Universidad
Carlos III de Madrid

Desarrollo de una aplicación Android para la detección de señales de tráfico

Trabajo de Fin de Grado

Autor:

Álvaro Carbajo Benito

Tutor:

Fernando García Fernández

Director de proyecto:

Juan Carmona

Grado en Ingeniería Electrónica, Industrial y Automática

Universidad Carlos III de Madrid

Junio, 2015

Índice general

Índice de figuras

Anexo:

Índice de tablas

Resumen

Los sistemas de visión por computador están ganando protagonismo en nuestra vida diaria. Durante los últimos 30 años, han pasado de ser una mera utopía a convertirse en una realidad. Encontramos estos sistemas en aplicaciones militares, aplicaciones de seguridad para detección de movimiento o intrusos; aplicaciones industriales como controles de calidad, aplicaciones para la gestión, administración y seguridad en medios de transporte, etc.

Por otro lado, los teléfonos inteligentes o 'smartphones' han adquirido también un papel fundamental en nuestras vidas, y el uso que les damos a estos dispositivos los ha transformado en auténticos asistentes personales, simplificando nuestras vidas de manera considerable.

Por tanto, era de esperar que ambos campos se uniesen y que los 'smartphones' comenzasen a ofrecer aplicaciones con funcionalidades de visión por computador o realidad aumentada, es decir, utilizar la cámara del teléfono para obtener información de nuestro alrededor.

En ese contexto de aplicaciones emergentes es donde se desarrolla este proyecto, un campo aun poco explorado, con mucho potencial pero también con mucho trabajo por delante.

Este trabajo se centra en la detección de señales de tráfico mediante el uso de la cámara de los teléfonos inteligentes de la plataforma Android. Para ello, se hará uso de los algoritmos implementados por las librerías OpenCV permitiendo localizar las áreas de la imagen con mayor probabilidad de contener una señal de tráfico primero, y verificando después si, efectivamente, el objeto detectado corresponde con una señal de tráfico.

Finalmente, se analizarán otras posibles alternativas para la detección de señales de tráfico no implementadas en este proyecto.

Abstract

Computer Vision systems are gaining prominence in our daily lives. During the last 30 years, they have gone from being a mere utopia to become a reality. We find these systems in military applications, security applications for motion detection or intrusion; industrial applications such as quality control, management applications, management and security in transport, etc.

On the other hand, smartphones have also acquired an important role in our lives, and the use we gave them has transformed the smartphones into our personal assistants, simplifying our lives in a great deal.

Therefore, it was to be expected that both fields unify and smartphones began to deliver applications with computer vision capabilities or augmented reality, which is using the phone's camera to get information about our surroundings.

In this context of emerging applications it is where this project is developed, a field still little explored, with lot of potential but also a lot of work ahead.

This paper focuses on the detection of road signs using the Android device's camera. To do this, we will make use of the algorithms implemented by the OpenCV libraries allowing the application to locate the areas of the image which are most likely to contain a road signal first, and then, verifying if indeed the detected object corresponds to a road sign.

Finally, other alternatives for the detection of road signs not implemented in this project will be as well analyzed.

Agradecimientos

"En esta vida hay dos clases de personas: las que siempre forman parte del problema, y las que siempre forman parte de la solución."

A mis padres, por estar a mi lado siempre que los necesito.
A mi familia, y amigos, por hacer de mí la persona que soy.
A mis profesores, compañeros y tutores, por recorrer conmigo este largo camino durante los últimos cinco años.

Acrónimos

CV	Computer Vision – Visión por Computador
SKD	Software Development Kit – Paquete de Desarrollo de Software
NDK	Native Development Kit – Paquete de Desarrollo Nativo
LSI	Laboratorio de Sistemas Inteligentes
ROI	Region Of Interest – Región de interés
FPS	Frames Per Second – Fotogramas por segundo
TFG	Trabajo de Fin de Grado
XML	Extensible Markup Languaje
OpenCV	Open Source Computer Vision
UC3M	Universidad Carlos III de Madrid

1. INTRODUCCIÓN

El nacimiento y rápido crecimiento de los teléfonos inteligentes (del inglés, 'Smartphones') ha hecho que entren a formar parte de nuestras vidas sin que apenas nos hayamos dado cuenta de ello. Sin embargo, poco a poco, los servicios y aplicaciones que estos dispositivos ofrecen se han integrado en nuestras actividades cotidianas, simplificando y asistiendo muchos de los quehaceres de nuestro día a día.

Es cada vez más frecuente descubrir nuevas aplicaciones con nuevas funcionalidades para realizar tareas que hace unos años habrían resultado difíciles de creer. Además, la continua mejoría de las especificaciones técnicas de estos dispositivos permite que dichas aplicaciones puedan ejecutar algoritmos más potentes y complejos.

Uno de los campos que está en pleno auge actualmente es del de los sistemas de visión por computador (CV) y realidad aumentada, no solo en aplicaciones para dispositivos móviles sino para toda clase plataformas y ámbitos. Este tipo de sistemas, utiliza las imágenes obtenidas mediante sensores ópticos y procesa las imágenes o señales obtenidas para obtener información del mundo real.

Como no podía ser de otra manera, los desarrolladores de Android no han tardado en crear las herramientas necesarias para trabajar con algoritmos de CV en esta plataforma. Las librerías de OpenCV (Opern Source Computer Vision), ponen a disposición del programador una gran variedad de algoritmos y clases utilizados para el procesamiento de imágenes y la visión por computador.

Una de las grandes ventajas de estas librerías, así como de la plataforma Android, es que son proyectos de código abierto, por lo que se encuentran en constante crecimiento y tienen detrás una gran comunidad de usuarios y programadores muy activos en el intercambio de información para impulsar el progreso de ambos proyectos.

Uno de los ámbitos donde se están desarrollando y probando aplicaciones de visión por computador es en la industria automovilística y en el entorno de los sistemas de transporte inteligente. Es en ese marco de investigación en el que se encuentra el TFG que esta memoria detalla.

Durante el desarrollo de este proyecto, se analizarán, implementarán y probarán diferentes alternativas para la detección y reconocimiento de señales de tráfico a través de las cámaras de los 'Smartphones' que utilicen el SO Android.

Este Trabajo de Fin de Grado se integra en un proyecto más amplio para la implementación de algoritmos de visión por computador para la plataforma Android desarrollado por el Laboratorio de Sistemas Inteligentes (LSI) de la Universidad Carlos III de Madrid.

2. ESTADO DEL ARTE

2.1. Android

Android es un sistema operativo libre y gratuito basado en el kernel (núcleo) de Linux. Inicialmente fue desarrollado por Android Inc., y más tarde fue comprado por Google, empresa que se encarga de su desarrollo actualmente. En un principio, fue diseñado para teléfonos móviles con pantalla táctil, convirtiéndose más tarde en un SO multiplataforma que se encuentra en una gran variedad de dispositivos: tablets, relojes, auriculares, coches, netbooks, portátiles, televisores… [1].

Para hacerse una mejor idea de qué es Android y cómo funciona, es necesario analizar brevemente su estructura y los diferentes niveles en los que se divide [2].

Empezando en el primer nivel nos encontramos con el núcleo de Linux. Android utiliza Linux para administrar controladores, memoria, procesos y redes. Sin embargo, las aplicaciones para Android no se programan directamente sobre este nivel.

A continuación, nos encontramos con las librerías (bibliotecas) nativas de Android. Están escritas en C/C++ y se accede a ellas mediante el uso de las Interfaces Nativas de Java (JNI).

En el siguiente nivel, conocido como entorno de ejecución, es en el que se encuentra la máquina virtual Dalvik – DVM, optimizada para reducir la memoria utilizada por las aplicaciones, y diseñada para poder ejecutar varias instancias de la máquina virtual simultáneamente, delegando al sistema operativo subyacente la gestión de memoria, los hilos y el SandBoxing (aislamiento de procesos).

En la última versión del sistema operativo Android (Lollipop), Dalvik ha sido sustituida por ART (Android Runtime).

Android pone al servicio de los desarrolladores una gran variedad de librerías que también se encuentran en este nivel, y que están escritas en Java. Entre estas librerías se encuentran: las bibliotecas 3D, las bibliotecas de medios gráficos, bibliotecas de entradas y salidas…

En el siguiente nivel nos encontramos con el marco de la aplicación. En este marco de la aplicación se alojan las herramientas necesarias para administrar el ciclo de vida de la aplicación, los recursos, paquetes, sensores y otros complementos necesarios para el funcionamiento de las aplicaciones.

Finalmente, en el último nivel es donde se desarrolla gran parte del código de las aplicaciones creadas por los desarrolladores y las aplicaciones de serie del teléfono como el marcador, contactos, navegador…

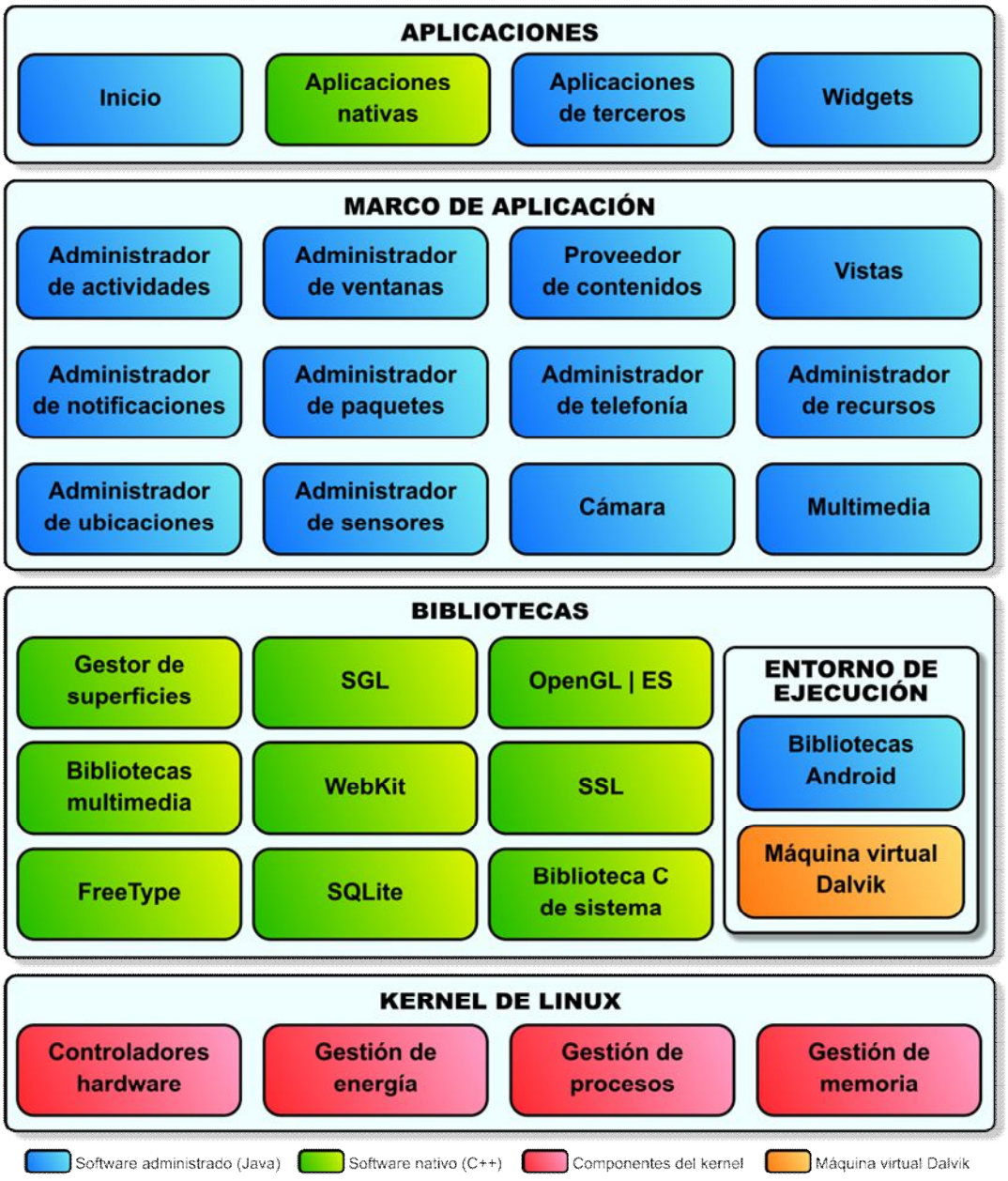

Figura 1: Estructura Android [2].

2.2. Open CV

OpenCV (Open Source Computer Vision Library) es una biblioteca de software de visión por ordenador de código abierto y de aprendizaje automático. Su principal objetivo es crear una infraestructura común para las aplicaciones de visión por computador y acelerar el uso de los sistemas de percepción por computador en sistemas comerciales.

La librería cuenta con más de 2500 algoritmos optimizados para numerosas aplicaciones, que pueden ser utilizados para la detección y reconocimiento caras, identificación de objetos, detección de movimiento, seguimiento de objetos, clasificación de acciones humanas, extracción de modelos 3D, reconocimiento de escenarios y superposición con realidad aumentada, comparación de imágenes y un largo etcétera que sigue creciendo con el avance de la tecnología, las cámaras y las tarjetas de procesamiento [3].

Además, son unas librerías multiplataforma, disponibles por tanto para Linux, Windows, Mac OS, Android… y pueden ser implementadas desde varios lenguajes también, como C, C++, Python, Java y MATLAB.

2.3. Visión por computador

La visión por computador es la disciplina que busca la manera de reconstruir, interpretar y entender una escena en 3 dimensiones mediante sus proyecciones en dos dimensiones. En otras palabras, la visión por computador permite obtener información no gráfica/visual a partir de una imagen o conjunto de imágenes. La meta principal de este campo es modelar el sistema visual de los seres humanos e incluso, en un futuro, sobrepasar nuestras capacidades visuales [4].

La vista es el sentido perceptivo que nos aporta más información del mundo exterior, pero también es el que presenta un sistema más complejo y por tanto es más difícil de modelar. En el momento actual de desarrollo de los sistemas de percepción por computador, ese objetivo de replicar el sistema visual humano aún está lejos, aunque se han conseguido grandes avances y dichos sistemas ya superan nuestras capacidades en algunas aplicaciones. Pese a esto, existe cierta polémica entre los expertos sobre si la visión por computador conseguirá algún día igualar o sobrepasar la visión de los seres humanos o si, en cambio, hay ciertas cosas que un computador no puede hacer.

Se podría decir que la visión por computador nace en la década de 1950, bajo el nombre de Visión Artificial. Durante los primeros años de desarrollo, el optimismo inicial en el rápido avance en este campo se vio truncado por las dificultades que fueron surgiendo. Posteriormente, los avances en sistemas de inteligencia artificial y redes neuronales impulsaron la creación de nuevos algoritmos de visión por computador capaces de obtener información de las imágenes mediante conceptos matemáticos abstractos.

La década de los 90 es el momento en el que se desarrolla la visión por computador moderna. El descubrimiento de los algoritmos de detección y extracción de características y las máquinas de soporte vectorial (SVM) permitieron resolver complejos problemas como la identificación y clasificación de objetos, reconocimiento de caras, etc.

En la actualidad, existe un sinfín de aplicaciones y ámbitos en los que los sistemas de visión artificial marcan la diferencia. La lista sigue creciendo con nuevas áreas de actuación según estos sistemas se van haciendo más populares y los avances tecnológicos van abriendo nuevas puertas.

Podemos encontrar sistemas de CV en el ámbito militar (impulsor de grandes avances en este campo), en sistemas de seguridad, en la industria automovilística, ferroviaria, aeronáutica (y de medios de transporte en general). También en entornos industriales, tanto al nivel de manufacturación, como para revisiones calidad, por ejemplo.

Respecto a las aplicaciones en el área de transporte, nos encontramos con las aplicaciones desarrolladas en los Sistemas de Transporte Inteligente, STI, o su acrónimo en inglés ITS. Estos sistemas se encargan de proveer soluciones tecnológicas, telemáticas e informáticas por lo general, que mejoran la calidad de las operaciones y la seguridad de los medios de transporte.

No todas las soluciones de estos sistemas se basan en la visión por computador, pero la implementación de sistemas de CV para aplicaciones de seguridad y optimización ha crecido significativamente en los últimos años.

Un claro ejemplo es la integración de cámaras y sensores infrarrojos en los vehículos, para asistir al conductor a la hora de aparcar, detectar los límites de la carretera, evaluar la proximidad con otros vehículos, detectar peatones con riesgo de atropello… y actuar automáticamente si la situación lo requiere.

En el ámbito de las aplicaciones de CV para la asistencia a la conducción es en el que se desarrolla este proyecto.

2.4. Procesamiento Digital de Imágenes

El procesamiento digital de imágenes tiene como objetivo optimizar el aspecto de las imágenes para hacer más o menos evidentes ciertas características que se quieran resaltar o eliminar. A diferencia de la visión por computador, la meta principal de este campo no es obtener información de las imágenes sino generar una nueva imagen a partir de la imagen original, cuyo resultado sea más adecuado para una aplicación específica.

Con las técnicas de procesamiento de imágenes se puede retocar una imagen para optimizar las características que el sistema visual humano percibe, como modificar el contraste, el brillo, eliminar ojos rojos, etc… O se puede modificar una imagen para favorecer las operaciones que un computador puede hacer con esa imagen.

En este segundo caso es en el que el procesamiento digital de imágenes entra en juego en el ámbito de la Visión por Computador. Resaltando ciertos aspectos de la imagen, se pueden aplicar después algoritmos para obtener información a partir de ella.

Para ello, se utilizan las técnicas de filtrado que permiten, entre otras cosas, suavizar (‘smoothering’) la imagen para reducir las diferencias de intensidad entre pixeles vecinos; eliminar ruido, realzar bordes…

Las técnicas de filtrado pueden ser aplicadas en el dominio de la frecuencia, con la Transformada de Fourier de la imagen, o en el dominio del espacio, aplicando las operaciones directamente sobre los píxeles de la imagen. Para el caso del procesamiento digital, se aplican las operaciones en el dominio espacial.

2.5. Asistentes de conducción

Como ya se ha mencionado brevemente entre las aplicaciones de la Visión por Computador, la industria automovilística es una de las áreas en las que se están desarrollando una gran variedad de aplicaciones de visión artificial. Las primeras aplicaciones que los vehículos integraron permitían asistir al conductor a la hora de aparcar, mediante sensores infrarrojos que medían proximidad, en sus comienzos, y mediante cámaras y videos con líneas de ayuda más tarde.

Actualmente los vehículos de gama alta de marcas como Audi, Volvo,… permiten por ejemplo, medir la distancia con el vehículo de delante y alertar al conductor o incluso reducir la velocidad automáticamente si es necesario.

El último proyecto de Volvo en este ámbito está enfocado a la seguridad de los peatones, detectando mediante una cámara la posición del peatón y calculando su trayectoria para ver si interfiere con la del vehículo. En caso afirmativo, se reduce la velocidad o se detiene el coche si es necesario.

Otro ejemplo de sistemas de Visón por Computador integrados en vehículos es la flota de Toyota Prius sin conductor regentada por Google, que ha recorrido ya más de 300.000 Km en entornos urbanos, autopistas, carreteras de montaña… con intervención humana únicamente en ocasiones puntuales.

En la figura 2 se muestra uno de estos vehículos y se detalla la posición y funcionalidad de los sensores que lleva integrados.

Todos estos proyectos se encuentran en sus primeras fases y todavía no son 100% fiables, algunos no son ni si quiera comercializables todavía, pero marcan el comienzo de una nueva etapa en nuestra manera de circular.

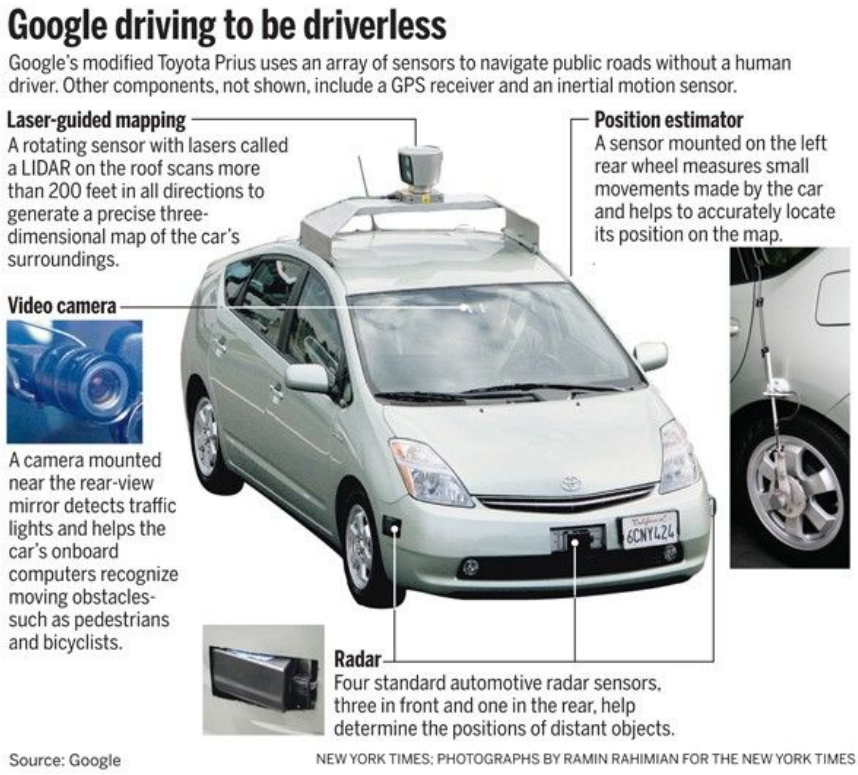

Figura 2: Coche sin conductor de Google.

En cuanto a los asistentes de conducción no integrados en vehículos, las librerías de OpenCV proporcionan las herramientas necesarias para el desarrollo de aplicaciones para la detección de vehículos, de peatones, de adelantamientos, de los límites de la carretera, de señales de tráfico, identificación de matrículas y control de tráfico… La principal dificultad con la que se encuentran todos estos sistemas es reducir los costes computacionales y los tiempos de procesamiento para conseguir que las aplicaciones funcionen en tiempo real. Este es uno de los motivos por el cual este tipo de aplicaciones son generalmente desarrolladas para computadores y no para teléfonos inteligentes u otros dispositivos con menor capacidad de procesamiento.

Durante el desarrollo de este proyecto se tratará de implementar una aplicación para la detección de señales de tráfico como las ya existentes para ordenadores en un dispositivo Android. Se utilizarán las técnicas y herramientas que ya se ha comprobado que funcionan en computadores más potentes que los 'smartphones' [5] [6] y [7].

2.6. Detección de señales de tráfico en Google Play

La presencia de aplicaciones desarrolladas con las librerías de OpenCV en el Google Play es todavía reducida, y la mayoría de aplicaciones existentes son comercializaciones de los ejemplos que las propias librerías OpenCV incluyen en su paquete SDK para Android.

En cuanto a detectores de señales de tráfico, nos encontramos con tres aplicaciones desarrolladas para la detección y reconocimiento de señales de tráfico. Además, dos de ellas se encuentra integradas en aplicaciones más completas de asistencia a la conducción con otras funcionalidades como alertas de distancia de seguridad, colisión frontal, salida de carril…

Se han realizado pruebas con las tres aplicaciones para analizar su rendimiento:

Road Sign Recognition

Figura 3: Road Sign Recognition Free - App Google Play [8].

Figura 4: Road Sign Recognition - App Google Play [8].

Esta aplicación presenta dos versiones en Google Play, una gratuita y la otra de pago [8]. La versión gratuita, que es la que se ha probado para realizar las pruebas no funciona, al ejecutarla, aparece un mensaje de error y se cierra.

Según la descripción de la aplicación, es capaz de detectar señales de limitación de velocidad con forma circular y color rojo. Añade también que es posible que la aplicación no funcione en tiempo real en todos los dispositivos.

iOnRoad Aumented Driving

Figura 5: iOnRoad Augmented Driving - App Google Play [9].

Figura 6: iOnRoad Augmented Driving Pro - App Google Play [9].

Esta aplicación, al igual que la anterior, ofrece también una opción gratuita y otra de pago. En este caso, se han hecho las pruebas con la versión pro [9].

En su descripción se detallan las funcionalidades que ofrece la aplicación y comprobamos que nos encontramos ante un asistente de conducción más completo:

- Advertencia de colisión frontal.
- Advertencia de distancia de seguridad.
- Advertencia de salida de carril.
- Localización del vehículo.
- Alerta de exceso de velocidad.

Para este último módulo, la aplicación detecta las señales de limitación de velocidad, es decir, señales circulares con borde rojo y con un número dentro.

Los resultados de las pruebas realizadas con esta aplicación han sido positivos y, en efecto, se detectan casi todas las señales de velocidad, amén de funcionar también el resto de alertas y detectores.

myDriveAssist

Figura 7: myDriveAssist (BOSCH) - App Gogle Play [10].

Por último, se ha probado una aplicación desarrollada por la empresa Bosch [10]. En la descripción informan que la aplicación es capaz de detectar señales de limitación de velocidad y adelantamiento en una "multitud de países (entre otros, España, Alemania, Austria, Suiza, Francia y el Reino Unido)".

Además, anuncian que la calidad de la detección puede variar en función del dispositivo utilizado y que las circunstancias lumínicas, a contraluz o por la noche, pueden afectar seriamente a la detección.

En las pruebas realizadas con esta aplicación se comprueba que la aplicación detecta correctamente las señales de velocidad. Al haberse desarrollado estas pruebas en un entorno urbano donde no hay señales de adelantamiento, no se ha podido comprobar si funciona o no la detección de las mismas.

Esta incursión en las aplicaciones de Google Play desarrolladas para la detección de señales de tráfico nos anuncia ya algo que se irá comprobando durante el desarrollo del proyecto, la detección de señales de tráfico con dispositivos móviles se encuentra en sus primeras fases de desarrollo y por tanto es complicado implementar un detector completamente funcional.

2.6. Proyectos LSI – Laboratorio de Sistemas Inteligentes

El Laboratorio de Sistemas Inteligentes de la UC3M tiene en su haber una gran variedad de proyectos dedicados a la implementación de algoritmos y aplicaciones de CV.

Destaca, entre otros, el proyecto del vehículo inteligente capaz de detectar peatones por la noche o en bajas condiciones lumínicas [11], premiado el año pasado [12] por la fundación Eduardo Barreiros con el 'Premio a la investigación en el campo de la automoción'.

Este sistema utiliza una cámara de infrarrojos para detectar el calor desprendido por los objetos y avisar al conductor en caso de que uno de estos objetos sea un peatón.

