SISTEMA DE DETECCIÓN DE CARRILES EN ANDROID

Sergio de Lara Toledo

INGENIERÍA INDUSTRIAL - UC3M

Tutor: Fernando García Fernandez

Table of Contents

Tablas y Gráficas

1. Introducción

Los avances en electrónica e informática están aumentando exponencialmente en los últimos tiempos. La potencia de los dispositivos aumenta a pasos agigantados, y surgen cada vez más nuevos aparatos y tecnologías que nos abren todo un mundo de posibilidades.

Uno de los cambios más significativos que ha experimentado este campo, y también nuestra sociedad, es la aparición de los *smartphones.* Hoy en día, se ha convertido en algo estándar tener acceso constante a un dispositivo con conexión a internet y alta capacidad de procesamiento. Con todas las posibilidades que ello conlleva.

Si a esto le unimos que todos estos nuevos dispositivos llevan cámara integrada, podemos entender como la visión por computador y el tratamiento de imágenes han pasado de ser una técnica de uso bajo o experimental, a estar completa y absolutamente en el primer plano de las nuevas tecnologías.

Las aplicaciones que pueden tener estos avances son prácticamente ilimitadas, y en los próximos años veremos muchas de ellas, como la realidad aumentada, cobrar una mayor relevancia. En nuestro caso, nos centraremos en la utilización de estas cámaras para el desarrollo de sistemas automatizados. Concretamente, para automatizar la conducción de vehículos.

Para ello, hay dos tipos de análisis que una aplicación deberá analizar para poder conducir de forma segura y efectiva. Por una parte, tendrá que poder procesar toda la información respectiva a la vía de circulación que está recorriendo. Tanto el carril a través del cual se mueve, como todas las señales que establezcan las normas de circulación en dicho carril.

Por otra parte, deberá ser capaz de recibir y procesar la información respectiva al resto de elemento que interfieran en la vía. Sean estos otros vehículos, peatones u obstáculos. Y poder reaccionar ante ellos de la forma más adecuada.

Cabe destacar que, dada la naturaleza de este tipo aplicaciones, un alto nivel de efectividad será exigido. Puesto que la conducción es una actividad en la que cualquier error podría derivar en la pérdida de vidas humanas. Por tanto, este tipo de aplicaciones requieren más trabajo que si se tratase de un simple juego o una aplicación destinada a un ámbito con menores riesgos.

En nuestro caso concreto, nos centraremos en el apartado correspondiente a la correcta detección de los diferentes tipos de líneas de carreteras que delimitarán el

desplazamiento del vehículo. Dicho programa, irá integrado en la aplicación del Laboratorio de Sistemas Inteligentes de la universidad Carlos III de Madrid.

Programa el cual contiene, además de la nuestra, otras dos aplicaciones de utilidad para la tarea que se propone. Un programa de reconocimiento de señales de tráfico, y otro para la detección de vehículos y peatones. Con todo esto en conjunto, se espera se podrá poner en marcha un vehículo completamente automatizado y funcional.

Por último, cabe añadir que el propósito de este proyecto se encuadra dentro de los ADAS (Advanced driver assistance system). Es decir, programas diseñados para ayudar al conductor en la labor de conducción, pero que no le sustituyen a la hora de controlar el vehículo.

