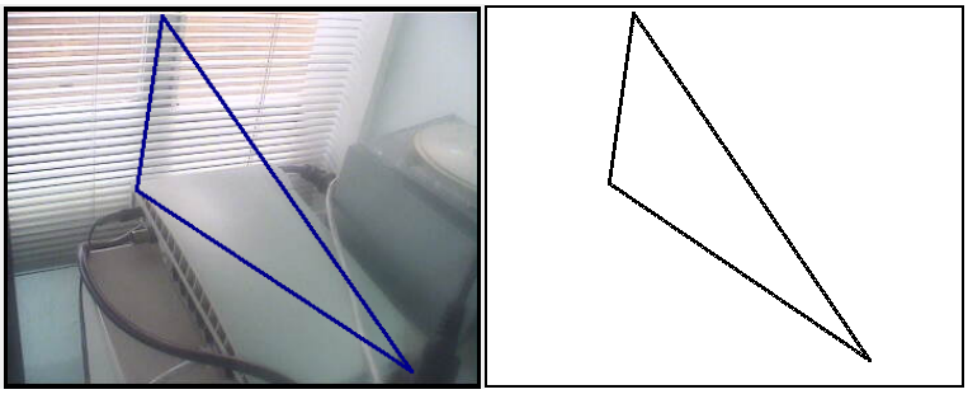

Figura 8. Máscara activa en la aplicación (izquierda) y máscara PBM (derecha)

2.1.3 Creación de Múltiples Plataformas

Otro de los elementos interesantes de la aplicación original es la capacidad de poder configurarse para poder ejecutarla en entornos variados (*desktop, smartphones, etc.*), tal y como ya comentamos en el inicio de esta sección.

Para ello, en el modo Proyectos de la herramienta de desarrollo integrado QtCreator se configuran las diferentes plataformas sobre las que se quiere compilar el proyecto (compilación cruzada). Además, en las propias opciones de construcción del proyecto con *qmake*, a través del fichero *.pro* del proyecto se pueden configurar diferentes *scopes* o "ámbitos" para construir nuestro proyecto. Los ámbitos establecen qué recursos (librerías, código fuente, etc.) han de cargarse en la fase de compilación en función de la plataforma seleccionada para la ejecución de la aplicación.

```
win32 {
    FORMS += plataforma/windows/mainwindow.ui
}
android {
    FORMS += plataforma/android/mainwindow.ui
}
```

De esta forma, si el entorno es por ejemplo Microsoft Windows, los recursos asociados a esta plataforma se pondrán bajo el ámbito *win32*. Si la plataforma es Android utilizaremos en cambio el ámbito *android*. De esta forma, se podrán cargar diferentes vistas para la aplicación personalizadas en función del tamaño del dispositivo, además de poder cargar diferentes parámetros de configuración, tales como resolución, tamaño de la fuente, etc. según la necesidad.

Gracias al proyecto *Necessitas Qt for Android*, se pudo configurar la aplicación para que ésta pudiera ser lanzada en dispositivos *Android*, realizándose una demostración en la propia presentación del PFC sobre una *Tablet PC* modelo *ViewPad-S10* con sistema operativo *Android ViewComb v3.2*.

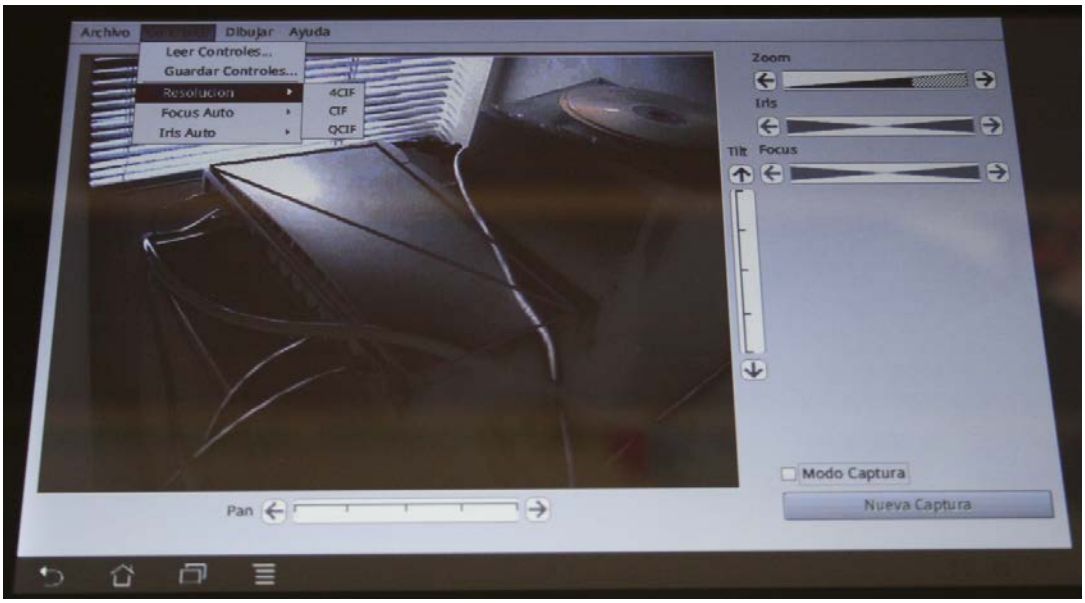

Figura 9. Vista de la interfaz gráfica para el Tablet PC *ViewPad-S10* con Android-*ViewComb*

Tras esta breve descripción de la aplicación *Control Camera IP* original, explicaremos a continuación las modificaciones que se han llevado a cabo para adaptar dicha aplicación a la versión 5 de Qt, y tener así la posibilidad de incorporar de forma oficial la plataforma *Android* al proyecto Qt.

2.2 Adaptación de la aplicación a Qt 5

Como ya hemos comentado con anterioridad, con el paso de Qt de la versión 4 a la versión 5 algunas de las funciones, sobre todo las relacionadas con los aspectos de red, que se utilizaron para la aplicación desarrollada en el PFC original se han redefinido en la nueva versión, por lo que han quedado obsoletas.

Esta incompatibilidad entre funciones requiere aplicar al código original una serie de modificaciones para adaptarlo a la nueva versión de Qt. A continuación detallamos las modificaciones realizadas en el código de la aplicación atendiendo a cada una de las funciones afectadas:

2.2.1 QWidgets como módulo separado

Este cambio no ha sido muy significativo pero sin embargo ha provocado la necesidad de tener que realizar algunas modificaciones en el proyecto Qt. El problema surge debido a que, a partir de la versión 5, el módulo *widgets* se encuentra disponible como un módulo separado por lo que hay incluirlo dentro del fichero de proyecto (*.pro*) como un módulo más.

En la versión 4.8 el módulo utilizado en su lugar era el módulo *gui*, por lo que ha sido necesario sustituir la siguiente línea del fichero de proyecto:

```
QT   +=   gui \
```

por la línea:

```
QT   +=   widgets \
```

Además, también será necesario modificar en la clase principal *MainWindow* la línea de código:

```
#include <QtGui>
```

por la línea:

```
#include <QtWidgets>.
```

De esta manera, el objeto de la clase MainWindow relativo a la interfaz de usuario principal que se crea en base a la clase QMainWindow de Qt, dejará de provocar en el momento de su creación el error de que la clase QMainWindow no ha sido encontrada.

2.2.2 Cambios en las funciones de red

Las modificaciones que se han tenido que llevar a cabo en las funciones que llevaban a cabo las comunicaciones con la cámara IP han sido mucho más importantes.

Para empezar, el módulo de red (Q*Network*) de Qt, ha sufrido un cambio radical al migrar a la versión 5, de forma que la clase QHttp utilizada para todas las peticiones realizadas a la cámara haya sido eliminada del módulo, y reemplazada a su vez por un conjunto de clases tales como QNetworkAccessManager y QNetworkReply, que realizan una labor parecida. Este hecho ha provocado tener que modificar gran parte del código relativo al módulo de control de la cámara y del programa principal, habiendo tenido que redefinir gran parte del esquema de la aplicación original. Sin embargo, estos cambios han permitido crear una estructura mucho más simple, ya que con las nuevas clases del módulo de red de Qt se ha simplificado bastante la forma en la que se realizan las peticiones HTTP sobre la cámara, y además ha permitido eliminar ciertas funcionalidades del código (tales como las temporizaciones), que a partir de ahora dejarán de ser necesarias para sincronizar las diferentes peticiones.

2.2.2.2 Las nuevas clases del módulo *Network*

QNetworkAccessManager y QNetworkReply son las clases oficiales para el manejo de comunicaciones básicas en el protocolo HTTP que incorpora la versión 5 de Qt, y que sustituyen en funcionalidad a la clase QHttp, la cual deja de existir para esta versión.

Lo primero que podemos observar del nombre de las clases es la separación clara que se hace entre la gestión y control de acceso a la red (función realizada por QNetworkAccessManager), y la respuesta de las comunicaciones (realizado por QNetworkReply). Así, las instancias de QNetworkAccessManager, de forma similar a como la hacía la clase *HttpManager*, permitirán crear peticiones (QNetworkRequest) tipo GET y POST para atender todas las demandas de recursos de la cámara que requiera aplicación.

Tras hacer una petición, la instancia de QNetworkAccessManager crea un objeto de tipo QNetworkReply y lo pone en estado abierto en modo lectura. Tras hacer esto, la instancia de QNetworkReply, al heredar de QIODevice, emite la señal *readyRead()* cada vez que hay nuevos datos disponibles para la lectura en el dispositivo. Por lo tanto, conectando esta señal convenientemente a un SLOT tras la petición, se puede capturar la información que está disponible conforme va llegando tras hacer la petición (que recordemos es asíncrona). No obstante, lo más interesante de QNetworkReply es que dispone de una señal *finished()*. Cuando se recibe esta señal, esto quiere decir que la respuesta se ha terminado de procesar, por lo que no se recibirán más datos. Conectando un SLOT a esta señal se puede utilizar el método *readall()* para obtener todos los datos almacenados en la respuesta y volcarlos a un *buffer*.

Una vez la instancia de QNetworkReply emite la señal *finished()* la instancia de QNetworkAccessManager también se hace eco de ello y emite a su vez su propia señal *finished(QNetworkReply*)*¸ pasando como parámetro al SLOT receptor (que puede ser cualquier que definamos) una referencia a la respuesta con todo su contenido. Con esto evitamos que conectar la señal *finished()* de QNetworkReply con ningún SLOT, tal y como comentábamos en el párrafo anterior, ya que directamente podemos tratar esa misma respuesta en el SLOT conectado a la señal *finished()* de QNetworkAccessManager.

Por otro lado, QNetworkReply dispone de una señal *error(NetworkError)* que envía un código de error al SLOT al que se conecte. En este caso QNetworkAccessManager no

dispone de una señal similar que se emita cuando exista un error en la respuesta (solo dispone de una para errores de certificación SSL), por lo que esta señal deberá manejarse dentro del ámbito de QNetworkReply.

2.2.2.1 Eliminación de la clase auxiliar *HttpManager*

La clase *HttpManager* del módulo de funciones auxiliares de *Control Camera IP*, era utilizada como puerta de salida o de enlace (*Gateway*) para realizar las distintas peticiones por parte de la aplicación a la cámara. Así, cada vez que algún elemento del módulo de control requería hacer una petición de un recurso (imágenes, actualización de controles, etc.), la entidad *HttpManager* montaba una petición tipo GET o POST a través de la URI (*Uniform Resource Identifier*) que recibía como entrada y la lanzaba. La respuesta era controlada a través de una conexión *SIGNAL/SLOT* de fin de petición, devolviéndose el recurso en forma de respuesta HTTP si todo había ido bien, o un error en caso de que hubiera sucedido algún fallo en la comunicación con la cámara.

En el nuevo esquema, haciendo uso de las clases QNetworkAccessManager y QNetworkReply, se prescinde de la clase *HttpManager*¸ debido a que ya no es necesario conectar la señal de error y finalización de QHttp con cada una de las clases que gestionan las peticiones, y capturar la respuesta en la clase principal MainWindow, sino que directamente cada una de estas clases pueden heredar de QNetworkAccessManager, permitiendo así gestionar la respuesta, y los mensajes de error directamente dentro de la propia clase de petición de recursos.

Este hecho simplifica bastante la estructura del módulo de control la cual detallaremos a continuación.

2.2.2.3 La nueva estructura del módulo de control

En la figura 4 del apartado 2.1.2 de este Trabajo, podíamos apreciar la estructura del módulo de control de la aplicación original. Tras los cambios comentados en los apartados anteriores, la nueva estructura quedaría del siguiente modo:

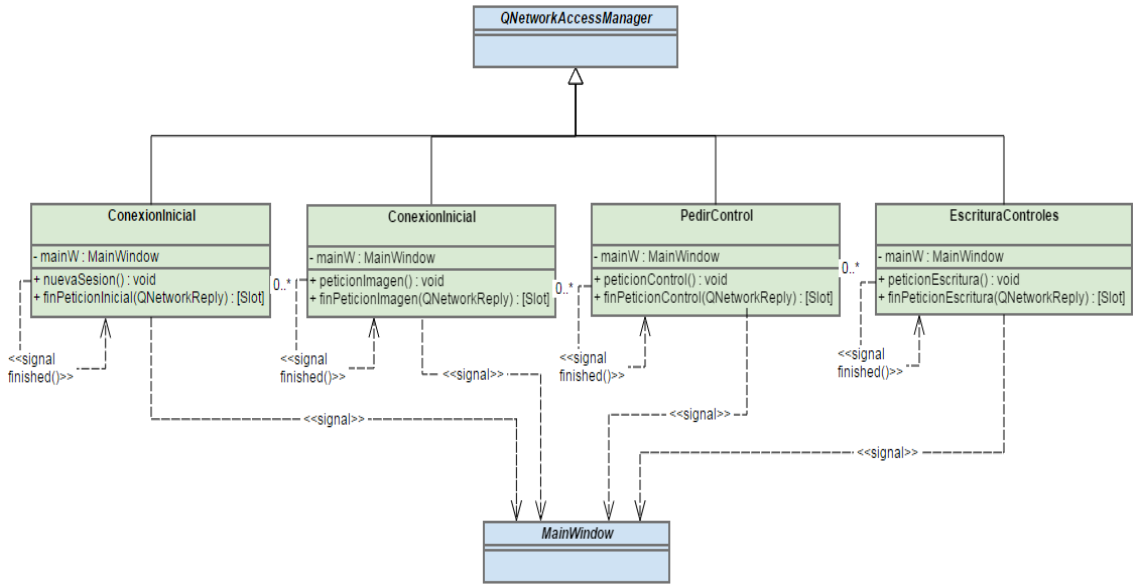

Figura 10. Nueva estructura del módulo de *control* de la aplicación *Control Camera IP*

Tal y como podemos apreciar en el esquema anterior, cada una de las clases del módulo de control se encargan ahora de gestionar sus propias peticiones haciendo uso de las funcionalidades de QNetworkAccessManager. Las señales de error y finalización se tratan directamente dentro de las mismas, permitiendo de esta manera obtener la respuesta en el mismo ámbito de la petición y poder así generar un *arrayBuffer* con la información que será enviada directamente a la interfaz principal MainWindow a través de las señales correspondientes para que pueda ser tratada por la aplicación.

Este mecanismo no solo nos ofrece un nivel de simplificación bastante elevado con respecto al anterior esquema, sino que también permite manejar de forma mucho más eficiente los tiempos de respuesta entre peticiones.

Internamente, cada una de las clases de petición de recursos a la cámara implementarán una petición GET o POST en función de las necesidades. Tal y como ya hemos comentado, este tipo de comunicaciones son de naturaleza asíncrona, por lo que la señal *readyRead()* de QIODevice se lanzará cada vez que existan datos disponibles en la respuesta, pero esto no querrá decir que toda la información haya sido recibida en su totalidad. A diferencia del mecanismo utilizado a través de la clase QHttp, las peticiones en este caso no necesitarán habilitar o deshabilitar el modo *ReadReady* para la lectura escalonada de los controles por ejemplo, o para la escritura de los mismos en un fichero de texto. Este mecanismo es cubierto como ya hemos comentado por la señal

finished(QNetworkReply)* de QNetworkAccessManager, que permite devolver la respuesta integra en el momento en que se lanza la señal de finalización de la petición.

Para construir una petición GET o POST en esta nueva estructura se incluirá como argumento de entrada a la misma un objeto de tipo QNetworkRequest. Esta clase permite crear una petición HTTP desde cero a partir de los diferentes elementos que la integran (URL, atributos, cabeceras, contenido, etc.). En principio, para nuestro caso, solo es necesaria la URL puesto que el programa CGI de la cámara se encarga de montar las cabeceras y el contenido de la respuesta sin necesidad de tener que enviarle más información. Sin embargo, aunque todas las peticiones se puedan hacer con GET pasando los argumentos en la URL, gracias a la clase QNetworkRequest es posible realizar peticiones POST a la cámara encapsulando los argumentos dentro del cuerpo (*body*) de la petición con QUrlQuery. De esta forma, se pueden separar las peticiones que requieran el envío de parámetros a la cámara (como por ejemplo las peticiones de modificación de algún control), por peticiones POST, dejando aquellas que solo requieren recibir los recursos de la cámara como peticiones GET.

2.2.3 Eliminación de la clase auxiliar *Temporizador*

Una de las cuestiones que se ha eliminado después de las modificaciones del módulo de control de peticiones a la cámara, ha sido la necesidad de incluir temporizaciones (*timers*) para evitar posibles conflictos que ocurrían al recibir la respuesta de las peticiones HTTP debido a la mayor dificultad que tenía la clase QHttp para el control de la recepción de respuestas múltiples a múltiples peticiones (tales como las peticiones de secuencias de imágenes). Había por tanto que evitar el conflicto entre peticiones a través de algún mecanismo.

La clase *Temporizador* del módulo de funciones auxiliares se creó para solventar este problema. Disponía de una serie de *timers* que lanzaban una señal de finalización tras un tiempo de finalización (*timout*) determinado y que permitían en ese momento lanzar la petición requerida. Antes de activar un *timer* (como por ejemplo la petición de un control a la cámara), el resto de *timers* que pudieran estar activos se paraban, de forma que no interfirieran en esa petición concreta. Tras finalizar la petición, el resto de *timers* volvían a habilitarse.

Este tipo de temporizaciones ayudaron también a implementar un mecanismo de carga de controles de un estado de la cámara a través de un fichero, y cuyo proceso era

manejado de forma escalonada por diferentes *timers* que realizaban la carga de controles de forma paulatina. A este mecanismo se le llamó *transición de carga*, y en la siguiente captura se puede ver un esquema simple de su funcionamiento:

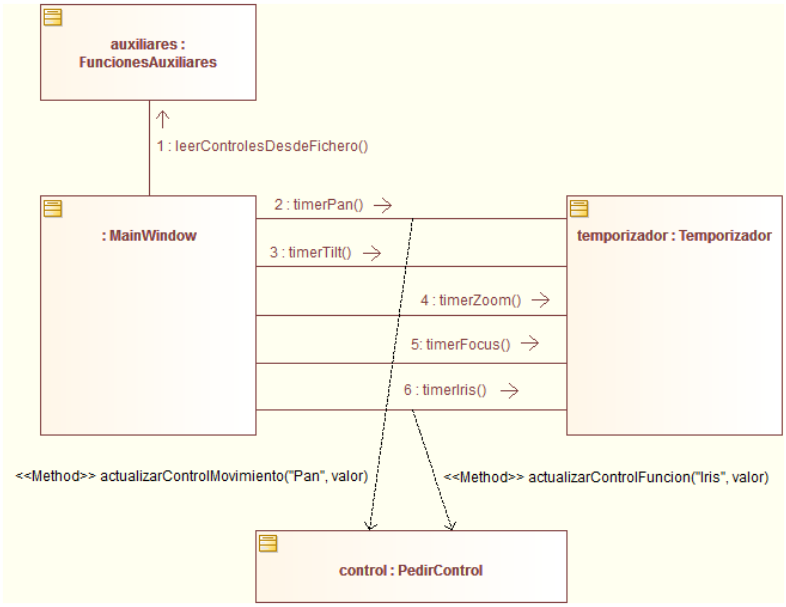

Figura 11. Lectura de controles mediante el proceso de "transición de carga"

Tal y como ya hemos comentado anteriormente, las nuevas funciones de comunicación con la cámara de la clase QNetworkAccessManager, se resuelve el conflicto entre peticiones debido a la diferenciación en el manejo de señales entre cada una de las clases que gobiernan las peticiones HTTP. En consecuencia, ahora las respuestas se procesan en SLOTs internos de la propia clase de forma excluyente, por lo que cualquier otra respuesta entrante deberá esperar su turno para procesarse.

Se elimina por tanto la clase *Temporizador* y todos aquellos puntos del código donde se crean y eliminan los *timers*. El temporizador de la carga de imagen deja de ser necesario. Con el slot *finish()* de QAccessNetwork se controla que no se cargue la siguiente imagen hasta que no se ha cargado la anterior. De esta forma, el tiempo de *delay* entre capturas se ajusta a la capacidad de transmisión de la red y no a un *timer*, tal y como ocurría antes.

Finalmente, siempre que se produzca alguna petición de control, se para la carga de imagen y se reanuda después de la petición, tal y como ya ocurría antes, con la diferencia de que ahora no tendremos que controlar los temporizadores. La transición de carga se ha adaptado de forma que ahora se utilizan una serie de *flags* que van emitiendo señales que lanzan de forma sincronizada la carga de los controles. Al igual

que con la carga de imagen, el tiempo de *delay* se ajustará a la transmisión de la red y no al *timer*.

2.3 Creación de Plataformas *Android* en Qt 5

A partir de la versión 5, Qt incorpora la plataforma Android para el desarrollo de aplicaciones heredando la funcionalidad directamente del proyecto *Necessitas* que se había iniciado unos años antes para hacer compatible la versión 4.8 con la creación de aplicaciones para la plataforma Android. El entorno QtCreator es prácticamente un calco al entorno de la versión *QtCreator Necessitas for Android*, utilizada para desarrollar la aplicación original, así como sus opciones de configuración.

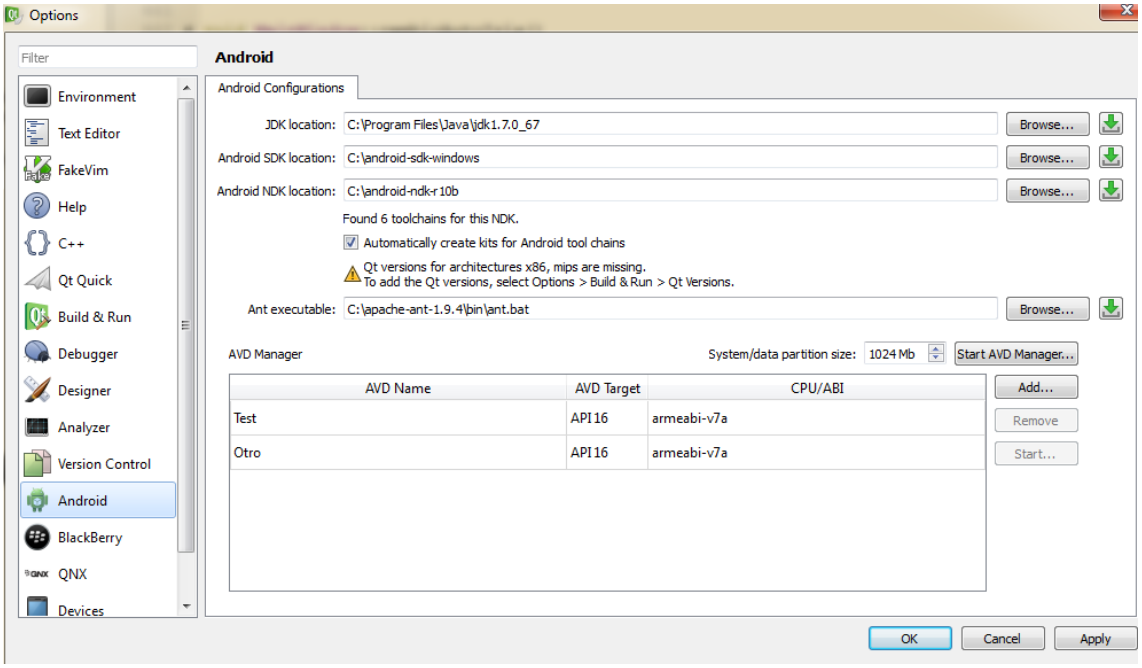

Figura 12. Configuración para los dispositivos Android en la versión 5 del entorno Qt Creator

Una funcionalidad nueva que incorpora el panel de configuración de opciones de Android de la imagen anterior, es la posibilidad de descargar las JDK, SDK y NDK directamente desde el mismo panel. Una vez configurados todos los parámetros, se pueden añadir *AVDs (Android Virtual Devices*) al entorno para probar en modo simulación. No obstante en este Trabajo Fin de Grado, del mismo modo que se hizo en el PFC original, las pruebas se realizarán sobre el dispositivo Tablet PC *ViewPad-S10* con sistema operativo Android-*ViewComb*.

La parte de creación de plataformas del modo Proyectos de QtCreator tampoco ha cambiado mucho en relación a la versión de *Necessitas*, a excepción de incorporar

algunas opciones más en la configuración de compilación y ejecución (como es la firma de paquetes y el uso de certificaciones).

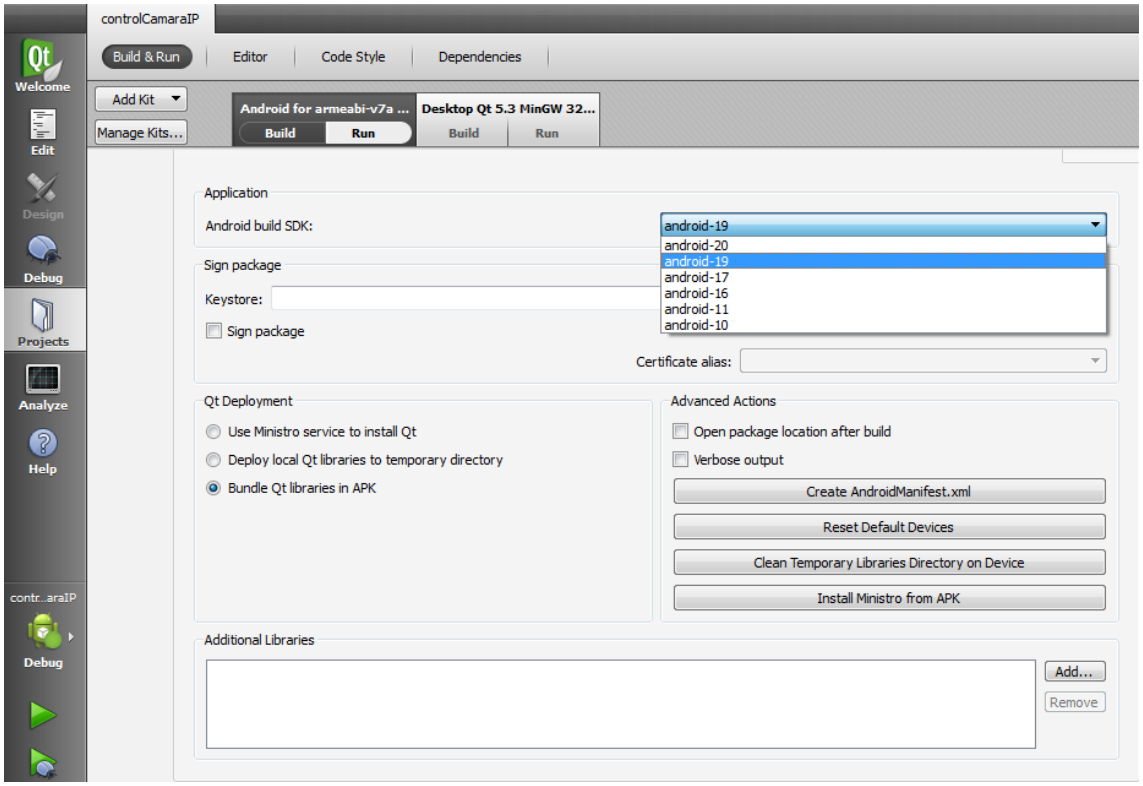

Figura 13. Configuración de plataforma Android en la versión 5 del entorno Qt Creator

Como observamos en la imagen, al configurar la plataforma podemos seleccionar entre otras cosas la versión del API de Android del dispositivo sobre el que deseamos instalar la aplicación. También nos permite instalar la aplicación Ministro [12] para cargar las librerías de Qt en el dispositivo, así como crear el *AndroidManifest.xml* para poder configurar parámetros de la aplicación Android dentro del propio QtCreator.