Figura 8: Coche inteligente LSI, UC3M [13].

Además de este proyecto, el LSI cuenta con otros trabajos relacionados con los sistemas de Visión por Computador aplicados al sector automovilístico, como es el proyecto en el que se integra este TFG, una aplicación base para la implementación de algoritmos de visión por computador en Android sobre la que ir añadiendo nuevas funcionalidades mediante, por ejemplo, otros TFG, como ocurre en este caso.

DESCRIPCCIÓN DEL SISTEMA

3.1. Elección del Software

A la hora de programar en Android, la primera cuestión que uno se plantea es el entorno de desarrollo. Existen dos IDE (integrated developement enviroment/entorno de desarrollo integrado) principales, el SDK de Android para Eclipse y Android Studio. El primero lleva más tiempo en acción, y hasta hace poco era la opción principal para el desarrollo en Android y recibía gran soporte por 'google' y la comunidad Android. Hace dos años (mediados 2013), desde google iniciaron el proyecto Android Studio para convertirlo en la herramienta principal de desarrollo y es el que actualmente recibe soporte por parte de la renombrada empresa.

Para el proyecto que nos atañe. Se eligió Eclipse sobre Android Studio por las negativas experiencias previas del tutor y del director de proyecto a lo hora de utilizar las librerías de OpenCV en Android Studio. Sin embargo, dichas experiencias se obtuvieron en proyectos realizados en las primeras fases de vida de Android Studio, por lo que es posible que en el punto de desarrollo y soporte en el que se encuentra ahora, se puedan utilizar estas librerías sin problema. Aun así, se eligió Eclipse por comodidad y simplicidad, ya que es el entorno en el que se desarrollan los proyectos de Android del Laboratorio de Sistemas Inteligentes.

Por otro lado, la versión de las librerías de OpenCV a utilizar es la 2.4.9. Al comienzo del desarrollo de este proyecto, Enero de 2015, la versión más actualizada de estas librerías era la 2.4.10. Se eligió la versión anterior por llevar más tiempo en funcionamiento y ser una versión más estable. A día de hoy, Junio de 2015, la versión 3.0.0 se encuentra ya disponible en fase beta [3].

Con respecto a la versión de Android, la aplicación está programada para funcionar con las versiones de Android 2.3 y superiores, es decir, desde Android GingerBread hasta Android Lollipop 5.1.

3.2. Introducción al problema

Un primer estudio del problema a resolver nos permite llegar a varias conclusiones que marcarán la trayectoria a seguir del TFG:

Las señales de tráfico son utilizadas en todos los países del mundo en los que los vehículos son el principal medio de transporte. Sin embargo, dichas señales no son iguales en todos los países y, aunque las diferencias entre las señales de unos lugares y otros no sean abismales y haya señales y símbolos en común, es conveniente reducir las señales a las de un único país, para optimizar los resultados a obtener. Para este proyecto, utilizaremos, evidentemente, las señales de tráfico españolas.

Por otro lado, las características más significativas a la hora de caracterizar las señales son su forma y color. Dichas características, permiten agrupar las señales en función de la información que contengan y el carácter que expresen: se utilizan señales triangulares y rojas para advertir peligro, señales circulares y rojas para anunciar prohibición u obligación, señales cuadradas y azules para recomendación, con fondo blanco para señalar fin de prohibición, señales provisionales con fondo amarillo para anunciar la existencia de obras en la carretera… y finalmente, señales rectangulares de diversos colores y tamaños para los paneles informativos de orientación, señalización, localización, identificación de la carretera, salidas próximas, puntos kilométricos…).

El siguiente y último punto a tener en cuenta en este primer análisis es el entorno en el que se encuentran la señales, urbano o interurbano. Para este proyecto nos centraremos en las señales del entorno urbano, principalmente por las complicaciones que surgen con las altas velocidades de circulación en las vías interurbanas y las limitaciones de las cámaras de los dispositivos a dichas velocidades. Además, esto nos permite centrarnos en las señales de circulación, dejando a un lado las señales o paneles de información, más características de los tramos interurbanos. En el apartado de limitaciones se ha estudiado más en detalle cómo afecta la velocidad al proceso de detección de la señal.

Por tanto, llegados a este punto, hemos reducido la gran variedad de señales de tráfico a las señales españolas situadas en entorno urbano, y finalmente, decidimos enfocar el problema a las señales de advertencia o peligro, características por su marco rojo.

3.3. Estructura de la aplicación

Reduciendo el problema a resolver hasta su estructura más simple, se ha decidido dividir el proyecto en dos grandes bloques: 'mapping' y verificación.

El primer bloque es el responsable de la localización de la señal o posible señal en la imagen ('mapear') y el segundo se encarga de verificar si, efectivamente, el objeto 'mapeado' en el proceso anterior es una señal o no; y en caso de que lo sea, obtener la información pertinente de dicha señal.

A su vez, estos dos grandes apartados pueden dividirse en sub–procesos y ser implementados de diversas maneras, como se mencionaba anteriormente. A continuación, se enumerarán brevemente los pasos necesarios para el desarrollo de cada bloque, y más adelante, serán explicados en detalle.

La idea del primer apartado es simplificar el trabajo del segundo, ya que es el que presenta un mayor coste computacional. De esta manera, se consigue que la entrada a la fase de verificación sea una región de la imagen con altas probabilidades de contener una señal, para evitar el tremendo impacto temporal que tendría realizar esta verificación en toda la imagen y no solo en la región de interés (ROI).

Para llevar a cabo dicha tarea, nos centramos primeramente en las señales de prohibición y advertencia o peligro, caracterizadas por tener un borde rojo y en su interior la información en negro sobre fondo blanco. Elegimos estas señales por ser las más comunes en el entorno urbano y porque consideramos que son las que menos problemas darán, ya que las azules 'comparten' color con el cielo.

El primer paso consiste en una segmentación por color. De esta manera, se obtienen las zonas de la imagen de color rojo, permitiendo al dispositivo centrarse en estas áreas y descartar todo lo demás. Hasta este punto, cualquier objeto de color rojo avanza al siguiente proceso. Para aumentar las probabilidades de que las ROI segmentadas contengan una señal de tráfico, se filtran de nuevo los resultados obtenidos en la segmentación de color, utilizando en este caso el tamaño y la forma de los objetos como medio para cribar las regiones de interés menos favorables. Por tanto, los objetos rojos demasiado pequeños para ser relevantes, o con formas extrañas y diferentes a las formas convencionales de las señales de tráfico (círculos y triángulos) son nuevamente descartados.

Una vez tenemos una región de interés con un objeto rojo y forma circular o triangular en su interior, se utiliza un método conocido como *detección y extracción de características* para comparar la sección de interés con una base de datos de imágenes de tráfico, y comprobar si coincide con alguna señal existente según la normativa española

[48]. En caso afirmativo, se confirma que el objeto detectado es una señal, y además se puede saber de qué señal se trata, pudiendo utilizar la información de la señal para alertar al conductor, mejorar la experiencia de la conducción, etc. En caso de que el objeto contenido en la ROI no concuerde con ninguna señal de la base de datos, se descarta esa región de interés.

Los algoritmos utilizados para el proyecto en cuestión serán explicados con más detalle en el apartado de implementación.

Aunque no lleguen a implementarse todas las posibles soluciones, se analizarán las ventajas e inconvenientes de las diferentes soluciones principales a lo largo de la memoria.

4 IMPLEMENTACIÓN

4.1. Aplicación base del Laboratorio de Sistemas Inteligentes

Esta aplicación para la detección de señales de tráfico se integra en una aplicación base desarrollada para el LSI por Alejandro Ramos para la implementación de algoritmos de Visión por Computador en la Plataforma Android.

Esta aplicación contiene diversas sub – aplicaciones con diferentes funcionalidades. Por un lado, nos encontramos con los ejemplos que el paquete de OpenCV pone a la disposición de los desarrolladores:

Figura 9: App Base LSI - Ejemplos OpenCV.

Y además, integra funcionalidades para la aplicación de peatones y coches.

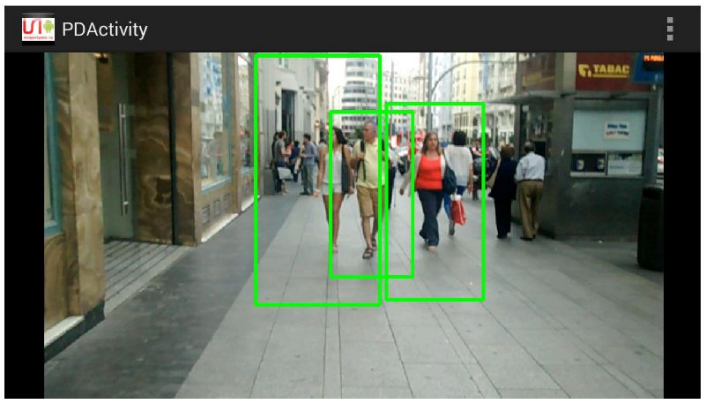

Figura 10: App Base LSI - Detector Peatones.

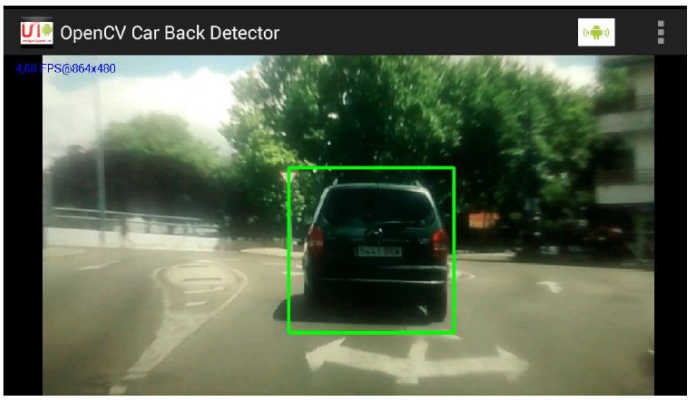

Figura 11: App Base LSI - Detector Coches.

La implementación de la aplicación para la detección de señales de tráfico se desarrolló de manera independiente a esta aplicación base y se integró en ella una vez completada.

El funcionamiento de esta aplicación será detallado en el anexo 'Manual de la aplicación'.

4.2. Conceptos Android

Para entender el funcionamiento y la estructura de una aplicación Android es necesario explicar una serie de conceptos y componentes básicos presentes en cualquier aplicación.

El primero de ellos es el concepto de actividad. Una actividad es un componente de aplicación que presenta una interfaz que permite a los usuarios interactuar con el fin de hacer algo, como llamar por teléfono, hacer una foto, enviar un correo electrónico... La actividad ocupa normalmente el total de la pantalla, pero puede ser más pequeña y flotar en la pantalla sobre otras ventanas o actividades [14].

Las aplicaciones suelen estar compuestas por una multitud de actividades interconectadas unas con las otras. La actividad que presenta la aplicación cuando se inicia es conocida como la actividad principal, y puede llevar a otras actividades para realizar diferentes funciones. Cada vez que una nueva actividad comienza, la anterior es detenida, aunque el sistema evita que se cierre, manteniéndola en un segundo plano en la pila de actividades ('back stack' en inglés). Las actividades tienen varios estados:

- Activa o 'Resumed': La actividad está encima de la pila de actividades, por lo que es visible y el usuario puede interactuar con ella.
- Pausada o 'Paused': La actividad es visible pero no tiene el foco, es decir, hay otra actividad que no ocupa el total de la pantalla sobre ella.
- Parada o 'Stopped': La actividad no es visible, el programador elige que preferencias guardar cuando la aplicación entra en este estado para poder recuperarlas más tarde.
- Destruida o 'Destroyed': La actividad es matada por el sistema.

Aquí entra en juego el conocido ciclo de vida de una actividad, que estipula cómo se comportan las actividades y como pasan de un estado a otro.

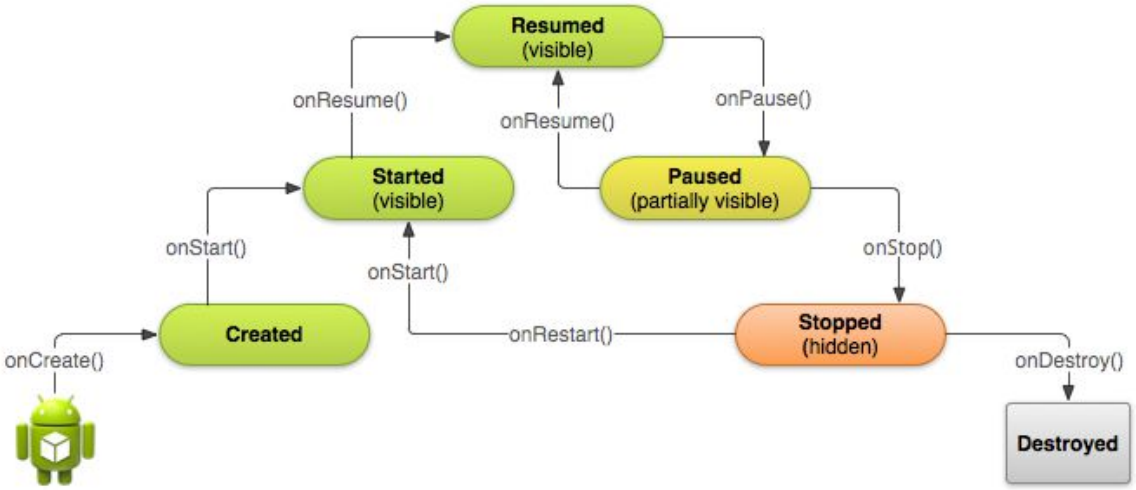

Figura 12: Ciclo de vida de una Actividad Android [15].

Otro concepto clave para entender el funcionamiento de una aplicación Android es el 'intent'. Un 'intent' es una solicitud para realizar una acción. En esencia, es un mensaje pasado entre los componentes de una aplicación. Son utilizados, por ejemplo, para pasar de una actividad a otra.

Durante el desarrollo de una aplicación Android, cada vez que una nueva actividad es incluida al proyecto se generan dos archivos, uno de tipo XML y otro de tipo Java. El primero se utiliza para el diseño de la interfaz de usuario, y el segundo para la programación de las funcionalidades de esa interfaz.

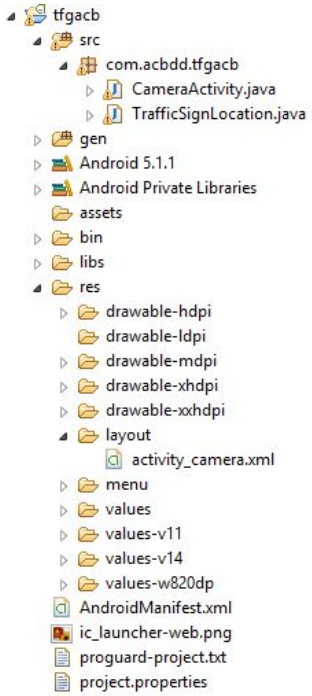

Figura 13: Estructura del Proyecto en Eclipse.

4.3. Conceptos OpenCV

Para este proyecto, además de entender el funcionamiento general de una aplicación Android, es pertinente también analizar las herramientas de Visión por Computador que las librerías de OpenCV ponen a disposición del desarrollador. Este conjunto de métodos, algoritmos y clases específicas permiten añadir nuevas funcionalidades a las aplicaciones.

A lo largo de la memoria, se hará referencia a los 'frames' o en español, fotogramas. Un frame es cada una de las imágenes que captura la cámara del dispositivo y que al ser representadas secuencialmente dan sensación de movimiento.

El concepto de frame nos lleva a los FPS – 'frames per second' o fotogramas por segundo. Esta variable representa el número de imágenes por segundo que tiene un vídeo o, en el caso que nos ocupa, la salida de la aplicación.

Las librerías de OpenCV proveen también diferentes tipos de variables para almacenar y operar con imágenes. La clase utilizada para almacenar los frames de entrada de la cámara así como las imágenes obtenidas tras realizar las operaciones pertinentes en dichos frames es la clase **Mat**. Estas matrices almacenan el valor de cada pixel de la imagen en sus filas y columnas correspondientes. La codificación utilizada para almacenar dichos valores depende de la cantidad de niveles de color que se quieran representar. Para este proyecto se utilizará la codificación CV_8U, de tal manera que cada pixel es representado por 8 bits sin signo, permitiendo representar 256 niveles de gris, que van desde el negro '0' hasta el blanco '255' pasando por diferentes intensidades de gris.

En caso de que se quiera almacenar una imagen en escala de grises, estas matrices están compuestas por un único canal. Sin embargo, si se quiere almacenar una imagen a color, se aumenta el número de canales de la matriz, de tal manera que el valor final de cada pixel es la suma de los valores para ese pixel de cada canal.

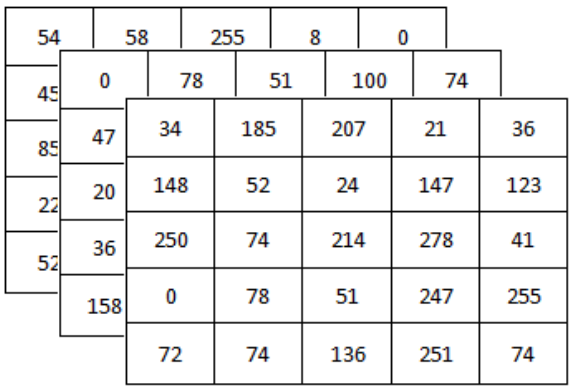

Figura 14: Layout Mat con 3 canales.

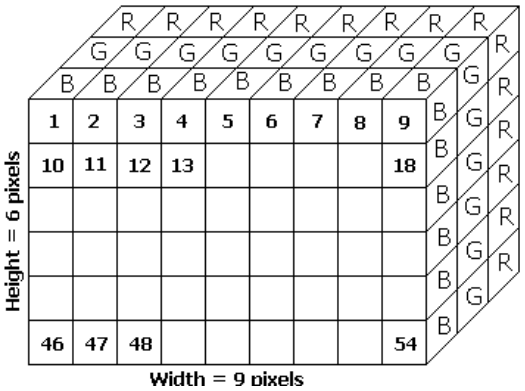

Figura 15: Altura y anchura de Mat.

Las filas y columnas de estas matrices son comúnmente denominadas como anchura, *Width*, y altura, *Height* de la matriz.

Para la obtención de los frames de la cámara se utiliza una clase de las librerías OpenCV llamada *JavaCameraView*. Esta clase es una implementación de un puente entre la cámara de Java y OpenCV [16].

El procesamiento principal de esta aplicación se realiza en un método provisto por las librerías OpenCV. Este método es conocido como *onCameraFrame*(inputFrame), su argumento es el frame obtenido anteriormente, y retorna una variable de tipo Mat. El código implementado dentro de este método es ejecutado cada vez que se obtiene un nuevo frame.

A lo largo de la memoria se hará referencia al concepto de vecindario de un pixel. Los píxeles vecinos son aquellos que se encuentran próximos, tanto vertical y horizontalmente como diagonalmente. El tamaño de un vecindario hace referencia al número de filas y columnas que lo forman.

Por último, cuando se aplican filtros u operaciones sobre una imagen o vecindario, se utilizan los términos 'kernel', 'máscara', 'ventana' o 'convolución' para referirse al tamaño de la región de la imagen sobre la que se aplica la operación, y que se irá desplazando por todos los píxeles hasta cubrir el total de la imagen.

4.4. Bloque 1: Localización de la región de interés – ROI

A continuación se detallan todos los aspectos analizados para la implementación de los algoritmos utilizados. Pese a que los algoritmos están ya implementados en las librerías OpenCV, la dificultad de las aplicaciones de CV se encuentra en saber cómo utilizar estas herramientas para optimizar el resultado final, teniendo en cuenta los diferentes parámetros y configuraciones posibles para cada uno de los algoritmos, así como los tiempos de procesamiento que conllevan.

Para optimizar el tiempo de procesamiento de la aplicación, los procesos de segmentación por color y post–procesamiento se han realizado sobre una versión re–escalada de los frames. De esta manera, el número total de píxeles disminuye y las operaciones, que trabajan directamente sobre los píxeles, requieren menor coste computacional.

4.4.1 Segmentación por color

Como se ha mencionado anteriormente, el color es una característica fundamental para la distinción de los objetos, y en el caso de las señales aún más, ya que permite no solo la identificación de una señal con respecto a otros objetos si no también la diferenciación entre unos tipos de señales y otras. Para el desarrollo de este primer proyecto de detección de señales de tráfico del LSI se ha decidido enfocar la detección a las señales de advertencia o prohibición, de color rojo, de los entornos urbanos. Esto permite que la segmentación por color sea una solución viable, ya que el color rojo no abunda en el ámbito de la circulación. Para poder detectar otros tipos de señales sería necesario aumentar el rango de colores a detectar, azules, blancos, amarillos… y la segmentación por color deja de ser útil. Más adelante se discuten alternativas a este método para poder aumentar el alcance de señales a detectar.

4.4.1.A. Espacio de color

Para el desarrollo de la primera fase, la segmentación por color, es necesario entender las diferencias entre la percepción del color del ser humano y la percepción del color de una cámara/sensor y su respectivo módulo de procesamiento de imágenes.

El uso del color en el procesamiento de imágenes simplifica la identificación de los objetos de una imagen ya que es un descriptor fundamental para la diferenciación entre unos objetos y otros. Sin embargo, conlleva un aumento en la memoria utilizada y en el coste computacional. Trabajar con imágenes en escala de grises, erróneamente denominado blanco y negro, minimiza la memoria a utilizar y los tiempos y costes de computación.

Para la codificación de los colores se utilizan los espacios de color. Estos modelos describen la forma en la que los colores pueden representarse utilizando 3 o 4 valores o componentes de color, también conocidos como canales.

El espacio de color más conocido es el RGB (rojo, verde, azul), y esto se debe a que es el más intuitivo para el ojo humano. En este espacio de color, todos los colores que se pueden reconocer son una combinación de los tres colores primarios: R (rojo), G (verde)

y B (azul). Sin embargo, este espacio no es el más óptimo para un computador, ya que existe una alta correlación entre sus tres canales y la información que aporta individualmente cualquiera de ellos no es relevante en sí misma. La manera más eficiente para que un computador pueda diferenciar colores con un menor coste computacional e impacto temporal es utilizar un espacio de color en el que sus tres canales representen variables independientes [17].

Existen diferentes características a la hora de definir un color, tales como: matiz, brillo, saturación, intensidad, rojo/verde, azul/amarillo… Y la elección de qué tres (o cuatro) características usar para formar un espacio de color dependen de la aplicación, el formato de los datos de entrada, etc.

De entre la gran variedad de espacios de color existentes, cinco modelos principales: RGB, HSV/ HSL, CIE, YUV y CMYK más sus subdivisiones; la experiencia en el campo de la visión por computador nos aconseja utilizar el modelo HSV.

Este modelo, conocido también como HSB, utiliza el matiz, la saturación y el valor o brillo para describir los colores. Para poder comprender las especificaciones de un espacio de color que represente mejor la información estas tres características, es necesario entender primero que representa cada una de ellas individualmente.

El **tono** o matiz, define el color en sí (rojo, verde, azul, naranja, amarillo…) y se representa como el grado de un ángulo, acotado entre 0º y 360º, donde el rojo es 0º, el verde 120º y el azul 240º.

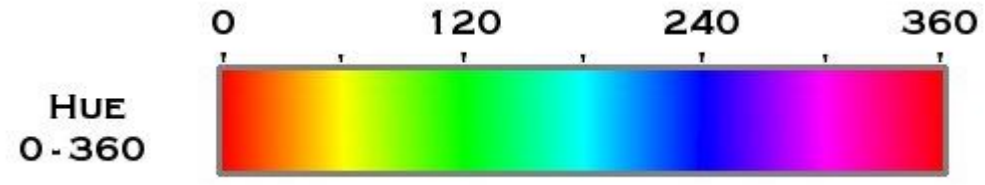

Figura 16: Representación gráfica del Matiz - Espacio de color HSV [18].

La **saturación**, representa la 'pureza' del tono del color, es decir, la cantidad de blanco que tiene un color. De esta manera, para un mismo matiz, por ejemplo 0º, un rojo intenso tendrá una saturación muy elevada (muy poco blanco) y un rojo claro será un color poco saturado, es decir, con mucho blanco.

Figura 17: Representación gráfica de la Saturación - Espacio de color HSV [18].

Finalmente, el **brillo**, representa la luminosidad u oscuridad relativa del color, suele expresarse en un porcentaje, siendo 0% la mayor oscuridad (negro) y 100% la mayor luminosidad (blanco).

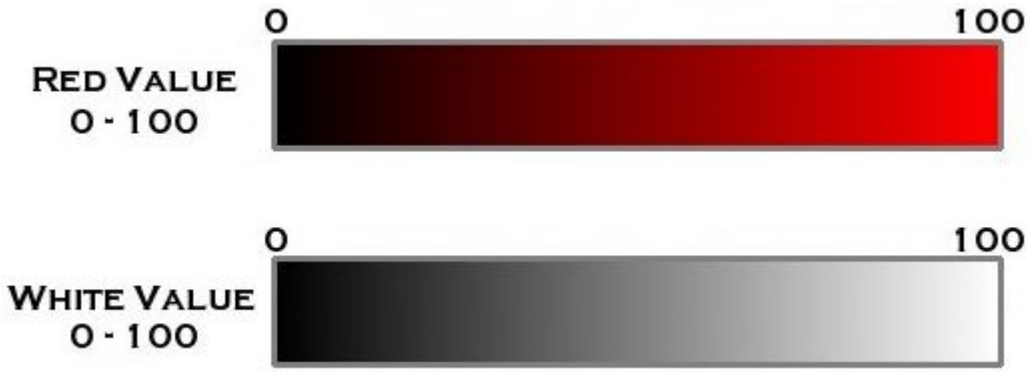

Figura 18: Representación gráfica del Brillo - Espacio de color HSV [18].

De esta manera, el negro puede tener cualquier matiz si el valor de V es cercano a 0 y de la misma manera, el color blanco puede tener cualquier matiz si la saturación es baja, es decir, cercana a 0.

Entendiendo bien los tres conceptos definidos en los párrafos anteriores, se puede construir un espacio de color en el que cada una de estas variables sea independiente, generando un modelo de tipo H [º], S [%], V [%].