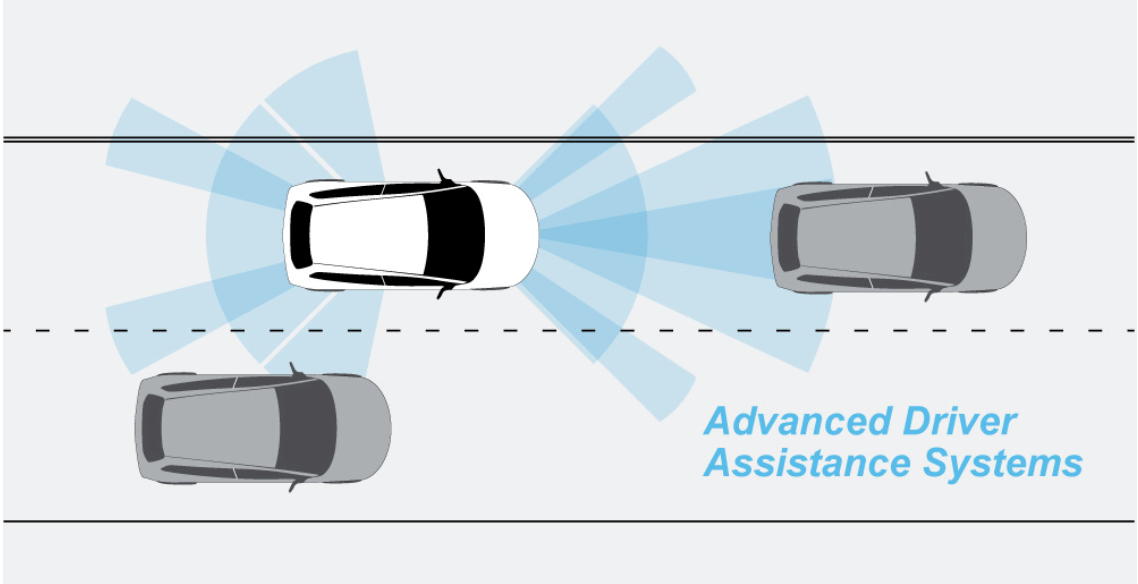


Figure 1

2. Estado Del Arte

2.1 Android

Para la realización de este proyecto se eligió *Android.* El sistema operativo con núcleo basado en *Linux* propiedad de la marca *Google.* Desde su salida al mercado en 2008 se ha convertido en el sistema operativo más usado en dispositivos móviles con un amplísimo margen de diferencia (1). Además, el incremento en el uso de este tipo de dispositivos, superando ya en uso doméstico al ordenador personal, lo convierten en el sistema operativo más usado del mundo.

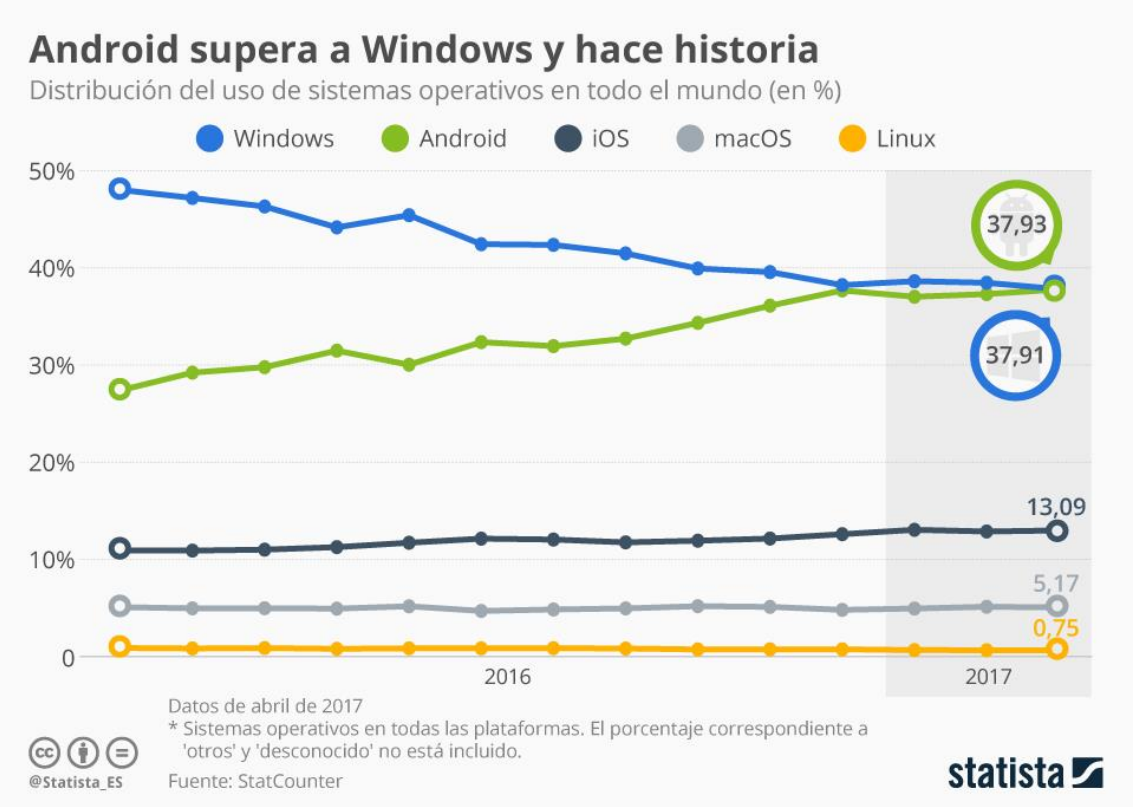


Figure 2

Su buen rendimiento en dispositivos móviles y su amplio uso, lo convertían en una de las opciones más atractivas. Además, se trata de un software libre para el que se ofrecen muchas facilidades a la hora de desarrollar aplicaciones.

Para su funcionamiento, *Android* posee un *kernel* que, como ya hemos dicho, está basado en el sistema operativo *Linux,* y se encarga de la administración de ciertos aspectos como la memoria o los controladores, pero no de las aplicaciones. También cuenta con su propio conjunto de librerías escritas en C y C++, a las cuales se accede utilizando JNI (Java Native Interface).

Además de estas, *Android* también ofrece a sus usuarios una gran multitud de librerías para cubrir todo tipo de funcionalidades. Siendo algunos ejemplos la librería de medios, la de gráficos o la de 3D.

Por último, la gestión de las aplicaciones se realiza a través de máquinas virtuales, ejecutadas de forma simultánea para cada una de las aplicaciones que se estén ejecutando. En un primer momento, *Android* utilizaba *Dalvik* como máquina virtual, pero posteriormente la cambio por *ART*.