A partir de aquí los pasos a realizar para instalar la aplicación Android en cualquier dispositivo serán los mismos que los llevados a cabo en el PFC original. Por otro lado, para incluir las librerías de OpenCV para el procesamiento de vídeo en la configuración de este tipo de plataformas hemos tenido que realizar algunos ajustes que incluyen entre otros, añadir la librería *opencv_java* de la versión de OpenCV para Android.

Estos ajustes y correcciones surgidos en la configuración de OpenCV serán descritos con más detenimiento en el siguiente capítulo, donde también realizaremos un análisis de las nuevas funcionalidades de procesado de video añadidas a la aplicación a través de esta librería.

Capítulo 3

Funcionamiento de la aplicación en Cámaras IP de bajo coste

Ante las nuevas funcionalidades añadidas en este Trabajo Fin de Grado a la aplicación *Control Camera IP*, en relación al tratamiento y a la aplicación de técnicas de detección sobre las imágenes recibidas desde la cámara IP A232D, utilizando para ello las librerías de la biblioteca de código abierto de OpenCV, se plantea la necesidad de verificar, también en este trabajo, que la aplicación es fácilmente adaptable para su funcionamiento en cámaras IP de distinto tipo y modelo al empleado hasta este momento en las pruebas realizadas.

De cara a la implantación de futuras soluciones "s*mart*" para la videovigilancia, es interesante la posibilidad de maximizar la compatibilidad de nuestra aplicación con el mayor número de modelos de cámaras IP del mercado, incluyendo tanto las gamas más altas como las de bajo coste, pudiendo por tanto flexibilizar al máximo la creación de soluciones de videovigilancia IP abiertas y fácilmente adaptables.

Sin embargo, este propósito no es fácil de conseguir. Para empezar, muchas de las empresas venden sus productos de video IP dentro de soluciones cerradas, donde sus cámaras IP utilizan protocolos específicos para la petición de recursos desde sus propias aplicaciones de gestión de video, también cerradas. Por otro lado, no todas las cámaras disponen de APIs (*Application Programming Interface*) abiertos para su libre uso y distribución, y/o estandarizados, de forma que una misma API común pueda ser utilizado en una amplia gama de productos de diferentes marcas y proveedores.

En relación a este último caso, sí que es factible hoy en día encontrar estándares de APIs para cámaras IP en aquellos modelos de bajo coste. La ventaja de las cámaras IP de bajo coste está ligada al hecho de que suelen estar dirigidas a un mercado de clientes mucho más amplio que el de las cámaras que forman parte de soluciones más sofisticadas, sobre todo dirigidas al ámbito de la seguridad, y que consecuentemente, suelen ser más caras de cara al usuario medio. Es por tanto interesante explorar las capacidades que nos ofrece las cámaras IP de bajo coste, no solo por su reducido precio,

sino también por la posibilidad que nos brindan para compatibilizar nuestra aplicación con una gran variedad de productos IP a través de un protocolo abierto y estandarizado.

Una de estas APIs estandarizadas que cada vez está siendo más seguida por fabricantes de cámaras IP de bajo coste, es el API IPCAM CGI de Foscam [13]. Se trata de una API basada en comandos CGI, que engloba una serie de funciones de las cámaras IP más comunes, incluyendo funcionalidades PTZ (*Pan*, *Tilt*, *Zoom*), modos de vigilancia, etc.

A diferencia del API CGI de las cámaras Axis (VAPIX) [14], el estándar abierto de Foscam incluye una aplicación CGI llamada *decoder_control.cgi* en la cual se integran todos los comandos más comunes que pueden aplicarse a las diferentes cámaras que sigan el estándar, de forma que si alguna cámara no dispone de un comando concreto, simplemente no tomará en cuenta la orden dada.

En este capítulo, estudiaremos más detenidamente este API estandarizado, y analizaremos también algunos de sus CGIs, incluido el CGI *decoder_control* con algunos de sus comandos más utilizados. No obstante, previamente recordaremos de forma breve el funcionamiento del API de las cámaras AXIS (VAPIX), y algunos de los comandos CGI utilizados en la aplicación *Control Camera IP*.

3.1 VAPIX, el API de AXIS

La aplicación original, utiliza para la conexión con la cámara A232D un API del fabricante AXIS denominado VAPIX. Este API, al igual que el API estandarizado IPCAM CGI que veremos a continuación, está conformado por un conjunto de funciones CGI que permiten la recepción de imágenes y parámetros, así como la ejecución de todas aquellas acciones para el control remoto de las cámaras IP de AXIS.

Entre las funciones más destacadas de VAPIX que son lanzadas dentro de la aplicación *Control Camera IP*, encapsuladas dentro de peticiones HTTP, podemos destacar las siguientes:

Recepción de imágenes con *image.cgi*

La función *image.cgi* de VAPIX, permite la recepción de imágenes JPG con una compresión y resolución establecidas por los parámetros *compression* y *resolution* respectivamente. Las posibles resoluciones de las imágenes para la cámara A232D son tres: 4CIF, CIF y QCIF, las cuales pueden ser obtenidas directamente de la aplicación a

través del fichero de configuración *valores.txt* de cada plataforma. Por tanto, la expresión utilizada para realizar peticiones de imágenes a la cámara sería la siguiente:

```
axis-cgi/jpg/image.cgi?resolution=<4CIF|CIF|QCIF>&compression=25
```

El valor de compresión se establece a 25 para todas las peticiones de imágenes en la aplicación.

Obtención de controles PTZ

La cámara A232D dispone de los tres controles PTZ que caracterizan a este tipo de cámaras IP, y que hace referencia a aquellas cámaras que son capaces de realizar movimientos en horizontal (*Pan*) hasta 360º y en vertical (*Tilt*) hasta 90º. Además también disponen de la función *Zoom*, que en el caso de la cámara A232D, su óptica integrada permite realizar un aumento de hasta 10X.

Para utilizar los controles PTZ en la aplicación original, se utiliza la función *ptz.cgi*, a la cual se le puede pasar una serie de parámetros en función de la acción a realizar. He aquí algunos de los utilizados en la aplicación *Control Camera IP*:

- Parámetro "move": con los valores *up, down, left y right*, hace que la cámara se mueva un solo paso a alguna de las direcciones indicadas.
- Parámetro "[r/d] zoom": permite incrementar *(rise*) o decrementar (*decrease)* el zoom de la imagen.
- Parámetro "<control>=<grados>": establece el control *Pan* o *Tilt* en una posición determinada por el valor en grados asignado al parámetro.
- Parámetros [r/d] focus e iris: similar al parámetro de zoom pero para incrementar o decrementar el enfoque de la imagen (*focus*), o la apertura (*iris*).

Además de los parámetros anteriores, el parámetro *query* con el valor *position*, permite obtener los valores de los controles actuales de la cámara en texto plano. Esto permite que se puedan almacenar los controles de la cámara en el fichero de controles (*controles.txt*), el cual es utilizado posteriormente para la carga en la aplicación de unos controles determinados a través de la acción de lectura de controles.

3.2 El API estandarizado *IPCAM CGI*

Foscam es una empresa de fabricación de productos de video ubicada en China, especializada en dispositivos de vigilancia, grabación de video y monitorización de

bebés. Hoy en día es mundialmente reconocida, gracias sobre todo a que sus productos ofrecen una alta funcionalidad práctica a un precio razonable, abarcando por tanto un mercado de clientes muy amplio.

Figura 14. Algunos modelos de Cámaras IP de Foscam

Es por eso mismo que Foscam decidió crear un sencillo API basado en CGI (IPCAM CGI), que permitiera a los usuarios desarrollar sus propias soluciones y aplicaciones para conectarse e interactuar con las cámaras Foscam, pero que no limitara la posibilidad de utilizar este mismo API con otro tipo de cámaras que quisieran seguir esta especificación. Así, con el tiempo, algunos fabricantes adoptaron este API, convirtiéndose en un estándar muy utilizado hoy en día sobre todo en productos de video IP de bajo coste.

Entre los fabricantes que hoy en día permiten el uso del API de Foscam en sus cámaras podemos encontrar por ejemplo a Solwise [15], HooToo o INSTAR [16].

A continuación, vamos a analizar brevemente, algunas de las funciones más utilizadas del API IPCAM CGI (sobre todo aquellas que utilizaremos para adaptar nuestra aplicación), comenzando por las funciones de estado y de obtención de parámetros de la cámara.

Hay que recordar, que una petición CGI se realiza a la cámara a través del protocolo HTTP, de igual forma que hacemos para conectarnos a la cámara A232D. En el caso del API IPCAM CGI, las peticiones son realizadas siempre con el mismo formato:

```
http://<dirección_ip>/<comandoCGI>.cgi[?param=valor&...]
```

donde la expresión `[?param=valor&...]`indica la secuencia de parámetros opcionales que puede recibir la función CGI.

3.2.1 Funciones de estado y de obtención de parámetros de la cámara

La función *getStatus.cgi* devuelve un contenido en texto plano (*plain/text*) con la información general de la cámara entre la que podemos incluir la siguiente:

- Id: identificador del dispositivo
- Now: la hora actual
- Alarm Status: estado de la alarma, donde el valor recibido indica si la alarma está desactivada, si está activa la alarma por detección de movimiento o si está activa la alarma de detección por voz.
- Wifi Status: en caso de que se trate de una cámara Wifi, indica si la conexión Wifi está activa o no.

La función get_camera_params.cgi en cambio, obtiene los parámetros establecidos en la cámara también a través de texto plano. Los parámetros estándar que devuelve esta función son:

- Resolution: resolución de la cámara (QVGA, VGA, etc…)
- Brightness: brillo de la imagen (valores en el rango [0 – 255])
- Contrast: contraste de la imagen con valores entre 0 y 6
- Flip: permite voltear la imagen de la cámara tanto horizontal como verticalmente (muy útil para cámaras fijadas al techo)

Hay que destacar que al ser un API diferente al de AXIS, muchos de los parámetros de las funciones CGI no coincidirán ni en su nombre ni en sus valores, por lo que habrá de ser tenido en cuenta al tratar de hacer la adaptación de un API al otro.

3.2.2 La función *camera_control.cgi*

Esta función permite modificar los parámetros del sensor de la cámara entre los que se incluyen algunos de los parámetros comentados en la función *get_camera_params*. Como ya vimos, para establecer los parámetros utilizaremos la expresión `[?param=valor&...]`. Por tanto, si queremos modificar por ejemplo el brillo de la imagen al valor 135 realizaremos la petición:

```
http://<dirección_ip>/camera_control.cgi?brightness=135
```

Además de los parámetros anteriores, la función *camera_control* permite modifcar el modo de trabajo (frecuencia) de la cámara a un valor de 50 o de 60 Hz, o bien al modo "outdoor", para exteriores.

Por último, el parámetro *patrol* permite establecer un modo predeterminado de barrido (vertical, horizontal, o una combinación de ambas). En algunas cámaras este modo es utilizado para invertir la imagen de la cámara (valor de barrido vertical) para cuando las cámaras están fijadas a un techo.

3.2.3 Funciones de permisos de usuario y configuración de red

La función *set_users.cgi* permite crear nuevos usuarios en el registro de acceso a la cámara, además de facilitar la asignación de permisos para los mismos. Para ello, se utiliza la siguiente nomenclatura en los parámetros de la función CGI:

```
set_users.cgi?[user1=<valor>&pwd1=<valor>&pri1=<valor>&]
                    [user1=<valor>&pwd1=<valor>&pri1=<valor>&]
                                . . .
                    [user8=<valor>&pwd1=<valor>&pri1=<valor>&]
```

De esta forma, en las cámaras que siguen el estándar API IPCAM CGI, se pueden crear hasta un total de ocho usuarios distintos a través del parámetro *user<n>*, donde *n* representa el número de usuario de ese total de ocho. El parámetro *pwd<n>* servirá para establecer la contraseña para el usuario *n*. Los valores asociados a estos parámetros serán introducidos como una cadena de texto que no podrá superar en ningún caso los doce caracteres de longitud.

El parámetro *pri<n>* hace referencia a los permisos del usuario *n*. Los permisos de acceso para este API son tres: el modo *visitor*, establecido con el valor 0, el modo *operator*, que se establece con el valor 1, y el modo *administrator*, asignado con el valor 2. Podemos apreciar, que los modos de acceso de este API son muy similares a los de las cámaras AXIS.

Es importante resaltar en este punto, que cualquier función o comando CGI que se lance haciendo uso del API IPCAM CGI, puede incorporar de forma opcional la siguiente secuencia:

```
[user=<valor>&pwd=<valor>]
```

Esta expresión dentro de cualquier petición CGI, indica el acceso a la cámara (a través de una función concreta) por parte del usuario *user* con contraseña *pwd* ya existente en

el registro de usuarios. Es decir, al igual que ocurría con el API de AXIS, las funciones o comandos CGI de API IPCAM CGI requieren diferentes permisos de acceso para poder ser ejecutadas correctamente, por lo que en muchos casos será necesario acceder a través de la identificación de un usuario que disponga de dichos permisos.

Otra funciones importantes para el acceso a la cámara son las relacionadas con la configuración de red. La función *set_network.cgi* permite establecer una red IP de acceso a través de los parámetros *ip* (dirección IP), *mask* (máscara de subred), *gateway* (puerta de enlace), etc.

Por otro lado, la función *set_wifi.cgi* permite configurar el adaptador Wifi de la cámara en caso de disponer de él. Para ello, se deben de especificar los parámetros *ssid* (identificador de red), *encrypt* (modo de encriptación, que puede ser WEP, WPA AES, etc.), *key* (clave de red), etc. Para activar o desactivar el adaptador de red se puede establecer el parámetro *enable* a 1 o 0 respectivamente.

3.2.4 Las funciones *snapshot.cgi* y *videostream.cgi*

Para obtener las imágenes de la cámara una a una podemos utilizar la función *snapshot.cgi*, que devuelve una captura en formato *jpg* al igual que hacíamos con la función *image.cgi* del API de AXIS. Sin embargo, no es posible especificar parámetros como la resolución, o la tasa de transferencia con este comando.

Por otro lado, la función *videostream.cgi* permite la descarga de un *stream* de video en *jpg*, donde sí que podremos establecer los parámetros anteriores. Esto hace, que para adaptar la aplicación *Control Camera IP* a este modo de recepción de video, tengamos que efectuar algunos cambios en la forma en que recibíamos las imágenes con la función *image.cgi* de AXIS. No obstante, la modificación es bastante sencilla.

La idea consiste en utilizar la clase *VideoCapture* de la librería de video de OpenCV. Esta función permite la recepción de un *stream* de video como entrada, incluyendo la entrada de video por defecto que tengamos en nuestro PC (como una webcam por ejemplo), o bien un *stream* ubicado en alguna dirección de internet a través del protocolo HTTP. Es por ello que utilizando una expresión como esta en nuestro código:

```
VideoCapture captura("http://<IP>/videostream.cgi"),
```

podemos conseguir un *stream* de imágenes *jpg* que se irán almacenando en la variable *capture* y que podremos ir leyendo en el momento que lo necesitemos. En la próxima

sección veremos la forma en la que podemos realizar esta adaptación y de cómo utilizar los parámetros *resolution* y *rate* para modificar la resolución de la imagen y tasa de transferencia de la misma respectivamente.

3.2.5 La función *decoder_control.cgi*

Como ya comentamos anteriormente, la función CGI *decoder_control* dispone de la capacidad de ejecutar cualquier función típica de la cámara (movimiento horizontal o vertical, zoom, apertura del sensor, enfoque, etc.) a través del parámetro *command*. El valor de este parámetro es un número entre 0 y 255, y cada uno de estos valores especifica una acción concreta a realizar por la cámara, o el cambio de algún ajuste de visualización en la misma. Entre los comandos más utilizados tenemos

Valor Command	Acción
0	Movimiento Tilt hacia arriba
2	Movimiento Tilt hacia arriba
4	Movimiento Pan a la izquierda
6	Movimiento Pan a la derecha
8	Apertura sensor (*Open Iris*)
10	Cierre del sensor (*Close Iris*)
16	Acercar Zoom
18	Alejar Zoom

Si nos fijamos, los comandos para los movimientos de la cámara en horizontal y vertical (*Pan* y *Tilt*) se corresponden con valores del teclado numérico en los que están ubicados los cursores de dirección ←→↑↓.

Al ejecutar la función *decoder_control.cgi* en algunas cámaras, los comandos de movimiento vertical y horizontal harán que ésta haga un barrido completo en la dirección seleccionada, en lugar de hacer un movimiento de un solo paso. Para evitarlo, podemos establecer el parámetro *onestep* al valor 1. Es decir, si queremos por ejemplo que la cámara se mueva un único paso a la derecha, tendremos que utilizar la siguiente expresión:

```
decoder_control.cgi?command=6&onestep=1
```

A diferencia del API de AXIS, esta función no dispone de comandos que permitan especificar una posición concreta a la que mover la cámara dentro del rango disponible de los controles *Tilt* y *Pan* (90º y 360º respectivamente en la cámara AD232). Sin embargo, sí que podemos especificar grados de movimiento en función de la posición

actual con el parámetro *degree*. Así, si estamos en una posición determinada y queremos mover la cámara 45 grados a la derecha, el comando a introducir sería:

```
decoder_control.cgi?command=6&degree=45
```

Es sin embargo destacable, que en este API no se ofrece la posibilidad de extraer la información de posición de la cámara en los valores angulares de los controles *Pan* y *Tilt*, lo cual nos introduce un problema de compatibilidad de cara a la adaptación de este API a la aplicación *Control Camera IP*, en la parte de la carga y guardado de ficheros de controles que fue implementada originalmente. No obstante, este API incluye la posibilidad de establecer hasta 32 *presets* existentes ya de forma predeterminada en la cámara, de forma que podemos utilizarlos para movernos rápidamente entre diferentes posiciones de la escena. Estos *presets* son establecidos también a través del parámetro *command* del CGI *decoder_control*, y vienen dados en tuplas de la forma [arriba-izquierda], [abajo-derecha-90º], etc, cubriendo prácticamente todo el rango de posiciones que permite la cámara.

El trabajo de adaptar la aplicación a esta nueva funcionalidad de *presets*, alternativamente al uso de ficheros de controles de la aplicación original, queda como propuesta de ampliación dentro del capítulo 5 de esta memoria.

3.3 Adaptación de la aplicación *Control Cámara IP* al nuevo API

En este Trabajo Fin de Grado hemos preferido evitar realizar cambios significativos en la estructura de la aplicación original, dado que no formaba parte de los objetivos del mismo. No obstante, hemos realizado algunas pruebas sobre una cámara IP de bajo coste Conceptronic modelo PTIW(*Pan & Tilt Wireless*), para las cuales hemos realizado modificaciones en el código, para mostrar la facilidad de adaptación de los elementos de la aplicación *Control Camera IP* a otras APIs de comunicación con cámaras IP, como puede ser la API IPCAM CGI estandarizada de Foscam.

A continuación, presentaremos las características de la cámara IP de bajo coste Conceptronic PTIW sobre la que vamos a realizar las pruebas, y acto seguido describiremos cuales han sido las modificaciones realizadas en la aplicación. Finalmente, mostraremos algunas capturas realizadas de la aplicación interactuando con la nueva cámara.

3.3.1 La cámara de bajo coste modelo *Conceptronic Pan & Tilt Wireless*

La cámara Conceptronic PTIW es una cámara IP que proporciona una alta calidad de imagen digital a un precio muy asequible (alrededor de 30 euros).

Figura 15. Cámara Conceptronic *Pan & Tilt Wireless*

Dispone de aplicaciones específicas, tales como *iSmartView*, adaptadas a las plataformas móviles con sistema operativo Android [17], o para *iPhone*. Se trata de una cámara con capacidad de conexión vía Wireless, y con diversas funciones de detección de movimiento y de conectividad con sensores externos.

Los motores de movimiento *Pan* y *Tilt* permiten un movimiento de hasta 270º en la horizontal, y de hasta 120º en la vertical. No dispone de zoom, aunque sí que dispone de funciones de apertura y cierre del sensor (*Iris*), así como de la posibilidad de modificar los parámetros de enfoque (*Focus*). Ofrece tres tipos de resolución: la máxima de 640x480 (VGA), resolución media de 320x240 (QVGA) y una resolución mínima de 160x120 (QQVGA), así como una tasa de imágenes (*frame rate*) de hasta 30fps.

Las imágenes pueden rotarse con las funciones *flip* y *mirror* y además pueden configurarse diferentes alarmas que alerten sobre la detección de intrusos u otras situaciones en el entorno.

El modo navegador

Una vez configurada la conexión a la cámara Conceptronic PTIW (vía Wifi o a través del puerto Ethernet), podremos acceder "al modo navegador" a través de una dirección IP y un puerto que introduciremos en la barra de direcciones de nuestro navegador

favorito. Antes de acceder, debido a que el acceso a este modo requiere privilegios de administrador, se nos pedirá el nombre de usuario y la contraseña.

Una vez dentro, se mostrará una interfaz de monitorización de la cámara con controles en el panel de la derecha, a través de los cuales podremos ejecutar la mayoría de funciones de las que dispone la cámara (*Tilt*, *Pan*, barridos verticales y horizontales, modos *flip* y *mirror*, resolución, *frame rate*, etc.).

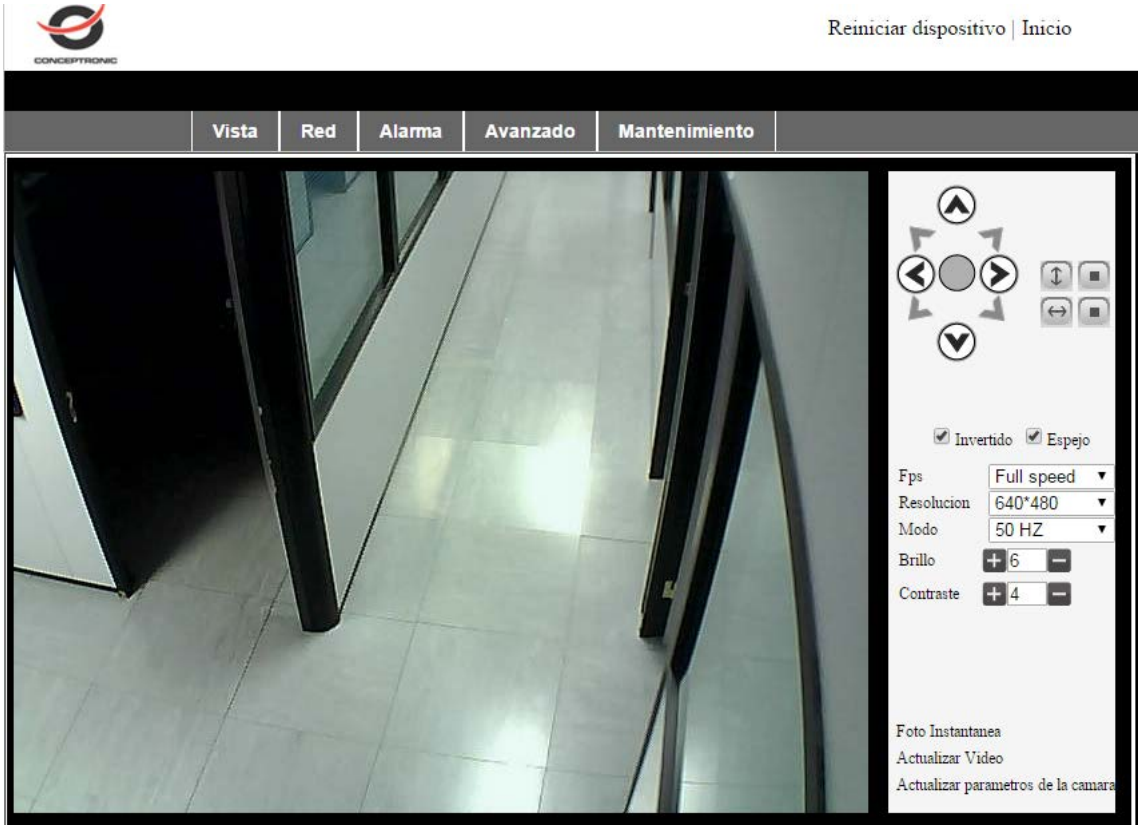

Figura 16. Modo Navegador de la cámara Conceptronic PTIW

A través de la barra de herramientas de la parte superior, se puede acceder a las opciones de configuración internas de la cámara, tales como la configuración de usuarios y permisos, la configuración de red o la configuración de alarmas.

Entre los grupos de usuarios se establecen tres categorías: *Administrador*, *Operator* y *Visitor*, las cuales son similares a la de los grupos de usuarios de las cámaras AXIS. En el caso de *Visitor*, se tiene solo la posibilidad de acceder a las funciones de monitorización de imágenes, pero no a las opciones de modificación ni de visualización de ciertos parámetros. En el grupo *Operator*, se da la posibilidad de modificar algunos parámetros de las funciones de la cámara, pero no aquellos relativos a la configuración de usuarios o de red. Finalmente, el usuario del grupo *Administrador* dispondrá de todos

los privilegios por lo que podrá realizar cualquier modificación en los parámetros y funciones de configuración de la cámara.

3.3.2 Modificaciones en el código de la aplicación

Para empezar, la primera modificación a llevar a cabo, indispensable para el funcionamiento de la aplicación, es la relativa a la recepción de imágenes desde la cámara.

3.3.2.1 Modificaciones para la recepción de imágenes

Para ello, y tal como ya hemos comentado, utilizaremos la función *videostream.cgi* del API IPCAM CGI en conjunción con la clase *VideoCapture* de la librería de video de OpenCV.

La clase *VideoCapture* permite capturar secuencias de video a través de diferentes dispositivos, en función del parámetro que pasemos en su constructor. El más típico suele ser un valor entero (0,1,…) que representa el identificador de dispositivo de captura de video de alguna cámara que esté conectada a nuestro PC (una webcam por ejemplo. También puede recibir como parámetro un fichero de video o de una secuencia de imágenes, a través del acceso a una ruta local o a una localización remota.

Este último caso es interesante debido a que podemos utilizarlo para abrir un flujo de captura de imágenes utilizando como fuente la secuencia de imágenes en *stream* que devuelve la función *videostream.cgi*.