Este espacio de color tiene una desventaja principal, y es que no es lineal. Sin embargo, los colores cercanos para el ojo humano también tienen valores cercanos en su codificación HSV. La conversión del modelo RGB al HSV se realiza de la siguiente manera:

$$R' = R/255 \qquad Cmax = \max(R', G', B')$$

$$G' = G/255 \qquad Cmin = \min(R', G', B') \qquad V = Cmax$$

$$B' = B/255 \qquad \Delta = Cmax - Cmin$$

$$H = \begin{cases} 0^\circ & \Delta = 0 \\ 60^\circ \times \left(\frac{G'-B'}{\Delta} mod 6\right) & , C_{max} = R' \\ 60^\circ \times \left(\frac{B'-R'}{\Delta} + 2\right) & , C_{max} = G' \\ 60^\circ \times \left(\frac{R'-G'}{\Delta} + 4\right) & , C_{max} = B' \end{cases} \qquad S = \begin{cases} 0 & , C_{max} = 0 \\ \frac{\Delta}{C_{max}} & , C_{max} \neq 0 \end{cases}$$

Figura 19: Ecuaciones de conversión de RBA a HSV.

Las siguientes imágenes muestran la representación gráfica de los espacios de color RGB y HSV. Como se puede apreciar, el espacio RGB se representa como un cubo en el que el origen de coordenadas es el color negro (0,0,0) y los tres ejes se corresponden cada uno con un color R (1,0,0), G (0,1,0) y B (0,0,1).

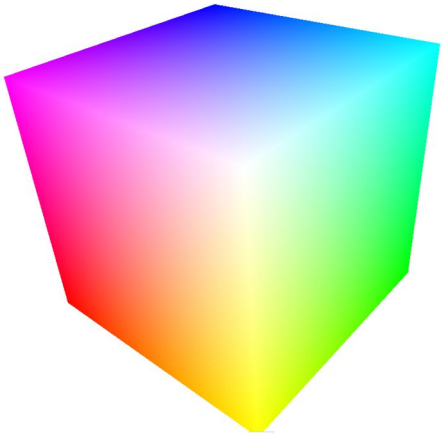

Figura 20: Cubo del Espacio de Color RGB [21].

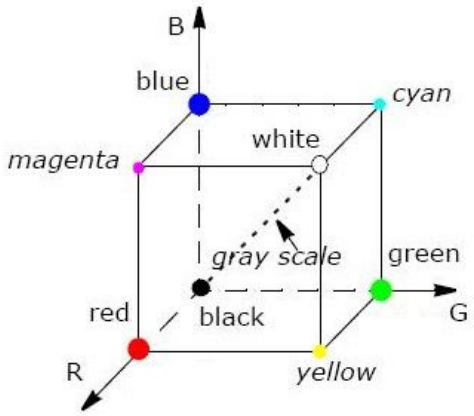

Figura 21: Colores del cubo RGB [20].

La representación del modelo HSV tiene, sin embargo, forma cónica, aunque también puede representarse sobre un cilindro, como se puede apreciar en las figuras 23 y 24. En ambos casos, el ángulo representa el matiz, el radio se corresponde con la saturación y la altura con el brillo o valor. En las secciones verticales del cono y cilindro se puede apreciar las diferencias entre uno y otro. Además, este modelo de color es representado también como una rueda de color, para facilitar el ajuste de los parámetro H, S, V en los programas que trabajan con este modelo de color, figura 25.

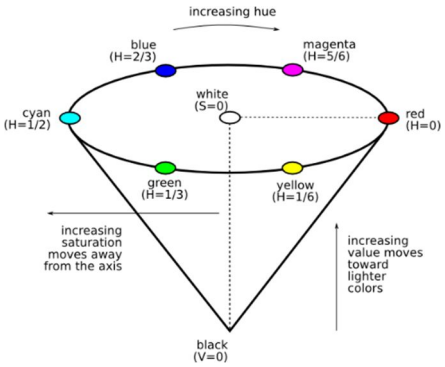

Figura 22: Colores cono HSV [21].

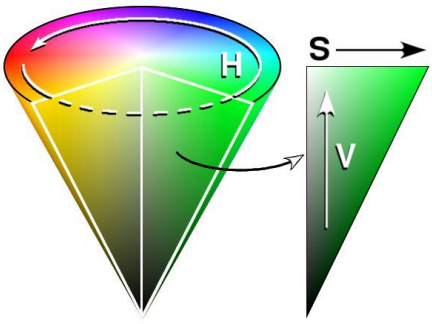

Figura 24: Cono HSV [22].

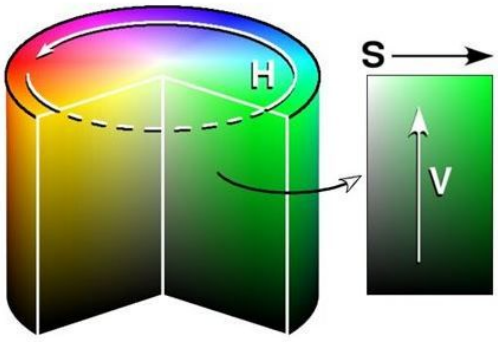

Figura 23: Cilindro HSV [23].

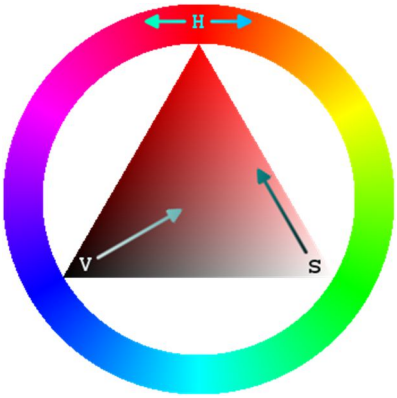

Figura 25: Color Wheel HSV [24].

4.4.1.B. Obtención de los colores de un objeto

Una vez decidido el espacio de color a utilizar, es fundamental encontrar los valores de H, S y V para los colores a detectar durante el funcionamiento de nuestra aplicación..

A estas alturas, el lector ya entiende la dificultad para especificar dichos valores, ya que para un mismo color (matiz) las condiciones de luminosidad y brillo exteriores, así como el material del objeto en cuestión y sus propiedades para reflejar la luz, afectan enormemente la percepción que tenemos de dicho color.

Para poder establecer los parámetros característicos del color de un objeto de una manera objetiva, se ha utilizado una aplicación de prueba que los desarrolladores de OpenCV incluyen en el paquete de distribución como ejemplo. Dicha aplicación permite, mediante el uso de la cámara, identificar los objetos de un mismo color de una imagen.

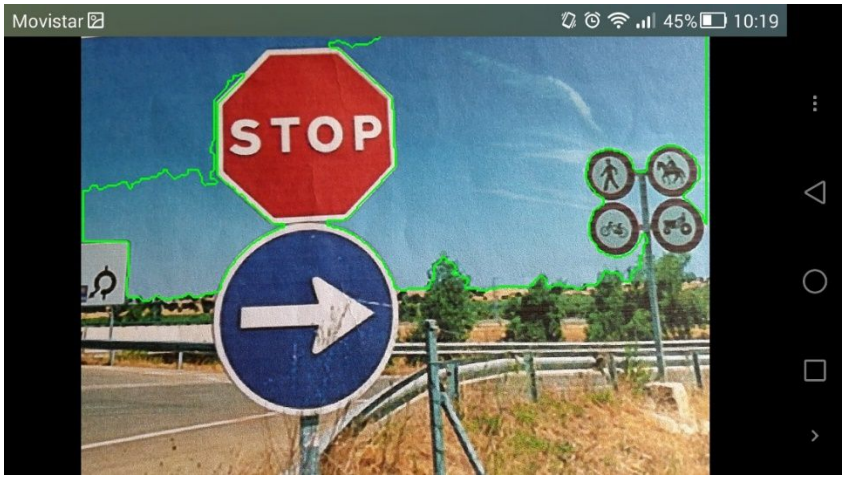

Figura 26: Captura de pantalla ColorBlob - Ejemplo OpenCV.

Con unas pequeñas modificaciones en el código de la aplicación, se consigue añadir una nueva funcionalidad para que además de mostrar los objetos del color seleccionado, aparezcan por pantalla también los parámetros H, S y V del color en cuestión. Esta modificación facilita enormemente la tarea de identificación de los colores de las señales de tráfico para acotar los rangos de búsqueda de color.

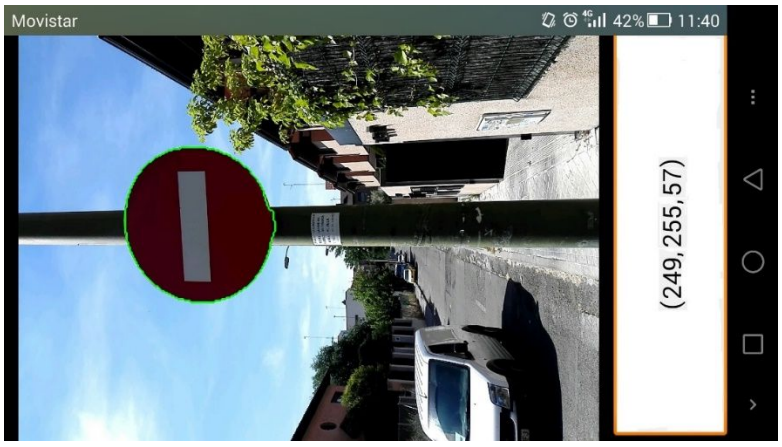

Figura 27: Captura de pantalla ColorBlob modificado para mostrar valores (H, S, V).

4.4.1.C. Implementación

Con el espacio y los rangos de color definidos, podemos comenzar a implementar los primeros pasos de la aplicación: la segmentación por color y el post-procesamiento de la imagen obtenida para mejorar los resultados.

Lo primero que necesitamos hacer es convertir el frame del espacio de color RGB al HSV. Para ello, utilizamos un método ya implementado en las librerías de OpenCV para Android, Imgproc.cvtColor(imagen, destino, BGR2HSV).

El resultado tras esta conversión puede apreciarse en las siguientes figuras. Es importante destacar que esta conversión no se realiza para mejorar la percepción del color al ojo humano, por lo que su representación gráfica no es significativa (no ayuda) para nosotros, pero sí que lo es para el computador.

Figura 28: Imagen 1 – Espacio de color RGB.

Figura 29: Imagen 1 - Conversión espacio de color HSV.

Figura 30: Imagen 2 – Espacio de color RGB.

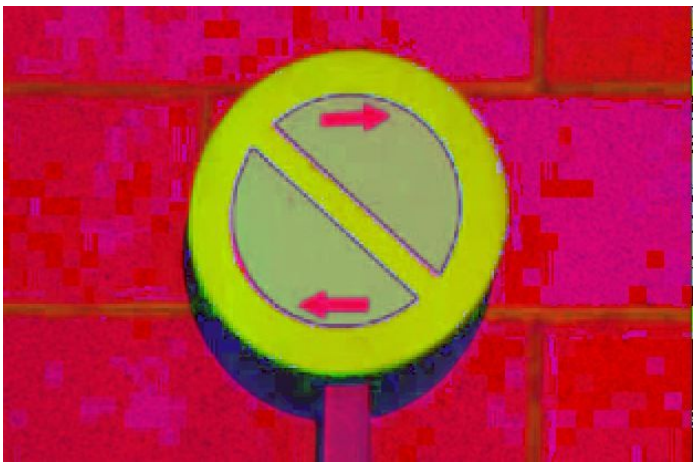

Figura 31: Imagen 2 - Conversión espacio de color HSV.

Con la nueva imagen en el espacio de color deseado, podemos identificar los objetos de la imagen en un rango de color específico gracias a otro método que nos proporcionan las librerías de OpenCV, Imgproc.inRange(imagen, color1, color2, destino).

Tras esta operación se obtiene una imagen binaria (solo dos colores, blanco y negro) en la que los objetos en el rango de color deseado, rojo en este caso, aparecen en blanco y todo lo demás en negro.

Figura 32: Imagen 1 - Resultado segmentación por color.

Figura 33: Imagen 2 - Resultado segmentación por color.

Como se puede apreciar en la imagen, y debido a las dificultades para percibir el color mencionado anteriormente, la imagen binaria presenta ruido y ciertas 'imperfecciones' que pueden dar problemas en los siguientes pasos de la aplicación. Para evitarlo, se realiza un post-procesamiento de la imagen para 'limpiarla'.

4.4.2. Post – Procesamiento

Las librerías de OpenCV también nos proporcionan las herramientas necesarias para realizar esta tarea:

El ruido que encontramos en estas imágenes se denomina ruido impulsional, es decir, píxeles ruidosos cuyo valor no tienen nada que ver con los píxeles circundantes.

Para eliminar este ruido se utiliza el filtro de la media, conocido como 'Average Blur' en las librerías de OpenCV. Esta operación utiliza un kernel de tamaño configurable (3x3, 7x7, 9x9…) que se desplaza sobre todos los píxeles de la imagen y da al pixel central el valor de la media de los pixeles del vecindario definido por el tamaño del kernel.

A continuación, se realizan una serie de operaciones morfológicas para eliminar los objetos pequeños suficientemente grandes como para no haber desaparecido con el filtro de ruido, y para rellenar los posibles huecos existentes en zonas que deberían ser de color blanco y no lo son.

Las operaciones utilizadas son la 'dilatación' y 'erosión' de los píxeles de una imagen, operaciones duales que consisten en expandir o degradar los píxeles de color blanco de la imagen. Ambas operaciones acabarían destruyendo la imagen si son aplicadas iterativamente. Sin embargo, si se combinan en procesos conocidos como apertura y cierre, pueden servir para eliminar objetos pequeños o rellenar huecos respectivamente [25].

De nuevo, el tamaño de las ventanas a utilizar para estos filtros es configurable y juega un papel decisivo en el resultado final de estas operaciones. El tamaño de estos filtros no solo afecta al resultado final sino también al tiempo de procesamiento, ya que filtros más grandes implican mayores cálculos.

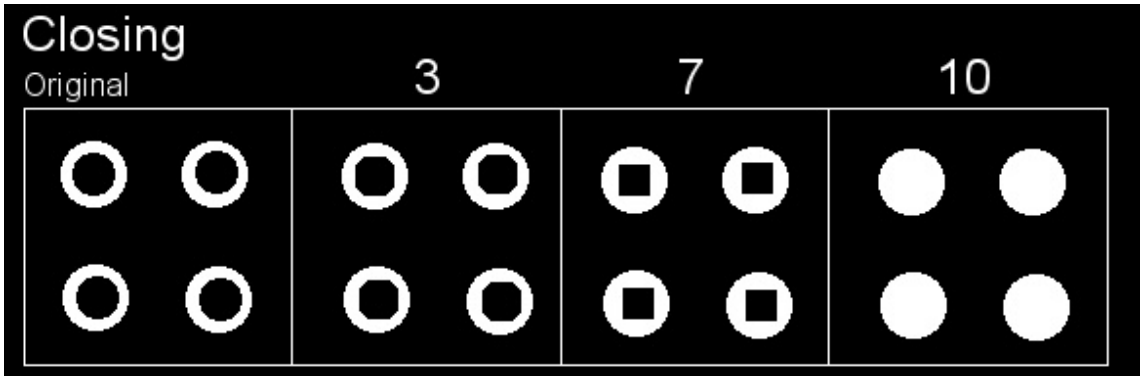

Figura 34: Ejemplo operación 'Closing' o Cierre.

Por tanto, el tamaño de los filtros debe optimizar el resultado de la operación y el tiempo de procesamiento, y no es sencillo encontrar un punto intermedio, como se detallará en el apartado de pruebas.

A continuación se muestran las imágenes con el resultado de estas operaciones, y como se puede comprobar, no todas las imágenes son igual de fácil de limpiar y en ocasiones ciertos objetos pequeños no son eliminados.

Figura 35: Imagen 1 - Resultado operaciones post – procesamiento.

Figura 36: Imagen 2 - Resultado operaciones post – procesamiento.

Finalmente, y para simplificar la tarea del siguiente paso, que es almacenar en un vector los puntos del contorno de los objetos segmentados, aplicamos un detector de bordes, también implementado en las librerías de OpenCV.

Como resultado, se obtiene una imagen similar a la anterior pero en este caso, únicamente los bordes de los objetos en blanco, y todo lo demás en negro.

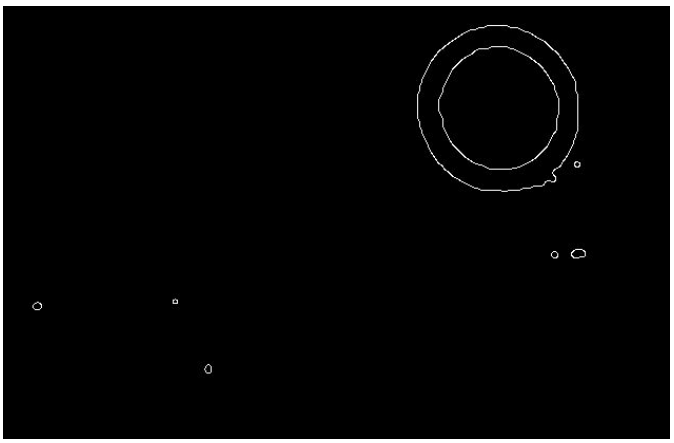

Figura 37: Imagen 1 - Resultado del detector de bordes – Canny.

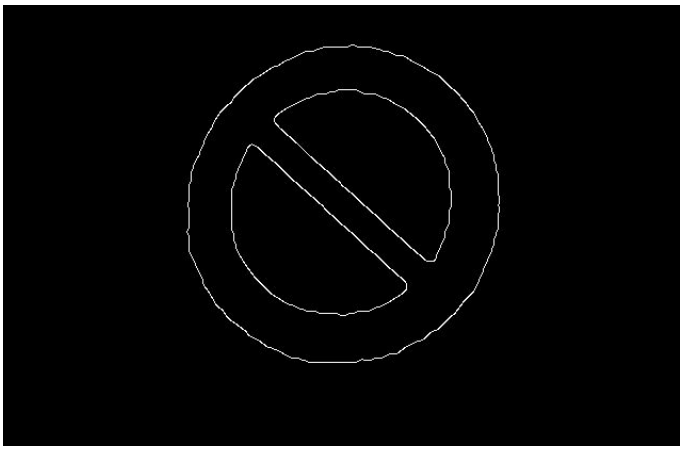

Figura 38: Imagen 2 - Resultado del detector de bordes – Canny.

4.4.3. Extracción y corrección de contornos

El siguiente paso consiste en encontrar y registrar los contornos de los objetos previamente segmentados para poder trabajar sobre/con ellos en los pasos posteriores. Para ello, utilizamos un algoritmo específico para la búsqueda de contornos. Una vez aplicado sobre la imagen, se obtiene un una lista que tiene tantas entradas como contornos se hayan detectado en la imagen, y cada elemento de la lista es de tipo MatofPoint, es decir, una matriz de puntos (x, y) que contiene todos los puntos que formal el contorno detectado.

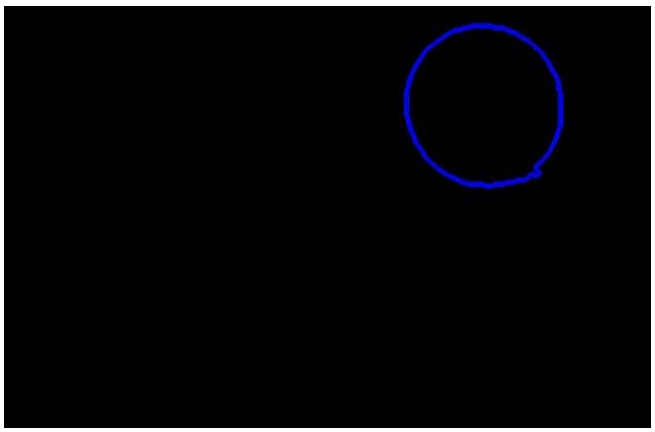

Figura 39: Imagen 1 - Resultado Extracción de Contornos.

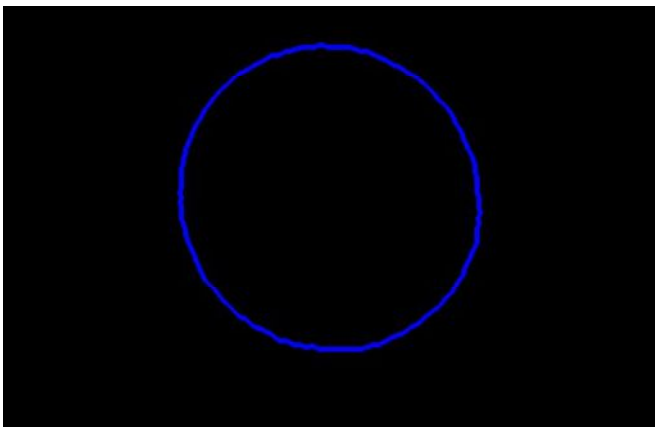

Figura 40: Imagen 2 - Resultado Extracción de Contornos.

Es importante resaltar las diferencias entre los resultados obtenidos en este paso, detección de contornos, y el anterior, resalte de bordes con el uso del algoritmo Canny. Con Canny el usuario puede identificar 'visualmente' el contorno del objeto, pero el computador no sabe dónde está. Tras encontrar los contornos, el computador tiene almacenado en un vector los puntos que forman ese contorno, y por tanto puede realizar las operaciones pertinentes con ellos.

Debido a los problemas relativos a la percepción de color y al posible ruido residual existente tras la segmentación de color y el post – procesamiento, en ocasiones, estos

contornos no siempre representen el objeto del que han sido obtenidos a la perfección y presenten formas extrañas, por lo que se suele buscar la envolvente convexa de cada uno de los contornos para optimizar su forma.

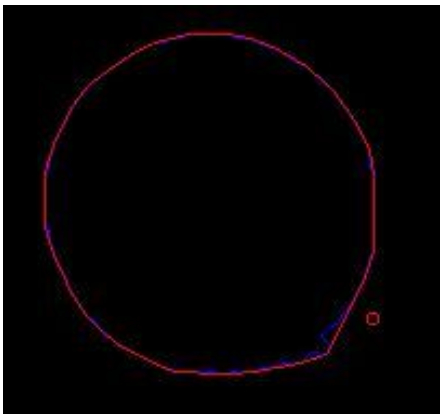

Figura 41: Imagen 1 - Comparación del contorno original, en azul, y su envolvente convexa, en rojo.

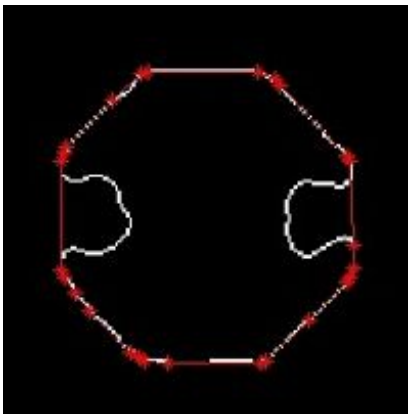

Figura 42: Comparación contorno y envolvente convexa [5].

Una vez que se han obtenido los contornos optimizados (envolvente convexa), el computador sabe con exactitud que pixeles de la imagen están dentro de dichos contornos y por tanto se puede saber que regiones de la imagen contienen objetos rojos y con formas razonables que tienen altas probabilidades de contener una señal de tráfico.

Para reducir el tiempo de procesamiento de los bloques de segmentación por color y post-procesamiento, todas las operaciones realizadas en estos módulos se hacen sobre una imagen re-escalada a un tamaño inferior, reduciendo la cantidad de píxeles de las imágenes y aumentando por tanto las velocidades de procesamiento.

4.4.4. Aproximación poligonal y detector de formas.

Como se mencionaba anteriormente, a partir de este punto, se puede trabajar sobre las regiones de interés para obtener más información sobre los objetos que contienen.

La primera idea que se tuvo era filtrar los contornos circulares mediante la Transformada de Hough, capaz de detectar círculos. Sin embargo, esta detección no era demasiado acertada y únicamente funcionaba con señales circulares. Por lo tanto se descartó el uso de este algoritmo y se implementó un simple detector de formas, aproximando los contornos a figuras poligonales planas y contando el número de lados para diferenciar entre las diferentes formas.

Lo primero que se hace para implementar el detector de formas es aproximar los contornos a formas poligonales. Aunque en teoría se podrían distinguir sin problemas las formas de las señales de tráfico, triángulos, cuadrados y círculos; en la práctica se ha demostrado que la diferenciación entre triángulos y cuadrados es bastante complicada. Esto se debe a que la envolvente convexa tiende a redondear las esquinas y al realizar la aproximación poligonal, suelen aparecer vértices o lados extra que 'convierten' un triángulo en un cuadrilátero dificultando su diferenciación con un cuadrado.

En las siguientes capturas de pantalla de la aplicación se pueden ver los contornos dibujados en amarillo y su aproximación poligonal en azul.

Figura 43: Aproximación Poligonal en señal triangular.

Como se puede apreciar en la figura superior, la aproximación poligonal no devuelve un triángulo. Esto se debe a los fallos arrastrados de los procesos anteriores.

Para los círculos, esta aproximación devuelve generalmente el octógono regular inscrito en la circunferencia que el contorno representa, siendo esta simplificación más que suficiente para distinguirlo de las otras formas geométricas de las señales cuadradas o triangulares.

Figura 44: Aproximación Poligonal en señal circular.

Figura 45: Aproximación Poligonal en señal octogonal.

4.4.5. Rectángulo asociado y extracción de ROI

Para finalizar el primer bloque, encargado de obtener la posición de las regiones de interés con altas probabilidades de contener una señal de tráfico, solo queda implementar un último paso, y encontrar el rectángulo que delimita cada uno de los contornos detectados en los pasos anteriores. Este proceso se aplica porque es más fácil trabajar con secciones de la imagen rectangulares o cuadradas, ya que mediante el uso de dos bucles 'for', es posible recorrer todos los píxeles de la sección; a diferencia de las complicaciones que implica recorrer todos los pixeles de una región de la imagen circular o triangular.

Para encontrar este 'rectángulo delimitador' (del inglés, bounding rectangle) podemos utilizar otro método ya implementado en las OpenCV, capaz de encontrar el rectángulo de mínima área que contiene un conjunto de puntos, los puntos del contorno en este caso [19]. Para la aplicación que nos ocupa, no buscamos el rectángulo de mínima área si no un rectángulo cuyos lados sean paralelos a los bordes de la imagen (vertical y horizontal), para simplificar, de nuevo, la manera de recorrer los píxeles de la región en las operaciones posteriores.

En la figura 46 pueden apreciarse las diferencias entre el rectángulo de área mínima, en rojo, y el rectángulo limitante, en verde.