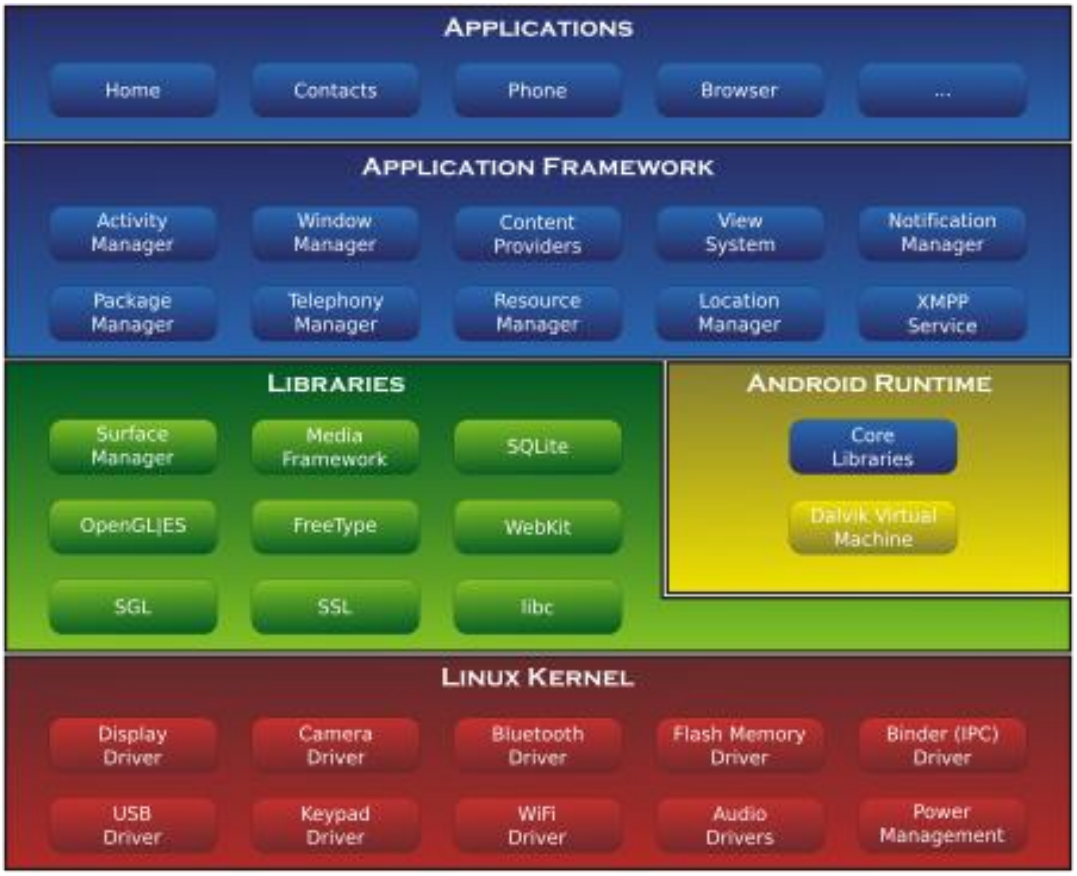


Figure 3

En cuanto al desarrollo de aplicaciones, se realiza en *Java* a través de *Android SDK* (Software Development Kit). Un kit de desarrollo de aplicaciones que ofrece *Google* y que se puede descargar de forma totalmente gratuita. Lo cual es otra de las razones por las que supone una excelente elección para la realización del proyecto. (2)

2.2 Open CV

Open CV es una librería diseñada para ofrecer un entorno de programación fácil y eficiente para el desarrollo de aplicaciones basadas en la visión por computador a tiempo real. Su código fue programado en C++, y está optimizado para el uso de procesadores con múltiples núcleos.

Se trata de una librería de desarrollo libre que creó *Intel* en el año 1999. Incluyendo la opción de uso multiplataforma, al haber versiones para *Linux, Mac Os y Windows.* (3)

Su uso se ha extendido a una gran variedad de aplicaciones. Incluyendo funciones como reconocimiento facial o de objetos, realidad aumentada, interacciones robot-humano, visión en estéreo (mediante dos cámaras), etc.

Siendo una librería totalmente abierta, cargada de funciones y fácil de usar, resulta la opción ideal para la realización de nuestro proyecto. Además, su uso en Android está bastante extendido, resultando tremendamente fácil de descargar e implementar.