Así pues, primera modificación a realizar en nuestro código es la inclusión de la línea que ya mostramos anteriormente:

```
VideoCapture captura("http://<IP>/videostream.cgi")
```

A la función *videostream.cgi* le añadiremos también los parámetros *admin* y *pwd* para identificarnos como usuario, y además podremos incluir los parámetros *resolution* y *rate* para establecer la resolución y la tasa de imágenes por segundo respectivamente.

El *frame rate* no lo hemos modificado, ya que por defecto está a *full speed* (valor 0), y ésta es una buena configuración para apreciar mejor los cambios en la imagen, a la hora de aplicar algunos de los métodos de detección y procesamiento sobre la imagen que se han desarrollado en este trabajo. La resolución en cambio, la hemos ajustado a los tres tipos de resolución que ya existían para su modificación en la aplicación, relativos a las cámaras AXIS: 4CIF, CIF y QCIF.

Aunque los valores de las resoluciones son diferentes, solo tenemos que modificar estos valores por los requeridos por la función *videostream.cgi* (por ejemplo, el valor "32" es el equivalente a la resolución 640x480) en el fichero *valores.txt* asociado a cada una de las plataformas, y utilizarlo en la línea de código anterior.

Utilizando el operador >> de C++, podemos volcar en cualquier momento, un *frame* capturado del *stream* de video de la siguiente manera:

```
captura >> frame
```

Al tratarse de una clase de OpenCV, la variable *frame* será una estructura *Mat* que es inicializada con el contenido de la imagen volcada desde el *stream* de video. Así, podemos disponer directamente de la imagen *Mat* para procesarla utilizando los diferentes métodos desarrollados en este trabajo. No obstante, la imagen *Mat* habrá que convertirla a un componente QPixmap para poder mostrarla en el monitor de la aplicación.

3.3.2.2 Modificaciones de los controles de la cámara

Con objeto de no extender demasiado el alcance de este Trabajo Fin de Grado, se han modificado únicamente a modo demostrativo los controles básicos de movimiento horizontal (*Pan*) y vertical (*Tilt*) en un paso, que son comandados en la interfaz de la aplicación *Control Camera IP* a través de las flechas de dirección en la parte derecha e inferior del monitor de recepción de imágenes.

Así, deshabilitaremos para las pruebas la posibilidad de seleccionar en las barras de posicionamiento intermedias un valor concreto de *Tilt* y *Pan*, y además deshabilitaremos la función de *Zoom* ya que la cámara Conceptronic PTIW no dispone del mismo. Los controles de enfoque (*focus*) y de exposición del sensor (*iris*) tampoco están disponibles en esta cámara por lo que también serán deshabilitados.

Por lo tanto, haremos únicamente que la cámara se mueva horizontalmente y verticalmente paso a paso. Como hemos comentado, como modo de demostrar la posibilidad de adaptar la aplicación a la API presentada en este capítulo, creemos que es suficiente con disponer únicamente de estos controles básicos.

Se podrían añadir no obstante algunos de los modos de ronda de vigilancia que vienen incluidos en la cámara Conceptronic PTIW, o bien barridos horizontales y verticales sin establecer el parámetro *onestep*. Sin embargo, nos decantaremos por proponer estas

posibles mejoras para una futura ampliación de este trabajo, en la que se pueda realizar una adaptación integral y mucho más completa de la API en nuestra aplicación.

En el siguiente apartado mostraremos las pruebas realizadas a lo largo de las pequeñas modificaciones que hemos realizado para hacer funcionar la aplicación *Control Camera IP* con la cámara Conceptronic PTIW, utilizando para ello la API IPCAM CGI estandarizada de Foscam.

3.3.3 Capturas de las pruebas realizadas

Tras adaptar la aplicación *Control Camera IP* a las funciones de la API IPCAM CGI, incluyendo la recepción de imágenes a través del comando *videostream.cgi*, y el movimiento básico de la cámara con los controles de *Tilt* y *Pan* utilizando el parámetro *onestep* en la función CGI *decoder_control*, obtenemos el resultado que se puede observar en la siguiente captura:

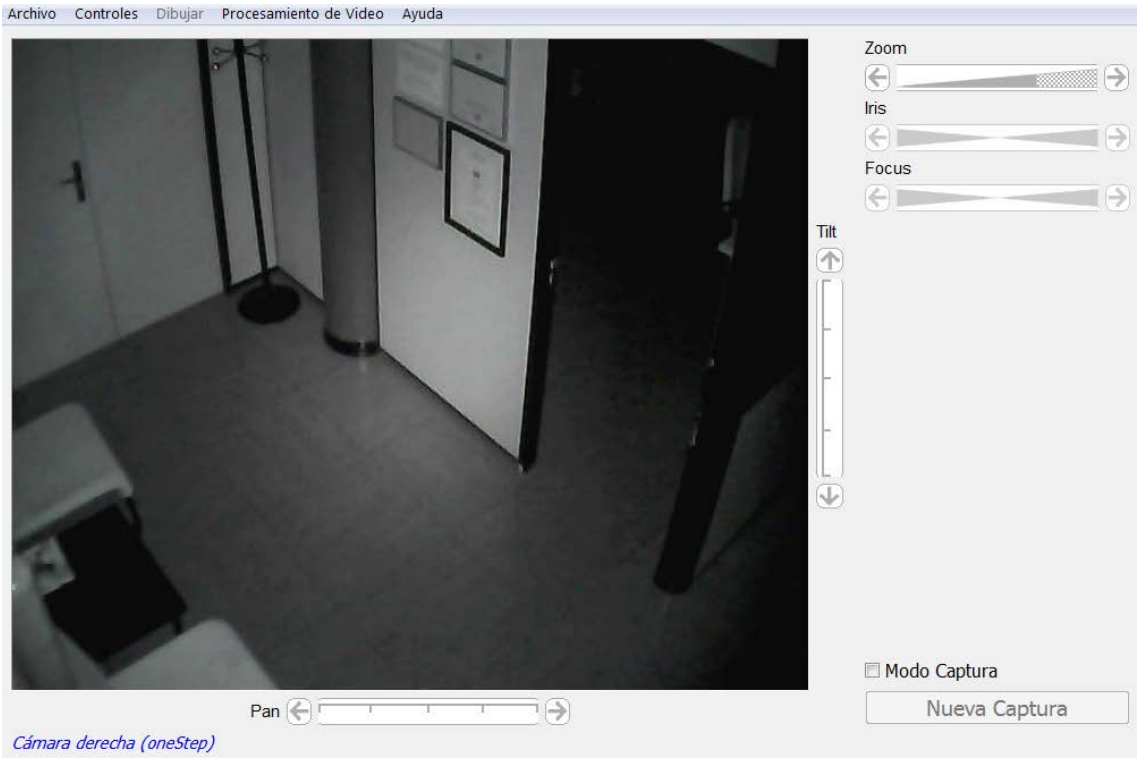

Figura 17. Captura de cámara Conceptronic tras aplicar el comando 6 (Pan) *onestep* a la derecha

Además, incluimos también algunas capturas aplicando los nuevos métodos de OpenCV (los cuales describiremos en el capítulo 4), sobre la imagen capturada por la cámara Conceptronic PTIW.

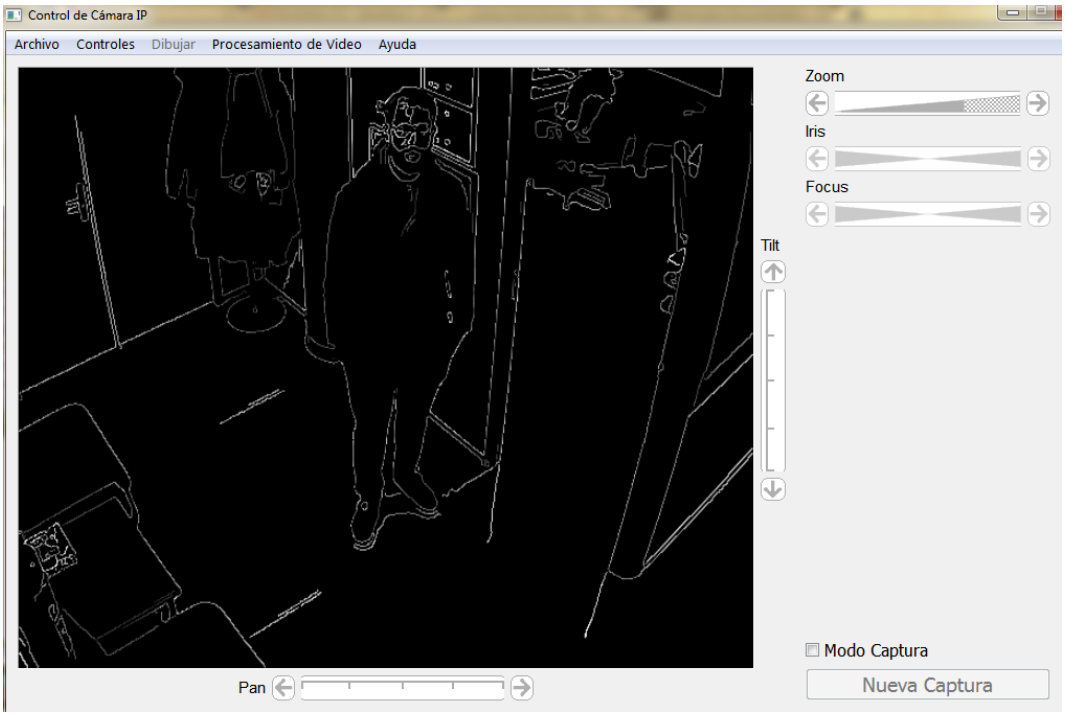

Figura 18. Captura de cámara Conceptronic aplicando el algoritmo de Canny

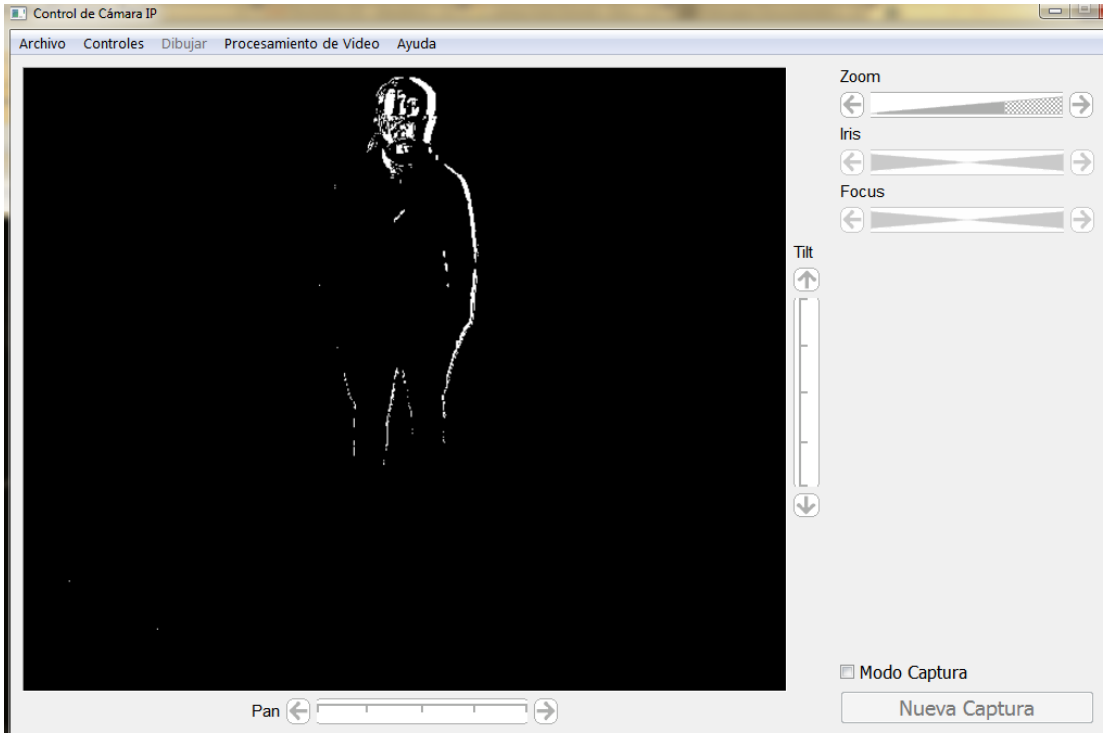

Figura 19. Captura de cámara Conceptronic aplicando el algoritmo de sustracción de fondo

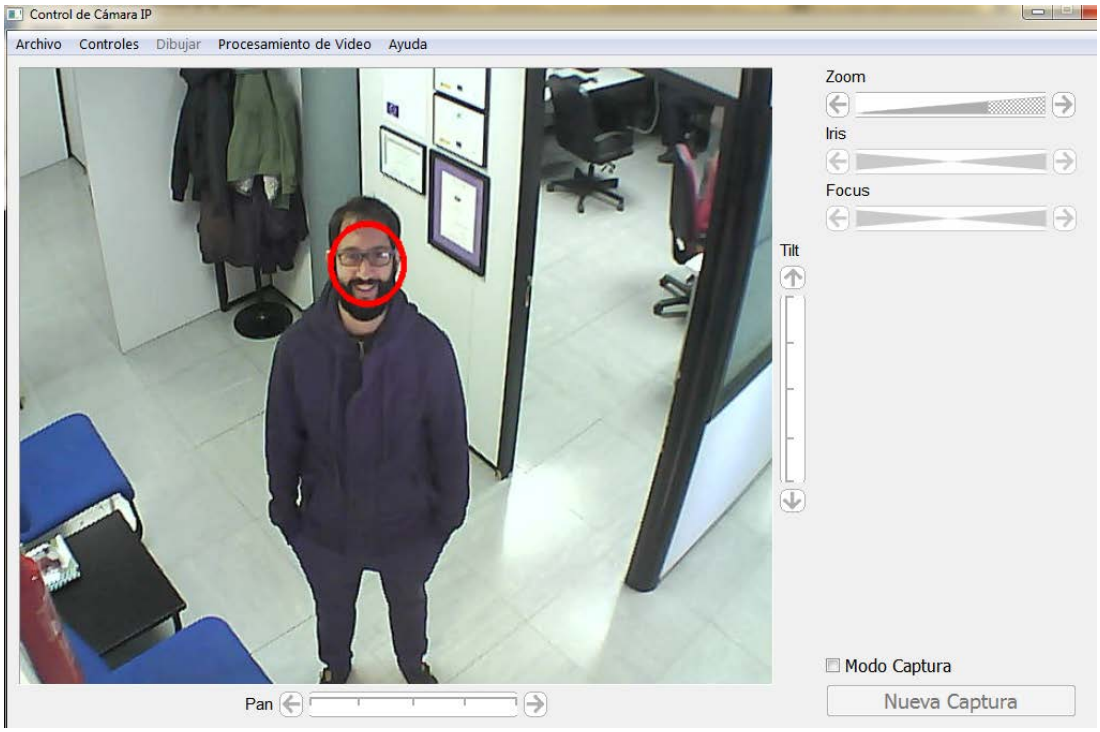

Figura 20. Captura de cámara Conceptronic aplicando el método de detección de caras

Capítulo 4

Integración de Procesamiento de Vídeo con la Biblioteca OpenCV

En este capítulo trataremos de abordar el desarrollo realizado sobre la aplicación *Control Cámara IP* que incluye varios métodos y técnicas de tratamiento sobre las imágenes recibidas desde la cámara, utilizando para ello la biblioteca de visión artificial OpenCV.

En un primer bloque, realizaremos una descripción del funcionamiento de los módulos y funciones de OpenCV utilizadas en los desarrollos de este trabajo. Acto seguido, veremos el aspecto de la integración de la biblioteca OpenCV en Qt, así como la inclusión de librerías particulares para cada una de las plataformas. Por último, nos dedicaremos a explicar los detalles de diseño e implementación de los desarrollos realizados, incluyendo una descripción detallada de los métodos y algoritmos específicos utilizados para la obtención de las distintas soluciones de procesamiento de video incluidas en la aplicación.

4.1 La biblioteca de visión artificial OpenCV

Tal y como ya hemos comentado en el capítulo de introducción, la biblioteca de procesamiento de video y visión artificial OpenCV está constituida por un conjunto de librerías de código abierto y de libre distribución, que ofrece amplias capacidades para todo tipo de aplicaciones relacionadas con la visión y el tratamiento de imágenes, con más de 500 funciones disponibles en cada uno de sus módulos.

Para los desarrollos realizados en este Trabajo Fin de Grado, los cuales explicaremos más adelante, se han utilizado varias de estas clases y funciones pertenecientes a varios de los módulos de OpenCV. A continuación, realizaremos un breve análisis de cada una de estas funciones, y de su utilidad de cara a su uso en nuestra aplicación.

4.1.1 Funciones del módulo *Core*

En el módulo *Core* están definidas todas aquellas estructuras de datos y funciones básicas que establecen la forma de trabajar con los datos en general, y con las imágenes en particular, en OpenCV.

Una primera cosa a entender cuando se trabaja con OpenCV es que los tipos de datos básicos típicos que se manejan en el lenguaje C++ (*unsigned char*, *bool*, *signed short*, *int*, *float*, *double*, etc.) pueden definirse a su vez como un identificador de la forma `CV_<profundidad-bits>{U|S|F}C(<número-de-canales>)`.

Esta manera particular de OpenCV de definir un tipo primitivo permite identificar tuplas de tipos básicos en una matriz de forma que, por ejemplo, una tupla de tres elementos de tipo punto flotante (*float*) puede definirse como CV_32FC3, donde el número 32 indica el tamaño o profundidad en bits del tipo (32 bits), el carácter 'F' representa el tipo básico (*Float*) y el número 3 el número de elementos, o como se suele llamar en el lenguaje de OpenCV, el número de canales.

El número de canales en OpenCV representa normalmente el número de componentes de un elemento dentro de un vector (*std::vector*) o en una matriz (*cv::Mat*). Este número de componentes suele definir los distintos canales de color de una imagen. Es decir, en el ejemplo anterior, la tupla con tres elementos podría definir un píxel de una imagen RGB con las tres componentes de color rojo, verde y azul (*Red*, *Green*, *Blue*).

En la siguiente imagen podemos observar un zoom realizado sobre una captura de una imagen mostrada con la función *imshow()* de OpenCV, donde se pueden apreciar las tuplas de tres elementos que definen cada uno de los píxeles con cada uno de los canales RGB que lo conforman.

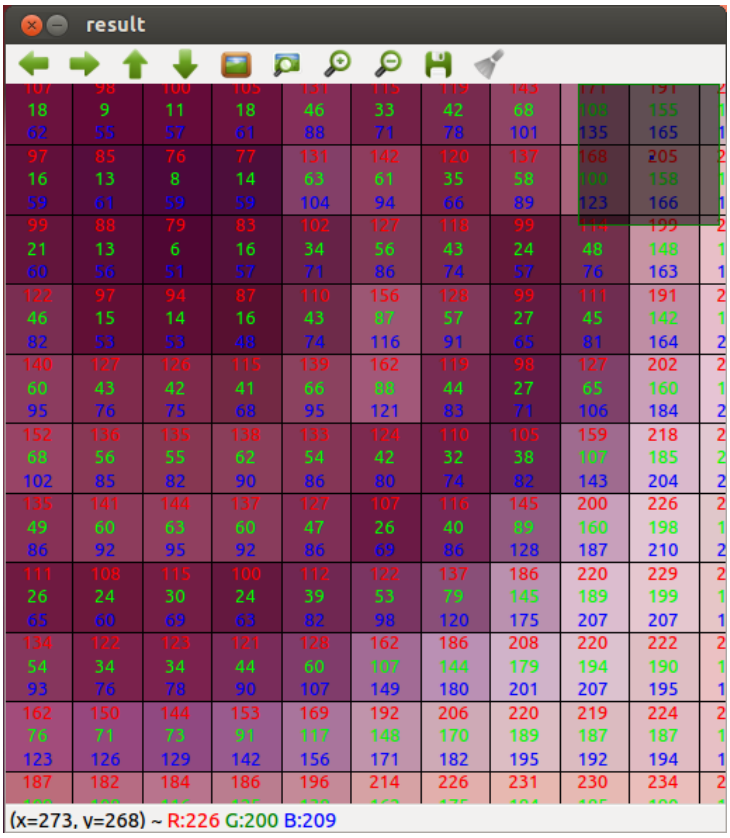

Figura 21. Visualización de las componentes RGB haciendo zoom sobre una imagen

La estructura de almacenamiento y manejo de imágenes más utilizada hoy en día en OpenCV es la clase *Mat*. Esta estructura definida en el lenguaje C++, fue incorporada para suplir las funcionalidades de la estructura *IplImage*, concebida originalmente cuando la biblioteca OpenCV fue creada en torno al lenguaje de programación C. Aunque la estructura *IplImage* sigue siendo muy utilizada en muchos tutoriales y como herramienta para el aprendizaje, actualmente cada vez es menos recurrida debido, por un lado, a que responsabiliza al programador de la gestión de memoria dinámica, cuestión heredada de la definición de estructuras de tipo puntero en C, y por otro, de las inherentes dificultades de depuración añadidas por el mismo motivo para los errores en el desarrollo de métodos o algoritmos complejos.

Es por ello que, desde el surgimiento de C++ orientado a objetos, la clase *Mat* se configura como la estructura más apropiada para el manejo de imágenes en OpenCV. Sin embargo, el problema de estar definida en C++ es que todavía hoy existen muchos sistemas de desarrollo empotrado que solo soportan el lenguaje C. Así pues, es muy común que nos encontremos con soluciones que puedan incorporar cualquiera de las dos estructuras comentadas e incluso, en muchas ocasiones, una combinación de las mismas. En nuestro código haremos uso de las dos estructuras para algunos de los

métodos desarrollados. En el caso de *IplImage*, utilizaremos esta estructura en el ámbito privado de algunos de los algoritmos matemáticos implementados que realizan las operaciones sobre las imágenes a más bajo nivel. De esta forma, en caso de necesitar en un futuro migrar estos métodos a otros entornos de desarrollo que solo soporten el lenguaje C, no tendremos problema para reutilizarlos.

A continuación veremos cómo se utilizan estas estructuras de datos para trabajar con imágenes en OpenCV.

4.1.1.1 Trabajando con imágenes *IplImage*

El nombre *IplImage* viene de *Intel Image Processing Library Image*, y define una estructura (*struct*) en C con un conjunto de campos para el almacenamiento de una imagen. No obstante, el número de campos que pueden ser utilizados al crear esta estructura no es fijo, por lo que puede ser utilizada para definir la cabecera de una imagen únicamente, donde solo se utilizan los campos referentes a los metadatos (profundidad, número de canales, etc.), o bien puede ser utilizada también para almacenar el contenido de la imagen, para lo cual se utiliza el campo *imageData*, así como la estructura *IplROI*, la cual permite procesar únicamente una región de interés (*Region Of Interest*) concreta de la imagen [18].

Los campos más destacados de una estructura *IplImage* son los siguientes:

- Campo *nChannels*: número de canales de la imagen (de 1 a 4 canales).
- Campo *depth:* Profundidad de cada canal en bits más un bit de signo opcional (*IPL_DEPTH_SIGN*). A diferencia de la estructura *Mat* los formatos soportados comienzan con el prefijo *IPL_DEPTH* en lugar de *CV*. Por ejemplo, el identificador de tipo punto flotante de simple precisión sería IPL_DEPTH_32F, que es equivalente a CV_32F.
- Campos *width y height*: definen la altura (número de filas) y anchura (número de columnas) de la imagen en pixeles.
- Campo *imageSize*: es el tamaño de la imagen en bytes, o lo que es lo mismo, el resultado de multiplicar la altura (o el número de filas) por el paso en anchura (*widthStep*).
- Campo *widthStep*: el paso en anchura es el número de bytes de todos los píxeles de una fila de la imagen, que vendrá dado por el tamaño del tipo básico

almacenado multiplicado por la anchura de la imagen (número de columnas) y por el número de canales.

- Campo *imageData*: se define como un puntero a la estructura de datos que almacena la imagen.

Para crear una imagen con la estructura *IplImage* se utiliza la función *cvCreateImage()*, que acepta como parámetros de entrada los elementos básicos que conforman la cabecera de la imagen, es decir, el tamaño (*cvSize*), que define la altura y la anchura de la imagen, la profundidad en alguno de los formatos válidos vistos anteriormente, y el número de canales. Esta función devuelve como resultado un puntero a una estructura *IplImage* con una cabecera establecida y asigna memoria dinámica al campo *imageData* de forma que pueda ser utilizado para almacenar el contenido de la imagen.

Por otro lado, si ya disponemos de una imagen almacenada como estructura *IplImage*, la función *cvCloneImage()* nos permite realizar una copia completa de la imagen, incluyendo la cabecera y el contenido de la misma. Por último, para liberar la memoria reservada para almacenar este tipo de estructura es necesario recurrir a la función *cvReleaseImage()*.

4.1.1.2 Trabajando con la clase *Mat*

Desde la inclusión de librerías de clases de C++ en la biblioteca de OpenCV, se eliminó la necesidad de reservar y liberar memoria de forma manual para las diferentes estructuras de datos. Así, las nuevas funciones permitirían reservar los datos de salida de forma automática, así como hacer uso compartido y más eficiente de los recursos del sistema.

La estructura *Mat* para almacenamiento y gestión de imágenes en OpenCV se crea en base a una matriz con dos partes bien definidas: la cabecera, que contiene entre otros datos el tamaño de la matriz, el método de almacenamiento y la dirección de memoria donde se almacena; y un puntero a la matriz conteniendo los valores de los píxeles tomando las dependencias de dimensionamiento en función del método de almacenamiento elegido. Si bien la cabecera de la matriz es fija, el tamaño de los datos contenidos en la matriz puede variar de imagen a imagen en varios órdenes de magnitud.

La idea es que cada objeto *Mat* tiene su cabecera, pero sin embargo, el contenido de la matriz puede ser compartido por dos instancias de la misma, de forma que sus punteros

de matriz apunten a la misma dirección de memoria. De esta manera, al copiar una imagen de tipo *Mat* con el constructor de copia (sin usar las funciones *clone* o *copyTo*), solo se copiarán las cabeceras y el puntero a la matriz, pero no los datos en sí. Este mecanismo es muy útil por ejemplo cuando se quiere trabajar sobre regiones de interés (ROI) de una misma imagen, de forma que se pueden definir dos objetos *Mat* con cabeceras que definan un tamaño de matriz diferente, para realizar operaciones sobre la imagen en diferentes zonas de la misma.

El método de almacenamiento de una imagen tipo *Mat* hace referencia al espacio de color (combinación de componentes de color) y al tipo de dato usado en el almacenamiento de la imagen. Como ya hemos visto, el método de almacenamiento más utilizado suele ser RGB (*Red Green Blue*), ya que es la forma en la que nuestros ojos construyen los colores (hay que recordar que el sistema de representación de imágenes en OpenCV utiliza el método BGR, que es igual al RGB pero con las posiciones invertidas).

Sin embargo, existen otros métodos de almacenamiento tales como HSV o HLS, que descomponen los colores en su matiz (*Hue*), su componente de valor (*Value*) o luminancia (*Luminance*) y en su saturación (*Saturation*), y que son utilizados como una manera más natural de describir los colores, permitiendo por ejemplo que los algoritmos sean menos sensibles a las condiciones de luz de la imagen de entrada (siempre y cuando se descarte el componente de valor).