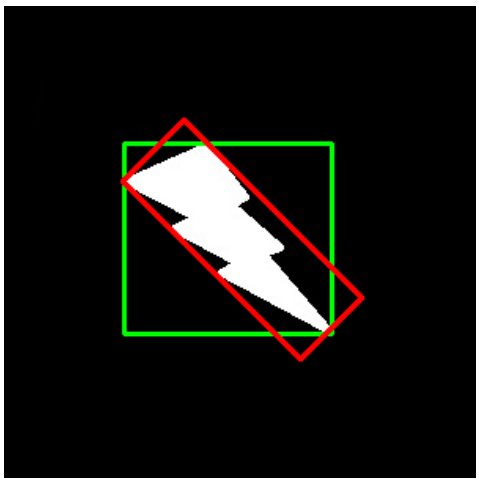

Figura 46: Comparación rectángulo área mínima, en rojo, y rectángulo limitante, en verde [27].

4.5. Bloque 2: Localización de la región de interés – ROI

4.5.1. Detección, extracción y comparación de características

Lo primero que hay que entender para entender el funcionamiento de este bloque es el concepto de 'características' (del inglés 'features') de una imagen. Una característica es un punto de interés de la imagen, es decir, con información relevante para resolver ciertas operaciones. Las características pueden ser estructuras, como puntos, esquinas, bordes… o resultados de operaciones de procesamiento aplicadas a una región parcial o sobre el total de la imagen, como máximos, mínimos, crestas, dirección del gradiente... [28].

Para que una característica pueda ser considerada buena ('good feature'), es decir, útil, es necesario que cumpla ciertos requisitos: ser consistente en varias imágenes de la misma escena u objeto, ser invariable a ciertas transformaciones (rotación, escala, translación…) y ser inmune al ruido entre otros [25].

La localización de las 'features' de una imagen o región es obtenida mediante un detector de características o 'feature detector' [30].

La información que describe dichas características o puntos de interés se almacena en vectores multidimensionales conocidos como descriptores, y es obtenida mediante un extractor de descriptores o 'descriptor extractor'. Existen diferentes algoritmos para obtener los descriptores, con diferentes maneras de describir y codificar la información (valores reales, binarios…) [31].

El último paso es comparar las dos imágenes o regiones. Para ello se utilizan los comparadores de descriptores o 'descriptor matchers'. Como su propio nombre adelanta, estos algoritmos comparan los descriptores utilizados para describir las regiones a comparar. Como resultado se obtienen parejas de descriptores (uno de cada imagen) con mayor similitud entre ellos [32].

Cada pareja está compuesta por dos vectores, uno de la primera imagen y el otro, es el vector (descriptor) de la segunda imagen con el índice de similitud más alto de todos los descriptores de la segunda imagen. De esta manera, se concluye que es una operación en la que el orden de las imágenes es determinante.

Existen diferentes maneras de calcular la similitud entre dos vectores, como por ejemplo 'La Distancia Euclídea' para valores reales o 'La distancia de Hamming' para valores binarios. De esta manera se introduce el atributo de distancia: el índice de similitud que tienen dos vectores de una pareja.

De la misma manera que las librerías de OpenCV proveen una clase de tipo Mat para almacenar imágenes, ofrece también una serie de 'hijas' de dicha clase que permiten

almacenar características, MatofKeyPoints o las parejas de descriptores asociadas o 'matches' en MatOfDMatch.

El siguiente diagrama muestra el proceso de detección, extracción y comparación de características partiendo de las dos imágenes iniciales. Es frecuente que la imagen que se utiliza como modelo se encuentre ya optimizada para este proceso y no requiera ningún tipo de procesamiento adicional.

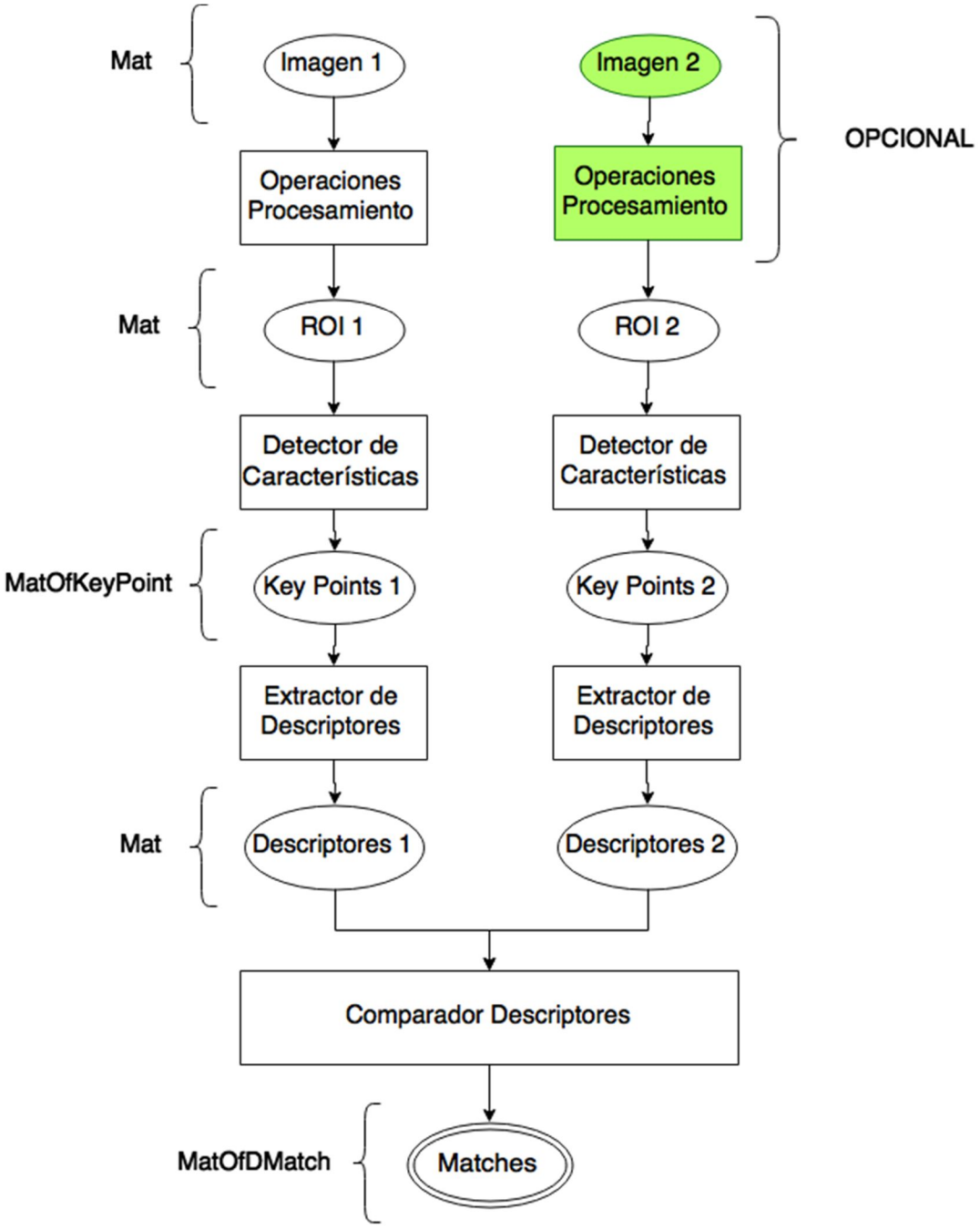

Figura 47: Diagrama de flujo del bloque de detección, extracción y comparación de características.

Como ya se ha mencionado anteriormente, los comparadores de descriptores asocian en parejas un descriptor de la primera imagen con su descriptor más parecido de la segunda imagen, pero eso no quiere decir que sean realmente similares. Por ello, el siguiente paso consiste en obtener lo que se conoce como 'good matches' o buenas parejas de descriptores. La distinción entre buenas o malas matches la establece el programador, y es la cantidad de 'good matches' la que permite decidir si el resultado de la operación es positivo, hay 'matching' o no.

Existen diferentes métodos para obtener las 'good matches', basándose la mayoría en la distancia entre ambos descriptores de la pareja. En general, se establece una distancia de 'matching' y se compara individualmente con la distancia entre cada una de las parejas. Si la distancia de la pareja es menor que la distancia de 'matching' esa pareja se añade a una nueva lista de 'good matches'. El criterio para la elección de la distancia de 'matching' depende de la aplicación, de los algoritmos utilizados, etc.

En este caso, se calcula distancia media entre todas las parejas de descriptores y se multiplica por un coeficiente. La ventaja de este método es que no se establece una distancia fija, si no que varía en cada comparación en función de la distancia media. A su vez, con tan solo modificar el coeficiente se varía la distancia de 'matching' por lo que se repetir el proceso iterativamente modificando dicho coeficiente en caso de que no haya 'matching' positivo.

Una vez se ha obtenido la lista de 'good matches', es necesario establecer un criterio para tomar la decisión de si hay 'matching' entre las dos imágenes o no. Como la cantidad de matches detectadas entre diferentes parejas de imágenes oscila bastante, desde 20 matches hasta 500 o incluso 1000, no se puede establecer una cantidad de 'good matches' a partir de la cual se puede dar un resultado positivo en la comparación. Se ha decidido por tanto establecer como criterio la relación entre el número de 'good matches' y el número total de 'matches'.

Finalmente, modificando los valores de la distancia de 'matching' (variando el coeficiente) y del ratio mínimo de 'good matches' frente a 'good matches' se puede regular el comportamiento del detector y ajustarlo en función de los algoritmos utilizados.

Los valores de ambas variables serán analizados más adelante en el apartado de entrenamiento y en el apartado de pruebas.

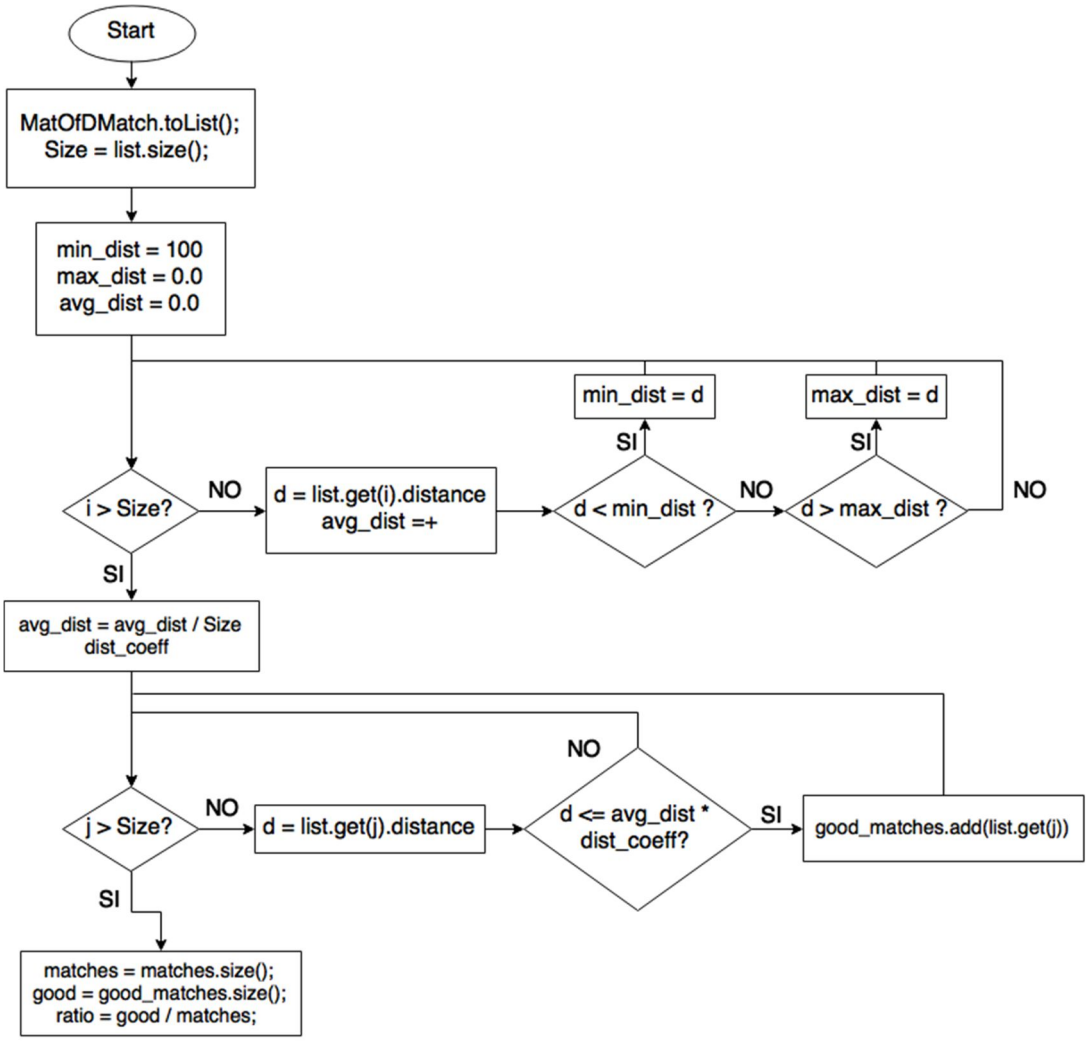

Figura 48: Diagrama de flujo del proceso de selección de 'Good matches'.

Las librerías de OpenCV ponen al alcance del desarrollador una gran variedad de algoritmos para la detección, extracción y comparación de características y descriptores:

Feature Detector
FAST
STAR
SIFT
SURF
ORB
BRISK
MSER
GFTT
HARRIS
Dense
SimpleBlob

Descriptor Extractor
SIFT
SURF
BRIEK
BRISK
ORB
FREAK

Descriptor Matcher
Brute Force (BF)
BF – L1
BF – L2
BF – SL2
BF – Hamming
Flann Based

Tabla 1: Algoritmos disponibles en las librerías OpenCV [30] [31] [32].

Ante tal cantidad de algoritmos, se ha realizado una breve búsqueda sobre las combinaciones más populares de 'detector | extractor | comparador' [33] [34] [35] [36], con la intención de implementar únicamente las combinaciones que mejores resultados ofrecen para el caso que nos ocupa, y son:

Detector	Extractor	Comparador
FAST / SURF	SURF	Flann Based
FAST / SIFT	SIFT	Flann Based
FAST / ORB	ORB	Brute Force
FAST / ORB	BRIEF	Brute Force
FAST / SURF	FREAK	Brute Force

Tabla 2: Combinaciones más populares de detector, extractor y comparador.

La principal clasificación para los extractores y comparadores de descriptores los divide en dos grupos: descriptores de tipo 'float' y descriptores de tipo 'uchar' [37]. Los descriptores más utilizados clasificados en estos dos grupos son:

Descriptores 'float':
SIFT
SURF

Descriptores 'uchar':
ORB
BRIEF

Tabla 3: Clasificación descriptores.

Y sus comparadores correspondientes:

Comparadores 'uchar':
Brute Force – Hamming
Brute Force – HammingLUT
Flann Based with LSH index

Comparadores 'float':
Flann Based
Brute Force – L2
Brute Force – L1
Brute Force – SL2

Tabla 4: Clasificación comparadores.

Esta división no afecta a los detectores de características ya que la información que almacenan es la localización de los puntos de interés, coordenadas de los píxeles de la imagen, y no necesitan codificación.

Los algoritmos SURF (Speed Up Robust Features) y SIFT (Scale Invariant Feature Transformation) son los más recomendados por los programadores que utilizan OpenCV, sin embargo, estos algoritmos están patentados por lo que no pueden usarse con fines comerciales ni están incluidos en el paquete de OpenCV para Android. Por tanto, estos dos algoritmos no serán utilizados en el desarrollo de este proyecto, ni como detectores de características ni como extractores de descriptores.

Pese a haber realizado una pequeña búsqueda sobre los algoritmos más utilizados para la detección y extracción de características, se ha realizado una prueba con cada uno de los algoritmos disponibles para comprobar su rendimiento. Se dispone de 9 detectores de características, 4 extractores de descriptores y 6 comparadores de descriptores.

Las siguientes tablas muestran las combinaciones de detector, extractor y comparador así como el número de 'matches' encontradas con cada una de ellas, el número de 'good matches' y los FPS de la imagen de salida. Este último dato nos aporta información sobre el tiempo de procesamiento de cada combinación, teniendo un ratio de FPS muy bajo los que mayor coste computacional requieren. Es importante tener en cuenta que su valor oscila incluso para un mismo grupo de algoritmos y que es por tanto meramente orientativo, simplemente para descartar los algoritmos más lentos.

Pruebas con Detectores de Características:

Detector	Extractor	Comparador	# maches	# good	FPS
FAST	ORB	BF	558	131	6.48
ORB	ORB	BF	235	19	5.89
GFTT	ORB	BF	760	168	4.52
STAR	ORB	BF	1	1	5.26
MSER	ORB	BF	233	6	3.84
DENSE	ORB	BF	5628	2854	2.75
SIMPLEBLOB	ORB	BF	0	0	5.71
HARRIS	ORB	BF	Error	Error	5.46
STAR	ORB	BF	1	1	3.16

Tabla 5: Pruebas con detectores.

Pruebas con Extractores de Descriptores:

Detector	Extractor	Comparador	# maches	# good	FPS
FAST	FREAK	BF	587	184	4.57
ORB	FREAK	BF	30	0	5.26
GFTT	FREAK	BF	822	243	3.13
FAST	BRISK	BF	638	221	2.49
ORB	BRISK	BF	184	5	2.87
FAST	BRIEF	BF	517	117	6.24
ORB	BRIEF	BF	235	25	6.13

Tabla 6: Pruebas con extractores.

Pruebas con Comparadores de Descriptores:

Detector	Extractor	Comparador	# maches	# good	FPS
ORB	ORB	BF	235	19	5.76
ORB	ORB	BF_HAMMING	235	19	6.19
ORB	ORB	BF_HAMMINGLUT	235	19	6.05
ORB	ORB	FlannBased	ERROR	ERROR	ERROR
ORB	ORB	BF_L1	235	19	5.71
ORB	ORB	BF_SL2	235	19	5.89

Tabla 7: Pruebas con comparadores

Además de estos algoritmos, existen varios adaptadores que se utilizan para cambiar el comportamiento de los detectores y extractores [38].

El adaptador 'dynamic', utilizado para detectores de características se utiliza para ajustar dinámicamente los parámetros de este algoritmo hasta que un número deseado de características es encontrado. Las aplicaciones en las que se aplica este adaptador generalmente son aquellas en las que se busca un número exacto de características en un

conjunto de imágenes relacionadas temporalmente, como 'streams' de video o fotografías de tipo 'time lapse'.

En principio este adaptador no aporta ninguna ventaja en esta aplicación, pero se realizarán también pruebas con este adaptador para comprobar su rendimiento.

El adaptador 'grid', utilizado también para detectores de características, se encarga de dividir la imagen en celdas (de ahí su nombre), y detectar las características de cada una de ellas.

El último adaptador disponible para los detectores de características es el adaptador 'pyramid'. En este caso, se modifica el detector para que busque características en múltiples niveles de una pirámide Gaussiana. Como se mencionó previamente en la definición de característica, es conveniente que sean inmunes a las transformaciones de escala. Este adaptador se utiliza para detectores que no re-escalan las características automáticamente.

Los extractores de descriptores también pueden modificar su comportamiento con el uso del adaptador 'opponent'. Este adaptador se utiliza para calcular los descriptores en el espacio de color opuesto, por lo que no es interesante en este caso.

Pruebas con adaptadores de detectores y extractores:

Feature Detector	ORB Extractor		BRIEF Extractor		OpponentORB Extractor		OpponentBRIEF Extractor	
	Matches	Good	Matches	Good	Matches	Good	Matches	Good
FAST	558	131	517	117	558	99	571	50
DynamicFast	276	64	227	53	276	39	277	9
GridFAST	478	105	489	91	478	73	489	33
PyramidFast	591	131	604	117	591	99	604	50
ORB	235	19	235	25	235	9	235	3
DynamicORB	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
GridORB	176	5	176	19	176	1	176	1
PyramidORB	262	20	262	25	262	9	262	3

Tabla 8: Pruebas con adaptadores

Tras estas pruebas con los diferentes algoritmos y modificadores para la detección, extracción y comparación de características, se llega a la conclusión de que los algoritmos que mejores resultados ofrecen son ORB y FAST para la detección de características, ORB para la extracción de descriptores y comparadores de tipo Brute Force.

Con los algoritmos a utilizar ya decididos, el siguiente paso es crear una base de datos con diferentes imágenes modelo y organizarla de tal manera que simplifique el proceso y reduzca el tiempo de procesamiento.

4.5.2 Base de Datos

La primera de las imágenes que entra al comparador es la ROI obtenida al final del bloque uno. Es una región con altas probabilidades de contener una señal, pero es necesario verificar que efectivamente se trata de una señal antes de confirmar la detección. Para ello, esta ROI es comparada con un conjunto de imágenes almacenadas en la base de datos.

La base de datos no necesita estar alojada en un servidor, ya que las señales de tráfico no varían con el tiempo y las imágenes utilizadas tienen un tamaño suficientemente pequeño como para que la base de datos no sea muy pesada.

La Normativa Española para la señalización vertical cuenta con alrededor de 250 (es difícil obtener un número exacto ya que existen diferentes variaciones de una misma señal en varios casos). Esta aplicación está enfocada a la detección de las señales de advertencia o peligro y obligación, es decir, señales con fondo blanco y marco de color rojo, reduciendo el número de imágenes a unas 100.

Aunque la idea inicial del proyecto era detectar y diferenciar todas estas señales de advertencia, según se ha ido avanzando con el proyecto y realizando pruebas, se ha comprobado que era una idea demasiado ambiciosa. Las limitaciones del dispositivo y de los algoritmos implementados para la detección de las señales no permiten trabajar con un espectro tan amplio de señales a detectar.

Se ha decidido organizar la base de datos por la forma de las señales. Si se pudiesen detectar señales de otros colores (futuras ampliaciones), habría que tener en cuenta el color también, pero no es necesario para este caso.

Por tanto, se separan las señales en circulares y no circulares, que es la distinción que se realiza en el detector de formas implementado en el primer bloque. La señal de STOP, octogonal, forma parte del grupo de las señales circulares. En el otro grupo, señales no circulares, nos encontramos con las señales triangulares, ya que no hay señales con marco rojo con otras formas. En función de la salida del detector de formas, se comparará la ROI con un grupo u otro.

A la hora de diseñar una base de datos es necesario tener en cuenta ciertos factores. En este caso, y para facilitar la tarea de los detectores y extractores de características, es fundamental que todas las señales de las imágenes de la base de datos tengan el mismo tamaño. Esto es importante porque aunque en teoría las características deben ser invariantes a variaciones en escala, los algoritmos de comparación de descriptores no siempre relacionan correctamente descriptores de características con distinto tamaño.

Los contenidos de la base de datos se han ido modificando en función de los resultados obtenidos tras las pruebas realizadas con los diferentes algoritmos implementados.

Ha sido bastante tedioso localizar bancos de imágenes de señales de tráfico para formar la base de datos. Se han encontrado dos páginas web con imágenes de señales de tráfico españolas [39] [40], pero aun así, no aparecen todas las señales existentes. En la página oficial del Ministerio de Fomento hay información disponible referente a la normativa española para la señalización vertical, pero no una base de datos con las señales.

4.5.3 Entrenamiento

Tras comprobar que los FPS de la salida se reducen cuando entra en acción el segundo bloque, como más delante se detallará en el apartado de pruebas limitaciones, se reafirman los problemas que implica ser capaz de detectar una gran variedad de señales, como ya anunciaban las aplicaciones encontradas en Google Play, capaces de detectar únicamente un tipo de señal.

Se ha decidido por tanto reducir la cantidad de señales a detectar a cuatro:

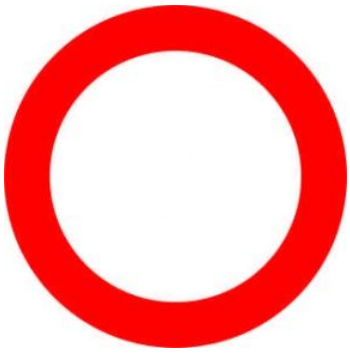

Figura 49: Señal r100 - Circulación Prohibida.

Figura 51: Señal R2 – STOP.

Figura 50: Señal r101 - Entrada Prohibida / Dirección Prohibida.

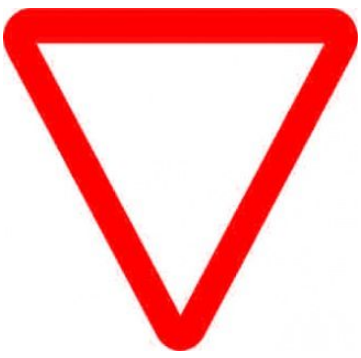

Figura 52: Señal R1 - Ceda el Paso.

Para que el detector funcione correctamente, es necesario optimizar los valores de distancia de 'matching' y del ratio de 'good matches' frente a las 'matches' totales.

Como ya se mencionó previamente, el algoritmo para la comparación de descriptores tiene en cuenta el orden de las imágenes, por lo que los resultados de comparar la Imagen1 con la Imagen2 no son los mismos que al comparar la Imagen2 con la Imagen1. De esta manera, se presentan dos alternativas, comparar las imágenes de la base de datos en primer lugar y como segunda imagen la ROI, o al revés, primero la ROI y después las imágenes de la base de datos.

Antes de integrar el comparador en la aplicación y de probarlo con las ROI obtenidas en el bloque uno, se ha implementado en una aplicación independiente un comparador que no utiliza las imágenes obtenidas a través de la cámara, si no imágenes almacenadas en la memoria del dispositivo.

Para ello, se han cogido dos señales reales para cada uno de estos cuatro modelos y se han comparado por parejas con distintos valores para el coeficiente multiplicador de la distancia y el ratio.

En este caso, se ha utilizado la imagen real como imagen 1 y las imágenes de la base de datos como imagen 2. Lo que se consigue al hacerlo así es que el número de 'matches' totales sea siempre el mismo, correspondiente al número de características y descriptores de la imagen real, y que lo que varíe sea el número de 'good matches' en función de la distancia con cada conjunto de características y descriptores obtenidos en la segunda imagen.

Se ha implementado la combinación de 'FAST', 'ORB' y 'Brute Force' para los algoritmos de detección de características, extracción de descriptores y comparador de descriptores respectivamente. Se han elegido estos algoritmos por ser una de las combinaciones más utilizadas y comprobarse que era una de las que mejor resultados ofrecía en las pruebas anteriores.