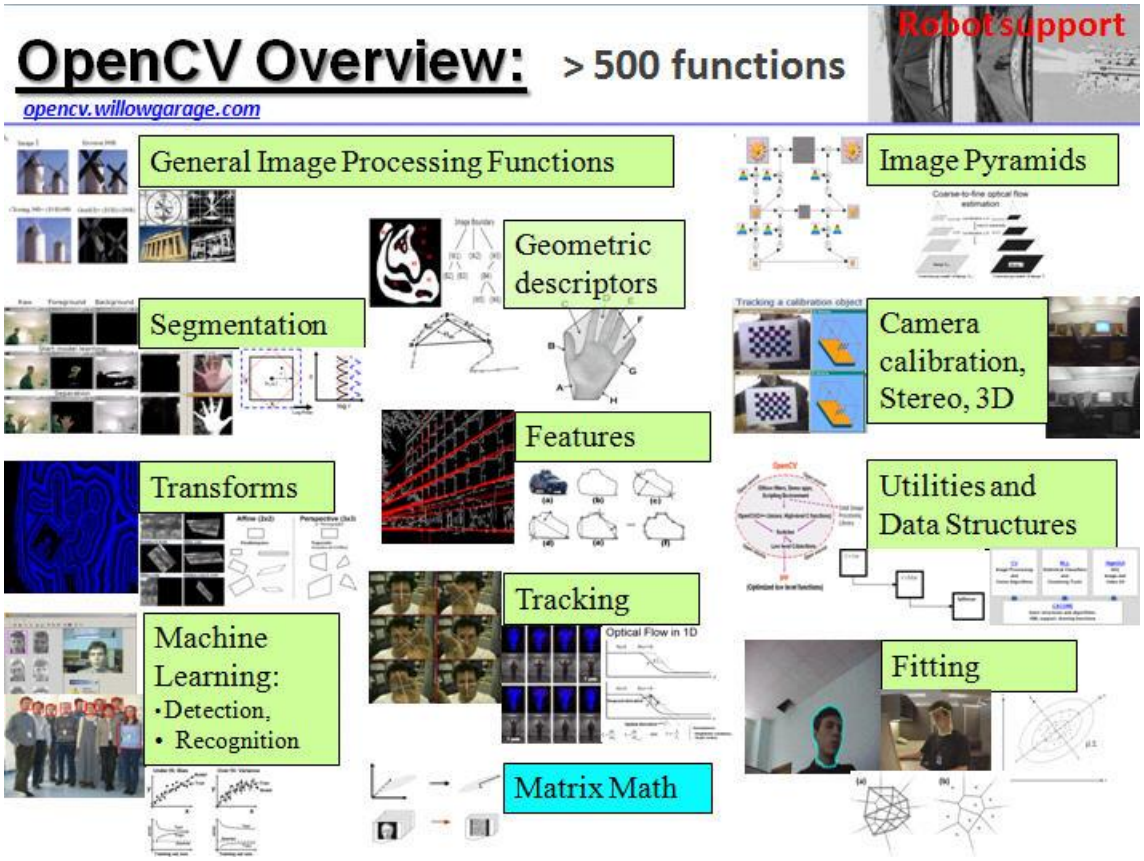


Figure 4

2.3 Visión por Computador

La visión por computador es la disciplina encargada de procesar las imágenes obtenidas del mundo real, para dar lugar a una información que sea analizable y utilizable por un ordenador. Al fin y al cabo, el funcionamiento del sentido de la vista es extremadamente complejo. Por lo que ciertas funciones de reconocimiento que nosotros realizamos sin pensar, resultan extremadamente complejas de desarrollar para un programa informático.

Figure 5

Su origen se remonta al final de los 60, cuando varias universidades que ya estaban realizando estudios en temática de inteligencia artificial comenzaron a realizar estudios en este campo. En un principio, se creyó que simplemente valdría con conectar una cámara a un ordenador, y hacer que el ordenador “describiera” la información que le llegaba. (4)

Durante los 70 se producen muchos estudios que establecen los fundamentos del campo. Técnicas de detección de bordes, modelado de formas o detección de movimiento surgen en esta época. (5)

En los 80 se produjo un aumento de estudios de ámbito matemático en el campo, que dieron lugar a grandes avances en temas de escala, calibración de cámaras y reconstrucciones en 3D entre otros.

El último gran avance se ha producido de manera más o menos reciente. Se trata de la aplicación de técnicas de aprendizaje automático. Mediante el cual, en lugar de fijar nosotros los patrones que debe seguir una imagen para clasificarse dentro de una categoría, se enseña al ordenador una amplia cantidad de imágenes de todo tipo de categorías y se deja que sea él mismo el que halle cuales son los patrones que marcan la diferencia en cada caso. (6)

Figure 6

Hay multitud de problemas que podemos encontrarnos a la hora de estudiar una imagen. Por un lado, tenemos los problemas causado por el ruido presente en una imagen digital, y que puede confundir al ordenador a la hora de analizarla. Por otro lado, tenemos los problemas causados por efectos como la perspectiva de la cámara, las diferencias de escala o el bloqueo de elementos de la imagen por otros situados delante suyo.

Para solucionar estos problemas tenemos a nuestra disposición una amplia multitud de técnicas. Basadas en las propiedades geométricas de los objetos, en los efectos de las leyes de la física o incluso en factores estadísticos.