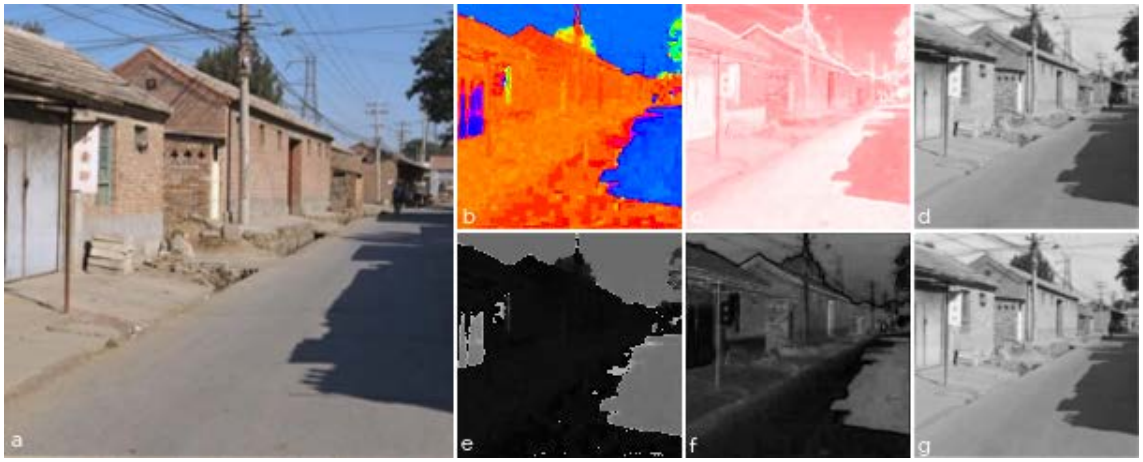

Figura 22. Imagen HSV y sus distintos componentes de color
Imagen HSV original(a) con sus canales de color: matiz(b), saturación(c), y valor(d). En la segunda fila podemos apreciar cada canal en escala de grises

La forma más simple de almacenar una imagen es en escala de grises, donde los colores son una combinación del blanco y el negro, lo cual nos permite crear muchos tonos de

gris. Este método de almacenamiento permite que al ser matrices de menor tamaño el procesamiento sea mucho más rápido en la mayoría de algoritmos.

Cada uno de los componentes de color tiene sus propios valores de dominio válidos, lo cual nos lleva al tipo de dato usado en cada uno de ellos. El tipo de dato más pequeño que se puede utilizar es el tipo carácter (*char*), que tiene un tamaño de 8 bits y que puede ser con signo o sin signo. Aunque el número de representaciones de un píxel con 8 bits pueda parecer limitado (256 posibles valores), si el número de componentes es por ejemplo de 3 (como por ejemplo con BGR), el número de posibles colores que se pueden representar son del orden de 16 millones.

Tal y como adelantábamos al comienzo de esta sección, los métodos de almacenamiento de OpenCV para los objetos *Mat* se representan con la notación `CV_<profundidad-bits>{U|S|F}C(<número-de-canales>)`.

En este trabajo haremos uso del constructor de copia de la clase *Mat* para crear las imágenes, de forma que hagamos que su puntero de matriz apunte a la posición de memoria donde tenemos almacenada la imagen en nuestra aplicación de Qt, es decir, en un objeto QPixmap. Más adelante veremos cómo realizar la conversión entre estos dos tipos de datos.

Otras funciones como *height()*, *width()*, *depth()*, *step()*, etc., que también hemos utilizado en los desarrollos realizados para este trabajo, hacen referencia a los diferentes campos de la cabecera de una imagen tipo *Mat* (tamaño, profundidad, paso de anchura,…), y tienen el mismo significado que el de la estructura *IplImage*.

4.1.2 Funciones del módulo *ImgProc*

Al hilo de la sección anterior, cabe destacar la utilidad de la función *cvtColor()* del módulo *ImgProc*, la cual nos permitirá fácilmente cambiar el método de almacenamiento de una imagen. Esta función recibe como argumento la imagen original, la imagen de salida, y un método de almacenamiento en el formato definido para las imágenes de tipo *Mat*.

Así, si por ejemplo tenemos una imagen que tiene un espacio de color BGR de 8 bits con 4 canales, y la queremos convertir a un formato de escala de gris, podríamos hacer algo como lo siguiente:

```
cvtColor(image, image, CV_BGRA2GRAY);
```

Este tipo de conversiones será un recurso muy utilizado en los métodos desarrollados en este trabajo tal y como podremos ver en los próximos apartados.

Por otro lado, entre las funciones de filtrado y de procesado de la imagen de las que dispone el módulo *ImgProc* hemos utilizado dos de ellas para la implementación del método de detección de bordes incluido en los desarrollos realizados para la aplicación *Control Camera IP*. Éstas son la función *blur()*, y la función *Canny()*.

4.1.2.1 Función *blur()*

La función *blur()* difumina o suaviza (*smooth*) la imagen permitiendo reducir el ruido incorporado, de forma que a la salida de la función ciertas partes de la imagen puedan apreciarse con una mayor claridad descartando los detalles. Para ello, se aplican filtros sobre los píxeles de la imagen (normalmente filtros lineales) donde el valor del píxel a la salida está determinado por una suma ponderada de los valores de los píxeles a la entrada. Esta suma ponderada viene dada por los coeficientes del filtro, también llamados semilla o *kernel*. La función *blur()* utiliza un filtro de caja normalizado (*Normalized Box Filter*), donde el píxel de salida se calcula en base a la media de los vecinos en función del tamaño de la semilla, donde todos contribuyen con el mismo peso. Así, dado un tamaño de semilla 3 por ejemplo, los coeficientes del *kernel* que se aplicaría a la imagen serían:

$$K = \frac{1}{3 \cdot 3} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

De esta forma, en función del tamaño de la semilla que se aplica sobre cada uno de los píxeles de la imagen, podremos aumentar el umbral del filtro de paso bajo aplicado sobre los mismos, produciendo un difuminado de la imagen más notable, tal y como se aprecia en la siguiente figura:

Figura 23. Función *blur* sobre una imagen usando diferentes tamaños de *kernel*

De izquierda a derecha: imagen original e imágenes filtradas con tamaño de *kernel* 3, 5 y 7.

4.1.2.2 Función *Canny()*

La función *Canny()* del módulo *imgProc* de OpenCV, permite buscar bordes en una imagen haciendo uso del algoritmo de Canny.

El algoritmo de Canny, desarrollado por John F. Canny en 1986, es un método de detección de bordes sobre imágenes óptimo y que está dirigido a satisfacer tres criterios básicos:

- Una baja tasa de error, con una detección efectiva de los bordes existentes.
- Buena localización, minimizando la distancia entre los píxeles reales y los píxeles de los bordes detectados.
- Minimización del detector a una respuesta por borde detectado.

La función *Canny()* del módulo *ImgProc* encuentra los bordes en la imagen de entrada y los marca en un mapa de salida utilizando para ello el algoritmo de Canny. La función utiliza dos valores de umbral de forma que el valor más pequeño entre ellos se utiliza para enlazar los bordes, y el más grande para encontrar los segmentos iniciales de los bordes más destacados.

Más adelante explicaremos el método de detección de bordes que se ha implementado para este trabajo, y que utiliza las funciones *blur()* y *Canny()* que acabamos de comentar.

4.1.2.3 Ecualización de histogramas

Finalmente, el módulo *imgProc* dispone de funciones útiles para modificar las imágenes previamente a ser procesadas o utilizadas por otros métodos. Es el caso de la función de ecualización de histogramas (*equalizeHist*), que utilizaremos para preparar la imagen en escala de grises que recibe como entrada el método de detección de caras implementado en este trabajo.

Un histograma de color en una imagen es una representación de la distribución de los colores de los diferentes píxeles dentro del dominio o espacio de colores en el que está construida dicha imagen. Básicamente para cada color del espectro, se asocia un número de píxeles de la imagen donde ese color está presente.

La ecualización de un histograma solo tiene sentido para el caso de una imagen en escala de grises (donde a cada tonalidad de gris se asocia un número de píxeles), ya que

se trata de que el histograma resultante normalice los píxeles de un mismo nivel de gris y los distribuya en todo el rango posible separando las ocupaciones de cada nivel. Es decir, la ecualización permitirá maximizar el contraste sin perder información y además normaliza el brillo de la imagen, distribuyendo los tonos de gris en todo el histograma de forma que la suma de los píxeles representados (barras del histograma) sea igual al número de tonalidades de gris (255).

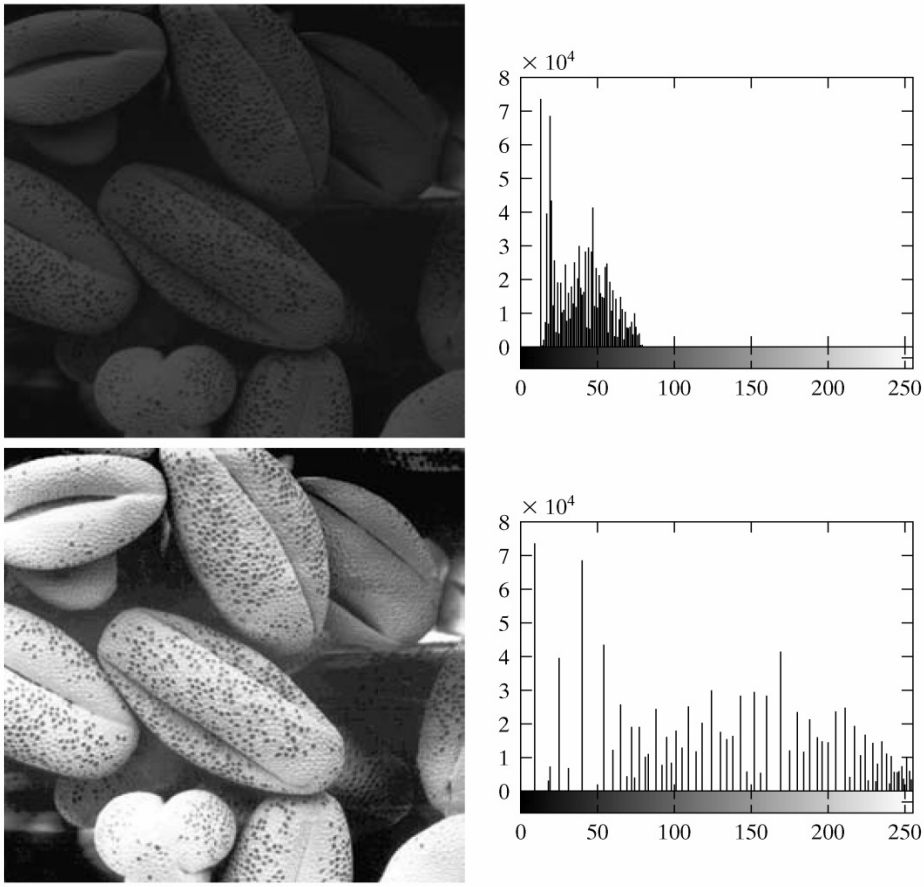

Figura 24. Ejemplo de imagen tras aplicar la ecualización de histograma.
Abajo podemos apreciar la imagen ecualizada y el histograma normalizado

La ecualización de histogramas es una herramienta muy útil para dotar a las imágenes de un mayor contraste, y por lo tanto es muy frecuente su uso en aplicaciones de imagen en medicina y otras áreas. En la detección de objetos en la imagen su uso es también muy interesante por razones obvias, ya que una mejor definición y contraste en la imagen permitirá que estos métodos tengan un mayor nivel de acierto al comparar las imágenes de entrada con las muestras o patrones deseados.

4.1.3 Funciones del módulo *ObjDetect*

Aunque en el primer capítulo de este trabajo introdujimos los módulos *HighGui* y *Features2D* de OpenCV, junto con algunas de sus funciones más destacadas, en este capítulo obviaremos entrar en más detalles acerca de estos módulos ya que no será necesaria su utilización para los desarrollos realizados dentro del ámbito de este Trabajo Fin de Grado. No obstante, en el capítulo dedicado a los posibles trabajos futuros que pudieran llevarse a cabo para ampliar el presente trabajo, se propondrá la utilización del módulo *Features2D* para la inclusión de funciones relativas a la detección de características sobre la imagen, utilizando algunos de los métodos que ya describimos brevemente en el capítulo de introducción.

Nos dedicamos a analizar ahora el clasificador en cascada basado en características de tipo *Haar* para detección de objetos de OpenCV (*Haar Feature-based Cascade Classifier for Object Detection*), y que ha sido utilizado para implementar el método de detección de caras que se ha añadido como nueva funcionalidad a la aplicación *Control Camera IP*.

4.1.3.1 Clasificador en cascada basado en características tipo *Haar*

En el ámbito de la detección de objetos en imágenes, un clasificador es una estructura de información obtenida en base a un entrenamiento a partir de imágenes de muestra de un objeto particular (una cara, un coche, etc.), de forma que, una vez finalizado el entrenamiento, es capaz de aplicarse sobre una región de interés concreta de una imagen de entrada.

El clasificador resultante del entrenamiento está formado por un conjunto de clasificadores más simples (de ahí el nombre de clasificador en cascada) a los que se les denomina etapas (*stages*), y que son aplicados iterativamente sobre la región de interés hasta que, o bien se encuentra una coincidencia con alguna característica, o bien todas las etapas son superadas.

El clasificador define sobre cada región de interés lo que se conoce como una ventana de detección (*detection window*), de forma que en ella se calculan las diferentes características asociadas al clasificador. Usualmente, las características más empleadas para la detección y reconocimiento de objetos en imágenes digitales son las de tipo *Haar*.

Las características tipo *Haar* deben su nombre a su similitud con la idea del *wavelet* de *Haar* [19]. Para el cálculo de estas características se consideran regiones rectangulares adyacentes dentro de la ventana de detección, y se calcula la diferencia entre las sumas de las intensidades de los píxeles de cada una de ellas. Estas diferencias permiten categorizar las subsecciones de una imagen, y de esta forma se puede comprobar de forma fácil y rápida si una determinada región de interés encaja con este tipo de características.

En el caso de una cara por ejemplo, es fácil observar que en todas ellas la región de los ojos es más oscura que la región de las mejillas. Así pues, se puede definir una característica tipo *Haar* que considere dos regiones rectangulares adyacentes que encajen, una sobre los ojos, y otra sobre la región de las mejillas.

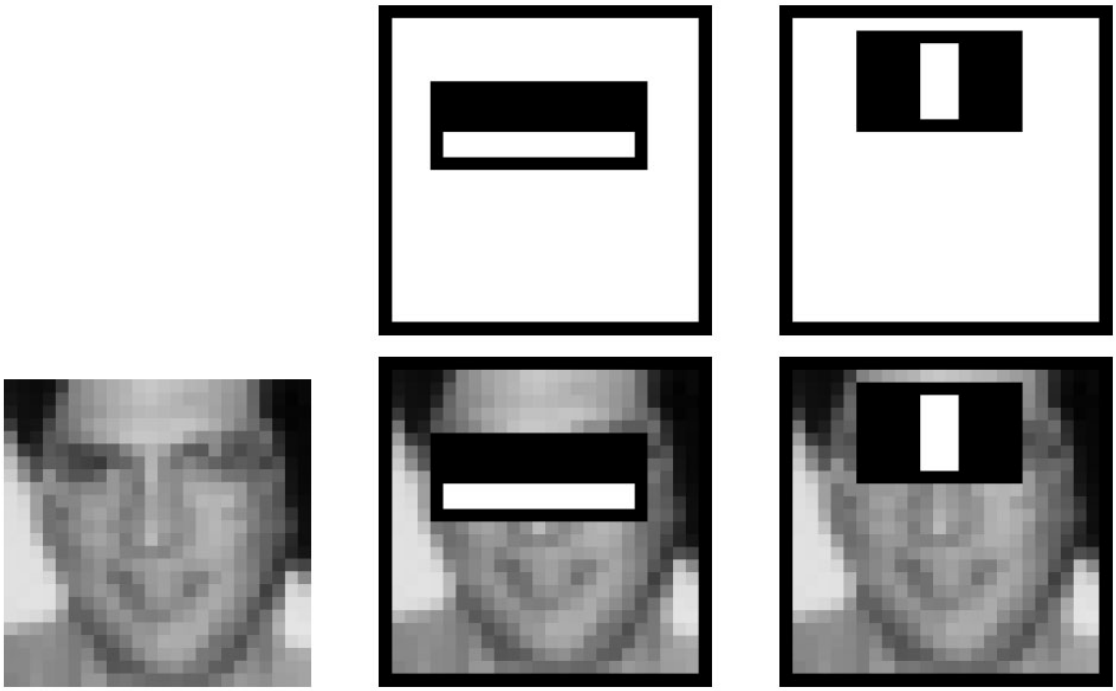

Figura 25. Ejemplo de cálculo de características tipo *Haar* para la detección de caras
La del centro clasifica la relación ojos-mejillas. La de la derecha clasifica la relación ojos-entrecejo.

El clasificador *Haar* está diseñado para que pueda ser redimensionado fácilmente, de forma que pueda ser capaz de encontrar objetos de interés de diferentes tamaños en la imagen, lo cual es mucho más eficiente que tener que redimensionar la imagen de entrada.

4.1.3.2 Cargando un clasificador con la función *load()*

La función *load()* del módulo *ObjDetect* nos permite cargar un nuevo clasificador en cascada a partir de un archivo que con el clasificador entrenado por algunas de las aplicaciones de entrenamiento [20].

Para el método de detección implementado en este trabajo utilizaremos el clasificador para detección de caras en perspectiva frontal incluido en las propias librerías de OpenCV [21], y que está definido en el archivo XML *lbpcascade_frontalface.xml*.

Una vez cargado, el clasificador puede ser utilizado por cualquier método de detección. Para este trabajo, hemos utilizado la función *detectMultiScale* del módulo *ObjDetect* de OpenCV.

4.1.3.3 La función *detectMultiScale*

Esta función permite detectar objetos de diferentes tamaños aplicando un clasificador previamente definido. La función recibe como entrada la imagen en una estructura *Mat* y devuelve por referencia un vector de polígonos rectangulares (tipo *Rect*) que definen las regiones donde el objeto ha sido detectado.

Entre otros parámetros de entrada que se pueden introducir podemos nombrar el factor de escalado, el número mínimo de vecinos o el tamaño mínimo o máximo del objeto a detectar.

El parámetro “número mínimo de vecinos” está relacionado con la forma de funcionamiento del algoritmo de detección cuando va deslizando la ventana del clasificador a lo largo de las regiones de la imagen. Como ya comentamos anteriormente, la ventana del clasificador es redimensionada para ajustarse a los distintos tamaños que pueda tener el objeto a detectar. Si el rango de tamaños es lo suficientemente amplio los falsos positivos serán muy frecuentes, sobre todo si se establece un tamaño mínimo muy pequeño. Esto hace además, que para un mismo objeto a detectar (una cara por ejemplo), se obtengan múltiples coincidencias para tamaños similares, por lo que el algoritmo obtendría a la salida varios rectángulos para un mismo objeto.

Para evitar los falsos positivos, se establece un número mínimo de vecinos (número de rectángulos cercanos) de forma que si el objeto tiene un número igual o superior a ese valor de rectángulos cercanos, ese será el objeto a detectar.

Más adelante, realizaremos algunas pruebas utilizando el método de detección de caras que se ha incluido en este trabajo, donde modificaremos algunos parámetros de la función *detectMultiScale()* para mostrar el funcionamiento del algoritmo.

4.2 Integración de las librerías de OpenCV en Qt5

El primer paso para añadir a la aplicación *Control Camera IP* las funcionalidades de detección y procesamiento sobre la imagen comentadas en apartados anteriores, es necesario incluir las librerías de OpenCV específicas para cada una de las diferentes plataformas sobre las que tenemos definidas una configuración de ámbito en el fichero *.pro* del proyecto Qt.

Esto es debido a que, evidentemente, existen diferentes versiones de kit de desarrollo software (*SDK*) de la biblioteca OpenCV optimizadas para su utilización en las diferentes plataformas (*MS Windows, Linux Debian, Android,* etc.), lo que hace que al momento de enlazar estas librerías dentro del proyecto Qt tengamos que establecer las diferentes rutas a los *SDKs* en los diferentes ámbitos del fichero *.pro* en función de la plataforma.

Así pues, nos disponemos a describir brevemente la instalación de OpenCV y la inclusión de sus librerías en el proyecto Qt, para las distintas plataformas que se han utilizado para probar los desarrollos realizados en este trabajo.

4.2.1 Integrando las librerías de OpenCV para la plataforma *Microsoft Windows 7*

La forma más sencilla de instalar OpenCV en Windows es a través de las librerías pre-compiladas habilitadas para ello, y que se pueden descargar directamente accediendo al siguiente enlace [22].

Al venir encapsuladas en la forma de un instalador típico para entornos *Windows*, el proceso de instalación de las librerías es sencillo. Una vez instaladas, dispondremos en el directorio de instalación de los diferentes subdirectorios que contienen entre otras, las librerías de enlace dinámico (*.dll*) para el compilador *minGW* utilizado por *Windows* para compilar los proyectos Qt.

MinGW (*Minimalist GNU for Windows*) es una implementación de los compiladores GCC de GNU (compiladores de código C/C++ para Unix) para la plataforma *Win32*. Al ser la forma en la que el entorno QtCreator compila los proyectos en las plataformas

Windows, es necesario que las librerías de OpenCV se ajusten a este compilador cuando se trabaje con este tipo de plataformas.

Por tanto, en el directorio de instalación de OpenCV en *Windows* localizaremos la siguiente ruta:

```
<dir_instalación_OpenCV>\install\x64\mingw\bin
```

En esta ruta se incluyen todas las librerías de OpenCV preparadas para ser enlazadas por el compilador *minGW* de Qt. Así pues, para configurar la plataforma *win32* para que cargue en la construcción del proyecto Qt las librerías de OpenCV, editaremos el ámbito de la plataforma *Windows* del fichero *.pro* con el siguiente contenido:

```
win32 {

    LIBS_PATH = "$$OPENCV_PATH\\install\\x64\\mingw\\bin"
    LIBS     += -L$$LIBS_PATH \
                -lopencv_core2410 \
                -lopencv_highgui2410 \
                -lopencv_imgproc2410 \
                -lopencv_features2d2410 \
                -lopencv_objdetect2410
}
```

Como podemos ver, las librerías incluidas son las relativas a los módulos de OpenCV que hemos explicado en secciones anteriores. Añadiremos también las librerías del módulo *Features2D* para dejar abierta la posibilidad de añadir nuevas funcionalidades a la aplicación en un futuro, y que éstas puedan utilizar algunas de las funciones de detección de características de las que dispone este módulo.

Asimismo, es necesario añadir también al fichero *.pro* las cabeceras *.h* de las clases C++ incluidas en las librerías, de forma que podamos importarlas directamente en el código de nuestro proyecto Qt:

```
INCLUDEPATH += $$OPENCV_PATH\\include
```

4.2.2 Integrando las librerías de OpenCV para la plataforma *Linux Ubuntu 14.04*

Para instalar la biblioteca OpenCV en una distribución *Linux* basada en *Debian* (en nuestro caso hemos utilizado la distribución *Ubuntu 14.04 – Trusty Tahr*), la forma más recomendable es hacerlo a través de la herramienta de generación de código *CMake*.

CMake permite generar ficheros de construcción (*MakeFiles*) a partir de una serie de reglas establecidas en un fichero de configuración llamado *CMakeLists.txt*, y que indica cómo se deben construir los programas y en que localización deben almacenarse. Muchos programas de instalación para Linux, vienen ya preparados con un fichero *CMakeLists* para que sean construidos a partir de *CMake.*

No obstante, quizás la forma más sencilla de instalar OpenCV en Ubuntu sea a través de la herramienta *APT* (*Advanced Packaging Took*) a través del comando:

```
>sudo apt-get install opencv-<versión>
```

Sin embargo, este método de instalación puede dar lugar a que algunas librerías no estén disponibles (tales como las relativas al módulo *nonfree*) tras la instalación, y además, no permite especificar opciones de configuración, tales como las librerías gráficas de visualización de componentes (GTK+, QtWidgets), y la opción de incluir aceleradores de rendimiento tales como OpenCL para las funciones de OpenCV.

Así pues, para instalar OpenCV en Ubuntu, podemos descargar la última versión del siguiente enlace [23]. Al descomprimir el directorio, podemos seguir los siguientes pasos para la instalación con *CMake*:

1- Nos situamos en el directorio de OpenCV

2- Creamos un directorio llamado *build* y nos situamos dentro de él.

3- Con *CMake* instalado escribimos en la consola:

```
>cmake ..
```

4- Tras el proceso de generación de MakeFiles escribimos:

```
>make
```

5- Y finalmente para la instalación en el sistema:

```
>sudo make install
```

Tras el proceso de instalación, los directorios de las librerías de OpenCV y de las cabeceras deberían de haber sido generados dentro del directorio *build*, y además estas deberían de haber sido incluidas en el *pkg-config* de la instalación. El software *pkg-config* permite acceder a las librerías y cabeceras de la instalación de OpenCV de la siguiente manera:

```
>pkg-config --cflags –libs opencv
```

Por tanto, para incluir las librerías y cabeceras de OpenCV dentro del ámbito de la plataforma *Linux Ubuntu* podemos realizarlo de forma sencilla añadiendo las siguientes líneas al fichero *.pro*:

```
INCLUDEPATH += “pkg-config --cflags opencv”
LIBS += “pkg-config --libs opencv”
```

No obstante, si se quieren incluir librerías y cabeceras de módulos específicos también es posible hacerlo de la misma manera que hacíamos con la plataforma *Windows* tomando como ruta base el directorio *build*.

4.2.2 Integrando las librerías de OpenCV para la plataforma *Android*

Las librerías de OpenCV para la plataforma Android podemos descargarlas del siguiente enlace [24].

Para este caso concreto, el proceso de incorporar las librerías es un poco más delicado que en el resto de plataformas, debido a que al compilar las librerías de OpenCV, al igual que las librerías de Qt y el resto de recursos necesarios para el funcionamiento de la aplicación, éstas además tienen que ser traducidas al mismo tiempo a código *Java*.

Así, la nueva versión 5 de Qt incorpora un conjunto de librerías y controladores nativos para la plataforma *Android*, que son necesarias para que el compilador pueda traducir el código nativo en C++ a código *Java*, de forma que éste pueda ser interpretado por la máquina *dalvik* del dispositivo *Android*, y por lo tanto éstas librerías también deberán ser cargadas en el dispositivo destino al realizar la instalación de la aplicación.

Sin embargo, entre este conjunto de elementos es necesario añadir alguna librería nativa que permita al dispositivo *Android* interpretar las funciones de OpenCV incluidas en la aplicación. Es por ello que el primer paso necesario antes de compilar la aplicación, consiste en copiar la librería nativa *opencv_java.so* en el directorio de librerías nativas de Qt para la plataforma *Android*. La librería *opencv_java.so* podemos encontrarla en la ruta del directorio de librerías nativas del *SDK* de OpenCV para *Android*.

Si queremos utilizar también otras librerías nativas tales como aquellas que permiten el funcionamiento de la cámara del dispositivo, podremos incluirlas de la misma manera en el directorio de librerías nativas de Qt para la plataforma (para la cámara nativa podríamos utilizar la librería *libnative_camera*). Sin embargo, en algunos casos es

posible que existan dependencias con otras librerías y obtengamos por ello problemas en tiempo de compilación o ejecución, por lo que es mejor evitar utilizarlas si no las vamos a necesitar.