A continuación se muestran los resultados obtenidos tras el proceso de entrenamiento:

Direción prohibida – r101:

	BASE DE DATOS											
Real r101	Entrada Prohibida			Circulacion Prohibida			Ceda el paso			Stop		
FAST + ORB + BF	r101			r100			R1			R2		
d = coeff * avg_dist	IGUALES			DIFERENTES			DIFERENTES			DIFERENTES		
	Matches	Good	Ratio	Matches	Good	Ratio	Matches	Good	Ratio	Matches	Good	Ratio
coeff = 0,50	32	4	0,125	32	1	0,031	32	1	0,031	32	1	0,031
coeff = 0,60	32	5	0,156	32	2	0,063	32	2	0,063	32	1	0,031
coeff = 0,70	32	6	0,188	32	4	0,125	32	5	0,156	32	5	0,156
coeff = 0,80	32	7	0,219	32	7	0,219	32	7	0,219	32	8	0,250
coeff = 0,90	32	10	0,313	32	11	0,344	32	9	0,281	32	12	0,375
coeff = 1,00	32	10	0,313	32	16	0,500	32	16	0,500	32	16	0,500
coeff = 1,10	32	16	0,500	32	19	0,594	32	21	0,656	32	21	0,656
coeff = 1,25	32	28	0,875	32	25	0,781	32	27	0,844	32	27	0,844
coeff = 1,50	32	31	0,969	32	31	0,969	32	31	0,969	32	30	0,938

Tabla 9: Entrenamiento con la señal – Entrada Prohibida.

Circulación prohibida – r100:

	BASE DE DATOS											
real r100	Entrada Prohibida			Circulacion Prohibida			Ceda el paso			Stop		
FAST + ORB + BF	r101			r100			R1			R2		
d = coeff * avg_dist	DIFERENTES			IGUALES			DIFERENTES			DIFERENTES		
	Matches	Good	Ratio	Matches	Good	Ratio	Matches	Good	Ratio	Matches	Good	Ratio
coeff = 0,50	56	1	0,018	56	1	0,018	56	0	0,000	56	0	0,000
coeff = 0,60	56	2	0,036	56	5	0,089	56	1	0,018	56	4	0,071
coeff = 0,70	56	5	0,089	56	8	0,143	56	6	0,107	56	6	0,107
coeff = 0,80	56	10	0,179	56	15	0,268	56	15	0,268	56	12	0,214
coeff = 0,90	56	16	0,286	56	21	0,375	56	20	0,357	56	18	0,321
coeff = 1,00	56	20	0,357	56	31	0,554	56	31	0,554	56	27	0,482
coeff = 1,10	56	39	0,696	56	42	0,750	56	41	0,732	56	37	0,661
coeff = 1,25	56	54	0,964	56	56	1,000	56	56	1,000	56	55	0,982
coeff = 1,50	56	56	1,000	56	56	1,000	56	56	1,000	56	56	1,000

Tabla 10: Entrenamiento con la señal r100 – Circulación Prohibida.

Ceda el paso – R1:

	BASE DE DATOS											
real R1	Entrada Prohibida			Circulacion Prohibida			Ceda el paso			Stop		
FAST + ORB + BF	r101			r100			R1			R2		
d = coeff * avg_dist	DIFERENTES			DIFERENTES			IGUALES			DIFERENTES		
	Matches	Good	Ratio	Matches	Good	Ratio	Matches	Good	Ratio	Matches	Good	Ratio
coeff = 0,50	304	6	0,020	304	4	0,013	304	3	0,010	304	3	0,010
coeff = 0,60	304	13	0,043	304	13	0,043	304	9	0,030	304	7	0,023
coeff = 0,70	304	27	0,089	304	20	0,066	304	21	0,069	304	22	0,072
coeff = 0,80	304	52	0,171	304	49	0,161	304	50	0,164	304	53	0,174
coeff = 0,90	304	90	0,296	304	97	0,319	304	98	0,322	304	91	0,299
coeff = 1,00	304	146	0,480	304	142	0,467	304	149	0,490	304	151	0,497
coeff = 1,10	304	183	0,602	304	189	0,622	304	195	0,641	304	201	0,661
coeff = 1,25	304	223	0,734	304	231	0,760	304	239	0,786	304	242	0,796
coeff = 1,50	304	265	0,872	304	274	0,901	304	286	0,941	304	282	0,928

Tabla 11: Entrenamiento con la señal R1 – Ceda el paso.

STOP – R2

	BASE DE DATOS											
real R2	Entrada Prohibida			Circulacion Prohibida			Ceda el paso			Stop		
FAST + ORB + BF	r101			r100			R1			R2		
d = coeff * avg_dist	DIFERENTES			DIFERENTES			DIFERENTES			IGUALES		
	Matches	Good	Ratio	Matches	Good	Ratio	Matches	Good	Ratio	Matches	Good	Ratio
coeff = 0,50	914	71	0,078	914	63	0,069	914	8	0,009	**914**	**53**	**0,058**
coeff = 0,60	914	129	0,141	914	114	0,125	914	21	0,023	**914**	**127**	**0,139**
coeff = 0,70	914	189	0,207	914	226	0,247	914	103	0,113	**914**	**195**	**0,213**
coeff = 0,80	914	240	0,263	914	311	0,340	914	259	0,283	**914**	**272**	**0,298**
coeff = 0,90	914	315	0,345	914	391	0,428	914	392	0,429	**914**	**348**	**0,381**
coeff = 1,00	914	399	0,437	914	465	0,509	914	484	0,530	**914**	**441**	**0,482**
coeff = 1,10	914	457	0,500	914	534	0,584	914	547	0,598	**914**	**526**	**0,575**
coeff = 1,25	914	574	0,628	914	619	0,677	914	611	0,668	**914**	**603**	**0,660**
coeff = 1,50	914	678	0,742	914	773	0,846	914	784	0,858	**914**	**759**	**0,830**

Tabla 12: Entrenamiento con señal R2 – STOP.

En las tablas de resultados se puede comprobar que las diferencias entre la cantidad de 'good matches' que hay cuando las señales a comparar son iguales que las del modelo y cuando son diferentes, no varían demasiado. Aun así, para los coeficientes más bajos, 0'5, 0'6, 0'7 y 0'8; la cantidad de 'good matches' es ligeramente mayor cuando ambas imágenes son iguales, excepto en el caso de la señal de STOP.

De esta manera, si se establece un ratio de 'matching' para cada uno de estos coeficientes, el detector es capaz de distinguir entre las diferentes señales con una cantidad de falsos negativos razonables.

Sin embargo, este funcionamiento es tan solo aceptable cuando se prueba el comparador con las imágenes utilizadas para el entrenamiento, ya que, como se verá en el apartado de pruebas, al utilizar otras imágenes reales o la ROI obtenida en el bloque uno, el detector no es capaz de distinguir entre las diferentes señales, aunque sí que permite distinguir entre señales en general y objetos rojos que hayan pasado la segmentación del primer bloque.

4.5.4. Integración en la aplicación

Una vez el bloque de validación ha sido implementado por separado, es hora de integrarlo en la aplicación principal y ver cómo responde ante las nuevas circunstancias.

El funcionamiento del comparador es el mismo que el desarrollado en el apartado anterior, pero en este caso, la primera imagen que entra al comparador es la sección de interés segmentada mediante los pasos anteriores.

Se aplica el detector de características primero, seguido del extractor de descriptores a ambas imágenes, y acto seguido, estos descriptores entran al comparador. Además del proceso explicado anteriormente para seleccionar las 'good matches', el proceso de comparación es iterativo en este caso, modificando el valor del coeficiente de la distancia de 'matching' y del ratio mínimo para dar un resultado positivo en la comparación. De esta manera, si con una distancia pequeña de 'matching' el ratio no es suficientemente elevado como para decidir si hay 'matching' o no, se repite el proceso con una nueva distancia de 'matching' mayor.

De las señales elegidas para formar la base de datos, tres de ellas pertenecen al grupo de señales circulares y una de ellas al grupo de señales no circulares.

Tras el detector de formas, se procede a comparar la imagen con el grupo de señales circular si la forma detectada es un círculo, y con el grupo de señales no circulares en caso contrario.

El siguiente diagrama muestra el funcionamiento del segundo bloque, con las diferentes iteraciones para comparar las tres señales del grupo de señales circulares de la base de datos individualmente y los diferentes valores para la distancia y el ratio.

El funcionamiento en el caso de que la señal no tenga forma circular es el mismo, evitando las iteraciones necesarias para comparar varias imágenes.

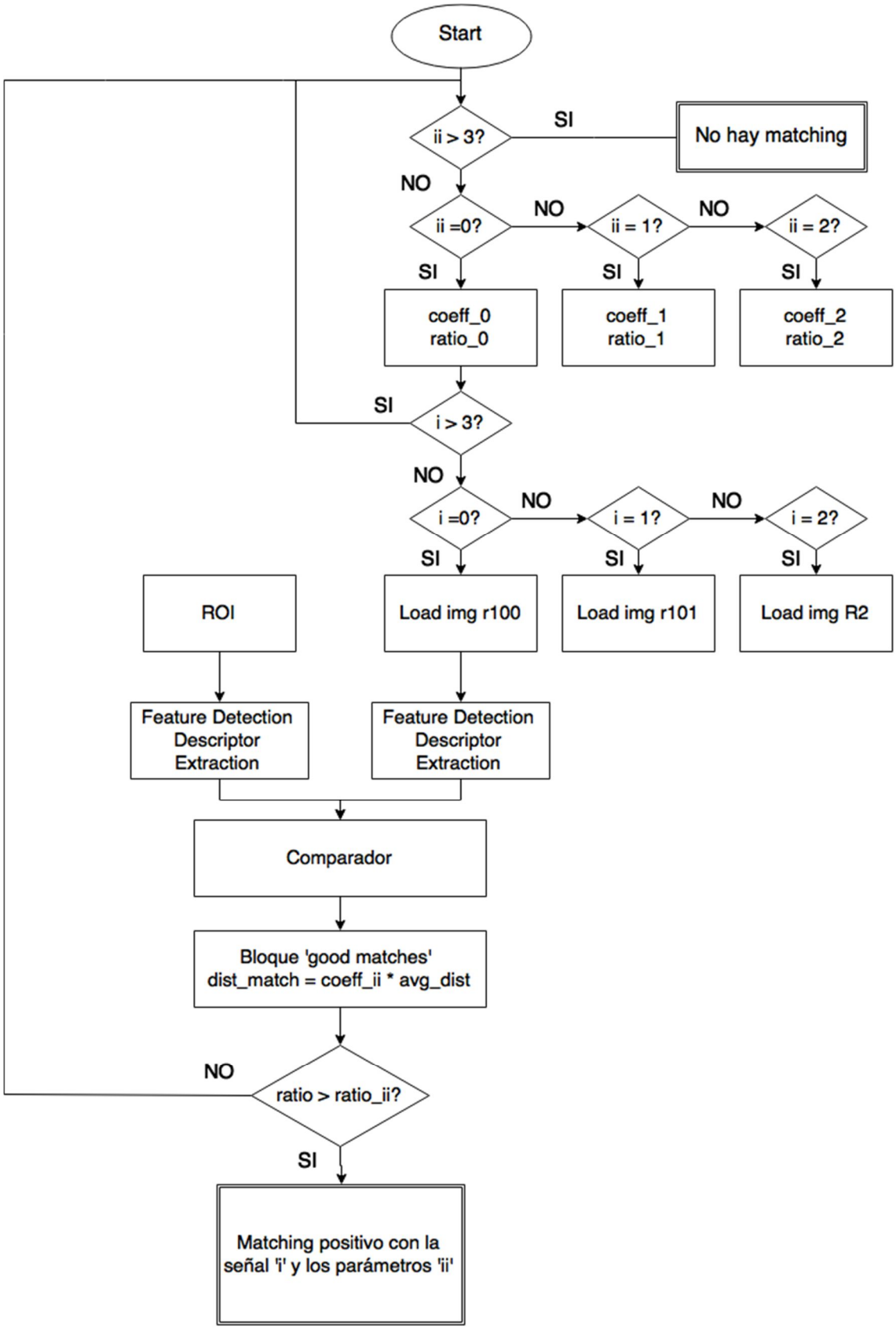

Figura 53: Diagrama de flujo con el funcionamiento de la comparación con la base de datos.

4.5.5. Actividades y clases implementadas

La aplicación para la detección de señales cuenta con una actividad principal y una clase con los métodos y algoritmos mencionados en el apartado anterior.

La interfaz de esta actividad principal es directamente la imagen de salida tras las operaciones y algoritmos utilizados para la detección de las señales. Si en el interior del método onCameraFrame(inputFrame) no se realiza ninguna operación, la salida coincide con el frame de entrada, como si de la interfaz de una cámara se tratase. En este caso, la salida se corresponde con el frame de entrada sobre el que se señalizan, dibujando un rectángulo, las señales detectadas.

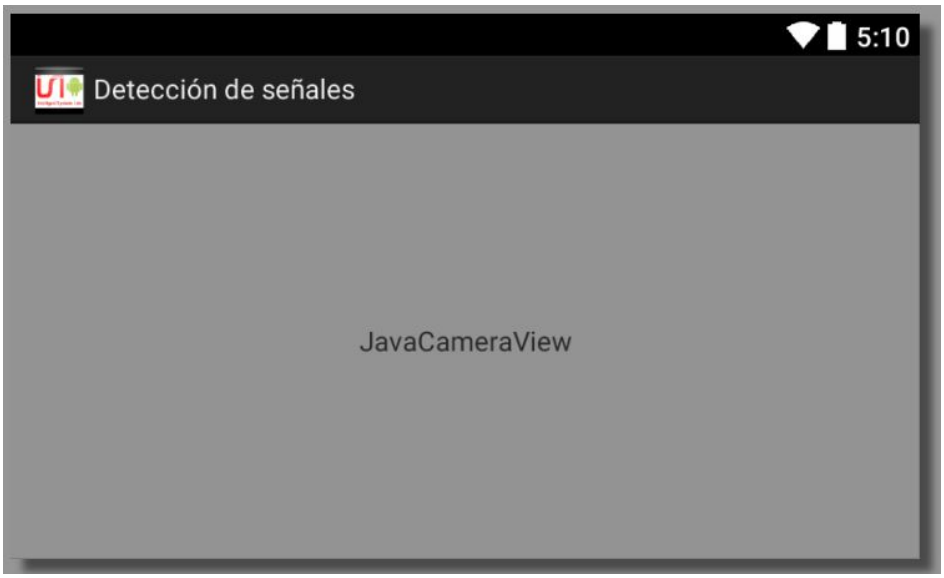

Figura 54: Interfaz actividad principal - JavaCameraView.

La actividad principal contiene en la cabecera las importaciones de todas las librerías utilizadas, seguidas de los métodos de Android necesarios de cualquier actividad así como los métodos adicionales que las librerías de OpenCV requieren para funcionar

Además, dentro del método onCameraFrame( ) se encuentran las llamadas a los métodos que se encuentran en la clase TrafficSignLocation.java y las variables necesarias para el funcionamiento de dichos métodos.

En esta actividad se ha implementado además un simple 'algoritmo' para mejorar la velocidad de procesamiento. Para ello, se ha decidido no procesar todos los frames de entrada, y hacerlo solo en uno de cada cuatro. Esto tiene sus ventajas e inconvenientes, que serán analizados en el apartado de pruebas.

```
private BaseLoaderCallback mLoaderCallback = new BaseLoaderCallback(this) {
private JavaCameraView mOpenCvCameraView;


protected void onCreate(Bundle savedInstanceState) {

public boolean onCreateOptionsMenu(Menu menu) {

public boolean onOptionsItemSelected(MenuItem item) {

public void onResume() {

public void onDestroy() {


public void onCameraViewStarted(int width, int height) {

public void onCameraViewStopped() {

public Mat onCameraFrame(CvCameraViewFrame inputFrame) {
```

Figura 55. Contenido Actividad Principal.

Por otro lado nos encontramos con la clase en la que se han implementado los métodos y algoritmos utilizados en los diferentes pasos mencionados anteriormente, segmentación por color, post – procesamiento, extracción de contornos, detección y extracción de características…

```
public class TrafficSignLocation {
    protected static final String TAG = null;

    public static Mat colorSegmentation(Mat input, int pickColor)

    static List<MatOfPoint> hullmop = new ArrayList<MatOfPoint>();

    public static List<MatOfPoint> getHull() {

    public static Mat ContourExtraction(Mat input, Mat rgba)

    public static Rect boundingRect = null;
    public static boolean rectExists = false;

    public static void setboundingRectState (boolean state) {

    public static Rect getboundingRect() {

    public static Mat Rectangulos(Mat rgba, Mat contornos) {  // SIN FEATURES

    public static Mat RectangulosandFeatures(Mat rgba, Mat contornos) {

    public static int signalMatch = 20;
    public static boolean detectedSignal = false;

    public static int getsignalMatch() {
    public static boolean getdetectedSignal() {

    public static void setsignalMatch(int a) {
    public static void setdetectedSignal(boolean a) {

    public static Mat featureDetectionCircles(Mat ROI_input, Mat rgba) {
}
```

Figura 56: Contenido Clase métodos.

5. Evaluación: Pruebas y Limitaciones

A continuación se muestran los resultados de las pruebas de los diferentes procesos que forman parte del total de la aplicación. Se comentarán los resultados de los distintos sub-apartados primero y después el funcionamiento de ambos bloques: localización de la ROI y verificación.

5.1. Limitaciones Externas

Antes de comenzar con las pruebas, se van a discutir las principales limitaciones ajenas a la implementación del proyecto, tales como los diferentes dispositivos que pueden ejecutar la aplicación, la velocidad del vehículo en el que se encuentra el dispositivo ejecutando la aplicación o el estado de las señales de tráfico.

En cuanto a la velocidad, es evidente que a mayor velocidad, menor es el tiempo que las señales están al alcance de la cámara y menores las probabilidades de que la señal sea detectada. Además, las altas velocidades pueden causar también problemas a la hora de enfocar la imagen, dificultando en gran medida o incluso impidiendo la detección. La tabla que se muestra a continuación.

Se ha estimado el alcance de la detección a través de la cámara en 15m (pudiendo variar en función de las especificaciones del dispositivo). A mayor distancia, la señal aparece en la imagen pero no tiene un tamaño significativo para ser detectada. Basándonos en esto y en la velocidad del vehículo en el que se encuentre el dispositivo, obtenemos con facilidad el tiempo y número de frames que hay para detectar una imagen, como se puede apreciar en la tabla número 13.

De esta manera, a 50km/h, la señal estará al alcance de la cámara alrededor de un segundo. Si se tiene en cuenta que la aplicación funciona a 5 FPS, nos encontramos con tan solo 5 frames para procesar y detectar esa imagen. Si en ninguno de esos frames se realiza una segmentación por color correcta, esa señal no será detectada.

		Procesamiento	
VELOCIDAD		Tiempo (s)	Número de frames
Km/h	m/s		
10	2,78	5,40	27,00
20	5,56	2,70	13,50
30	8,33	1,80	9,00
40	11,11	1,35	6,75
50	13,89	1,08	5,40
60	16,67	0,90	4,50
70	19,44	0,77	3,86
80	22,22	0,68	3,38
90	25,00	0,60	3,00
100	27,78	0,54	2,70
110	30,56	0,49	2,45
120	33,33	0,45	2,25

Tabla 13: Relación entre la velocidad y el tiempo de procesamiento.

Este es uno de los motivos por el que se ha decidido limitar el ámbito de actuación de esta aplicación al entorno urbano, ya que por ciudad, la máxima velocidad permitida es de 50km/h.

Por otro lado, tenemos los inconvenientes que implica no procesar todos los frames, como se ha mencionado en el apartado de implementación. Al reducir el número de frames procesados a uno de cada cinco, se aumentan los FPS de la salida a costa de perder la información existente en esos frames y haciendo que si, por ejemplo, a 50km/h la aplicación disponía de 5 frames para procesar una señal, solo uno de esos cinco se utilice, reduciendo las probabilidades de obtener una buena segmentación por color.

Otro factor a tener en cuenta es la capacidad de procesamiento del dispositivo y la calidad de su cámara. Una de las ventajas de la plataforma Android es que está presente en una amplia variedad de teléfonos y tabletas, cada uno con diferentes especificaciones, variando estas enormemente entre los diferentes dispositivos. Podemos encontrar núcleos de procesamiento de tipo single-core, dual-core, quad-core y octa-core, además de funcionar a diferentes frecuencias, desde 1,2GH hasta 2.3GHz. Por otro lado nos encontramos también con diferencias en la memoria RAM del teléfono, desde 512Mb o 1Gb en los dispositivos más antiguos o de gama más baja hasta 2 o 3 Gb en los más punteros del mercado. De manera similar, las especificaciones de las cámaras también varían, pasando de los 5MP de las cámaras de los dispositivos más limitados a los 16MP de las cámaras de los smartphones de última generación.

Para realizar las pruebas de la aplicación se ha utilizado un dispositivo Huawei Honor 6, con una cámara de 12MP, un procesado octa-core y 3 Gb de RAM.

Por otro lado, el estado de las señales juega un papel fundamental en el proceso de la detección, y aunque la apariencia de las señales de tráfico varía principalmente por las condiciones lumínicas, esta puede verse también afectada por:

- Condiciones ambientales (lluvia, nieve, niebla…).
- Vandalismo (señales dobladas o inclinadas, graffitis o dibujos sobre las señales…).
- Manchas o descoloración debido a la antigüedad de la señal.
- Sombras y reflejos.
- Oclusión parcial.

Estas variaciones en la apariencia de la señal, causan fallos en el proceso de segmentación por color que se arrastran durante los pasos sucesivos y que impiden la detección de algunas señales.

Para poder seguir el funcionamiento de los diferentes procesos de la aplicación, se ha añadido un menú provisional que permite ver las imágenes de salida de cada uno de ellos, y que no estará presente en la versión final de la aplicación. Además, este menú contiene un botón que permite activar y desactivar el funcionamiento del segundo bloque, para comparar el rendimiento de la aplicación en los dos casos. Para indicar el estado de este segundo bloque (activo o inactivo) se dibuja un rectángulo rojo en la esquina superior izquierda de la pantalla cuando se encuentra activo.

5.2. Pruebas y resultados del primer bloque

Uno de los apartados que más problemas ha dado durante todo el desarrollo de la aplicación ha sido el de segmentación por color. Además, los fallos que se producen en este paso son arrastrados a los siguientes procesos, impidiendo el correcto funcionamiento de la aplicación.

Uno de los primeros problemas que surgieron y que retrasó el avance de la implementación de este bloque fue el hecho de que en el espacio de color HSV, los colores rojos estén comprendidos entre los ángulos 0º – 15º y 345º – 360º, o en su codificación binaria (8 bits), entre los valores 0 – 10 y 245 – 255. El problema surge porque el método utilizado para establecer los rangos de color deseado no permite el paso del 255 al 0, por lo que no se puede segmentar todo el espectro de rojos con un único rango entre 245 –

10. De esta manera, es necesario utilizar dos rangos diferentes (valores 0 – 10 y 245 – 255) y sumar ambas segmentaciones. El alumno sabía que las herramientas de OpenCV permitían hacer esta operación pero debido a la escasa documentación existente para Java, resultó complicado encontrar la manera de hacerlo.

Tras comprobar experimentalmente la dificultad para obtener el color de una señal real, que amén de variar en función de las condiciones lumínicas puede variar también en función de la perspectiva de la cámara con respecto a la señal, por cuestiones de reflejos, sombras, brillos… Se comenzó a buscar una manera para determinar los rangos de manera más objetiva, y no mediante el método de prueba y error que se estaba utilizando.

Analizando una de las aplicaciones ejemplo de OpenCV para Android, denominada '*ColorBlob detection*', que permite seleccionar un objeto de la imagen y resaltar ese objeto y todos los que tengan el mismo color; se llegó a la conclusión de que se podía utilizar esta aplicación para obtener el color de las señales de tráfico. De esta manera, y como ya se ha explicado previamente en el apartado de implementación, se puede obtener con mayor precisión el color de las señales.

Además, las zonas muy oscuras de la imagen presentan otro inconveniente, ya que para el espacio de color HSV, los colores negros no dependen tanto del matiz del color, si no de su saturación y luminosidad, por lo que en ciertas ocasiones estas áreas de la imagen pasan este primer proceso de segmentación por color.

Tras múltiples pruebas con diferentes valores, se decidió optimizar este proceso para detectar la mayor cantidad de señales posibles en diferentes condiciones lumínicas, con la intención de eliminar estas áreas oscuras que no eran rojas en realidad en los siguientes pasos.

En el siguiente paso, el post – procesamiento, nos enfrentamos al problema de tratar de optimizar las operaciones de filtrado. Como ya se ha comentado previamente, si estos filtros son demasiado agresivos, tratando de eliminar el ruido y los objetos segmentados en el paso anterior que no se corresponden con color rojo, degradan en exceso los objetos (posibles señales) que sí que son de color rojo. Por otro lado, si estos filtros son demasiado blandos, no se consigue eliminar correctamente los objetos indeseados. Además, hay que tener en cuenta el tiempo de procesamiento que esas operaciones requieren y su impacto en los FPS de la salida. Encontrar un punto óptimo entre el buen funcionamiento del filtro en diferentes condiciones lumínicas y los tiempos de cómputo no ha sido tarea fácil.

De nuevo, se decide dejar parte de la eliminación de algunos objetos no deseados para los siguientes pasos, optimizando así el funcionamiento de la segmentación por color y su post – procesamiento.

El siguiente paso, la extracción de contornos, facilita bastante la eliminación de los elementos indeseados, ya que una vez que el computador tiene localizados los objetos, se puede trabajar con su área y su forma para seleccionar los que pasan al bloque de eliminación.

De esta manera, los objetos demasiado pequeños para ser relevantes, o los objetos que tienen una proporción entre altura y anchura que no encajan con la silueta de una señal, generalmente inscribibles en un cuadrado, son descartados.

Además la aproximación poligonal de los contornos permite diferenciar entre las diferentes formas de las posibles señales, circulares y triangulares. Aunque el detector de formas implementado es capaz de distinguir perfectamente entre cualquier tipo de figuran, teniendo en cuenta el número de lados para hacer la distinción; el ruido proveniente de los pasos anteriores suele hacer que la aproximación poligonal genere más de tres lados.

A continuación se exponen varias capturas de pantalla que reflejan los problemas y soluciones mencionadas en los párrafos anteriores. (Pruebas contraluz, pruebas oscuridad, falsos positivos, oclusión).