Todo este tipo de técnicas nos permiten desarrollar programas de visión computador que cumplan todo tipo de tareas:

- Detección de objetos (detección de rostros, gestos, señales de tráfico…)
- Análisis en tiempo real (sistemas de seguridad, detectores de movimiento…)
- Visión tridimensional (Recreación de imágenes en tres dimensiones a partir de una o varias imágenes bidimensionales)

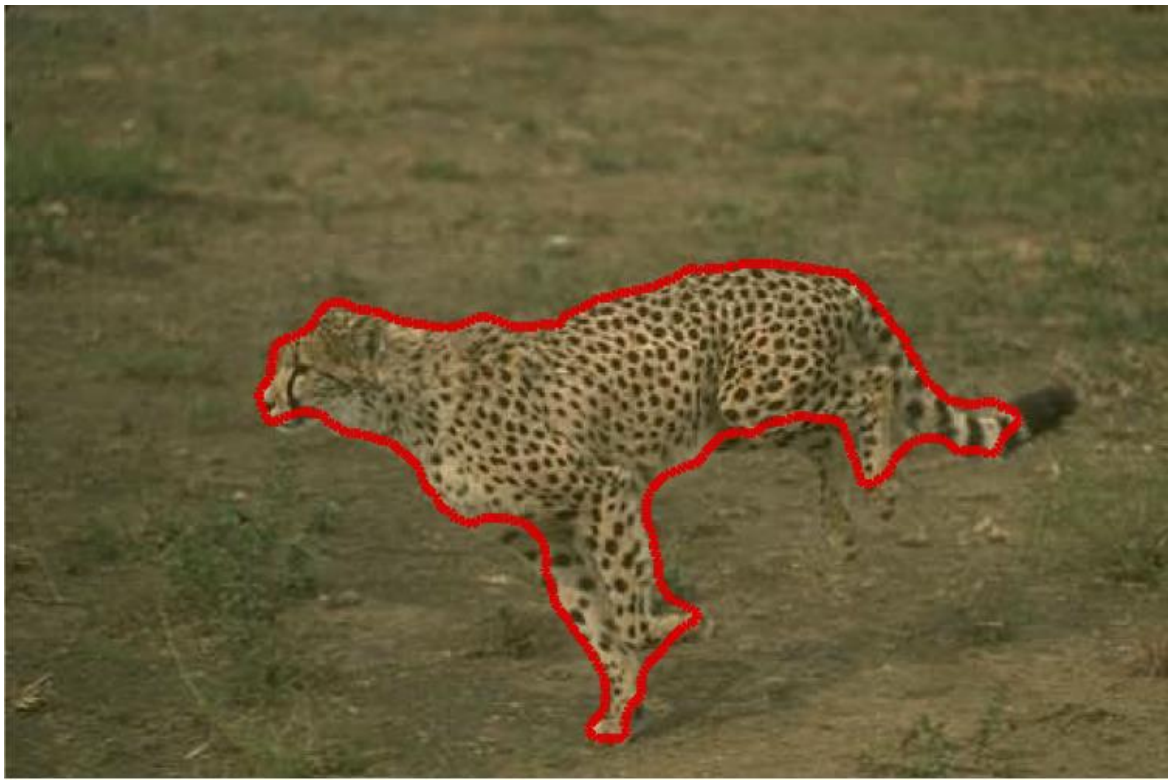


Figure 7

2.4 Sistemas de conducción automatizada

Un sistema de conducción automatizada consiste en un conjunto de sistemas de percepción, sistemas de toma de decisiones y sistemas de operación del automóvil. Su propósito es, mediante el trabajo combinado de todos estos elementos, poder realizar trayectos en un automóvil sin necesidad de ayuda humana.

En la práctica la conducción completamente automatizada aún no es posible al cien por cien. Todavía se sigue requiriendo de la labor de un conductor humano que se haga cargo de la conducción en diversos momentos de mayor complejidad.

Hasta ahora los mejores resultados se han obtenido, como cabía esperar, en la conducción por zonas poco pobladas. Siendo la conducción en el centro de una ciudad la labor más compleja de todas, debido a la alta cantidad de elementos que hay que tener en cuenta y procesar en cada instante por su alto volumen de tráfico.

Según el nivel de automatización que alcance el sistema de conducción se pueden distinguir 6 niveles (según SAE International): (7)

- Nivel 0 [No automatizado]: El conductor maneja por completo el vehículo, pero tiene algún sistema que puede alertarle como el sistema de detección de colisiones.

- Nivel 1 [Asistencia al conductor]: El conductor maneja el vehículo, pero el sistema pude alterar algunos aspectos como la velocidad o la dirección.

- Nivel 2 [Automatización parcial]: El conductor puede tomar el control si necesita ajustar algo, pero en general el sistema toma el control del coche para alguna labor en concreto. Por ejemplo, un sistema de aparcado.

- Nivel 3 [Automatización condicionada]: El sistema puede conducir el coche de forma completa, aunque sigue requiriendo de la presencia de un piloto humano para corregir cualquier posible fallo.

- Nivel 4 [Alta automatización]: El sistema no necesita de la presencia de un piloto humano (por ejemplo, el Google car).

- Nivel 5 [Automatización completa]: El sistema no solo no necesita un piloto, sino que además se puede adaptar a cualquier tipo de condición que se produzca en la conducción. Actualmente no hay ningún sistema en esta categoría.