Una vez incorporada la librería *opencv_java* entre las librerías nativas de Qt para la plataforma *Android*, tenemos que incluirla también en el fichero *.pro* para que pueda enlazarse con la aplicación, además de incluir también el conjunto de librerías de OpenCV de los distintos módulos que utilizaremos:

```
android {

    ANDROID_LIBS = "$$ANDROID_OPENCV/sdk/native/libs"
    ANDROID_JNI = "$$ANDROID_OPENCV/sdk/native/jni"
    ANDROID_3PARTY = "$$ANDROID_OPENCV/sdk/native/3rdparty"

    INCLUDEPATH += \
     $$ANDROID_JNI/include/ \                 #core module
     $$ANDROID_JNI/include/opencv2/highgui/ \ #highgui module
     $$ANDROID_JNI/include/opencv2/imgproc/   #imgproc module

    LIBS += \
    #Esta línea es necesaria!
    $$ANDROID_LIBS/armeabi-v7a/libopencv_java.so \
    #$$ANDROID_LIBS/armeabi-v7a/libnative_camera_r2.2.0.so \
    $$ANDROID_LIBS/armeabi-v7a/libopencv_contrib.a \
    $$ANDROID_LIBS/armeabi-v7a/libopencv_legacy.a \
    $$ANDROID_LIBS/armeabi-v7a/libopencv_ml.a \
    $$ANDROID_LIBS/armeabi-v7a/libopencv_objdetect.a \
    $$ANDROID_LIBS/armeabi-v7a/libopencv_calib3d.a \
    $$ANDROID_LIBS/armeabi-v7a/libopencv_video.a \
    $$ANDROID_LIBS/armeabi-v7a/libopencv_features2d.a \
    $$ANDROID_LIBS/armeabi-v7a/libopencv_highgui.a \
    $$ANDROID_LIBS/armeabi-v7a/libopencv_androidcamera.a \
    $$ANDROID_LIBS/armeabi-v7a/libopencv_flann.a \
    $$ANDROID_LIBS/armeabi-v7a/libopencv_imgproc.a \
    $$ANDROID_LIBS/armeabi-v7a/libopencv_core.a \

    $$ANDROID_3PARTY/libs/armeabi-v7a/liblibjpeg.a \
    $$ANDROID_3PARTY/libs/armeabi-v7a/liblibpng.a \
    $$ANDROID_3PARTY/libs/armeabi-v7a/liblibtiff.a \
    $$ANDROID_3PARTY/libs/armeabi-v7a/liblibjasper.a \
    $$ANDROID_3PARTY/libs/armeabi-v7a/libtbb.a
```

Podemos observar que en este caso en el fichero *.pro* se incluyen un número más elevado de librerías que en las plataformas anteriores. Esto es necesario por dos motivos principales.

El primero es que, aunque algunos de los módulos de OpenCV no los vayamos a utilizar en la aplicación actual, es aconsejable incluir todos los módulos para hacer más flexible la escalabilidad de la aplicación en un futuro en este tipo de plataformas.

El segundo motivo, y más importante, es el mismo que comentábamos anteriormente sobre las dependencias de las librerías nativas. Debido a ciertos aspectos de enlazado de las librerías por el compilador GCC *armeabi (ARM Extended Application Binary Interface*) de código nativo para *Android* utilizado por Qt, el orden de la inclusión de las librerías es importante (de hecho el orden en el que aparecen las librerías en las líneas anteriores del fichero *.pro* no es casual), debido sobre todo a las dependencias existentes entre ellas. Por ello, hay que tener cuidado con el orden a la hora de incluir nuevas librerías, sobre todo si implican código nativo para plataformas con arquitecturas diferentes a las más usuales.

También podemos observar que se han incluido en el fichero *.pro* la ruta a las cabeceras de las funciones de OpenCV, y además otras librerías llamadas *3rdParty*. Estas librerías son necesarias para aspectos tales como la decodificación de formatos de imagen en OpenCV.

4.3 Desarrollo de los métodos de procesamiento de video

A continuación nos dedicaremos a describir los desarrollos que se han llevado a cabo en la aplicación *Control Camera IP* para la inclusión de procesamiento de video haciendo uso de algunas de las librerías de OpenCV, las cuales ya hemos avanzado en apartados anteriores.

Para acceder a cada una de las funcionalidades de OpenCV se ha añadido a las diferentes vistas de la interfaz gráfica un nuevo elemento en la barra de menús denominado *Procesamiento de Video*, de forma que al desplegarlo aparecen las diferentes acciones para lanzar alguno de los métodos de procesamiento de video implementados. Estas acciones tienen como nombre "Detección de Bordes", “Brillo y Contraste”, “Sustracción de Fondo”, “Blobs Color” y “Activar/Desactivar Detección de Caras”.

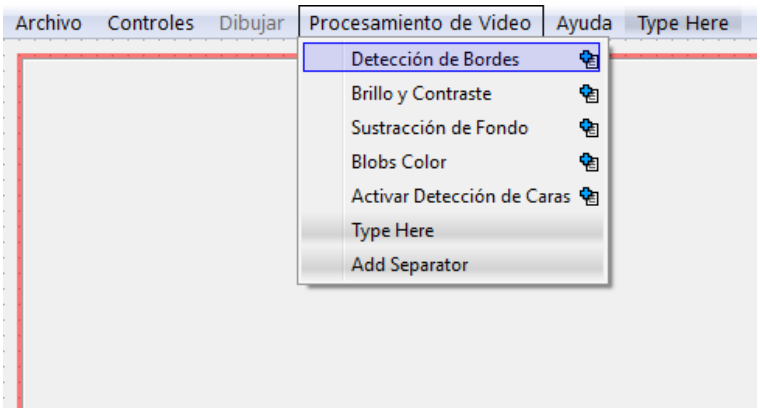

Figura 26. Diseño del nuevo menú *Procesamiento de Video* con sus distintas acciones

Al activar alguna de las cuatro primeras acciones, se visualizará un diálogo (QDialog) que permitirá al usuario modificar algunos parámetros del método o algoritmo, pudiéndose apreciar estos cambios sobre el monitor de imágenes. Estos diálogos estarán definidos en el proyecto Qt como clases de formulario *Form* dentro de un nuevo paquete denominado *procesamientoVideo*, de forma similar a como está definido el paquete *dialogoConexion*. Asimismo, para organizar los distintos métodos desarrollados con OpenCV, además de las funciones auxiliares utilizadas por los mismos, se ha creado una nueva clase dentro del paquete *auxiliares* denominada "FuncionesOpenCV".

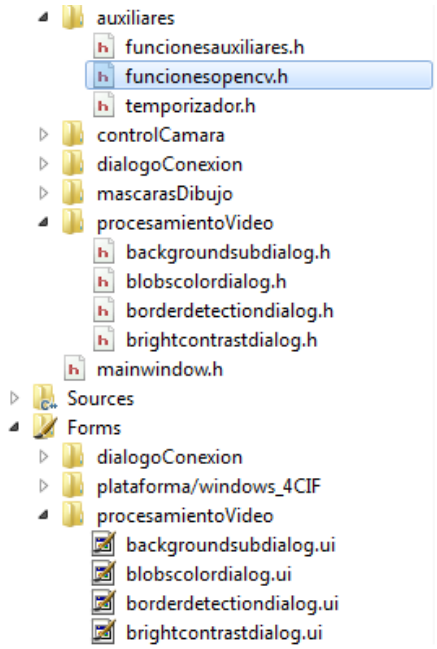

Figura 27. Nueva estructura del proyecto Qt

Clase *FuncionesOpenCV* y paquete *procesamientoVideo* con los diálogos de los métodos de OpenCV

4.3.1 La nueva clase auxiliar FuncionesOpenCV

La clase FuncionesOpenCV definida dentro del paquete *auxiliares*, ha sido creada con el propósito de contener todas aquellas funciones y algoritmos que están dentro del ámbito de los nuevos desarrollos realizados con OpenCV dentro de la aplicación. Es decir, en esta clase están definidas todas aquellas operaciones de conversión, funciones de tratamiento sobre la imagen o algoritmos de detección que son utilizados por el programa principal para ejecutar los distintos métodos de procesamiento de vídeo que explicaremos más adelante.

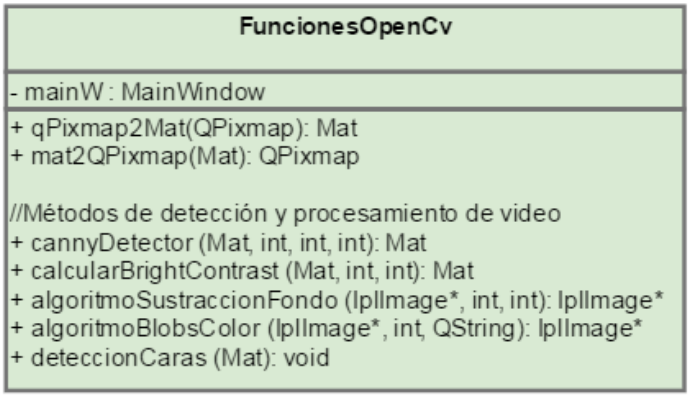

Figura 28. Nueva Clase FuncionesOpenCV del paquete auxiliares

A parte de los algoritmos o funciones propias de los métodos desarrollados, en la clase FuncionesOpenCV se han incluido dos herramientas auxiliares de conversión que tienen la capacidad por un lado, de transformar las capturas de la cámara obtenidas por la aplicación y que son almacenadas como una imagen de Qt (QPixmap) en una imagen OpenCV con estructura *Mat*, así como también, realizar la operación inversa (de *Mat* a QPixmap).

Estas funciones son fundamentales para poder aplicar las operaciones de OpenCV sobre la imagen cargada en el componente QLabel (monitor) de la aplicación. Así, previamente a la utilización de cualquier método de procesamiento sobre la imagen con OpenCV, se convertirá la imagen QPixmap a una estructura *Mat*. Asimismo, una vez realizado el tratamiento sobre la misma, la imagen *Mat* volverá a convertirse en una imagen QPixmap para ser cargada de nuevo en el monitor.

A continuación explicamos de forma breve el funcionamiento de estas dos funciones de conversión:

4.3.1.1 Conversión de QPixmap a *Mat*

La función *qPixmap2Mat()*, definida en la clase FuncionesOpenCV, recibe como parámetro de entrada una imagen QPixmap y devuelve como resultado una estructura *Mat* del mismo tamaño y formato. En realidad, la llamada a esta función lo que hace simplemente es llamar a su vez a otra función auxiliar llamada *qImage2Mat()*, que recibe como parámetro de entrada una imagen QImage creada a partir del QPixmap original utilizando para ello el método *toImage()* de la clase QPixmap.

Es decir, el proceso de conversión propiamente dicho se realiza tomando como origen una imagen QImage. Esto es así, ya que esta clase está implementada para el acceso y manipulación directa sobre píxeles a diferencia de la clase QPixmap, que está diseñada y optimizada para mostrar imágenes por pantalla.

De esta forma, para contener la información en la estructura *Mat*, la clase QImage dispone de la función *bits()*, que devuelve una referencia de tipo *unsigned char* a la primera posición (primer píxel) de la imagen. Tal y como ya comentamos en secciones anteriores, al crear una estructura *Mat* con el constructor de copia, podíamos hacer que su puntero de matriz apuntara a una posición de memoria donde se tuviera almacenada una imagen. Por lo tanto, haremos que el puntero de matriz de la estructura *Mat* de destino para la conversión desde QImage, apunte a la referencia obtenida a partir de la función *bits()* de esta última.

Por poner un ejemplo, partiendo de una imagen QImage con formato RGB32, donde cada píxel es de 8 bits (*unsigned char*) con 4 canales (RGB+*alpha*), la conversión a una imagen *Mat* sería:

```
Mat mat(inImage.height(), inImage.width(), CV_8UC4,
    const_cast<uchar*>(inImage.bits()), inImage.bytesPerLine());
```

En la expresión anterior, utilizamos las funciones *height()* y *width()* de QImage para establecer el tamaño de la estructura *Mat* en el constructor de copia. La profundidad (*depth*) y el número de canales son traducidos directamente al formato de OpenCV para imágenes *Mat*. Por otro lado, la función *bytesPerLine()* de QImage devuelve el número de bytes por línea, o lo que es lo mismo, el número de bytes de todos los píxeles de una fila de la imagen, lo que traducido a OpenCV sería el paso en anchura (*widthstep*).

Finalmente, y tal y como comentábamos antes, el puntero de matriz de la estructura *Mat* se establece a la referencia devuelta por la función *bits()* de QImage, por lo que la imagen *Mat* ya estaría completamente definida.

Imágenes RGB con otro número de canales

La imagen recibida desde la cámara Axis232D+, utilizada para las pruebas en el PFC original y en este Trabajo Fin de Grado, son recibidas en un método de almacenamiento RGBA de cuatro canales (el canal *alpha* es utilizado como byte de opacidad). Como cada canal corresponde a un byte, este formato también puede expresarse como RGB32.

Qt maneja varios formatos para los métodos de almacenamiento de la imagen (al igual que OpenCV). Además del formato RGB32 comentado, tenemos también por ejemplo el formato RGB888, con profundidad de 8 bits y 3 canales, y el formato *Indexed8* con una profundidad también de 8 bits y un número de canales igual a 1.

La función *qImage2Mat()* distinguirá entre cada uno de estos formatos para poder tener en cuenta otros tipos de imágenes que pudiéramos recibir en caso de conectarnos con otras cámaras. En este caso distinguimos los casos más típicos que son: imágenes monocromáticas o de escala de grises (1 canal), imágenes RGB (3 canales) e imágenes RGBA (4 canales).

4.3.1.2 Conversión de *Mat* a QPixmap

Para hacer la conversión inversa desde la imagen *Mat*, obtenida una vez procesada por algún método de procesamiento de imagen, a la imagen QPixmap que se cargará en el monitor, se procede de una forma muy similar a la explicada en el apartado anterior.

Para empezar, y del mismo modo que ocurre con el caso anterior, la conversión propiamente dicha no se hace a través de la función *mat2QPixmap()*, sino que esta función utiliza el método estático *fromImage()* de QPixmap, que recibe como parámetro de entrada una imagen QImage, y devuelve el objeto QPixmap en cuestión. Esta imagen QImage se obtiene a su vez realizando una llamada al método *mat2QImage()* que recibe como parámetro de entrada la estructura *Mat* de origen.

El método *mat2QImage()*, es muy similar en cómo está estructurado al método *qImage2Mat()*, ya que consiste en crear un nuevo objeto QImage a partir de su constructor, en el que recibe como argumentos un conjunto de parámetros de la imagen

Mat que, aunque en diferente orden, se corresponden con los parámetros recibidos por el constructor de copia de *Mat* en la operación de conversión inversa.

Así pues, el mismo ejemplo que establecíamos con un formato RGB32 en la conversión partiendo de una imagen QImage, la conversión partiendo de una imagen *Mat* con el formato equivalente (CV_8UC4) sería:

```
QImage image (inMat.data, inMat.cols,
      inMat.rows, inMat.step, QImage::Format_RGB32);
```

Como podemos ver, el campo *data* de la estructura *Mat* (puntero de matriz) se establece como referencia al primer pixel de la imagen QImage. El campo *cols* y *rows* establecen la anchura (*width*) y altura (*height*) de la imagen, el campo *step* establece el paso en anchura o número de bytes por línea (*bytesPerLine*), y por último se establece el formato de la imagen al equivalente para QImage.

Al igual que en la conversión inversa, en la función *mat2QImage* también se distinguirá entre los formatos para imágenes monocromáticas (CV_8UC1) e imágenes RGB sin el canal *alpha* (CV_8UC3).

El resto de funciones de la clase FuncionesOpenCV las detallaremos en las siguientes secciones al mismo tiempo que vayamos desgranando la implementación de los métodos desarrollados.

También, veremos los diseños de las interfaces de diálogo (QDialog) realizados para algunos de estos métodos, con el propósito de poder controlar, a partir de ellos, diferentes parámetros de corrección que permitan modificar el comportamiento de estos algoritmos y métodos de detección y procesamiento de imagen.

4.3.2 Implementación del método de detección de bordes

Este método para la detección de bordes sobre la imagen recibida desde la cámara, es posible activarlo a través de la acción “Detección de Bordes” asociada al nuevo elemento de menú *Procesamiento de Video*.

Una vez activada, la señal *triggered()* asociada a la acción de menú, lanza un *Slot* ubicado en la clase principal *MainWindow* llamado *mostrarDeteccionBordes()*. Esta función de SLOT lanzará la creación de un nuevo diálogo de detección de bordes (*BorderDetectionDialog*) en caso de que no exista, o bien, en caso de que ya exista alguno creado en memoria, hará que se visualice. El diseño del diálogo con la herramienta QDesigner podemos verlo en la siguiente figura:

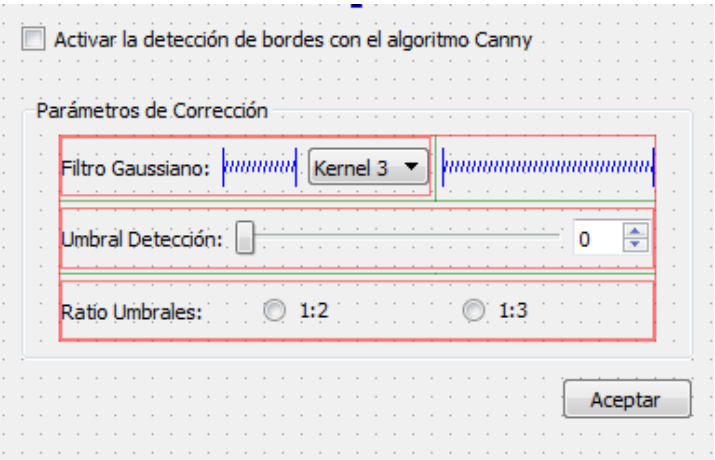

Figura 29. Dialogo de control para la detección de bordes en la imagen

Tal y como podemos ver en la imagen, este diálogo está compuesto de un elemento QCheckBox que permitirá activar la detección de bordes sobre la imagen, y de un grupo de parámetros de corrección para poder realizar modificaciones de los valores de umbral de detección y valores de filtrado.

Una vez activamos el *CheckBox* del diálogo, la señal *clicked()* del componente lanza el Slot *activarDeteccion()* de la clase BorderDetectionDialog. En este Slot, se habilitan los parámetros de corrección (inicialmente deshabilitados), y se establece a *True* el valor de la variable global booleana *deteccionBordesActiva* definida en la clase principal.

Esta variable permite saber a la interfaz principal (*MainWindow*) si el método de detección de bordes ha sido activado. De esta forma, podemos detectar el momento a partir del cual podemos interceptar las imágenes recibidas para que sean procesadas por nuestro método de detección.

El mecanismo anterior es idéntico en el resto de métodos de procesamiento de video con OpenCV, con la excepción de que el método de detección de caras es comandado directamente desde la acción asociada en el menú *Procesamiento de Video*, sin tener que ser activado a través de un QDialog.

La imagen por tanto, es interceptada por alguno de los métodos de procesamiento de video siempre y cuando el *flag* de activación de dicho método tenga el valor *True*. Mientras el *flag* esté activo, la imagen será procesada por el método elegido. Por último, al desactivar el método, dicho *flag* volverá a su valor *False* y por lo tanto la aplicación volverá a su estado original, en el que seguirá mostrando la imagen de la cámara sin alterar.

Tras la activación de alguno de estos *flags*, lo primero que se hace es una conversión de la imagen capturada (componente QPixmap) a una imagen OpenCV con la estructura *Mat*, tal y como vimos en el apartado anterior. Tras el procesado, la imagen *Mat* vuelve a convertirse en un componente QPixmap, y vuelve a cargarse en el monitor.

4.3.2.1 Descripción del método

Al activar en el QDialog el método de detección de bordes, se lanzará la función *cannyDetector()* definida en la clase FuncionesOpenCV. El método utilizado para la detección de bordes está implementado en base al algoritmo de *Canny*, y utiliza las funciones *blur()* y *Canny()* de OpenCV que ya introdujimos en capítulos anteriores.

El método consiste básicamente en reducir el ruido digital de la imagen a través de un filtro de caja normalizado (función *blur()*) con un tamaño de *kernel* determinado (que será el mismo tamaño que el que se aplicará al filtro gaussiano de la función *Canny*). Este ruido digital suele ser provocado usualmente por condiciones de poca luz en las que la sensibilidad del sensor electrónico (CCD) suele ser elevada para captar la máxima cantidad de luz posible. En este caso aumentan los datos aleatorios generados por la alta actividad eléctrica del sensor que aumentarán a su vez el ruido en la imagen.

Tras aplicar este primer filtro, la imagen es introducida como parámetro de entrada a la función *Canny()*, la cual se encargará de aplicar sobre ella el algoritmo del mismo nombre.

El propósito fundamental de la detección de bordes en general es el de reducir de forma significativa la cantidad de datos en una imagen, de forma que se preserven las propiedades estructurales de la misma con el propósito de ser utilizadas para procesamientos posteriores. En 1986, John F. Canny halló que los requerimientos para una aplicación de detección de bordes en diferentes sistemas de visión son relativamente los mismos. Por lo tanto, concluyó que el desarrollo de una solución para la detección de bordes que apele a estos requerimientos era posible implementarla en una gran variedad de situaciones.

Entre estos criterios o requerimientos necesarios se incluía la capacidad de maximizar la probabilidad de detección de bordes efectivos, así como de minimizar la detección de falsos positivos; esta detección de bordes debería ser tan cercana como fuera posible a los bordes reales de los objetos en la imagen; y finalmente, minimizar los bordes

detectados a una respuesta por borde real (lo cual podría enmarcarse también dentro del primer criterio).

El algoritmo de Canny establece una solución óptima para ciertos tipos de bordes (conocidos como bordes de paso o *step edges*) formulada en base a los criterios anteriores, y está definido en base a los siguientes pasos:

Aplicación de filtro Gaussiano

Aunque hemos filtrado buena parte del ruido de la imagen con la función *blur()*, la función *Canny()* incorpora como primer paso la aplicación de un filtro Gaussiano para la reducción significativa del ruido en la imagen con el objetivo de prevenir falsos positivos en la detección de bordes [25]. El filtro Gaussiano se diferencia del filtro de caja normalizado de la función *blur()* en que utiliza una semilla o *kernel Gaussiano* donde los píxeles vecinos intervienen en el cálculo de convolución con una distribución de pesos centrados en la media (píxel central), donde el peso de los vecinos más alejados del píxel central decrece.

$$K = \frac{1}{159}\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}$$

Figura 30. Coeficientes de un *kernel Gaussiano* de tamaño 5
El valor 15 es el peso del píxel central situado en la media. Conforme nos alejamos del píxel central, el valor de los pesos aplicado a los píxeles va decreciendo.

Aunque la aplicación de este filtro es más costosa en términos de tiempo que en el caso del filtro de caja normalizado, su eficacia en la eliminación de falsos positivos causados por el ruido en la detección de bordes es muy elevada.

Obtención del gradiente de intensidad de la imagen

Antes de continuar, hemos de destacar que previamente a aplicar cualquiera de las operaciones y pasos a realizar por las funciones *blur()* y *Canny()*, es conveniente (si no necesario) convertir la imagen original, normalmente en RGB o RGBA, en una imagen en escala de grises. Esto es motivado por un lado por la cuestión de limitar los

requerimientos computacionales que supone la utilización de este método detección, pero fundamentalmente, y también por el mismo motivo, es debido a que uno de los pasos del algoritmo de Canny tiene como objetivo localizar cambios en la imagen donde la intensidad de los píxeles en la escala de grises ha tenido una variación significativa. Este paso se conoce como "búsqueda de gradientes de intensidad".

Los gradientes en cada píxel de la imagen filtrada se obtienen con lo que se conoce como el operador *Sobel (Sobel-operator)* [26] . El primer paso consiste en aproximar el gradiente (primera derivada) en la dirección *x* y en la dirección *y* respectivamente, aplicando los *kernels* mostrados a continuación:

$$K_{GX} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad K_{GY} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Así, las magnitudes de los gradientes se pueden determinar como una medida de la distancia Euclídea, o bien aplicando una medida de la distancia Manhattan (con esta última se reduce la complejidad computacional) sobre las aproximaciones de gradiente en la dirección *x* e *y* para cada uno de los píxeles de la imagen, aplicando los *kernels* anteriores.

Supresión de los no máximos

Este paso consiste básicamente en convertir los bordes difuminados obtenidos en la imagen de magnitudes de gradiente, en bordes más "refinados". Este efecto se consigue preservando todos los máximos locales en la imagen de gradientes y eliminando todo lo demás (los no máximos). Para ello se realiza una comparación de los píxeles teniendo en cuenta la dirección de su gradiente y su módulo con respecto a sus píxeles vecinos. Si los valores de los vecinos en la línea positiva y negativa de su gradiente son mayores, el valor del píxel es suprimido.

Umbral doble

Los píxeles que quedan tras aplicar la supresión de los no máximos están todavía marcados con su "fuerza" o valor de gradiente. Es posible que muchos formen parte de bordes verdaderos, pero algunos serán producto del ruido o de las variaciones de color en la imagen. La mejor manera de dejar únicamente los bordes más "fuertes" (con mayor valor de gradiente) es a través de un valor de umbral o *threshold*.

El algoritmo de Canny utiliza dos valores de umbral. Por un lado, los píxeles de bordes más fuertes que el valor de umbral alto son marcados como fuertes (*strong*). Por otro, los píxeles de bordes más débiles que el valor de umbral bajo son suprimidos. Finalmente, los píxeles de bordes que están entre los dos valores de umbral serán marcados como débiles (*weak*).

Rastreo de bordes por histéresis

La idea es que los bordes fuertes sean interpretados como bordes verdaderos, y por lo tanto serán incluidos inmediatamente en la imagen final. Sin embargo, los bordes débiles solo serán incluidos si éstos están conectados a bordes fuertes, ya que es más probable que, si este no es el caso, estos bordes sean producto del ruido o de variaciones en el color de la imagen.

El rastreo de los bordes puede ser implementado utilizando análisis de conjuntos de píxeles o BLOBs (*Binary Large OBjects*) en la imagen, de forma que si estos BLOBs contienen al menos un píxel fuerte, entonces estos conjuntos serán preservados.

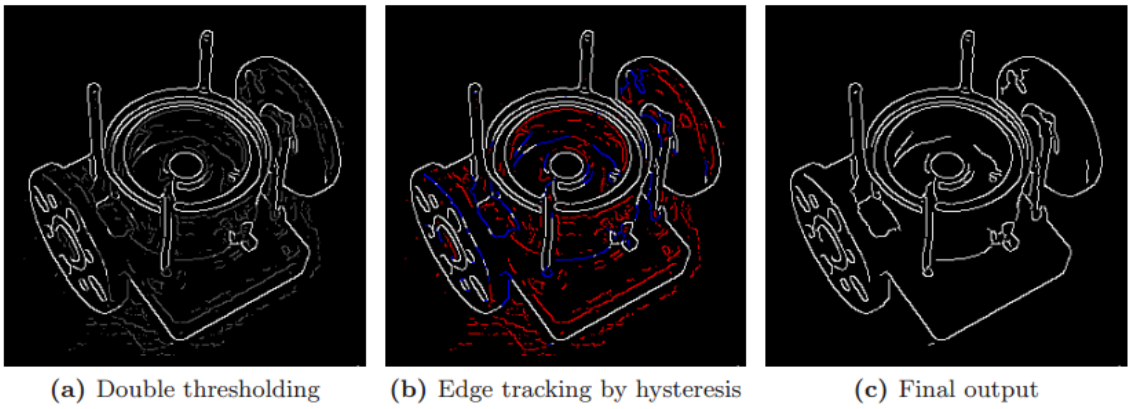

Figura 31. Rastreo de bordes y salida final
La imagen del medio muestra los bordes fuertes en blanco, los bordes débiles conectados a los fuertes en azul, y el resto de bordes débiles que son descartados para la imagen final en rojo

4.3.2.2 Diálogo *BorderDetectionDialog*

Volviendo al diálogo (QDialog) mostrado en la figura 19 de un apartado anterior, podemos distinguir una serie de parámetros de corrección con los que podemos interactuar para modificar ciertos valores que inciden en la detección de bordes, y que ya hemos comentado en el punto anterior. Son tres los parámetros que pueden ser modificados para este método de detección.