Conversión a espacio de color HSV:

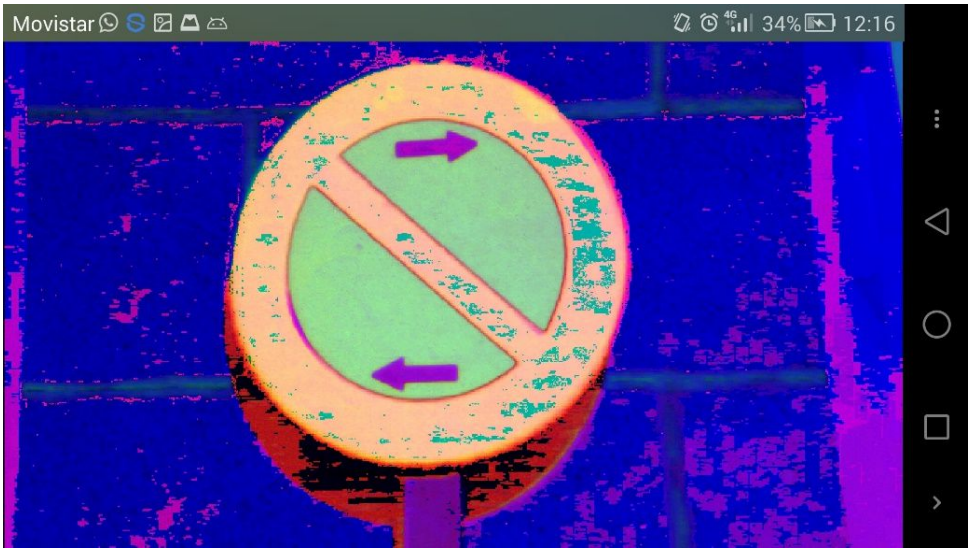

Figura 57: Resultados conversión HSV.

Segmentación de los colores rojos:

Figura 58: Resultados segmentación por color.

Post – procesamiento de la imagen obtenida tras la segmentación:

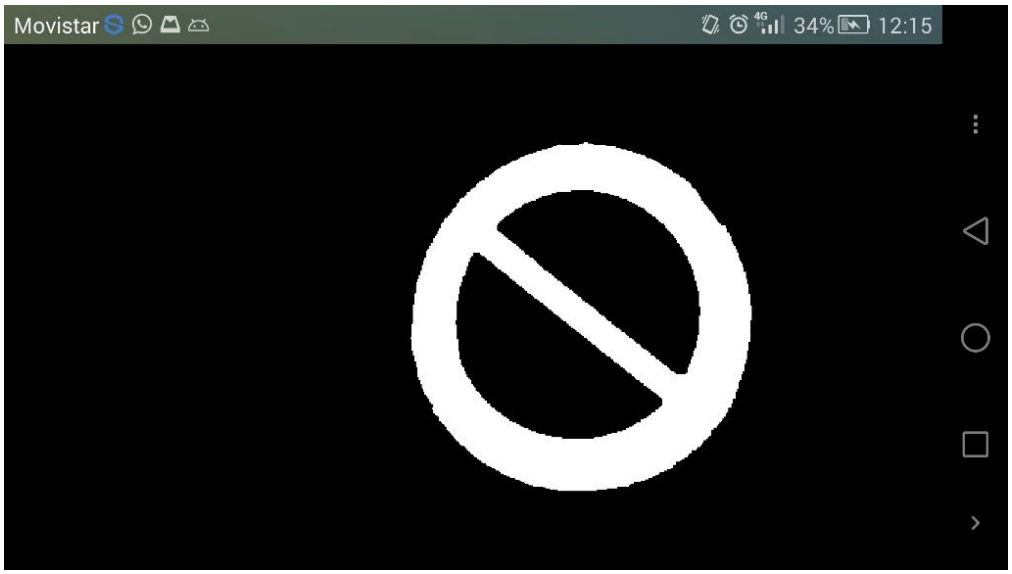

Figura 59: Resultados post – procesamiento.

Resultado de la extracción de contornos y la envolvente convexa:

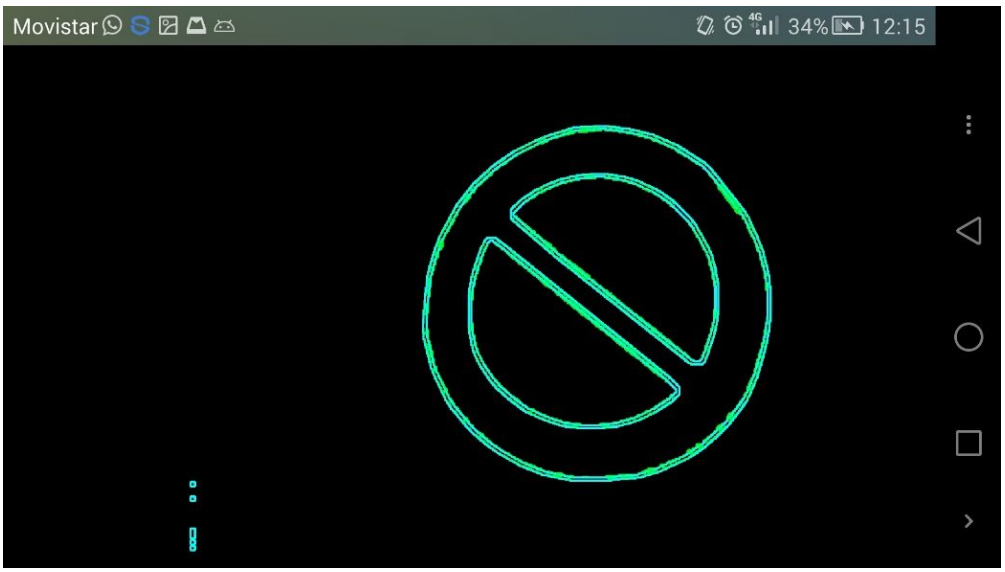

Figura 60: Resultados de la extracción y correción de contornos.

Funcionamiento de la aplicación en condiciones de contraluz:

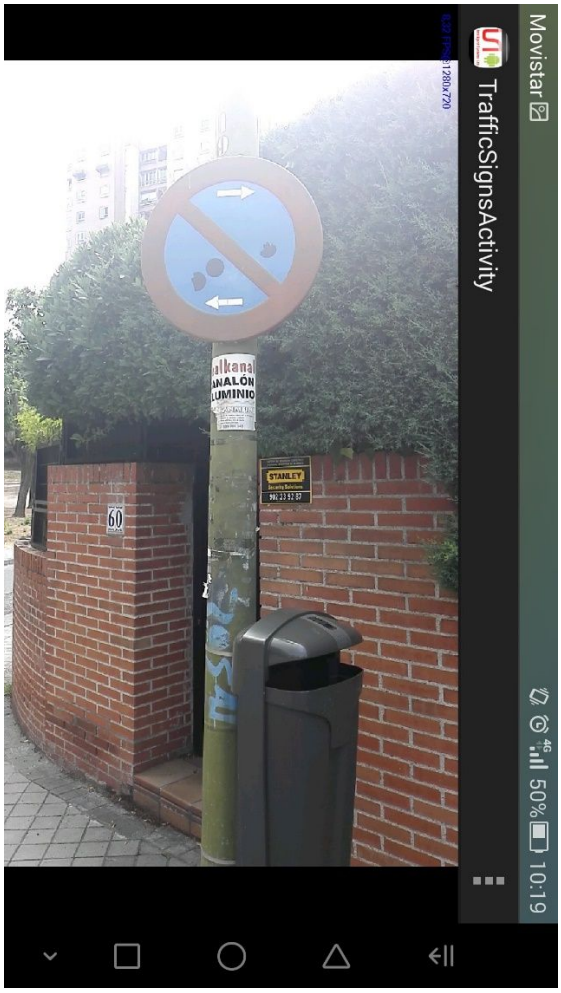

Figura 61: Fallos por condiciones lumínicas – CONTRALUZ.

Figura 62: ColorBlob Detection en contraluz.

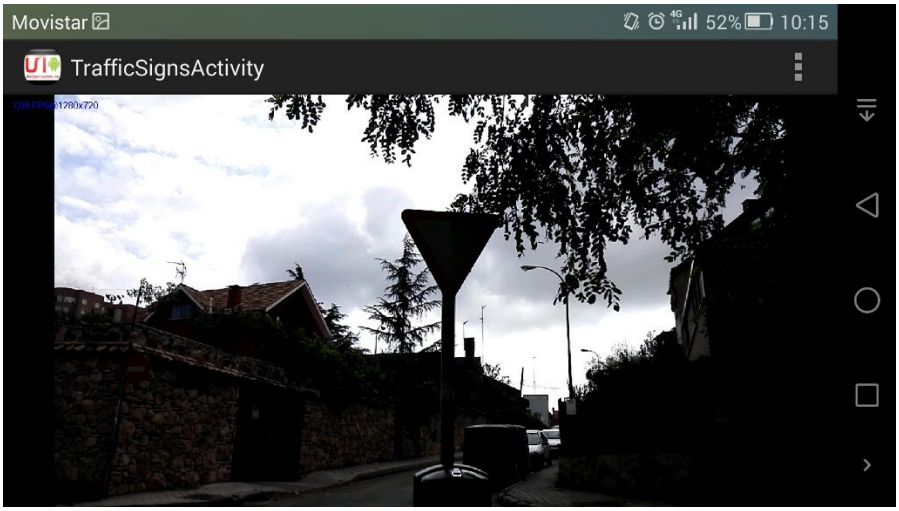

Figura 63: Fallos por condiciones lumínicas – CONTRALUZ.

Resultados en condiciones lumínicas demasiado oscuras:

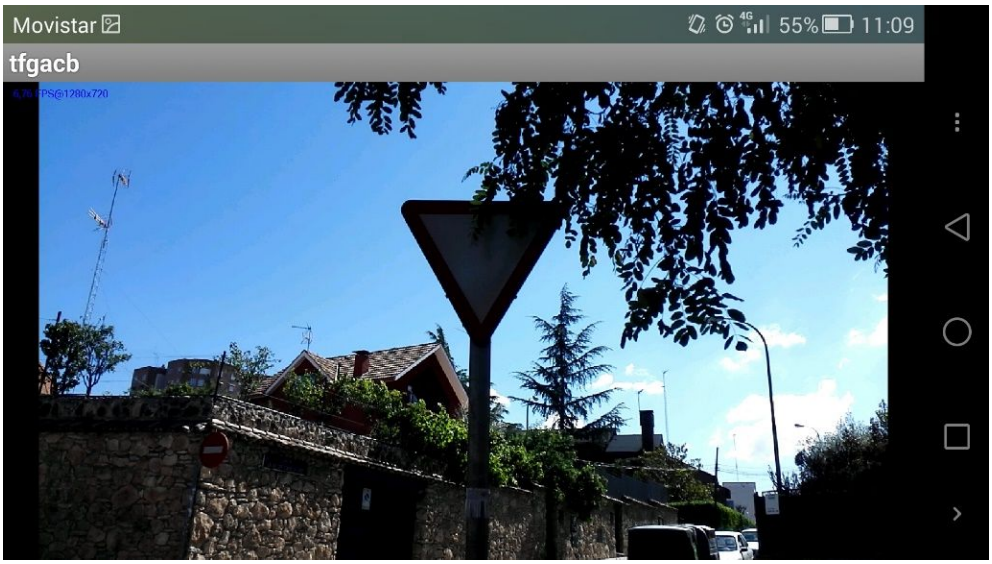

Figura 64: Fallos por condiciones lumínicas – OSCURIDAD.

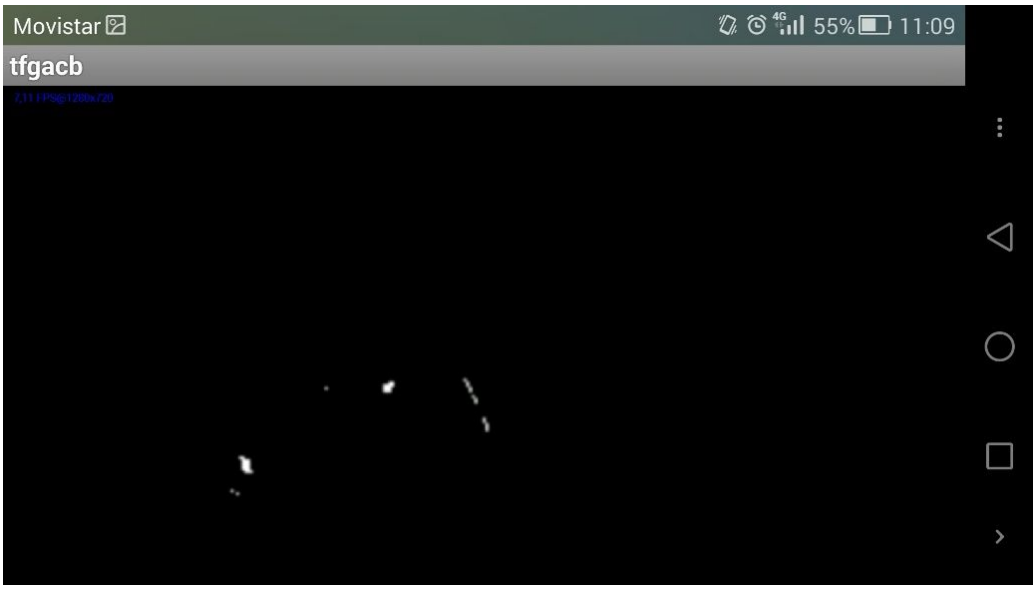

Figura 65: Fallos por condiciones lumínicas – OSCURIDADl. Salida del apartado de post – procesamiento.

La poca luz que recibe la señal impide que la segmentación por color sea posible, ya que no reconoce el marco de la señal como rojo.

Aunque las condiciones exteriores sean correctas, si la señal está parcialmente tapada por otro objeto, la detección no se realiza correctamente:

Figura 66: Fallos por condiciones lumínicas – OCLUSIÓN. Salida de la extracción de contornos.

Figura 67: Fallos por condiciones lumínicas – OCLUSIÓN. Salida bloque 1.

El rectángulo que se ve en la figura 67 no pasa el filtro de tamaño relevante por lo que es descartado en los siguientes pasos.

Otro problema encontrado al realizar las pruebas es la existencia de sobras de otros objetos sobre la señal, dificultando o incluso impidiendo una detección correcta.

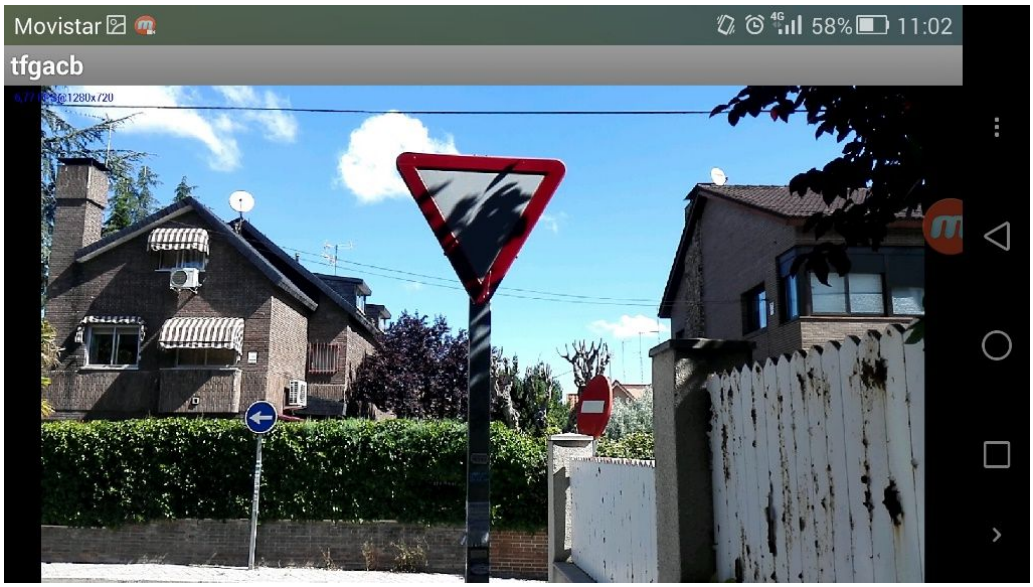

Figura 68: Fallos por condiciones lumínicas – SOMBRAS.

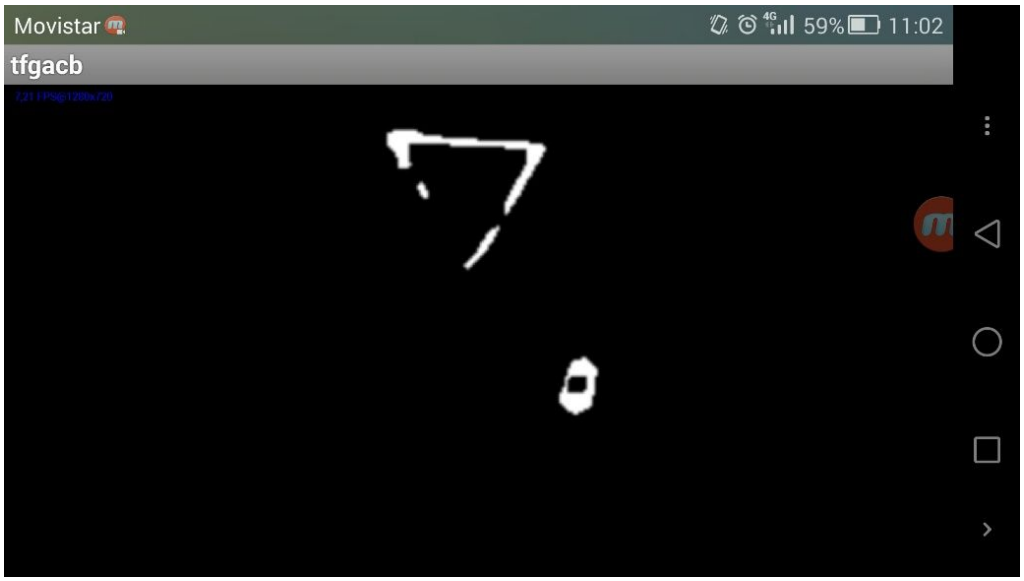

Figura 69: Fallos por condiciones lumínicas – SOMBRAS. Salida post – procesamiento.

En este segundo caso, se puede apreciar cómo pese a la existencia de sombras, se detecta la señal como región de la imagen que puede contener una señal de tráfico, sin verificar todavía por el segundo bloque.

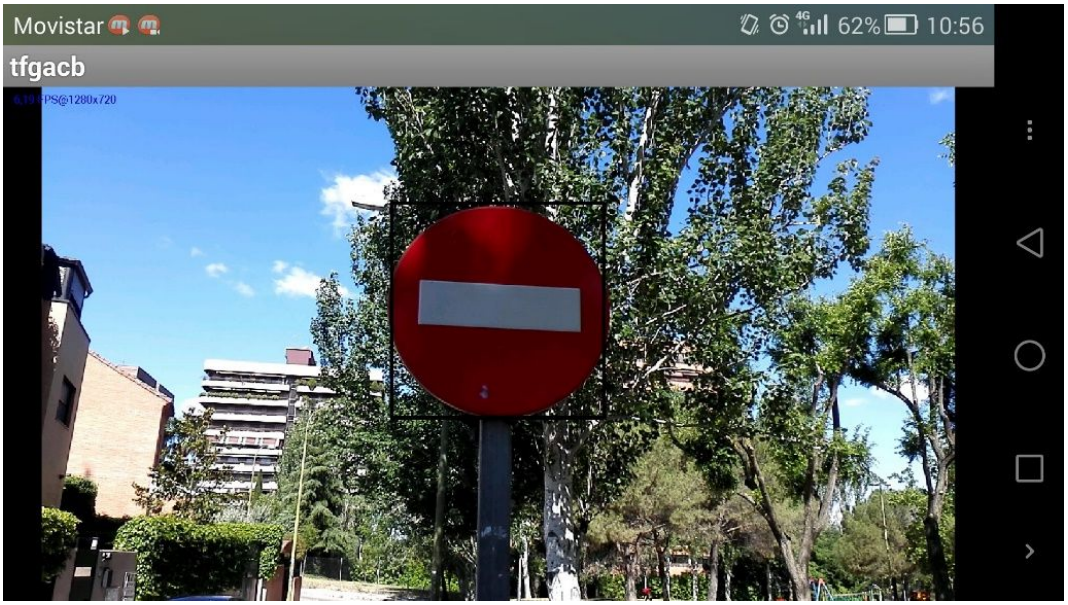

Figura 70: Detección positiva pese a la existencia de sombras.

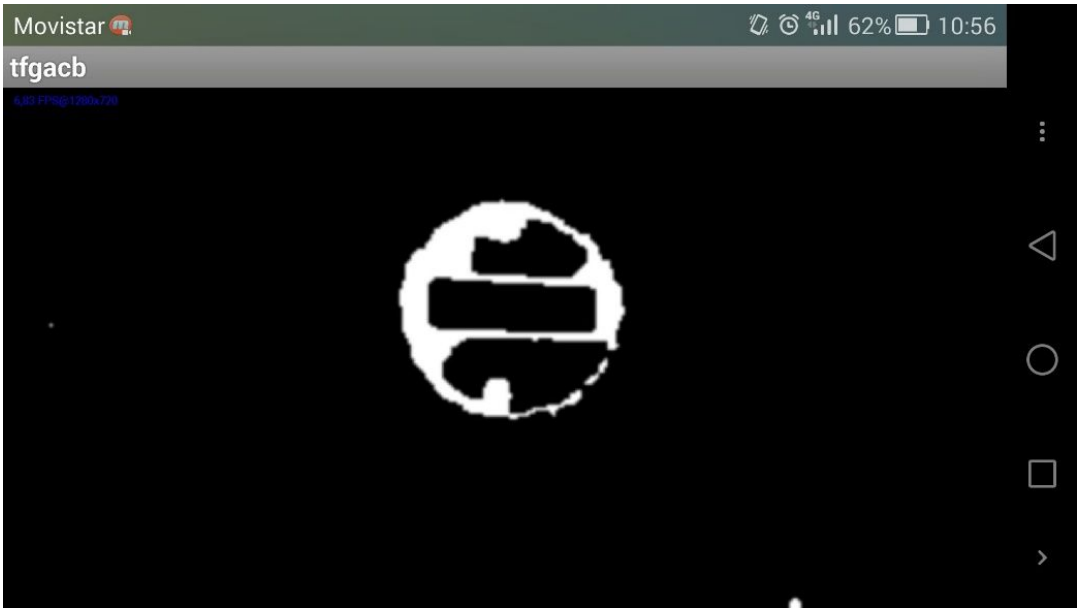

Figura 71: Detección positiva pese a la existencia de sombras. Salida del post – procesamiento.

Por otro lado, y como ya se ha comentado anteriormente, la segmentación por color tiene problemas con las zonas oscuras y de tonos rojizos de las imágenes. Como resulta complicado establecer unos rangos de color que segmenten únicamente las señales de tráfico rojas, la eliminación de ciertas regiones del frame no deseadas se realiza en función a su forma y tamaño. A continuación, las figuras 72, 73 y 74 muestran los resultados de la salida de la aplicación sin la implementación de los filtros de forma y tamaño.

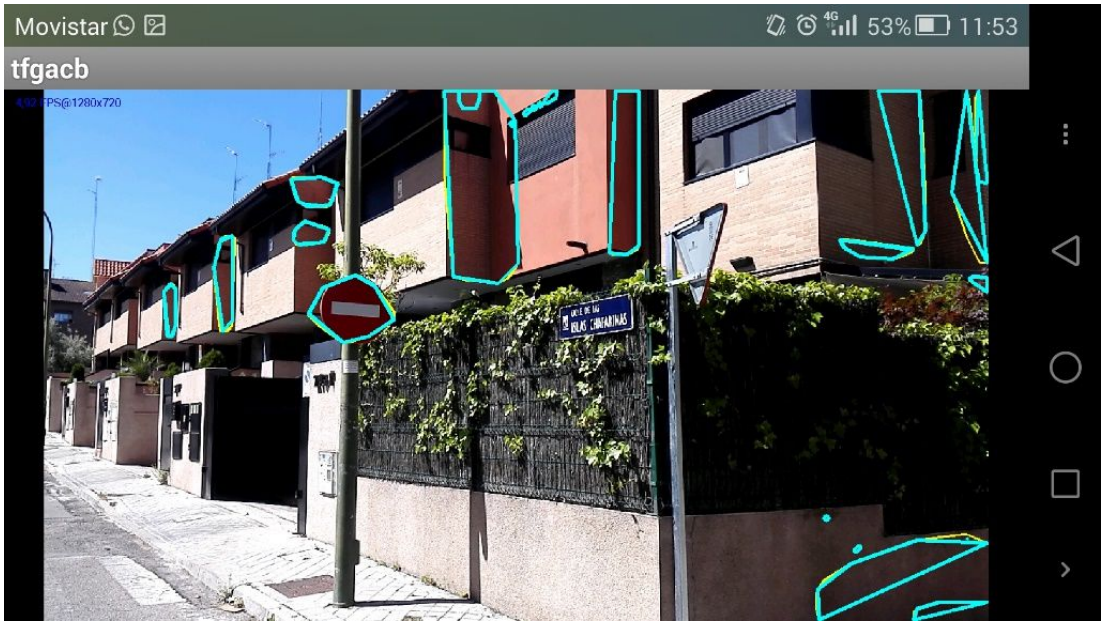

Figura 72: Resultados sin filtro de tamaño y forma. CONTORNOS.

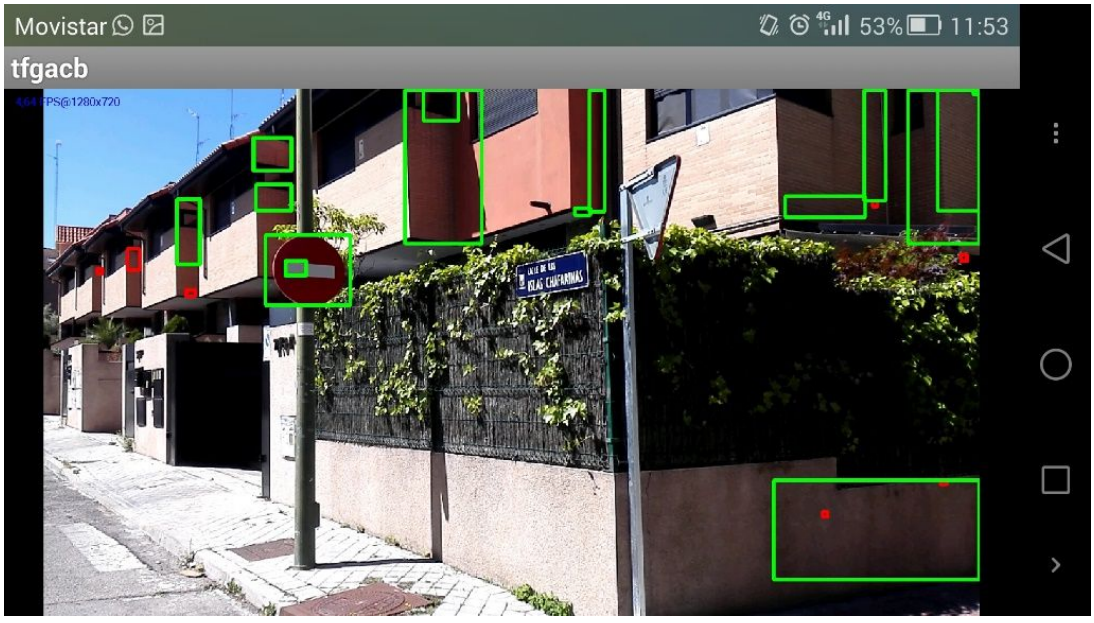

Figura 73: Resultados sin filtro de tamaño y forma – ROI.