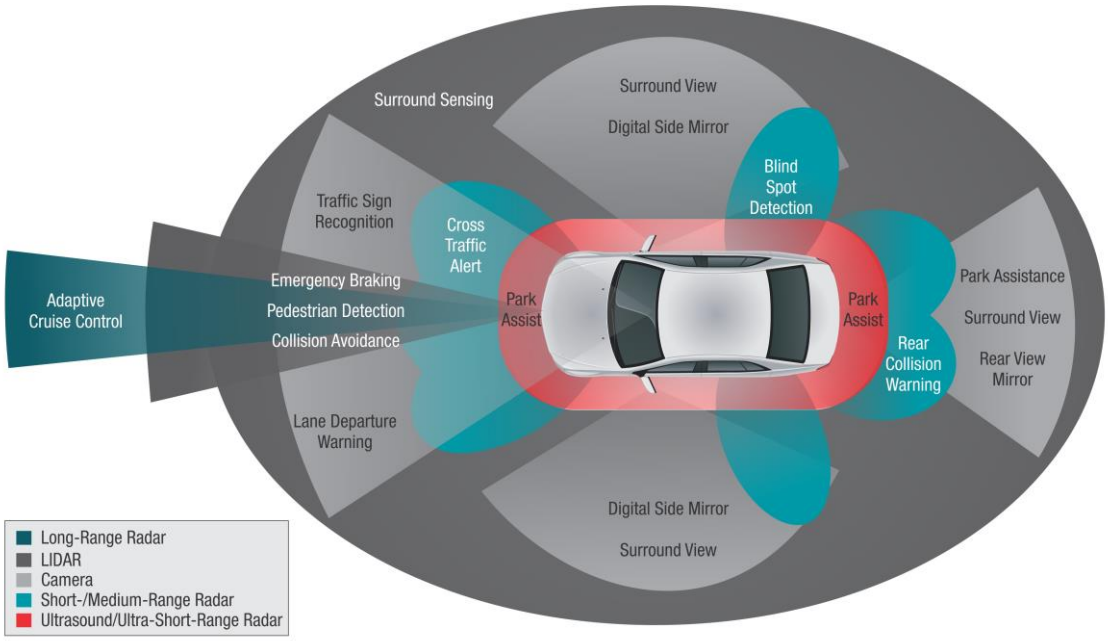


Figure 8

En la actualidad los casos más avanzados de conducción automatizada que se han producido han sido los Google car, los vehículos automatizados de Uber y los vehículos de la compañía Tesla. (8) (9) (10)

Figure 9

2.5 Sistemas de asistencia a la conducción

Se trata de los sistemas encuadrados en el nivel 1 de la clasificación que hicimos en el anterior punto. Son aquellos sistemas que facilitan la labor de conducción del vehículo, pero que, en última instancia, dejan en manos de un ser humano el manejo del mismo. (11)

Es uno de los sectores que más ha crecido en el mundo de la conducción, y uno de los más extendidos a nivel comercial. No en vano, al seguir requiriendo un humano al volante, se eliminan muchos de los problemas de seguridad y detección de errores que ocurren en el caso de los sistemas completamente automatizados.