En primer lugar, el tamaño del *kernel* utilizado al aplicar el filtro Gaussiano en el primer paso del algoritmo de Canny. El elemento QComboBox nos permite seleccionar entre dos valores de tamaño de *kernel*: valor 3 y valor 5.

Este tamaño de *kernel* es pasado como argumento tanto a la función *blur()* como a la función *Canny()* de OpenCV, de forma que tanto el filtro de caja normalizado como el filtro Gaussiano se realice utilizando el tamaño de *kernel* indicado. El efecto producido en la imagen resultante en función de si se establece uno u otro tamaño dependerá también de los valores de umbral, pero aunque pueda parecer que un valor más elevado de tamaño de *kernel* produciría mejores resultados en la detección de bordes, podemos comprobar en las siguientes capturas que en nuestro caso esto no ocurre así:

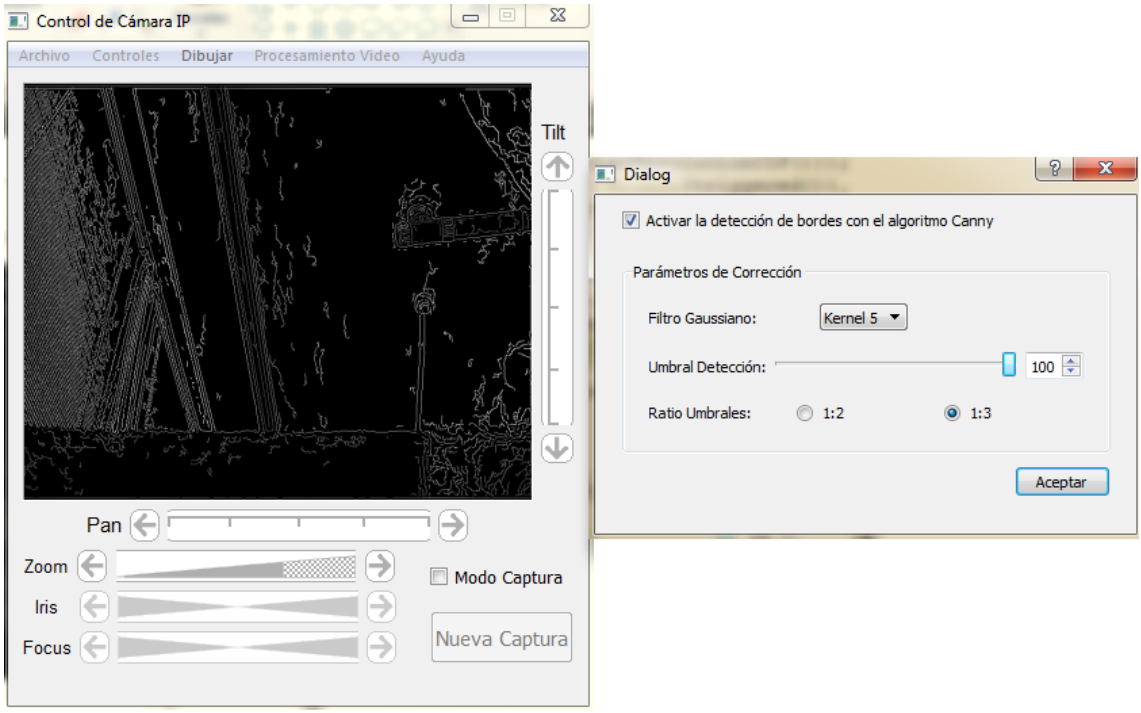

Figura 32. Detección de bordes con un tamaño de *Kernel* igual a 5

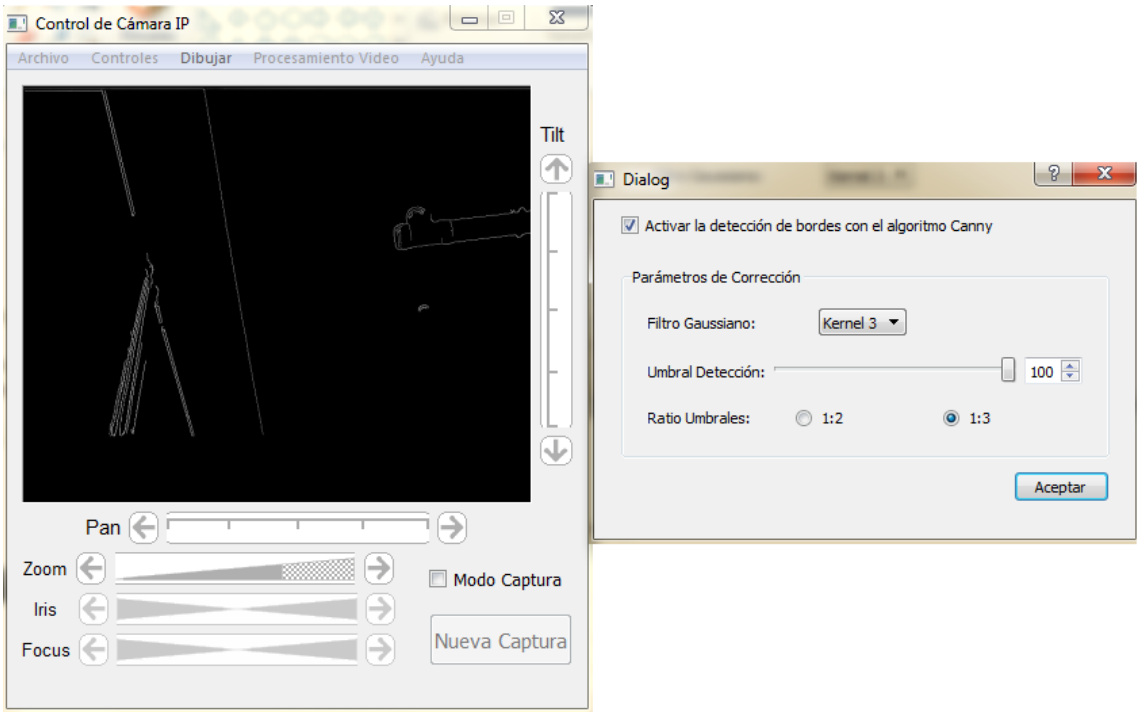

Figura 33. Detección de bordes con un tamaño de *Kernel* igual a 3

Esto es debido a que el filtro Gaussiano viene determinado por el valor de la desviación típica (σ) de la distribución obtenida para las primeras derivadas de *x* e *y* que establecen los valores del *kernel* sobre cada píxel de la imagen. Seleccionando valores bajos de desviación típica (valor bajo de tamaño de *kernel*), se filtrarán los cambios o transiciones de intensidad más agudas entre los píxeles. En cambio, con valores altos de desviación típica, estas transiciones serán más graduales.

En ocasiones, el hecho de que las transiciones de intensidad entre píxeles sean graduales (gran suavizado), nos permitirá tener una imagen con menos ruido, pero por otro lado podremos perder definición, lo cual para el computo de los gradientes en la detección de bordes puede resultar contraproducente, generando muchos falsos positivos. Es por ello, que para cada imagen debe buscarse el tamaño de *kernel* más apropiado que permita que los cambios de intensidad entre los píxeles no sean ni muy bruscos ni muy graduales.

También como ya hemos visto anteriormente, otro de los pasos a aplicar en el algoritmo de Canny es el de la utilización del doble umbral. En el método que hemos implementado, hemos utilizado un único valor de umbral (umbral bajo) como parámetro de corrección en el diálogo. El umbra alto sin embargo, es calculado utilizando un valor de ratio que establece una relación 1 a 2 o 1 a 3 con el umbral bajo. Es decir, podremos seleccionar que el umbral alto sea o bien el doble o el triple del valor del umbral bajo. De esta forma, modificando cualquiera de los dos parámetros de corrección del diálogo

relativos al umbral bajo y al ratio, podemos obtener diferentes resultados tal y como se puede apreciar en las siguientes capturas:

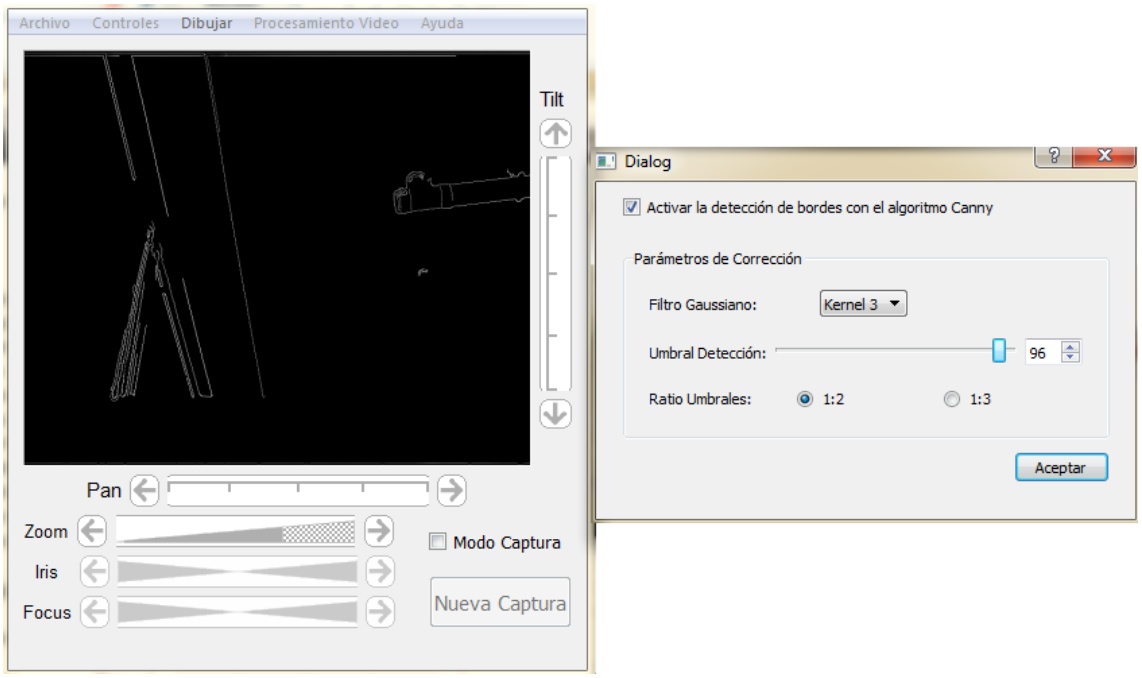

Figura 34. Detección de bordes con ratio 1:2 y umbral bajo a 96

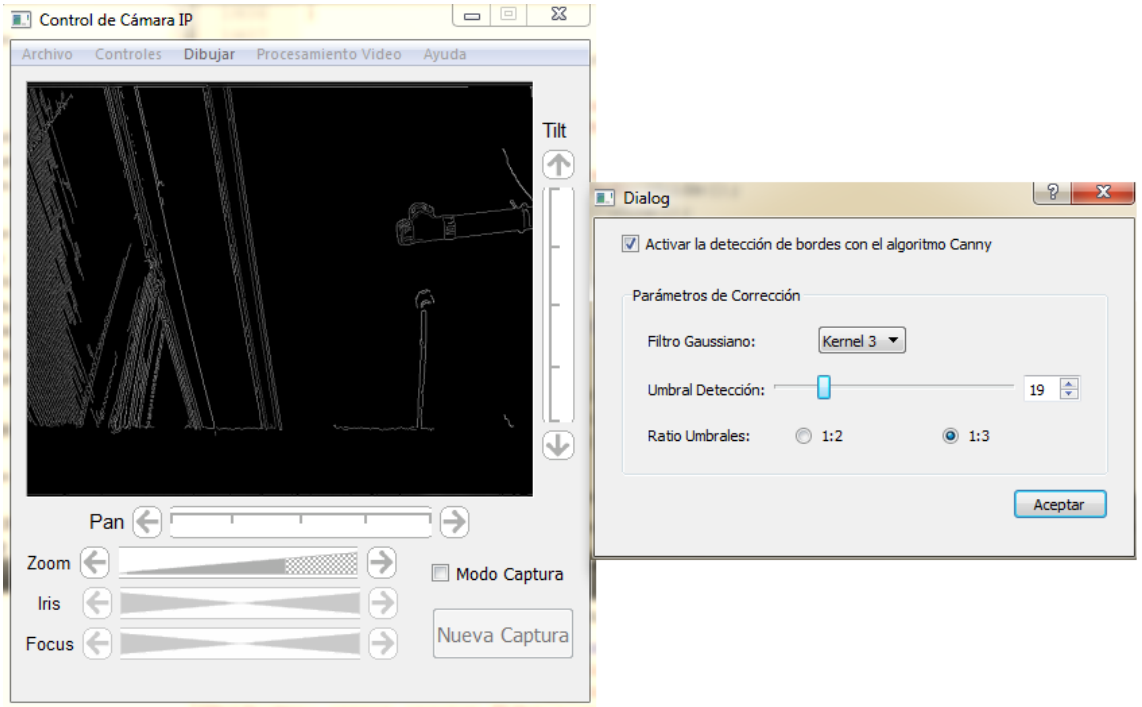

Figura 35. Detección de bordes con ratio 1:3 y umbral bajo a 19

4.3.3 Implementación de las funciones de brillo y contraste

Las funciones de regulación de brillo y contraste que se han implementado pueden activarse a través de la acción "Brillo y Contraste" del menú *Procesamiento de Video.*

De igual forma que ocurría con la detección de bordes, la señal *triggered()* asociada a esta acción del menú lanzará un *Slot* de la clase *MainWindow* llamado *mostrarBrilloContraste()*, que creará un nuevo diálogo para activar las funciones de brillo y contraste:

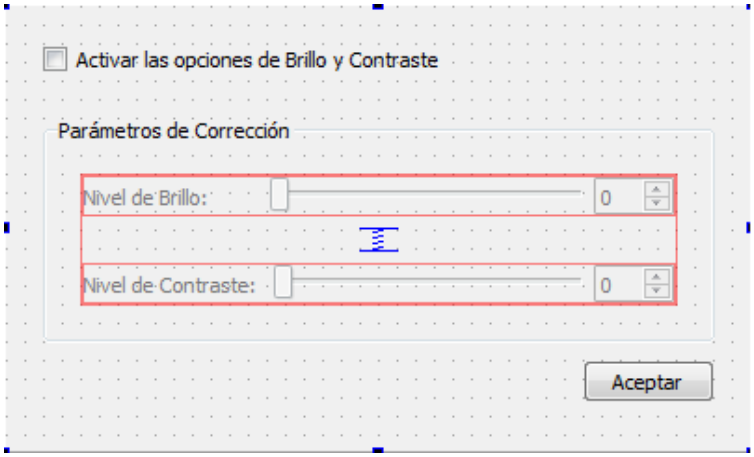

Figura 36. Dialogo de control para las funciones de brillo y contraste

Como en el método de detección de bordes, las funciones de brillo y contraste se activan a través del elemento QCheckBox del diálogo. Al clicar sobre este elemento, se lanzará el Slot *activarBrilloYContraste()* de la clase BrightContrastDialog (la clase donde está definido el diálogo). En este Slot, se habilitarán los parámetros de corrección para regular el brillo y contraste de la imagen, y se establecerá el valor de la variable de la clase principal *brilloContrasteActivo* al valor *True*, de forma que a partir de ese momento se puedan interceptar las imágenes recibidas desde la cámara para ser procesadas por estas funciones de brillo y contraste.

Por lo tanto, entramos en el mismo mecanismo común a todos los métodos y que permite interceptar las imágenes, convertirlas al formato de OpenCV, aplicar el método deseado y volver a convertirlas de nuevo al componente QPixmap del monitor. Al poner a valor *False* el *flag* de activación del método (clicando de nuevo en el QCheckBox del diálogo) la aplicación volverá a su estado original mostrando la imagen de la cámara sin procesar.

4.3.3.1 Descripción del método

Al activar en el QDialog estas funciones, es lanzado el procedimiento *calcularBrightContrast()* definido en la clase FuncionesOpenCV.

El ajuste de brillo y contraste es una utilidad disponible en la mayoría de aparatos de imagen digital, la cual permite ajustar la calidad de la imagen en función de dos parámetros que pueden aplicarse a cada uno de los píxeles. Estos parámetros son el nivel de luminosidad (brillo) y la diferencia de intensidades entre píxeles (contraste).

Estos dos parámetros funcionan muy bien en consonancia, ya que la alteración del brillo en la imagen permite obtener diferenciaciones de intensidad en los píxeles de la imagen que hace que si estas variaciones son amplias, y el factor de contraste es superior a un determinado umbral, los objetos en la imagen puedan mostrarse más definidos.

La aplicación de estas modificaciones a los píxeles de la imagen resulta en un cálculo fácil y sencillo, que consiste en operar cada píxel con un valor de ganancia (*gain*), que representa el factor de luminosidad de la imagen (brillo), y un valor discriminante (*bias*), que marca la diferenciación de la luminosidad entre los píxeles (contraste). Estos dos valores, también denotados como parámetro *alpha* (α) y parámetro *beta* (β) respectivamente, serán los utilizados para realizar el ajuste del brillo y contraste de la imagen en nuestra implementación. La operación a realizar en cada uno de los píxeles de la imagen *Mat* que se recibe como entrada en la función de cálculo de brillo y contraste es la siguiente:

$$g(i,j) = \alpha \cdot f(i,j) + \beta$$

Donde la función f (i, j) es el valor de cada color del píxel. Si el formato es RGB por ejemplo, la función deberá ser aplicada tres veces por cada píxel, una por cada color del mismo.

Los valores del parámetro α estarán definidos en el rango [1.0 – 3.0]. En cambio, el parámetro β podrá tener un valor entero definido entre 0 y 100.

4.3.3.2 Diálogo *BrightContrastDialog*

Tal y como acabamos de ver, el proceso de cálculo sobre la imagen para modificar el brillo y el contraste es muy sencillo. Para alterar cualquiera de estas dos magnitudes en nuestra aplicación, simplemente tendremos que modificar los valores de corrección del diálogo de la figura 26.

Así, otorgando diferentes valores a los parámetros α y β en el diálogo, podemos obtener resultados como los que se muestran en las siguientes capturas:

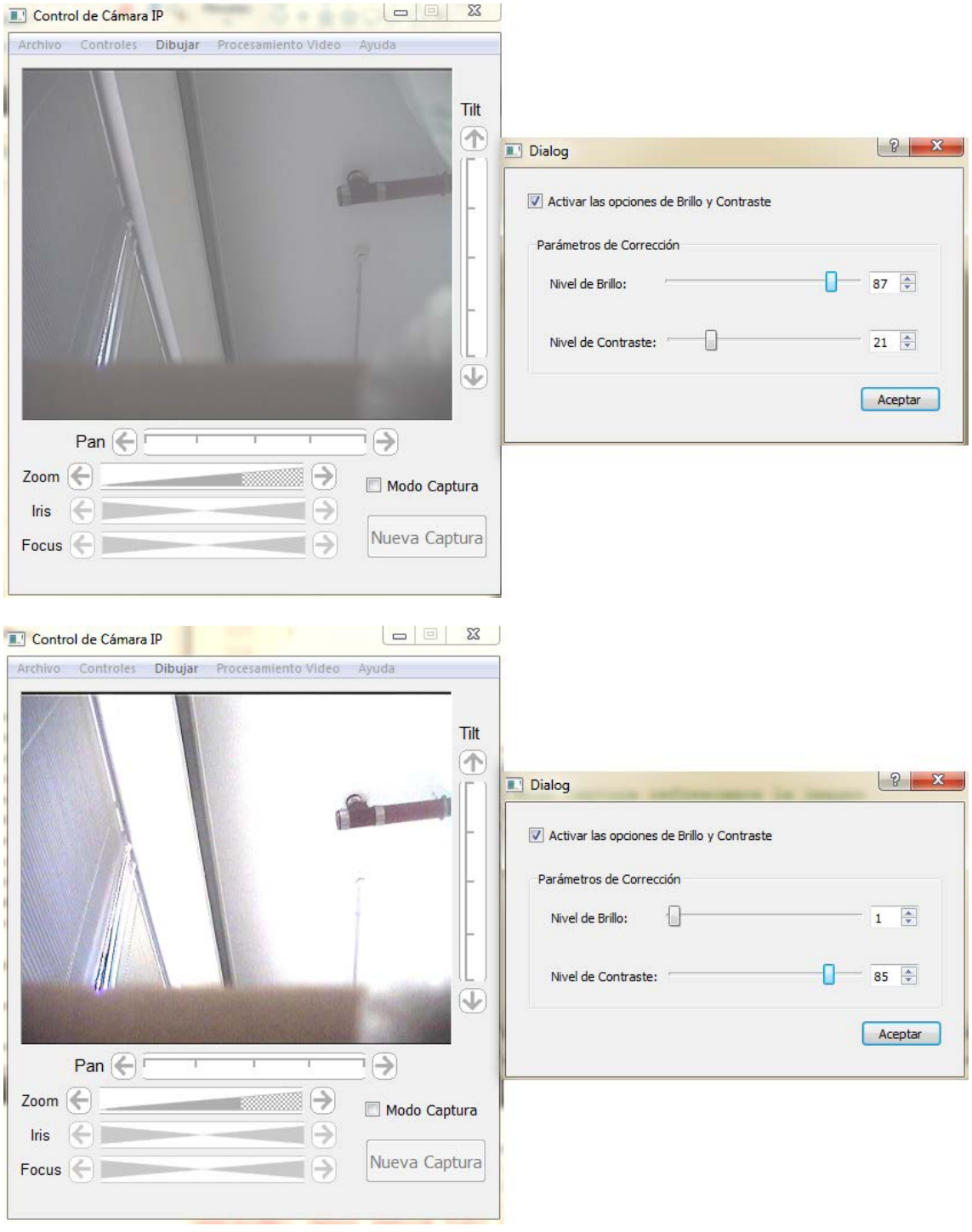

Figura 37. Operaciones de Brillo y Contraste sobre la imagen con diferentes niveles de corrección

4.3.4 Implementación del algoritmo de sustracción de fondo

La sustracción de fondo consiste básicamente en la obtención de una máscara que contenga los píxeles de los objetos móviles en la imagen. Este método es interesante en

aplicaciones de videovigilancia para la detección de personas u objetos en movimiento, ya que con una regulación correcta de sus parámetros, es capaz de sustraer toda la información de la imagen en la que no haya habido variación entre fotograma y fotograma. De lo anterior se deduce que para este método será necesario tener en cuenta no solo la imagen actual, sino la imagen recibida en la petición anterior, con el objeto de obtener las diferencias entre ambas.

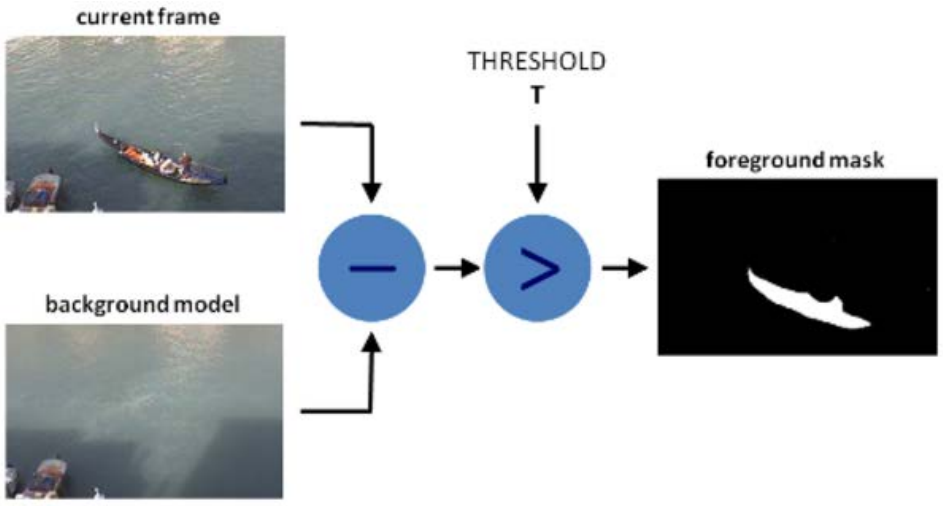

Figura 38. Esquema que refleja la idea del método de sustracción de fondo

El algoritmo de sustracción de fondo es activado a través de la acción “Sustracción de Fondo” del menú *Procesamiento de Video*. De igual forma que en los métodos ya explicados, esta acción lanza un diálogo a través del Slot *mostrarSustraccionFondo()*, el cual se muestra a continuación:

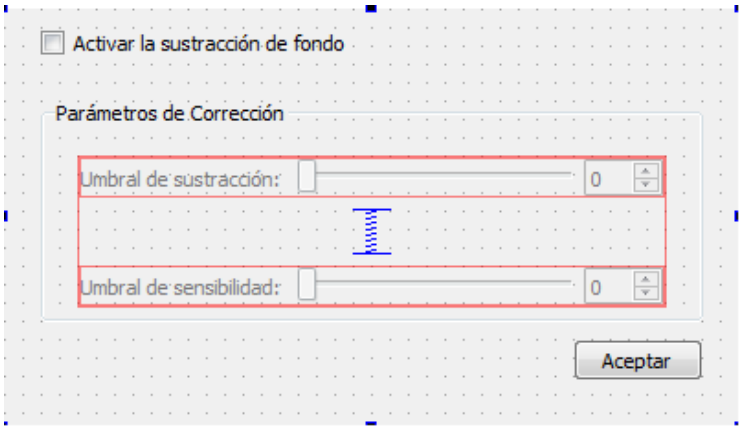

Figura 39. Dialogo de control para el algoritmo de sustracción de fondo

Al clicar sobre el elemento QCheckbox se lanzará el Slot *activarSustraccionFondo()* de la clase BackgroudSubDialog, que realizará las mismas operaciones que los Slots de los diálogos del resto de métodos. El valor de la variable *sustraccionFondoActivo* se pondrá a *True* para comenzar a ejecutar el algoritmo.

Aunque existen funciones del módulo de análisis de video de OpenCV (el cual no será tratado aquí) para realizar la sustracción de fondo, tales como *createBackgroundSubstractorMOG2()*, hemos preferido realizar la implementación basándonos en la idea que subyace en la definición del algoritmo, obteniendo de esta manera un método básico de comparación entre imágenes, que permite sustraer aquellos píxeles que se han mantenido con el mismo valor.

4.3.4.1 Descripción del método

Para la implementación del algoritmo de sustracción de fondo se ha utilizado la estructura *IplImage* de C, teniendo en mente la posibilidad de portar esta implementación en otras plataformas que no soporten C++ o versiones de OpenCV más recientes. El mecanismo utilizado consiste en clonar la imagen *Mat* original convertida a escala de grises (función *cvtColor()* con el modificador `CV_BGR2GRAY`), en una estructura *IplImage* que contenga la referencia a los datos de la imagen (puntero *imageData*).