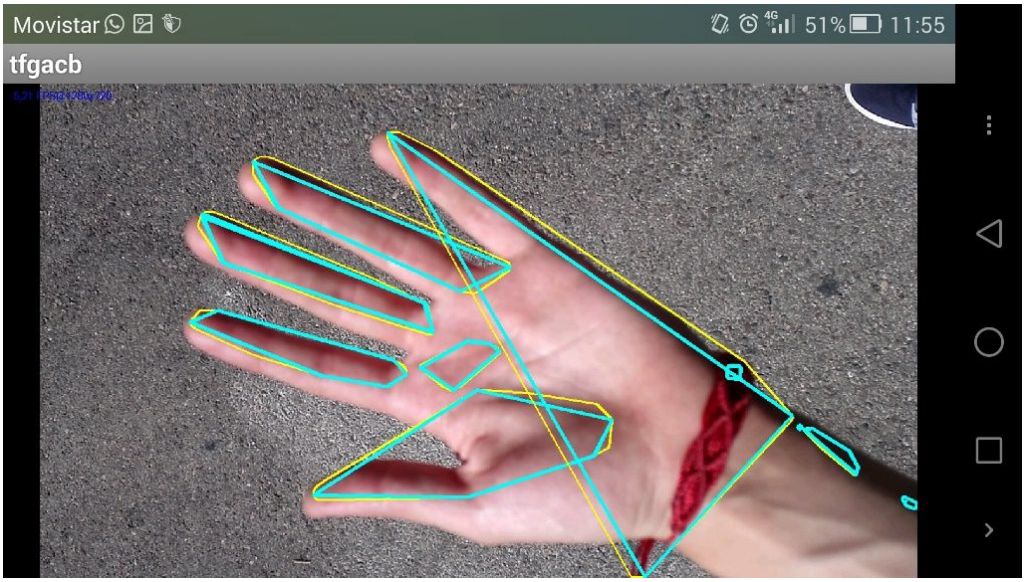

Figura 74: Resultados sin filtros de forma o tamaño. Problema zonas oscuras de color diferente al rojo.

Como se puede ver en la figura 75, el contorno del triángulo exterior se ve afectado por las sombras de la casa que hay detrás. Sin embargo, el triángulo interior no tiene ese problema y se puede apreciar como la aproximación poligonal del triángulo suele dar como resultado un cuadrilátero en vez de un triángulo, por lo que es difícil que el detector de formas distinga entre triángulos y cuadrados o rectángulos.

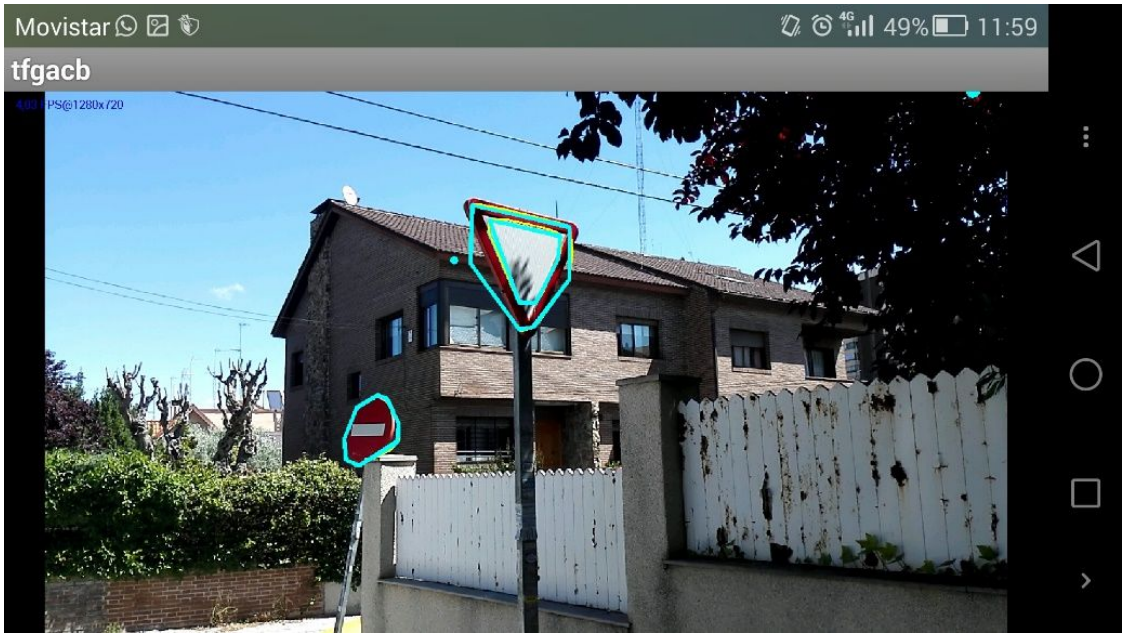

Figura 75: Problemas aproximación poligonal en triángulos.

Otro de los problemas que tiene el primer bloque de la aplicación es que las luces traseras de algunos vehículos son detectadas tras la segmentación por color y pasan también el filtro de forma y tamaño.

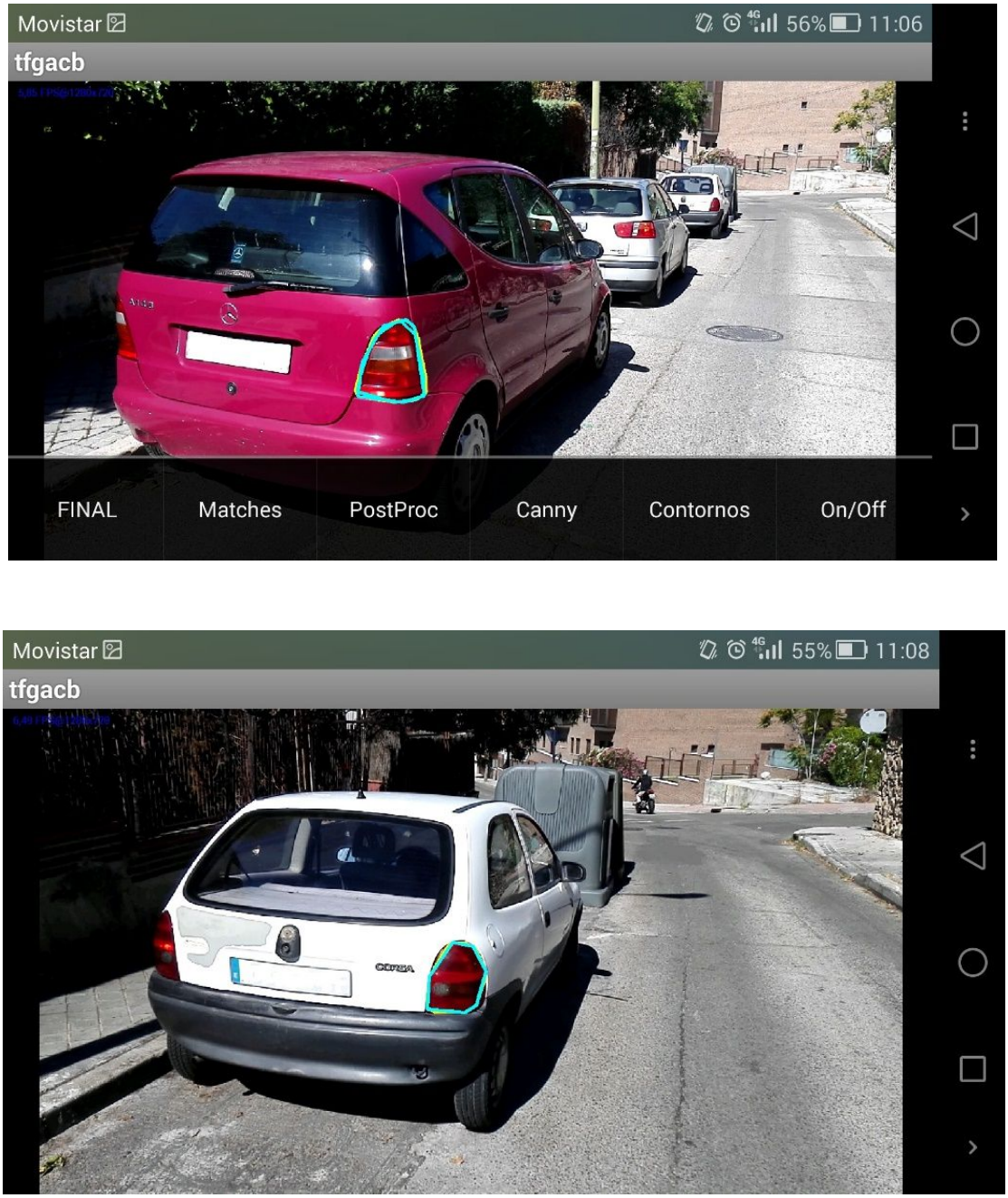

Figura 76: Falsos Positivos – Luz trasera de algunos vehículos.

Como se puede apreciar en la salida de las operaciones de post – procesamiento, es difícil realizar la distinción entre estas luces traseras y una señal de prohibido entrar, por ejemplo, solo con los filtros de color, forma y tamaño.

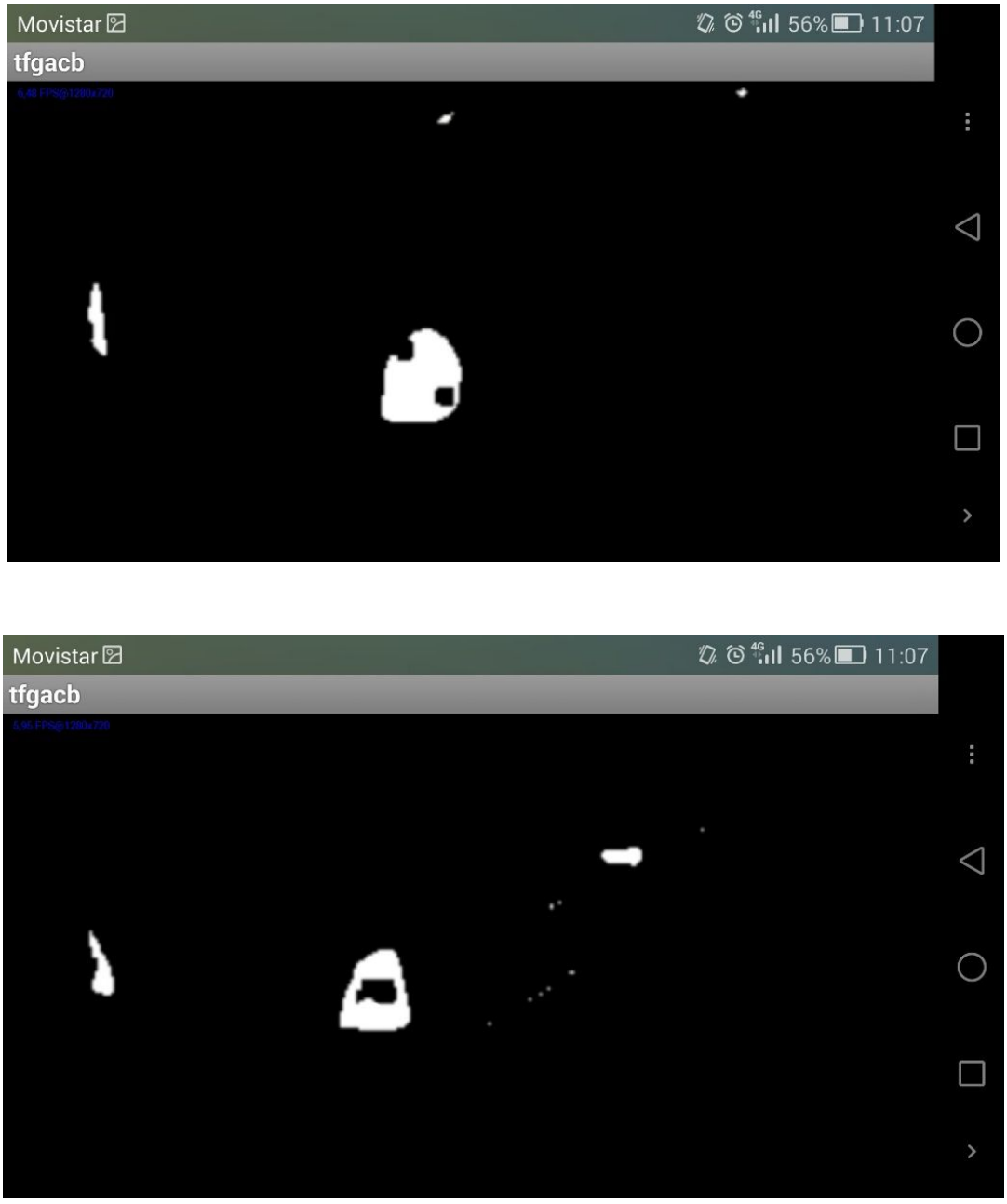

Figura 777: Falsos Positivos – Luz trasera de un vehículo. Salida post – procesamiento.

Estas pruebas muestran la necesidad de implementar el segundo bloque, para reducir la cantidad de falsos positivos, además de obtener información sobre la señal en cuestión.

Por último, la figura 77 muestra los resultados completos de este primer bloque, con los contornos en azul claro, la aproximación poligonal en morado y el rectángulo limitante en azul oscuro.

Además, en la esquina superior derecha se muestra la ROI que pasará al comparador implementado en el segundo bloque.

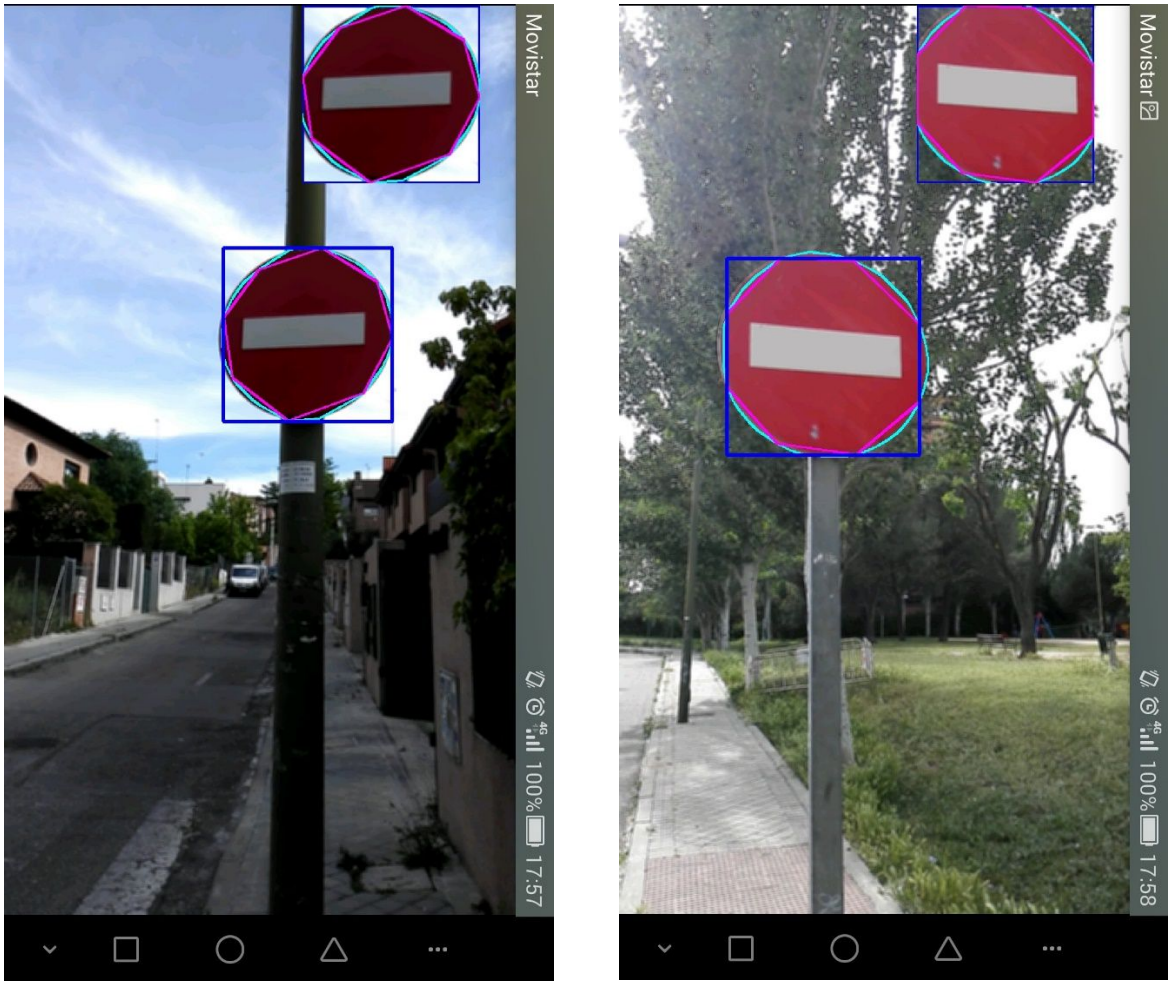

Figura 78: Resultados bloque uno – ROI.

Con respecto a la multi – detección, pese a que los diferentes procesos del primer bloque son capaces de trabajar con varias ROI en un mismo frame, el segundo bloque no permite la verificación de más de una sección con posible señal en su interior a la vez, por lo que la aplicación completa no permite detectar varias señales en un mismo frame.

5.3. Pruebas y resultados del segundo bloque

Una vez se ha conseguido que el primer bloque sea funcional, se puede comenzar con las pruebas de esta segunda parte. Como se ha explicado en el apartado de implementación, el conjunto detector, extractor y comparador ha sido entrenado con unas imágenes de prueba en condiciones estáticas, es decir, una imagen cargada desde la memoria del teléfono que no varía con el tiempo.

Al probar con el detector integrado en la aplicación, haciendo que la primera de las imágenes que entra al comparador sea la ROI obtenida en los pasos anteriores y como segunda imagen los diferentes modelos de la base de datos; los resultados obtenidos no son nada satisfactorios. Las diferencias entre la cantidad total de 'matches' (pareja de descriptores, uno de cada imagen) y 'good matches' no es suficientemente grande entre unos modelos de la base de datos y otro como para generar una detección acertada.

Después de numerosas pruebas con diferentes criterios para la selección de 'good matches' (distancia de matching) y para la decisión de si hay 'matching' o no (proporción entre 'good matches' y 'total matches') se llega a la conclusión de que pese a funcionar el detector durante el entrenamiento, este método no es el adecuado para casos reales.

Se decide por tanto cambiar el orden en el que las imágenes entran al comparador. En este segundo caso, la primera imagen a entrar es la imagen de la base de datos, en un proceso iterativo para ir comparando una a una. De esta manera, cada modelo de la base de datos tendrá un criterio individual para decidir si hay 'matching' o no. Recordemos que el número de 'matches' totales para una pareja de imágenes depende del número de características y descriptores encontrados en la primera imagen, por lo que para cada imagen de la base de datos se detectará siempre el mismo número de matches, permitiendo fácilmente establecer un número mínimo de 'good matches' para el resultado positivo de la detección con esa señal.

Para que este nuevo método pueda funcionar, solo queda establecer los valores del coeficiente utilizado para la distancia de 'matching' y el ratio entre good y total matches de cada modelo de la base de datos.

Tras los resultados del anterior entrenamiento con imágenes que no procedían del bloque anterior, en este caso, estos dos parámetros han sido ajustados con el detector integrado en la aplicación, por el método de prueba y error, estudiando el número de matches y el ratio para cada señal.

Como ya se ha mencionado, la comparación con la base de datos se realiza en dos grupos señales circulares y señales no circulares. Para este segundo caso, la base de datos cuenta con una señal modelo, la de ceda el paso. El problema que nos encontramos con este grupo es que todas las detecciones parciales de una señal, por cuestión de sombras,

oclusión o cualquier otro problema entran también al comparador con este grupo, por lo que se genera bastante ruido y falsos positivos y negativos.

A continuación se muestran las detecciones obtenidas con este segundo bloque en acción. Como se puede apreciar, la distinción entre las señales de prohibido entrar y prohibido circular es errónea en ciertas ocasiones.

El logotipo circular naranja que se ve en algunas imágenes corresponde a la aplicación *Mobizen*, utilizada para tomar capturas de pantalla y grabar videos.

Se ha establecido un código de colores para la diferenciación entre las señales a detectar:

Entrada Prohibida → **Azul**

Stop → **Rojo**

Circulación prohibida → **Verde**

Ceda el paso → **Amarillo**

Detección de objeto sin verificación positiva → **Negro**

Figura 79: Detecciones positivas para diversas señales de tipo 'Prohibido Entrar o Dirección Prohibida'.

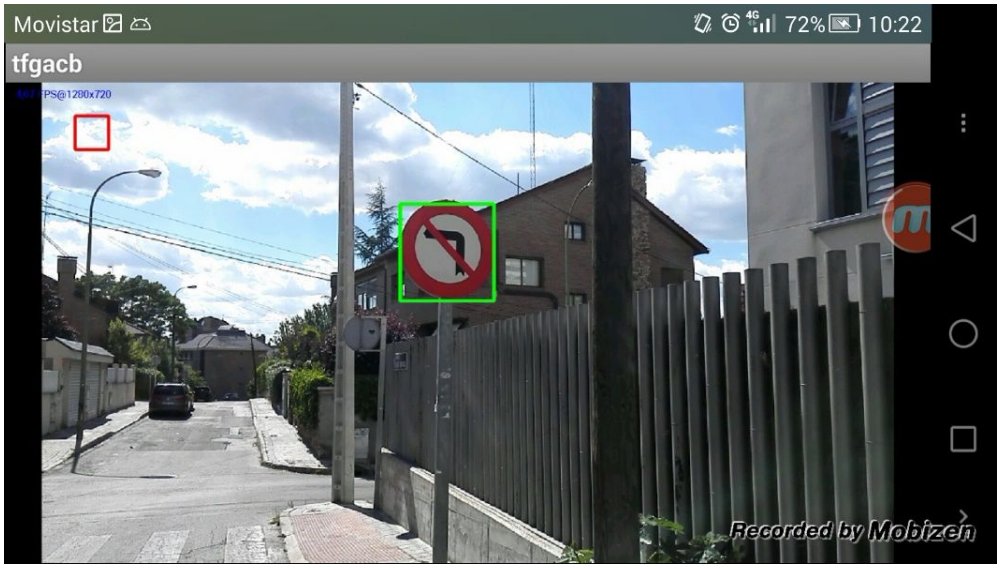

Figura 80: Detecciones positivas para diversas señales de prohibición.

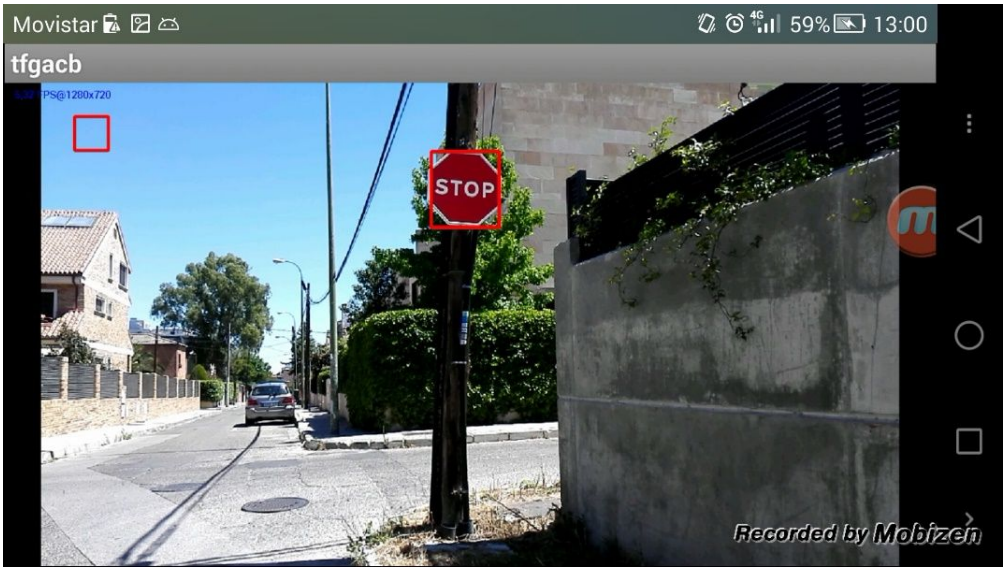

Figura 81: Detección positiva para señal de STOP.