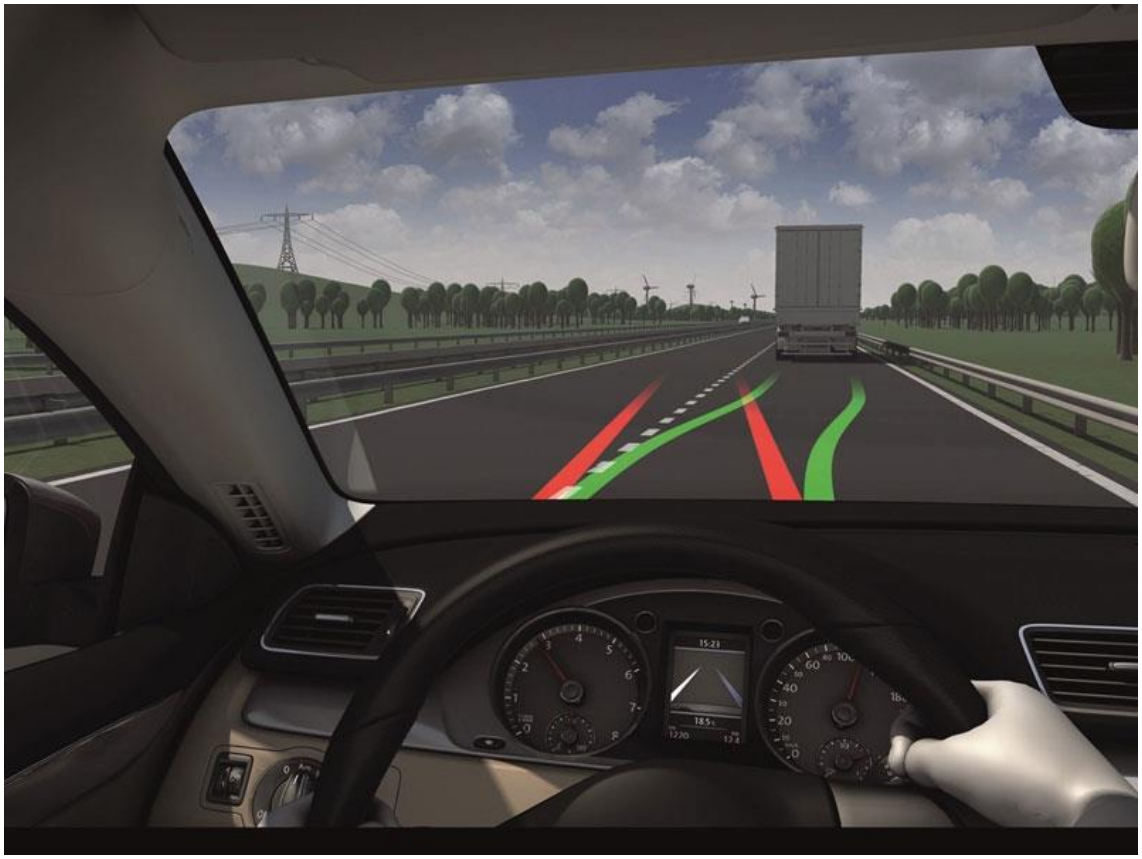


Figure 10

Algunos de los sistemas encuadrados en esta categoría son: (12)

- Sistema de aparcado automático
- Monitorización de “puntos ciegos”
- Sistema de prevención de colisiones
- Detector de proximidad de obstáculos
- Aviso de salida de carril
- Detección de conductor somnoliento
- Ayuda al descenso de pendientes
- Ayuda al cambio de carril
- Visión nocturna
- Comunicación inter-vehicular
- Detección de señales de tráfico
- Protección de peatones
- Sensor de lluvia

Y muchos otros sistemas, que siguen aumentando en número cada día. Ya sea incluidos en el sistema de navegación del coche, a través de dispositivos electrónicos independientes o como aplicaciones para un smartphone.

Figure 11

2.6 Otras aplicaciones del mercado

En el mercado de las aplicaciones son muchas las relacionadas con la conducción. Sin embargo, la mayoría se limitan a realizar tareas simples, o realizan tareas complejas, pero sin demasiado éxito.

Muchas, como la nuestra, se basan en la cámara para detectar elementos presentes en la carretera. Otras, en cambio, utilizan el GPS para proporcionarnos información que nos ayude a elegir ruta o prestar más atención a posibles problemas.

Hemos seleccionado algunas de las más comentadas, tanto dentro de la función de la nuestra, como en otro tipo de funciones, para ver todas las posibilidades que nos ofrece el mercado hasta la fecha. (13)

iOnRoad Augmented Driving

Figure 12

Esta aplicación utiliza los sensores y la cámara del móvil para avisarnos de cualquier tipo de peligro que se pueda encontrar ante nosotros. Solo hay que colocarlo enfocado a la carretera y nos mostrará información sobre la distancia de seguridad necesaria, la velocidad que debemos llevar o si conducimos dentro del carril.

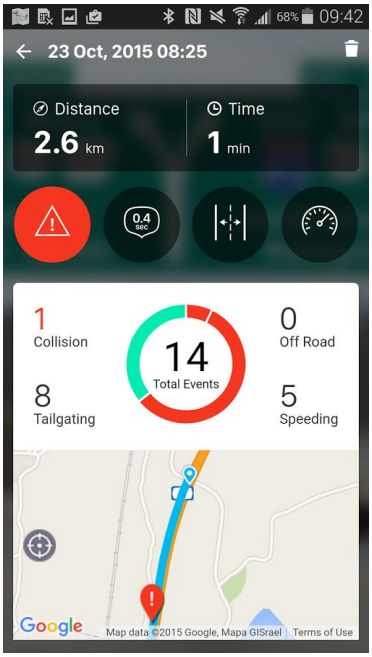


Figure 13

Sin embargo, según las últimas opiniones de la app parece que en las últimas actualizaciones de la versión gratuita su utilidad ha empeorado mucho, y se han aumentado en gran medida los errores, haciendo casi imposible su uso.

Drivies

Figure 14

No se trata exactamente de una aplicación de asistencia a la conducción, aunque mucha de las funciones que realiza están relacionadas con las aplicaciones de este campo. Por lo que resulta interesante tenerla en cuenta.

La aplicación registra datos de tu conducción para ofrecerte datos sobre la calidad de la misma. De esta forma podrá calificar tu estilo de conducción y premiarte con puntos en función de lo bien que lo hagas, así como darte consejos para que mejores. Dichos puntos podrán canjease por ofertas dentro de la propia aplicación, como descuentos en el seguro del coche.