Con la primera imagen recibida, realizamos una llamada al procedimiento *algoritmoSustraccionFondo*() definido en la clase FuncionesOpenCV, donde la estructura *IplImage* de entrada es copiada en la variable global *imagenAnterior*, que es otra imagen *IplImage* definida en la clase principal, y que será la encargada de ir guardando las imágenes sucesivamente anteriores a las imágenes que se van recibiendo. De esta forma, en la segunda imagen recibida, se clonarán dos estructuras *IplImage* a partir de la imagen actual y de la imagen anterior, almacenada previamente.

Acto seguido, la imagen original es recorrida píxel a píxel, realizándose para cada dato correspondiente al valor de gris contenido en el píxel (número de canales igual a uno) la siguiente operación:

```
Si:          |DatoOrig[índice] – DatoAnt[índice]| >= sensibilidad
Y Si:        |DatoOrig[índice]| >= umbral
Entonces:    DatoSalida[índice] = 255
EnOtroCaso:  DatoSalida[índice] = 0
```

Es decir, si la diferencia entre el valor del píxel de la imagen original y el de la imagen anterior es mayor que un determinado valor de sensibilidad (*sensitivity*), y si además el valor de gris es mayor que un determinado umbral (*threshold*), eso quiere decir que ese píxel ha cambiado entre la imagen anterior y la siguiente, y que por tanto hay que marcarlo en la máscara de color blanco (255). En caso contrario, el píxel se representará en la máscara de color negro (0), ya que no ha sufrido variación alguna.

Una vez aplicada la operación anterior a todos los píxeles de la imagen, se debe de almacenar la imagen actual en la estructura global *imagenAnterior*, de forma que la próxima vez que se ejecute el método (con la siguiente imagen recibida) la imagen actual sea la imagen anterior, y así sucesivamente.

4.3.4.2 Diálogo *BackgroundSubDialog*

Tal y como acabamos de ver, en el algoritmo de sustracción de fondo existen dos parámetros que permiten alterar la capacidad del método de detectar cambios en los píxeles entre dos imágenes recibidas de forma consecutiva.

Por un lado, se puede ajustar un umbral o *threshold* que determinará el nivel de tono de gris del píxel de la imagen original a partir del cual se aplique la diferenciación. Por otro lado, esta diferenciación entre píxeles es medida a través del parámetro de sensibilidad.

Como su propio nombre indica, el parámetro de sensibilidad nos permite establecer a partir de qué nivel de sensibilidad en la diferenciación entre valores de los píxeles entre dos imágenes consecutivas podemos tomar como referencia, para mostrar los cambios que resultan en las imágenes a lo largo del tiempo. De este modo, estableciendo un nivel de sensibilidad elevado, cualquier pequeño en la imagen aparece reflejado en la máscara de forma mucho más evidente que modificando este valor a un nivel más reducido.

En las siguientes capturas podemos apreciar los resultados producidos estableciendo diferentes niveles de sensibilidad y de *threshold* en la imagen, utilizando para ello el los componentes del diálogo BackgroundSubDialog.

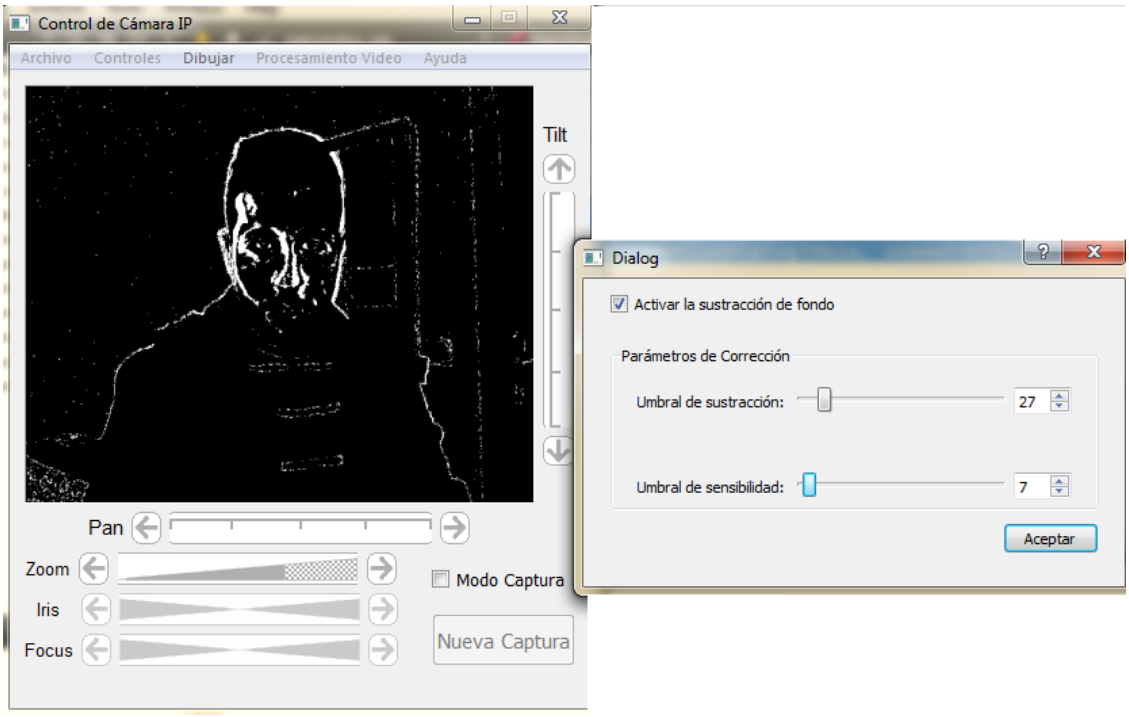

Figura 40. Algoritmo de sustracción de fondo con valor de umbral 27 y sensibilidad 7

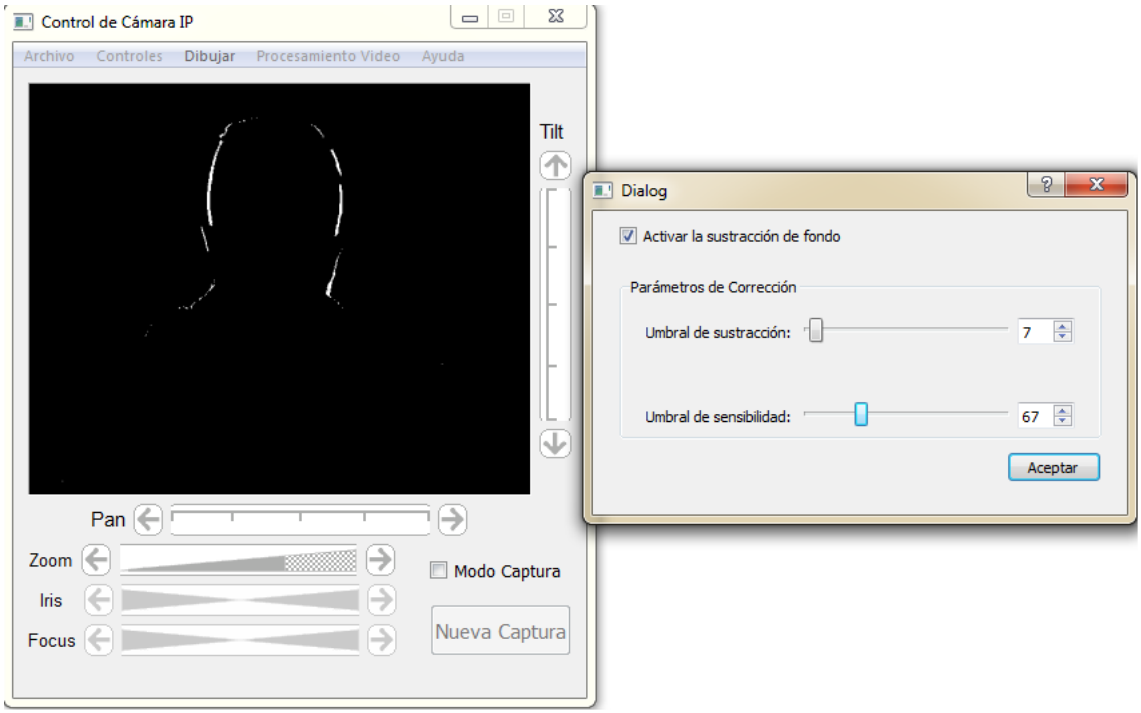

Figura 41. Algoritmo de sustracción de fondo con valor de umbral 7 y sensibilidad 67

4.3.5 Implementación del algoritmo de detección de BLOBs de color

Otra opción que se ha querido incluir entre las nuevas funcionalidades de procesamiento de imagen en la aplicación *Control Camera IP*, es un algoritmo de detección de BLOBs de color.

Un BLOB (*Binary Large OBject*) es un grupo de píxeles conectados entre sí en una imagen y que se caracterizan por tener una cierta propiedad (color, luminosidad, etc.) que es común a cada uno de ellos, o dicho de otra manera, los BLOBs engloban áreas de la imagen que difieren en las propiedades de sus píxeles. Se emplea la palabra *Large* para indicar que para definir este tipo de conjuntos, usualmente se toma en más consideración las regiones más "grandes" y no las más pequeñas, ya que éstas son consideradas normalmente como ruido.

Los métodos de detección de BLOBs reciben como entrada una imagen, y son capaces de definir una región de píxeles de forma que permitan clasificar ciertas propiedades de la imagen, lo cual puede aportar información valiosa de cara a la detección de formas u objetos, reconocimiento de patrones, etc.

Uno de los métodos más básicos y utilizados en muchas aplicaciones, es el de detección de BLOBs de color. Consiste esencialmente en determinar zonas en la imagen en donde los píxeles tienen un determinado color, o bien, que los colores de los píxeles de una región concreta sean lo más parecidos posibles entre ellos. En principio, la tarea se plantea sencilla (no más compleja que la del algoritmo de sustracción de fondo visto en el apartado anterior). Sin embargo, existe una dificultad derivada de la forma en que están definidos los rangos en los espacios de color en la imagen digital, sobre todo si utilizamos, como viene a ser muy común, el espacio de color RGB.

El principal problema de este espacio de color es que la representación de los colores como valores decimales en el rango [0-255] para cada una de las componentes *Rojo*, *Verde* y *Azul*, no está sujeta a limitadores que establezcan gamas o rangos de colores concretos. Es decir, si por ejemplo tenemos que el color azul viene representado por la tupla [0,0,255], donde la componente de *Azul* tiene el máximo valor, y el resto de componentes el valor cero, también es posible encontrar otros valores de azul por ejemplo en el rango determinado por las tuplas [93,138,168] , [161,202,241], o [59,68,75].

De esto podemos extraer que, aunque podamos suponer que para que un color sea azul la componente *Verde* tenga que ser mayor que la *Roja*, y que la componente *Azul* tenga que ser mayor que todas las demás, esto no nos garantiza que todos los colores dentro de esos límites vayan a ser de una tonalidad azul, o lo contrario, que cualquier tupla fuera de esos límites pueda representar a un color que podamos clasificar como azul. Por ejemplo, la tupla [127, 255, 212] representa el color *AquaMarine*, cuya tonalidad aturquesada podría estar clasificada dentro de lo que cualquier persona podría identificar como un color azul.

Figura 42. Degradación del color *AquaMarine* (RGB[127,255,212])

Es por esta dificultad a la hora de clasificar los colores en el espacio RGB por lo que hemos decidido implementar nuestro método de detección haciendo uso del espacio de color HSV, donde como ya vimos al comienzo de este capítulo, los colores de este espacio se descomponen en el matiz (*Hue*), la saturación (*Saturation*) y una componente de valor (*Value*). Pero en realidad, la única componente de las tres que nos interesa y que necesitaremos para implementar nuestro algoritmo es la componente *Hue* de matización.

La componente de matiz del espacio HSV establece un rango de valores en los cuales se puede representar todas las tonalidades de colores, sin tener en cuenta las degradaciones correspondientes a cada una de ellas. A diferencia de RGB, la componente *Hue* representa las tonalidades de color en una rueda o círculo dividido por cada una de estas tonalidades, y donde con un valor en el rango de [0 – 360] se puede delimitar ciertas tonalidades según estén dispuestas en la rueda de color de forma más sencilla que en el espacio RGB.

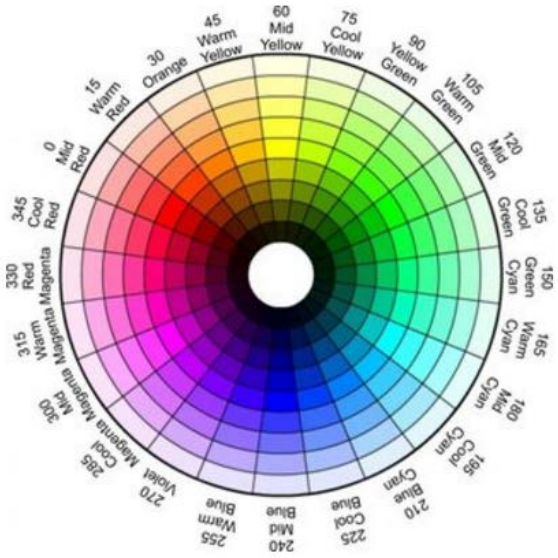

Figura 43. Rueda de color de la componente de matiz *Hue*

El algoritmo de detección de BLOBs de color que se ha implementado en este trabajo es activado a través de la acción "Blobs Color" del menú *Procesamiento de Video*. Esta acción lanza un diálogo a través del Slot *mostrarDetBlobsColor()*.

Haciendo clic sobre el elemento QCheckbox se lanza el Slot *activarDetBlobsColor()* de la clase BlobsColorDialog, que realizará las mismas operaciones que los Slots de los diálogos del resto de métodos. El valor de la variable *detBlobsColorActivo* se pondrá a *True* para comenzar a ejecutar el algoritmo.

A continuación, haremos una breve descripción del algoritmo de detección de BLOBs de color que se ha implementado para este trabajo

4.3.5.1 Descripción del método

Al igual que con el método de sustracción de fondo del que hablamos en los apartados anteriores, para la implementación del algoritmo de detección de BLOBs de color en la imagen, hemos utilizado la estructura *IplImage* para dotar de mayor flexibilidad de cara al futuro, para los desarrollos que se quieran ampliar de cara a implementarlos en otras plataformas que no soporten el lenguaje C++, o que la versión de OpenCV en esos sistemas no sea lo suficientemente reciente.

Del mismo modo que hacíamos en el método anterior, clonaremos la imagen *Mat* original convertida a escala de grises, de forma que una nueva estructura *IplImage* referencie los datos de imagen a través del campo *imageData*. Una vez realizamos una llamada al procedimiento *algoritmoBlobsColor*() definido en la clase

FuncionesOpenCV, el primer paso consistirá en transformar el espacio de color BGR de OpenCV en el espacio HSV de la siguiente forma:

```
cvtColor(in_image,in_image, CV_BGR2HSV);
```

A continuación, recorremos el campo *imageData* de la estructura *IplImage* para cada uno de los píxeles, estableciendo la componente *Hue* de cada uno de los píxeles en el índice de la estructura de la siguiente manera:

```
índice = fila + (columna * num_componentes) ,
```

donde `num_componentes` es igual a 3.

Luego el valor de *índice* marcará siempre la posición de la estructura *imageData* que contiene el valor del componente *Hue* o matiz del píxel actual. Así pues, para obtener los límites de los rangos de color simplemente nos basta con consultar la rueda de color *Hue* y limitar los valores de los píxeles a algunos de las tonalidades.

No obstante, en OpenCV, la representación del componente *Hue* se realiza en el rango [0 – 128] en vez del rango [0 – 360]. Al ser los valores proporcionales, simplemente habrá que realizar una sencilla regla de tres para ir obteniendo los valores que limiten las tonalidades de los BLOBs que queramos detectar. Para nuestro caso, hemos definido una serie de rangos para las tonalidades *Rojo, Verde* y *Azul*, dentro de una estructura llamada *LIMITES_BLOBS*, que contiene los campos *rl, rh, g y b*. Los campos *rl* y *rh* representan los límites de *Rojo*, que están partidos en el rango [0 - 10] (*rl*) y en el rango [160 – 179] (*rh*), El campo *g* representa el límite de *Verde* y los valores para estas tonalidades están definidas en el rango [45 – 80]. Finalmente, el límite de *Azul* (*b*) se encuentra en el rango [90 – 145].

La selección de color vendrá dada por el usuario a través del diálogo BlobsColorDialog. Así, en función del color seleccionado, se verifica si cada píxel está dentro del rango especificado para ese color en la rueda de tonalidades *Hue*, y se atiende al siguiente criterio:

```
Si:          Dato[índice] >= límite_inferior
Y Si:        Dato[índice] <= límite_superior
Entonces:    DatoSalida[índice] = 255
EnOtroCaso:  DatoSalida[índice] = 0
```

La máscara resultante visualizará en color blanco (255) el conjunto de píxeles (BLOBs) que se correspondan con el color especificado, y en negro (0) el resto de píxeles que no entren dentro de dicha clasificación.

Finalmente, a la salida del método, y antes de convertir la imagen de nuevo de *Mat* a QPixmap, realizamos un filtro gaussiano utilizando la función *GaussianBlur*, con un tamaño de *Kernel* de valor 9, para suavizar la imagen de salida debido sobre todo a la existencia de ruido que puede haber en la imagen como consecuencia de las condiciones de luminosidad del entorno.

4.3.5.2 Diálogo *BlobColorDialog*

En el diálogo de detección de BLOBs de color se permitirá seleccionar el color a detectar (*rojo, azul y verde*) utilizando el elemento QComboBox incluido en el mismo.

Estos valores se almacenarán en una variable global llamada *colorBlob* que contendrá una cadena con el tipo de color existente actualmente en el ComboBox. Una vez en el procedimiento de detección, los píxeles serán filtrados en un rango de color determinado por el valor de esta variable.

A continuación podemos apreciar algunas capturas del algoritmo en funcionamiento:

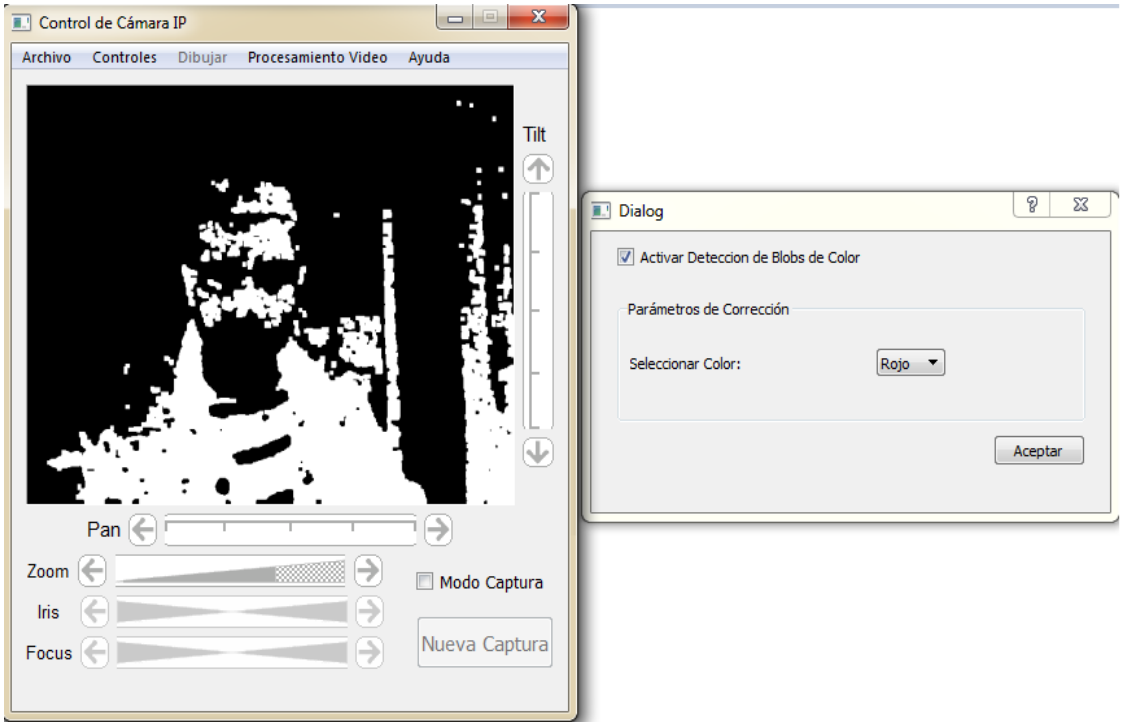

Figura 44. Captura del algoritmo de BLOBs de color seleccionando la opción de tonos *rojos*

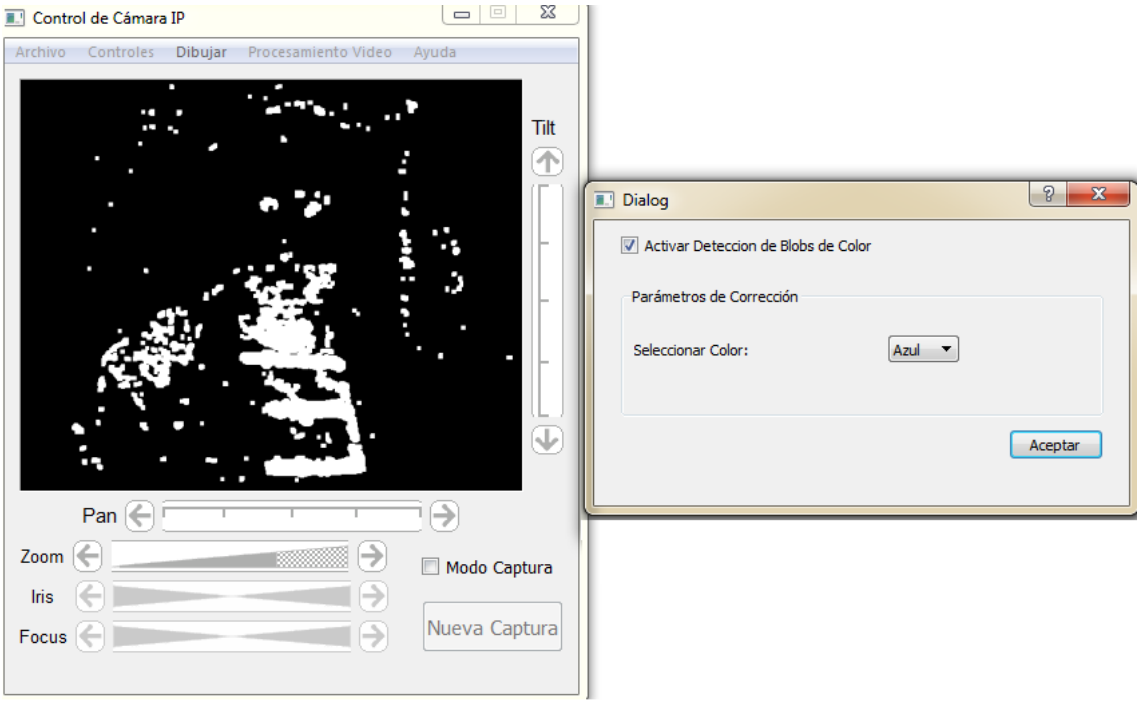

Figura 45. Captura del algoritmo de BLOBs de color seleccionando la opción de tonos *azules*

4.3.6 Implementación del método de detección de caras

Finalmente, en este trabajo hemos incluido un método de detección de caras utilizando para ello el clasificador en cascada basado en características tipo *Haar* que ya explicamos al comienzo de este capítulo.

Para lanzar la ejecución del método haremos clic en la última acción del menú *Procesamiento de Video* donde inicialmente aparece con el mensaje "Activar Detección de Caras".

A diferencia del resto del métodos, en este caso no existe un diálogo de parámetros de corrección ya que, si bien podríamos modificar algunos parámetros de la función *detectMultiScale()*, tales como el factor de escalado o el número mínimo de vecinos, hemos preferido establecer unos valores fijos con los que se obtiene una buena tasa de detección en relación y una minimización de falsos positivos.

La creación de un diálogo que incremente por ejemplo el número de vecinos u otras técnicas que permitan reducir aún más la tasa de falsos positivos, queda pendiente para una posible ampliación de cara al futuro. Aunque también se plantea en este caso algún otro tipo de enfoque que permita no solo la detección, sino también el reconocimiento de patrones más complejos, o de detección de características más específicas en las

personas o en los objetos. En el capítulo dedicado a los trabajos futuros ahondaremos un poco más en esta cuestión.

El método por tanto, se inicia en la aplicación directamente activando la acción del menú antes comentada. En ese momento, el Slot *activarDeteccionCaras()* es lanzado y cambia en primer lugar el texto de la acción del menú al mensaje "Desactivar Detección Caras". Con este mecanismo, cuando se lance de nuevo la acción, se detectará que el método está activo, y por lo tanto volverá de nuevo a modificarse la acción con el texto original.

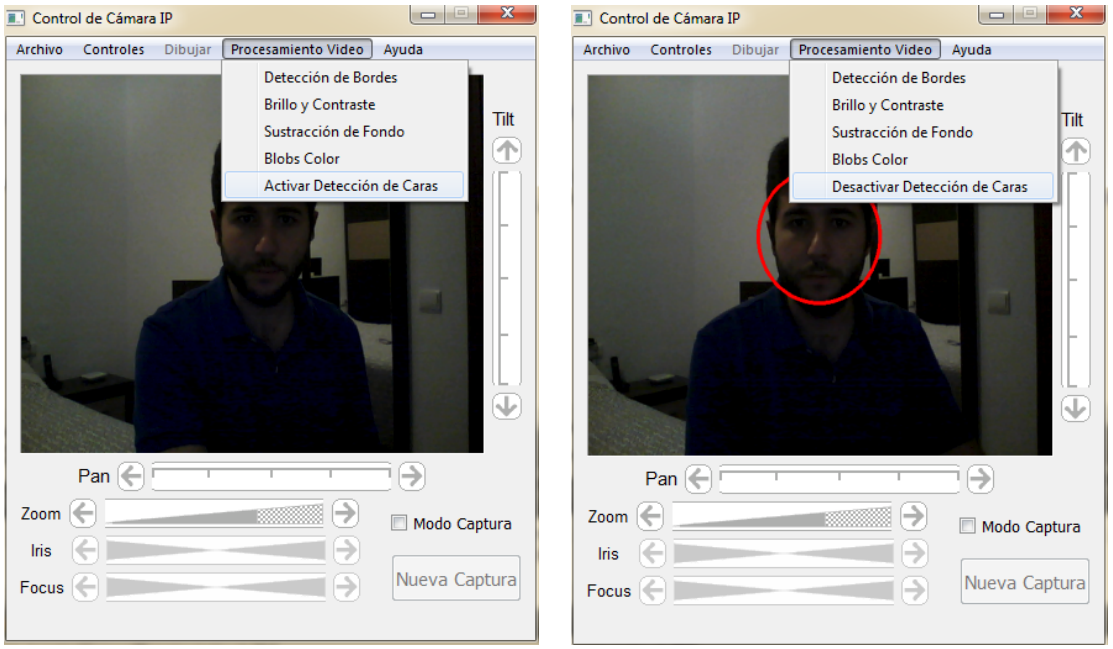

Figura 46. Activando y desactivando el método de detección de caras

4.3.6.1 Descripción del método

Previamente al lanzamiento del método *detecciónCaras()*, el cual está definido en la clase FuncionesOpenCV del paquete *auxiliares*, en el momento que activamos la detección de caras desde la acción de menú correspondiente, la clase principal (*MainWindow*), ya tiene inicializada una nueva variable global de tipo CascadeClassifier llamada *faceCade*, definida en el mismo momento en que se arranca la aplicación por primera vez.