El siguiente grupo de capturas de pantalla muestra los fallos que se producen en ciertas ocasiones, confundiendo unas señales con otras:

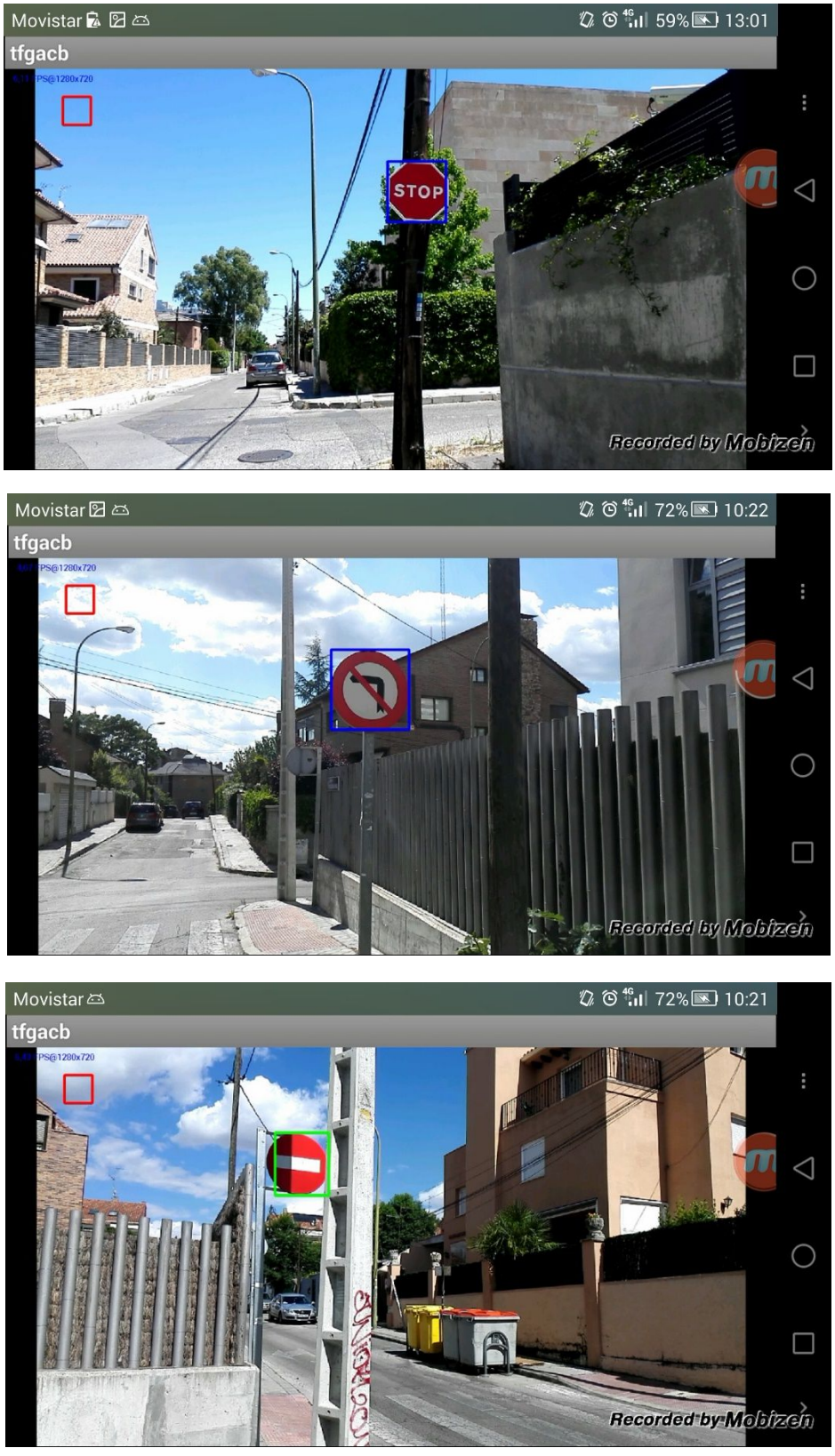

Figura 82: Ejemplos de falsos positivos. Confusión entre diferentes tipos de señal.

Todas las capturas de pantalla mostradas en este último apartado han sido realizadas a velocidades muy bajas, caminando despacio o incluso completamente detenido.

Los resultados de las pruebas en coche no generan buenas detecciones, en la mayoría de ocasiones no detectan ninguna señal. Además, dependen mucho de la velocidad, como ya se ha explicado, y de las circunstancias y apariencia de señal.

Las señales de STOP y 'Dirección Prohibida' son las señales que mejor se detectan, ya que son las que más rojo tienen, permitiendo que pese a la existencia de sombras o reflejos, estas señales sigan siendo detectadas, como se puede ver por ejemplo en las figuras 70 y 79.

Por otro lado, los falsos positivos que se producían en las pruebas del bloque uno dejan de ser detectados cuando el segundo bloque comienza a funcionar.

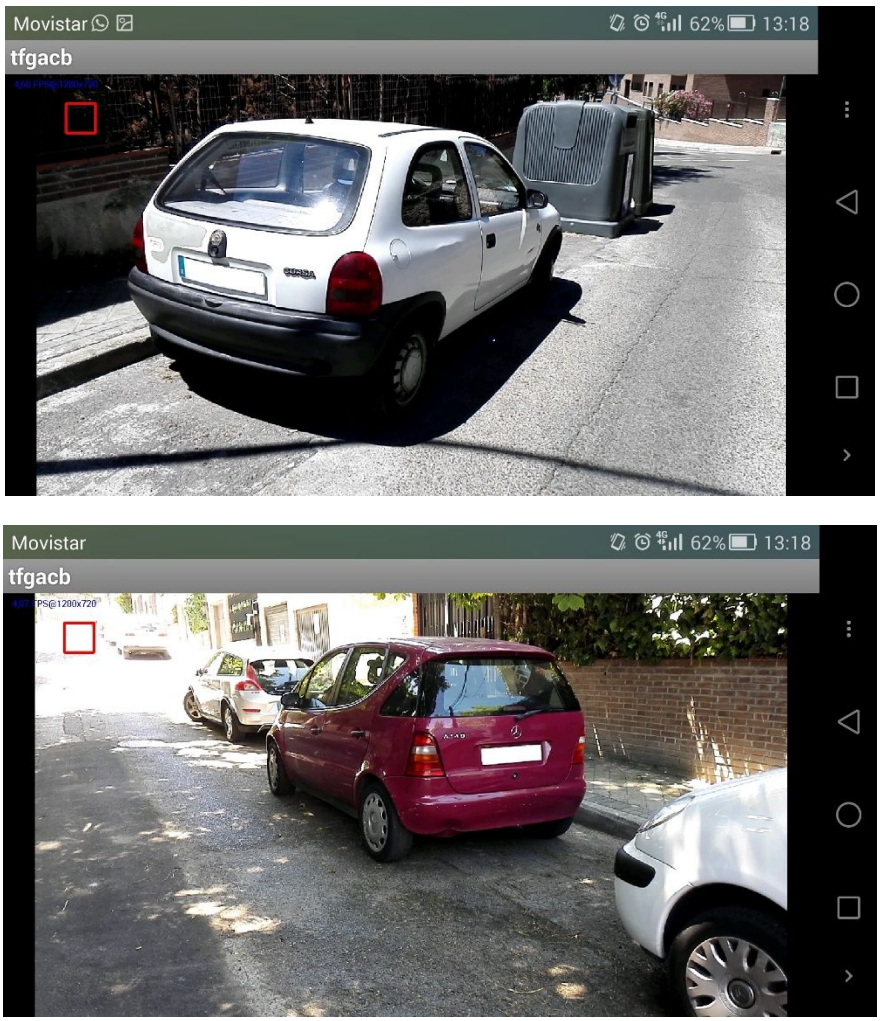

Figura 83: Falsos positivos de las luces traseras de los coches eliminados.

5.3. Posibles ampliaciones futuras y alternativas para la detección de señales.

A lo largo del desarrollo del proyecto, se han planteado diferentes opciones para implementar los diferentes procesos de la aplicación y corregir los problemas que iban surgiendo. En este apartado de enuncian y analizan brevemente algunas de esas posibles soluciones planteadas que por diferentes motivos no han llegado a ser implementadas.

Uno de los principales problemas que se ha encontrado está relacionado con la información obtenida y no almacenada en los frames previos al frame actual que está siendo procesado. Si tras el procesamiento de un frame la detección ha dado un resultado positivo, es de suponer que la señal detectada seguirá presente en el siguiente frame, y no solo eso, si no que estará localizada más o menos en la misma sección de la imagen. Sin embargo, la solución desarrollada en este proyecto no tiene en cuenta esta información, aumentando el tiempo de procesamiento y reduciendo los FPS de salida por cálculos redundantes.

La solución implementada resolver para este problema consiste en no procesar todos los frames, con los inconvenientes que esto genera explicados anteriormente. Por tanto, desarrollar una manera de tener en cuenta el frame anterior sin dejar de procesar todos los frames optimizaría la calidad de la detección.

Otro de los aspectos de la aplicación que se podría mejorar es el de la multi – detección de señales en un mismo frame. Habría que implementar un algoritmo que permita ejecutar el segundo bloque (detección, extracción y comparación) en varias ROI a la vez y no de una en una como hace actualmente.

Uno de los métodos que mejores resultados ofrecen a la hora de detectar objetos (señales, coches, peatones, caras, ojos…) son los clasificadores Haar – Cascade. Estos clasificadores trabajan con características más avanzadas.

El principal inconveniente que tienen estos clasificadores y motivo por el cual no se ha utilizado en este TFG es que requieren una gran cantidad de imágenes para ser generados. Para entrenar un clasificador Haar – Cascade son necesarias imágenes positivas (con el objeto a detectar en la imagen) e imágenes negativas (sin el objeto a detectar presente en la imagen), con diferentes opiniones entre los expertos sobre si lo importante es la cantidad o calidad de los modelos de entrenamiento [40] [41]. De cualquier modo, estamos hablando de aproximadamente 2000 muestras positivas y 8000 muestras negativas. Esta cantidad de imágenes está fuera del alcance de este TFG ya que no hay ninguna colección tan grande de imágenes de las señales de tráfico españoles y no se disponía del tiempo ni recursos suficientes para generar una.

De hecho, el LSI considera para futuros proyectos, TFG o tesis de fin de máster la implementación de un detector más avanzado.

Las librerías de OpenCV ofrecen, varios clasificadores Haar – Cascade para la detección de caras, personas/peatones, coches… que funcionan bastante bien, como puede comprobarse en las otras sub-aplicaciones de la aplicación de LSI en la que se integra este TFG. Sin embargo, no hay ningún clasificador de este tipo específico para señales de tráfico entre sus muestras ni en ninguno de los foros ni páginas oficiales de OpenCV.

Otra posible alternativa para mejorar el funcionamiento de esta aplicación podría ser la implementación de una red neuronal para sustituir el segundo bloque [43]. De la misma manera que el clasificador Haar – Cascade, podría entrenarse la red para detectar una señal específica, un grupo de señales (rojas, azules, circulares o triangulares, por ejempo) o simplemente un clasificador que detecte o no la presencia de una señal en la imagen, sea cual sea, sin distinguir entre ellas.

De nuevo, esta red debería ser entrenada con un conjunto de imágenes, menor que para el caso anterior, con las diferentes imágenes de señales de tráfico en diferentes condiciones lumínicas, ambientales, aspecto físico de la señal…

6. Presupuesto

A continuación se detallan los costes materiales y de personal que este proyecto ha generado.

Para el cálculo de los costes de materiales del proyecto y su amortización, se ha tenido en cuenta las tablas de amortización oficiales del Ministerio de Hacienda para los equipos informáticos [44] [45] [46].

Según estos datos, el coeficiente de amortización para sistemas y equipos electrónicos e informáticos se establece en un 20% con un periodo máximo de 10 años.

El material utilizado para el desarrollo del proyecto ha sido un ordenador portátil Asus con Windows 8 y un Smartphone Huawei modelo Honor 6.

La siguiente tabla refleja los costes de amortización de dichos dispositivos durante el periodo de desarrollo del proyecto, 6 meses.

Unidad	Precio	Coeficiente Amortización	Coste Proyecto
Asus N61 - JV	1.200 €	20%	120 €
Huawei Honor 6	300 €	20%	30 €
			150 €

Tabla 14: Presupuesto equipos informáticos.

Para calcular los costes de personal se ha tenido en cuenta el salario medio bruto español en el año 2014 [47], establecido en 26.162 € para una jornada de 8h diarias y 21 días trabajados por mes. Esto nos lleva a un total de 2016 horas trabajadas al año y un salario de aproximadamente 12.98 € / hora.

El alumno ha dedicado una media de 20 horas semanales al desarrollo de este proyecto, sumando un total de 480 horas, y no tiene experiencia profesional, por lo que es razonable asumir un contrato en prácticas, con un salario de como mínimo el 60% del sueldo de un contrato normal.

Teniendo en cuenta estos dos factores, y aplicando un 60% para el sueldo del contrato en prácticas, los costes de personal ascienden a 3738,24 €.

Por lo tanto, el coste total aproximado del proyecto es:

Tipo de Coste	Presupuesto
Materiales	150 €
Personal	3738,24 €
Total	3888,24 €

Tabla 15: Presupuesto total del proyecto.

7. CONCLUSIONES

La realización de un proyecto de esta envergadura ha supuesto un reto interesante, no solo a nivel académico o profesional sino también a nivel personal. La dedicación y esfuerzo requeridos para su desarrollo, así como la satisfacción de haberlo conseguido, han otorgado al alumno una nueva perspectiva para la resolución de futuros problemas.

Cualquier persona que haya programado alguna vez entiende los problemas y frustraciones a los que se enfrenta el desarrollador a la hora de hacer funcionar el código, sobre todo si no se tienen conocimientos avanzados del lenguaje y el entorno de desarrollo utilizado.

Es importante destacar que la solución propuesta en este trabajo no es ni la única ni la más completa, como ya se ha discutido en los apartados anteriores, si bien es la que mejor encajaba y la más factible a realizar con los recursos disponibles.

Dejando a un lado las limitaciones debidas a las capacidades de procesamiento de los teléfonos inteligentes, que con el tiempo irán aumentando y permitirán que este tipo de aplicaciones funcionen mejor, se ha conseguido implementar una aplicación funcional, y se ha podido estudiar el rendimiento de los algoritmos de OpenCV utilizados en un dispositivo con menos recursos que un ordenador, donde ya se había comprobado el correcto funcionamiento de dichos algoritmos.

Si bien es cierto que en el área de los sistemas de Visión por Computador todavía queda mucho por hacer, y tanto este proyecto como todos los que se están desarrollando actualmente son tan solo la punta del iceberg de lo que se conseguirá en un futuro.

Dicho esto, los resultados obtenidos son muy positivos establecen una buena base continuar implementando nuevos métodos y algoritmos para la detección de señales de tráfico. Esta es la primera aplicación para Android desarrollada por el LSI y por lo que se ha podido comprobar en Google Play, de las primeras del mercado.

Además, y pese a que las librerías de OpenCV están en auge y poco a poco irán ganando protagonismo, se ha sufrido la escasez de documentación existente para la plataforma Android (Java), por lo que el alumno ha tenido que encontrar la solución a gran parte de los problemas encontrados él solo con la ayuda del tutor y del director de proyecto.

8. NORMATIVA

Licencias del Proyecto:

- Open CV [3].
- Eclipse Public License v 1.0 [48].
- Android Open Source Project [49].
- Apache 2.0 License [50].
- Señales Verticales de Circulación [51].

9. BIBLIOGRAFÍA

[1] Google Android Development.
http://developer.android.com/index.html

[2] ZDNet.
http://www.zdnet.com/article/how-android-works-the-big-picture/

[3] OpenCV.
http://opencv.org/

[4] Efstratios Gavves – Computer Vision & Machine Learning.
http://www.egavves.com/a-brief-history-of-computer-vision/#sthash.fn85KcCM.dpbs

[5] Road Sign Recognition System – OpenCV.
https://sites.google.com/site/mcvibot2011sep/background/opencv

[6] Emgu – CV. Traffic Sign Detection in C#.
http://www.emgu.com/wiki/index.php/Traffic_Sign_Detection_in_CSharp#Traffic_Sign_Detection

[7] Fast Colour Based Object Tracking with OpenCV.
http://suksant.com/2013/04/26/fast-colour-based-object-tracking-with-opencv/

[8] Google Play – Road Signs Recognition.
https://play.google.com/store/apps/details?id=road.signs.recognition

[9] Google Play – iOnRoad Augmented Driving Pro.
https://play.google.com/store/apps/details?id=com.picitup.iOnRoad.pro

[10] Google Play – myDriveAssist (BOSCH).
https://play.google.com/store/apps/details?id=com.bosch.mydriveassist

[11] Oficina de Información Científica – OIC UC3M.
http://portal.uc3m.es/portal/page/portal/actualidad_cientifica/noticias/coche_inteligente_peatones

[12] Fundación Eduardo Barreiros.
http://www.fundacionbarreiros.com/es/fundacion/noticia/148

[13] Alef Revista.
https://www.youtube.com/watch?v=21XdtROQOfI

[14] Google Android Development – Activities.
http://developer.android.com/guide/components/activities.html

[15] StackOverflow – Android Activity Life Cycle.
http://stackoverflow.com/questions/8515936/android-activity-life-cycle-what-are-all-these-methods-for

[16] OpenCv Documentation – JavaCameraView.
http://docs.opencv.org/java/org/opencv/android/JavaCameraView.html

[17] Espacios de Color – PFC.
http://bibing.us.es/proyectos/abreproy/11875/fichero/Proyecto+Fin+de+Carrera%252F3.Espacios+de+color.pdf

[18] The Atomic Guide to Basis Gimp Stuff.
http://www.tadpolewebworks.com/web/atomic/highlights.html

[19] The Luminous Landscape – Cubo Espacio de Color RGB.
http://forum.luminous-landscape.com/index.php?topic=37695

[20] Computer Science at Virginia Tech – Detalles Color Cubo RGB.
http://courses.cs.vt.edu/~cs4624/s98/sspace/imgproc/Image11.gif

[21] New Mexico Tech – Introduction to Color Theory.
http://infohost.nmt.edu/tcc/help/pubs/colortheory/web/hsv.html

[22] StackOverflow – HSV Color Space.
http://stackoverflow.com/questions/3339692/modeling-hsv-color-space-in-matlab

[23] StackOverflow – HSV Color Space Cylinder.
http://stackoverflow.com/questions/470690/how-to-automatically-generate-n-distinct-colors

[24] Computer Graphics – HSV Color Wheel.
http://www.ccs.neu.edu/course/cs4300old/s10/L5/L5.html

[25] Open Course Ware (OCW) UC3M – Sistemas de percepción (2010).
http://ocw.uc3m.es/ingenieria-de-sistemas-y-automatica/sistemas-de-percepcion

[26] Pyevolve – Coin Segmentation using OpenCV.
http://blog.christianperone.com/?p=2711

[27] OpenCV – Python Tutorials. Contour Features.
http://opencv-python-tutroals.readthedocs.org/en/latest/py_tutorials/py_imgproc/py_contours/py_contour_features/py_contour_features.html

[28] Richard Szeliski (2010). Computer Vision: Algorithms and Applications, Springer 2010, pp. 209, 222, 225.
http://szeliski.org/Book/drafts/SzeliskiBook_20100903_draft.pdf

[29] StackOverflow – Detección de características y Extracción de descriptores.
http://stackoverflow.com/questions/6832933/difference-between-feature-detection-and-descriptor-extraction

[30] OpenCV Documentation – Detección de características.
http://docs.opencv.org/modules/features2d/doc/common_interfaces_of_feature_detectors.html

[31] OpenCV Documentation – Extracción de descriptores.
http://docs.opencv.org/modules/features2d/doc/common_interfaces_of_descriptor_extractors.html

[32] OpenCV Documentation – Comparación de descriptores.
http://docs.opencv.org/modules/features2d/doc/common_interfaces_of_descriptor_matchers.html

[33] StackOverflow – Clasificación de detectores, extractores, y comparadores.
http://stackoverflow.com/questions/14808429/classification-of-detectors-extractors-and-matchers/14912160#14912160

[34] Computer Vision Talks – Comparación SURF, FREAK y BRISK.
http://computer-vision-talks.com/articles/2012-08-18-a-battle-of-three-descriptors-surf-freak-and-brisk/

[35] Little Cheese Cake – Feature Detectors and Descriptors.
http://littlecheesecake.me/blog1/2013/05/25/feature-detection.html

[36] Computer Vision Talks – Comparación de algoritmos para la detección de características.
http://computer-vision-talks.com/articles/2011-01-04-comparison-of-the-opencv-feature-detection-algorithms/

[37] StackOverflow – Tipos de extractores y comparadores.
http://stackoverflow.com/questions/7232651/how-does-opencv-orb-feature-detector-work

[38] OpenCV Documentation – Adaptadores para la detección de características.
http://docs.opencv.org/modules/features2d/doc/common_interfaces_of_feature_detectors.html

[39] Trafico y servicios – Referencias señales de código.
http://www.traficoyservicios.com/esp/senalizacion-vial-vertical/senales-de-codigo-referencias/index.htm

[40] Enciclográfica – Señales de tráfico.
http://www.sitographics.com/enciclog/trafico/Localizacion/Localizacion.html

[41] Erogol – Large data really helps object detection?
http://www.erogol.com/large-data-really-helps-object-detection/

[42] Massachusetts Institute of Technology – Do we need more training data or better models for object detection?
http://web.mit.edu/vondrick/largetrain.pdf

[43] Center for Biomedical and Robotics Technology (BART LAB) & Department of Biomedical Engineering, Faculty of Engineering, Mahidol University – Traffic Sign Recognition using Neural Network on OpenCV.
http://bartlab.org/Dr.%20Jackrit's%20Papers/ney/1.TRAFFIC_SIGN_Lorsakul_ISR.pdf

[44] Gábilos Software de Gestión – Tablas oficiales de amortización para sociedades y autónomos en estimación directa.
http://www.gabilos.com/webcontable/amortizacion/estimacion_directa_simplificada.htm

[45] Durán Sindreu – Nuevo tratamiento de las amortizaciones en el IS 2015.
http://www.duransindreu.com/nuevo-tratamiento-de-las-amortizaciones-en-el-impuesto-sociedades/

[46] Asesor Contable – Nuevas tablas de amortización 2015.
http://asesor-contable.es/tablas-amortizacion-2015/

[47] DatosMacro – Salario Medio en España 2014.
http://www.datosmacro.com/mercado-laboral/salario-medio/espana

[48] Eclipse Public License v 1.0.
http://www.eclipse.org/org/documents/epl-v10.php

[49] Android Open Source Project License.
http://source.android.com/

[50] The Apache Software Foundation – Apache License Version 2.0.
http://www.apache.org/licenses/LICENSE-2.0

[51] Ministerio de Fomento – Señalización Vertical.
http://www.fomento.gob.es/MFOM/LANG_CASTELLANO/DIRECCIONES_GENERALES/CARRETERAS/NORMATIVA_TECNICA/EQUIVIAL/SENAVERTI/

Anexo I – Manual de la aplicación

Como ya se ha mencionado anteriormente, la aplicación implementada en este TFG se ha integrado en una aplicación base implementada para el LSI por Alejandro Ramos en su TFG, en septiembre de 2014.

El diseño de la interfaz y estructura de esta aplicación base, llamada *LSIappDet,* son por tanto independientes al desarrollo de este TFG.

A continuación, se explica brevemente el funcionamiento de *LSIappDet*, y como llegar hasta la sub – aplicación para la detección de señales.

La actividad principal presenta un menú con varias opciones. El botón que nos lleva a la aplicación para la detección de señales es el botón 'Aplicaciones LSI'. Además, desde este menú se puede acceder a las aplicaciones ejemplo de OpenCV, a las páginas web de la Universidad Carlos III de Madrid y del Laboratorio de Sistemas Inteligentes, y a otra actividad con información sobre los desarrolladores.

Figura 84: Actividad principal de la aplicación LSIappDet

Una vez el botón 'Aplicaciones LSI' es pulsado, nos dirige a otra actividad con dos botones: 'Ejecutar Aplicación' y 'volver'. Al pulsar el primero de ellos se abre un menú

flotante que permite escoger la aplicación a ejecutar: ‘detector de peatones’, ‘detector de coches’, ‘ejemplo libro OpenCV’ y ‘detector de señales’.

Figura 85: Actividad para iniciar subaplicaciones

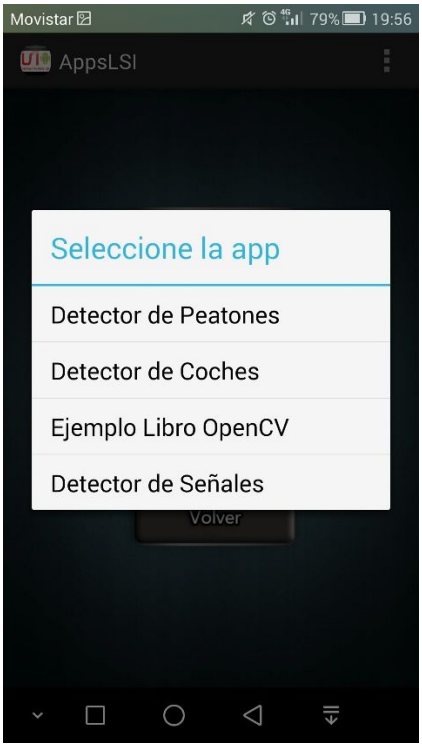

Figura 86: Menú subaplicaciones

Al pulsar sobre ‘detector de señales’ se abre una nueva actividad que, a través de otro botón, nos permite iniciar la aplicación.

Figura 87: Actividad para iniciar aplicación del detector de señales

Una vez la aplicación está iniciada, la interfaz muestra directamente los fotogramas capturados por la cámara en tiempo real, con los FPS en la esquina superior izquierda y con las señales recuadradas en color sus respectivos colores en caso de que haya una detección positiva.

El menú situado en la esquina superior derecha presenta seis botones que permiten ver las imágenes de salida de las diferentes fases implementadas y activar o desactivar el detector (segundo bloque).

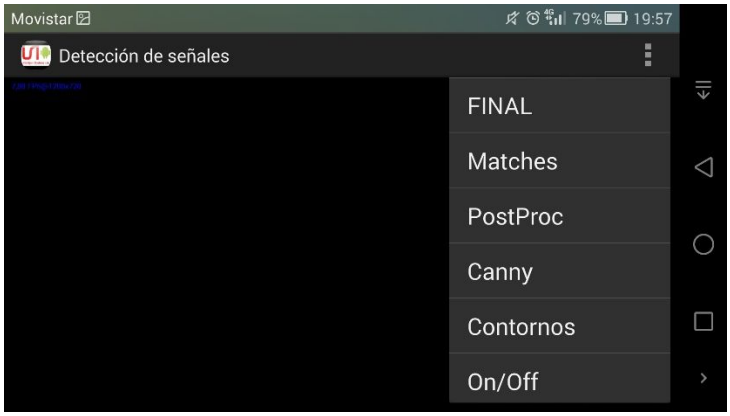

Figura 88: Interfaz del la aplicación para la detección de señales de tráfico

Además de esta sub – aplicación, *LSIappDet* cuenta con aplicaciones para la detección de peatones, coches y otros ejemplos de las librerías OpenCV.

El botón 'Quienes somos' abre una nueva actividad con información sobre los alumnos, tutor y director de proyecto que han participado en el desarrollo de la aplicación.

Figura 89: Actividad 'Quienes somos'