En esta inicialización se carga a través de la función *load()* un nuevo clasificador en cascada basado en características *Haar* a partir del fichero *lbpcascade_frontalface.xml*, que ya dispone de la información de un clasificador en cascada entrenado para la

detección de características *Haar* asociadas a las regiones en imágenes identificadas como “caras de personas”.

Una vez cargado este clasificador, y la aplicación es iniciada, podemos entonces activar la detección de caras, permitiendo por tanto que se lance el método *deteccionCaras()* de la clase FuncionesOpenCV. Lo primero que hace este método es realizar una ecualización del histograma asociado a la imagen recibida desde la cámara en escala de grises.

Esta operación, que como ya vimos se puede invocar haciendo uso de la función *equalizeHist()* del módulo *imgProc* de OpenCV, es importante de cara a evitar falsos positivos en la detección de caras debido ante todo a que la maximización que realiza sobre el contraste y a la normalización del brillo de la imagen en todos sus píxeles, hace que las características *Haar*, que son calculadas en base a la diferencia entre las sumas de intensidades de los píxeles de cada uno de sus regiones adyacentes, puedan ser más fácilmente detectadas y comparadas en las regiones de la imagen, por lo que la tasa de detección será mucho más eficiente.

Previamente por tanto, habrá que convertir la imagen de entrada en escala de grises con la función *cvtColor()* de OpenCV. Tras ecualizar el histograma, la imagen resultante es pasada a la función *detectMultiScale()* con los siguientes valores en sus argumentos:

```
<Arg1>: grayScale               -> Imagen escala de grises ecualizada
<Arg2>: faces                   -> Vector de ROIs con las detecciones
<Arg3>: 1.1                     -> Valor de factor de escalado
<Arg4>: 3                       -> Número de vecinos
<Arg5>: CV_HAAR_SCALE_IMAGE     -> Flag de escalado tipo HAAR
<Arg4>: Size(30,30)             -> Tamaño mínimo
```

Los dos últimos argumentos hacen referencia por un lado a la heurística empleada para reducir el número de áreas analizadas, donde en este caso se utilizará una optimizada para la detección de características *Haar* (*CV_HAAR_SCALE_IMAGE*), aunque también se pueden utilizar otras funciones heurísticas tales como por ejemplo una poda Canny (*CV_HAAR_DO_CANNY_PRUNING*), basada en el algoritmo del mismo nombre.

Por otro lado, el último argumento hace referencia al tamaño mínimo de objeto a analizar, por lo que nos abstendremos de analizar caras que en la imagen contengan menos de 900 píxeles cuadrados de área.

El segundo argumento (*faces*) es pasado como parámetro por referencia y en él, la función *detectMultiScale()* devuelve un vector de rectángulos (tipo *Rect* de OpenCV) con las regiones de interés (ROIs) de la imagen donde han existido coincidencias en la detección, es decir, devuelve las zonas de la imagen donde se han detectado "caras".

Para representar las detecciones sobre la imagen de salida, buscamos para ello el centro de cada uno de los rectángulos de la variable *faces* con la función *center()* y pasamos el mismo a la función *ellipse()* de OpenCV, la cual permitirá dibujar una forma circular con ese centro encerrada dentro del rectángulo.

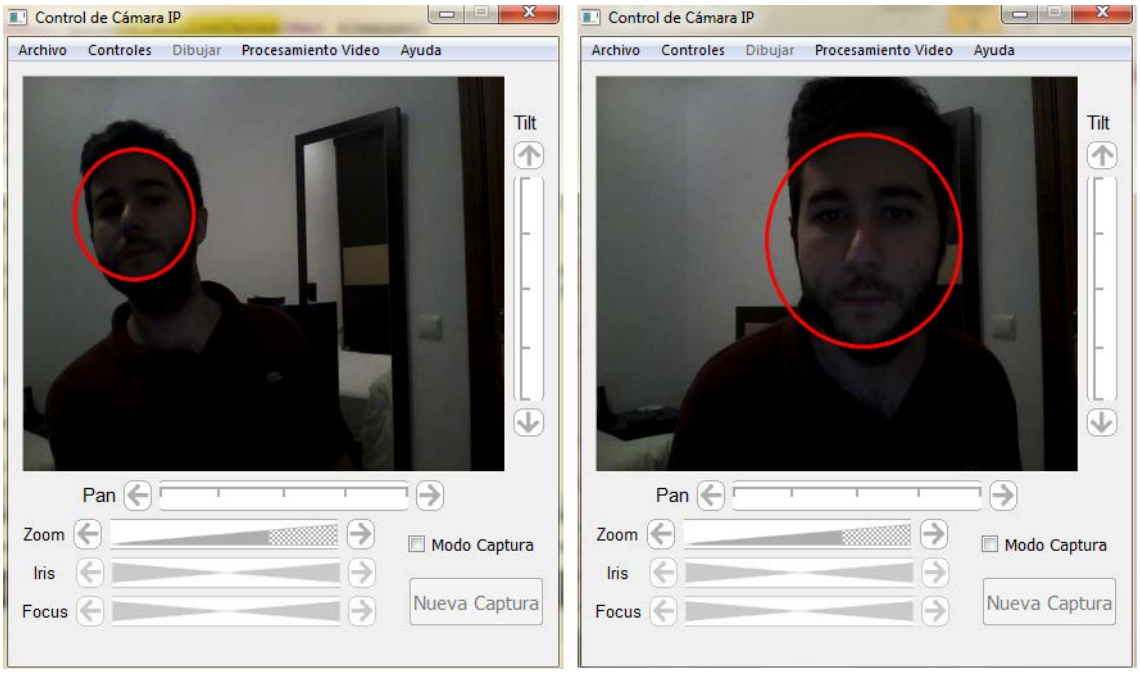

Figura 47. Detección de caras en la aplicación *Control Camera IP*

Las funciones del módulo de dibujo de OpenCV no han sido explicadas en este trabajo, ya que la única que utilizaremos será la función *ellipse()* solo en este caso. No obstante, en el capítulo dedicado a los trabajos futuros, propondremos la utilización de algunas de estas funciones para mejorar la detección de los métodos desarrollados en este trabajo.

Capítulo 5

Conclusiones y Trabajos Futuros

Con la elaboración de este Trabajo Fin de Grado hemos conseguido una mejora sustancial en la funcionalidad de la aplicación *Control Cámara IP*, introduciendo un conjunto de opciones en la interfaz gráfica que permiten aplicar algunas funciones de procesamiento sobre la imagen, incluyendo la detección de bordes y caras, así como la detección de objetos en movimiento gracias a un algoritmo de sustracción de fondo.

La facilidad de integración de la biblioteca OpenCV en las distintas plataformas definidas en el proyecto Qt, ha permitido que estas nuevas funciones puedan ser utilizadas en dispositivos móviles con sistemas Android entre otros, produciendo unos resultados bastante satisfactorios. En este sentido, también ha sido de gran ayuda la facilidad de configuración de plataformas Android que ofrece ahora Qt en su nueva versión 5, liberándonos de la necesidad de utilizar versiones de Qt no oficiales, tales como *Qt Necessitas for Android*.

Por último, también hemos conseguido en este trabajo encontrar una API estandarizada para la conexión de nuestra aplicación con cámaras de bajo coste de diversos fabricantes (Foscam, Conceptronic, HooToo, etc.), y se ha adaptado parte del código de la aplicación original, para poder utilizar algunas de las funciones básicas de las que este API dispone, sobre una cámara Conceptronic modelo *Tilt & Pan Wireless*, logrando por tanto incrementar en gran medida la capacidad de interconexión de nuestra aplicación con una gran variedad de cámaras existentes en el mercado.

Gracias a los resultados obtenidos, podemos vislumbrar muchas posibilidades de cara a la mejora de la aplicación, como por ejemplo, la creación de sistemas de detección y reconocimiento de objetos más sofisticados, o de una solución integrada que permita utilizar las diferentes funciones de la aplicación en conjunción con sistemas empotrados en vehículos no tripulados, tales como drones o vehículos submarinos (*UUV – Unmanned Underwater Vehicles*), o en otro tipo de sistemas inteligentes de visión.

A continuación, describiremos algunas de estas posibles mejoras que podrán ser aplicadas en un futuro para la ampliación de este Trabajo Fin de Grado.

5.1 Mejoras en la detección de objetos en la imagen

Los algoritmos de detección desarrollados en este trabajo haciendo uso de la biblioteca OpenCV, están realizados en base a procedimientos muy sencillos de tratamiento sobre la imagen, ya que la información que se obtiene de ellos solo implica la operación sobre una única característica de la imagen, como el color o la luminosidad de los píxeles (a excepción del método de detección de caras, que utiliza un conjunto de características *Haar* para buscar coincidencias en las zonas de la imagen).

Sin embargo, en muchas ocasiones, es necesario para la detección de objetos de ciertas características determinadas, extraer otros elementos de información sobre la imagen, de forma que nos ofrezca la posibilidad de conjugar la combinación de todos los elementos para obtener una información más precisa sobre el objeto.

Para no extendernos demasiado, propondremos por un lado la utilización del módulo *Features2D* de OpenCV para la detección de esquinas y de puntos característicos (*key* points) sobre la imagen, y por otro la inclusión de otros métodos de detección de BLOBs basados en la circularidad.

5.1.1 Detección de esquinas

La idea de detección de características sobre la imagen (*features detection*), consiste básicamente en buscar coincidencias sobre *frames* extraídos del entorno, buscando zonas que son fácilmente identificables debido a que contienen características muy reconocibles, tales como esquinas, colores, luminosidad, etc.

Las esquinas en una imagen son elementos que pueden aportar mucha información, ya que la mayoría de los objetos del mundo real están constituidos por zonas que, en una imagen digital sus pixeles experimentan un cambio brusco de gradiente (véase detección de bordes con el algoritmo *Canny*), sobre todo cuando dos de sus bordes confluyen en una esquina.

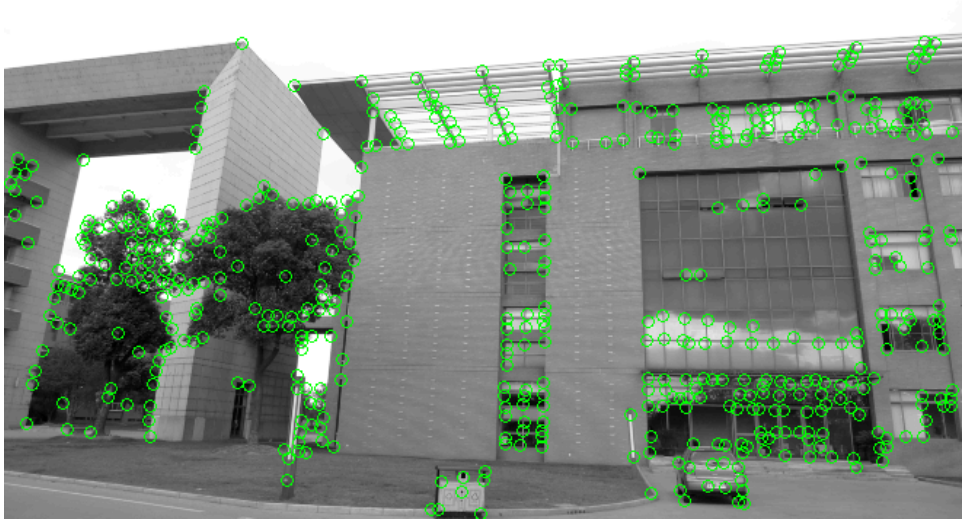

Figura 48. Aplicación de un algoritmo de detección de bordes sobre una imagen

La detección de esquinas en imágenes suele ser utilizada por algunos métodos de detección de puntos característicos o puntos de interés (*key points*) sobre la imagen, normalmente invariables en caso de escalado de la imagen.

5.1.2 Blobs de circularidad

Del mismo modo que se pueden detectar BLOBs de color en la imagen para detectar objetos que tienen un determinado color, también se pueden crear algoritmos que detecten otro tipo de BLOBs o conexiones de píxeles en las imágenes. Entre ellos podemos destacar los BLOBs de circularidad.

La detección de circularidad consiste básicamente en saber cómo de "circular" es una región de una imagen, de forma que esta información pueda ser representada en la imagen de salida respondiendo a un cierto umbral, que usualmente suele estar determinado por la longitud de la región obtenida. La longitud de un BLOB es la distancia entre los dos píxeles del mismo más distantes entre sí.

Así, la circularidad se define como el resultado de dividir el valor del área del BLOB, entre el área del círculo de diámetro la longitud del BLOB, que está definido dentro del mismo.

```
Circularidad = (4 * AreaRegion) / (Pi * (longitud)^2)
```

El resultado anterior, también llamado "factor de circularidad" puede ser aplicado en algún método similar al realizado en este trabajo para la detección de BLOBs de color, de forma que pueda representar en una imagen aquellas zonas o figuras que tengan una cierta circularidad.

5.2 Mejoras en la detección de caras

Se propone también en este capítulo la creación de un diálogo para la aplicación capaz de mejorar la detección de caras, modificando para ello los parámetros de corrección de la función *detectMultiScale()*, tales como el número de vecinos, o el heurístico de búsqueda de características *Haar*.

Además también se sugiere el desarrollo o utilización de algún método existente de reconocimiento de caras, como por ejemplo la clase de reconocimiento de caras FaceRecognizer de OpenCV, incorporada a partir de su versión 2.4, y que dispone de una serie de funciones de detección basadas en *eigenfaces* y *fisherfaces*, utilizando fórmulas de transformación de vectores invariantes en transformaciones lineales (vectores *eigen*), o variaciones en la reflexión de la luz, para detectar rasgos característicos en las expresiones faciales.

Existe un tutorial de OpenCV que puede ayudar a entender cómo funciona el reconocedor de caras, y poder utilizarlo así en un futuro en nuestra aplicación en conjunción con el método de detección ya implementado.

5.3 Integrar la aplicación en una solución para el análisis de información visual con cámaras instaladas en Drones

El auge que cada vez más hoy día está habiendo en la utilización de vehículos aéreos no tripulados (drones), para la realización de tareas específicas relacionadas con la visión u otras áreas, en lugares donde el acceso a través de otros medios es complicado, hace que nos planteemos la posibilidad de integrar nuestra aplicación en un futuro, en algún tipo de solución que permita el análisis de la información visual capturada por este tipo de vehículos en movimiento.

Figura 49. Dron con varios sensores entre los que se incluyen varias cámaras

En tal caso, podrían aplicarse todas las mejoras propuestas en los apartados anteriores, de forma que se pudiera extraer información variada del entorno donde el dron esté operando en ese momento, analizar dicha información, e incluso implementar mecanismos de control remotos que permitieran a la propia aplicación alertar al dron sobre elementos del entorno como obstáculos fijos, con el objetivo de poder evitarlos.

5.1.2 Técnicas de visión artificial para la navegación y análisis de información con drones

Aprovechando la inclusión en este Trabajo Fin de Grado de capacidad de análisis y procesamiento de video a través de la biblioteca OpenCV, se podrían también añadir algunas técnicas de visión por computador para recoger la información visual de una o varias cámaras con conectividad IP instaladas en un dron, de forma que podamos realizar estimaciones de posicionamiento para la navegación y el análisis para la detección de elementos visuales en el entorno (como obstáculos por ejemplo). Muchas de estas técnicas están basadas en detectores de puntos de interés o *key points*.

Detectores de *key points* y *matching* de características

Los detectores de *key points* (más conocidos como *features detectors*), utilizan técnicas de detección en la variación de intensidad en los píxeles, entre las que se incluyen la detección de esquinas, para obtener un elevado número de puntos de interés en la imagen, que pueden ser comparados con otras imágenes para determinar coincidencias en las mismas. Algunos de estos algoritmos de detección de características más destacados son *FAST, ORB, SIFT y SURF*. Los dos primeros pertenecen a la librería *Features2D* de OpenCV, por lo que son de libre distribución. Sin embargo, los algoritmos *SIFT* y *SURF* están patentados, y por lo tanto su uso está más restringido (están ubicados en el módulo *NonFree* de OpenCV).

Para el *matching* de características suele utilizarse la librería FLANN (*Fast Library for Approximate Nearest Neighbors*), que contiene un conjunto de algoritmos optimizados para la búsqueda de coincidencias en dos imágenes.

Estas coincidencias podrán ayudar a obtener información precisa sobre ciertos entornos en tiempo real, o bien, en el caso de drones con cámaras incorporadas, se podrían aplicar técnicas de predicción entre *frames* consecutivos, pudiendo estimar la posición del dron en el espacio 3D, así como mapear los objetos en el espacio tridimensional (reconstrucción 3D).

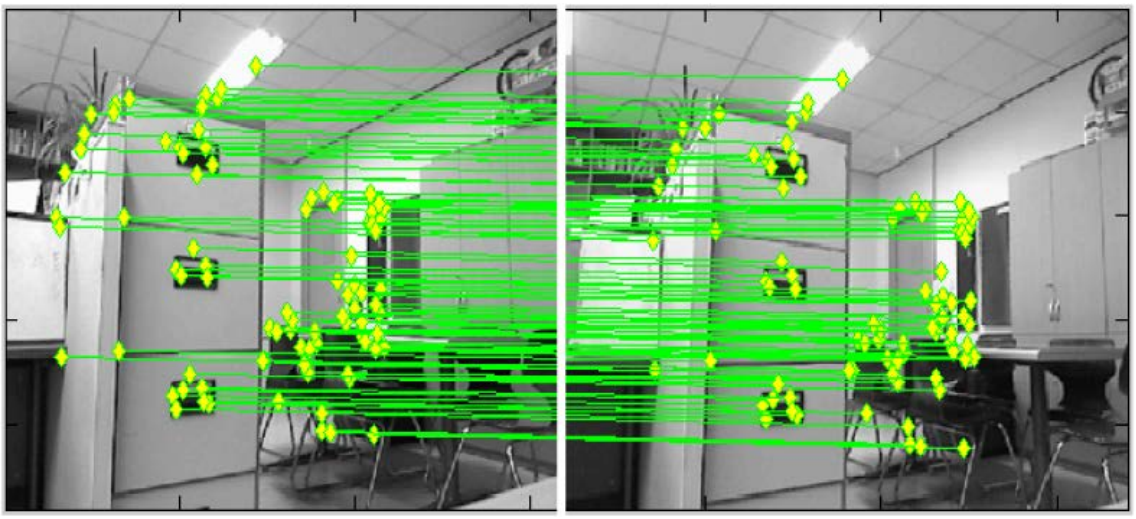

Figura 50. *Matching* entre dos frames consecutivos para establecer la localización

Lo anterior está situado en las bases de un problema bien conocido, denominado SLAM (*Simultaneous Location And Mapping*), que trata de buscar solución a la posibilidad de determinar (o estimar de forma bastante precisa) la localización del robot móvil (en este caso el dron) en el espacio, a la vez que poder realizar un mapeado o reconstrucción del entorno a través de la utilización de diferentes sensores. En el caso de que los sensores sean cámaras, estaremos hablando de V-SLAM (*Visual-SLAM*).

Visual SLAM

Existen muchos enfoques para la solución del V-SLAM, basadas en visión monocular y estéreo, pero todavía no hay soluciones estandarizadas que resuelvan este problema de una manera fiable al cien por cien, sobre todo debido al error acumulativo producido a largo plazo en la estimación de la posición, la cual debe ser corregida por sensores de posicionamiento global (GPS), o sensores inerciales (IMU – *Inertial Meassurement Unit*) de gran precisión.

El SLAM visual está basado en el *matching* de características entre *frames* consecutivos. El mecanismo consiste en estimar la posición basándonos en los cambios de posición de los puntos de interés en ambas imágenes en relación a la distancia focal y a otros parámetros de la cámara. Estos cambios permiten estimar la posición relativa de la cámara (y por tanto la del dron), y permiten gracias a la triangulación, realizar una operación de los puntos de la imagen en un espacio tridimensional (espacio de coordenadas XYZ), donde la coordenada Z determina la profundidad o distancia del punto del objeto en el espacio 3D.

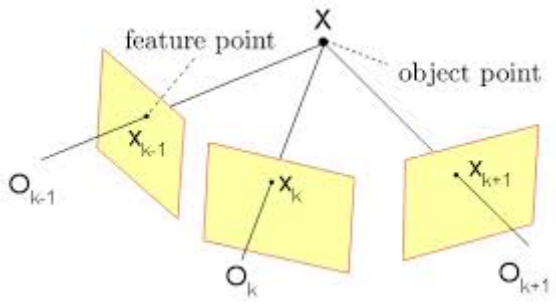

Figura 51. Proyección de un punto en el espacio 3D

El punto en el espacio es calculado a partir de un punto característico en tres *frames* consecutivos

Podemos ver por tanto que utilizando técnicas como V-SLAM, las posibilidades de implementar métodos de navegación y detección de objetos en el espacio 3D para poder ser incorporados en los drones serían muy interesantes de incluir en nuestra aplicación, aunque dado su nivel de complejidad, estas técnicas podrían ser objeto de otro Trabajo Fin de Grado, o si el proyecto es muy ambicioso, de un trabajo de Máster o Tesis Doctoral.

5.4 Adaptación completa de la aplicación para su funcionamiento con APIs estandarizados

Tal y como hemos visto en el capítulo 3, la adaptación del API estandarizado API IPCAM CGI a la aplicación *Control Camera IP* no ha sido completada, debido a que hubiera sido necesario cambiar gran parte de las funciones relacionadas con la carga de controles, así como haber incluido nuevas funcionalidades interesantes tales como la carga de *presets*, o la posibilidad de hacer barridos horizontales y verticales, o modos de ronda de vigilancia.

Es por ello, y porque creemos que esta labor queda fuera de los límites de este trabajo, que se deja esta posible adaptación para un trabajo posterior que pudiera querer realizarse sobre este proyecto.

Concretamente, para la API IPCAM CGI estandarizada de Foscam se podrían incluir las siguientes funciones:

- Funciones de alarma a través de la función *getstatus.cgi*, entre las que se incluyen la alarma de detección de movimiento y la alarma de detección de voz.
- El resto de funciones de control, tales como *Pan* y *Tilt* con grados predefinidos en la barra de navegación. Podrían incluirse por ejemplo, opciones en la

aplicación que permitieran seleccionar valores de grados de barrido para establecer en el *Pan y Tilt*.

- Establecer comandos de la función *decoder_control.cgi* relativos al modo patrulla (*patrol*), estableciendo algunos *presets* de los 32 predefinidos.
- Permitir la creación y configuración de usuarios directamente desde la aplicación a través de la función *set_users.cgi* utilizando también el parámetro *pri* para establecer permisos de usuario a cada uno de ellos.

Dentro del mismo API, y para cámaras que dispongan de opción PTZ, también podemos utilizar la función extra *set_misc.cgi*, que permite establecer algunas opciones específicas para aquellas cámaras de este tipo que utilicen también el API IPCAM CGI de Foscam. Algunos de los parámetros PTZ de esta función CGI son:

- *Ptz_center_onstart*: Inicializa la cámara a una posición central, cada vez que se activa.
- *Ptz_patrol_rounds*: Especifica el número de rondas a realizar por la cámara (el valor cero hace que la cámara haga un número de rondas indeterminado).
- *Ptz_patrol_rate*: Establece la velocidad de la ronda de vigilancia
- *Ptz_preset_onstart*: Inicializa el *preset* especificado para que se ejecute cada vez que la cámara es activada.

Como ampliación final, también sugerimos la posibilidad de cargar esta y otras APIs de forma externa (a modo de plugin), en la aplicación, de forma que se pueda ampliar todavía más la posibilidad de utilizar nuestra aplicación en otro tipo de cámaras que dispongan de APIs propias o estandarizadas.

Bibliografía

[1] D. Molina Alarcón, «Interfaz Gráfica Multiplataforma para Control de Cámaras IP,» [En línea]. Available: http://jabega.uma.es/record=b1912401~S4*spi.

[2] Qt, «Biblioteca de desarrollo Qt,» [En línea]. Available: http://www.qt.io/developers/.

[3] A. d. Canny, «Wikipedia,» [En línea]. Available: https://es.wikipedia.org/wiki/Algoritmo_de_Canny.

[4] IProNet, «Video Management Systems,» [En línea]. Available: http://ipronet.es/productos/vms.php?language=english.

[5] G. Security, «IP Video Supports Quality Management,» [En línea]. Available: http://www.git-security.com/topstories/security/ip-video-supports-quality-management.

[6] A. y. Videovigilancia, «alarmasyvideovigilancia.com,» [En línea]. Available: http://alarmasyvideovigilancia.com/camaras-ip-serie-ultra-smart-ipc/.

[7] V. S. a. a. S. (VSaaS), «VSaaS,» [En línea]. Available: http://www.vsaas.com/.

[8] OpenCV, «Wikipedia,» [En línea]. Available: http://es.wikipedia.org/wiki/OpenCV.

[9] OpenCV, «Módulo NonFree,» [En línea]. Available: http://docs.opencv.org/modules/nonfree/doc/nonfree.html.

[10] Digia, «Digia adquiere Qt de Nokia,» [En línea]. Available: http://www.digia.com/en/Company/Press/2012/Digia-to-acquire-Qt-from-Nokia/.

[11] Qt, «Necessitas Qt for Android,» [En línea]. Available: https://wiki.qt.io/Necessitas.

[12] G. Play, «Descarga App Ministro II,» [En línea]. Available: https://play.google.com/store/apps/details?id=org.kde.necessitas.ministro&hl=es.

[13] Foscam, «IP Camera CGI API,» [En línea]. Available: http://www.foscam.es/descarga/ipcam_cgi_sdk.pdf.

[14] AXIS, «VAPIX HTTP API Specification,» [En línea]. Available: http://www.axis.com/files/manuals/HTTP_API_VAPIX_2.pdf.

[15] Solwise, «Solwise IP Cameras,» [En línea]. Available: http://www.solwise.co.uk/sec-ip-cameras.html.

[16] C. SDK, «INSTAR supported cameras,» [En línea]. Available: http://www.camera-

sdk.com/p_186-how-to-connect-to-your-instar-ip-camera-onvif.html].

[17] C. A. U. Manual, «iSmartView,» [En línea]. Available: http://download.conceptronic.net/manuals/CIPCAMPTIWL_Android_User_Manual_V2.0.pdf.

[18] OpenCV, «IplImage Struct Reference,» [En línea]. Available: http://docs.opencv.org/master/d6/d5b/structIplImage.html#gsc.tab=0.

[19] W. d. Haar, «Wikipedia,» [En línea]. Available: https://es.wikipedia.org/wiki/Wavelet_de_Haar.

[20] OpenCV, «Cascade Classifier Training,» [En línea]. Available: http://docs.opencv.org/2.4/doc/user_guide/ug_traincascade.html.

[21] GitHub, «lbpCascades,» [En línea]. Available: https://github.com/Itseez/opencv/tree/master/data/lbpcascades.

[22] SourceForge, «OpenCV-Win,» [En línea]. Available: http://sourceforge.net/projects/opencvlibrary/files/opencv-win.

[23] SourceForge, «OpenCV-Unix,» [En línea]. Available: http://sourceforge.net/projects/opencvlibrary/files/opencv-unix/.

[24] SourceForge, «OpenCV-Android,» [En línea]. Available: http://sourceforge.net/projects/opencvlibrary/files/opencv-android.

[25] UIB, «Filtrado de Imagenes,» [En línea]. Available: http://dmi.uib.es/.

[26] OpenCV, «Sobel Derivatives,» [En línea]. Available: http://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/sobel_derivatives/sobel_derivatives.html.