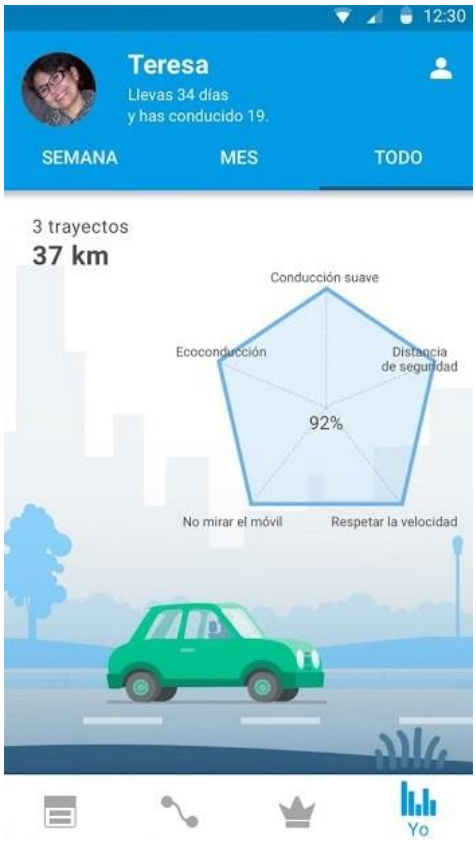


Figure 15

La aplicación analiza tu velocidad, para ver si rebasas el límite, así como acelerones o frenazos bruscos. También ofrece datos sobre la trayectoria realizada, el consumo, etc.

Según las opiniones parece que se trata de una forma de hacer publicidad a las compañías de seguros. Pero en cualquier caso la utilización de los sensores para medir velocidades y comprobar datos de la conducción resulta una idea ciertamente interesante.

Waze

Figure 16

En este caso la aplicación no recurre a la cámara para ayudarnos, sino al GPS. En función de nuestra posición, *Waze* nos dirá que trayectoria es mejor seguir. Basandose en datos de tráfico, accidentes y retenciones proporcionados en tiempo real por otros usuarios del mercado.

Figure 17

Se trata de una de las aplicaciones más exitosas de este ámbito, con alrededor de 40 millones de usuarios y el apoyo de Google, que la adquirió hace unos años.

Autoboy Dash Cam

Figure 18

Se trata de una aplicación que actúa como “caja negra” del vehículo. Recogiendo datos como la grabación de la cámara, la posición o la velocidad, para que estén totalmente disponibles en caso de accidente. De esta forma pueden resultar de utilidad en caso de un posible juicio.

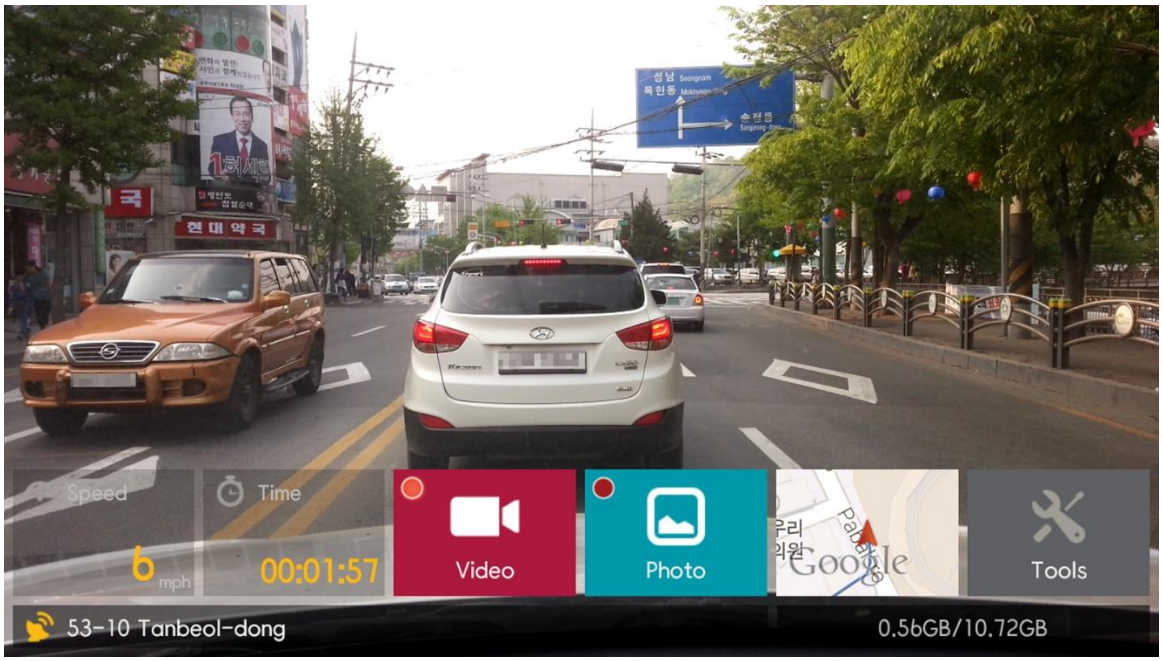


Figure 19

aCoDriver 5

Figure 20

Un de las aplicaciones más completas. Registra datos como el último límite de velocidad detectado, el carril actual en el que estamos o la distancia con otros vehículos. Todo en un sistema completamente integrado y con una interfaz de fácil comprensión.

Las opiniones son, en general, positivas. Aunque parece que aún está lejos de resultar perfecta. Sin embargo, sigue siendo una de las mejores opciones que hemos podido encontrar en android.

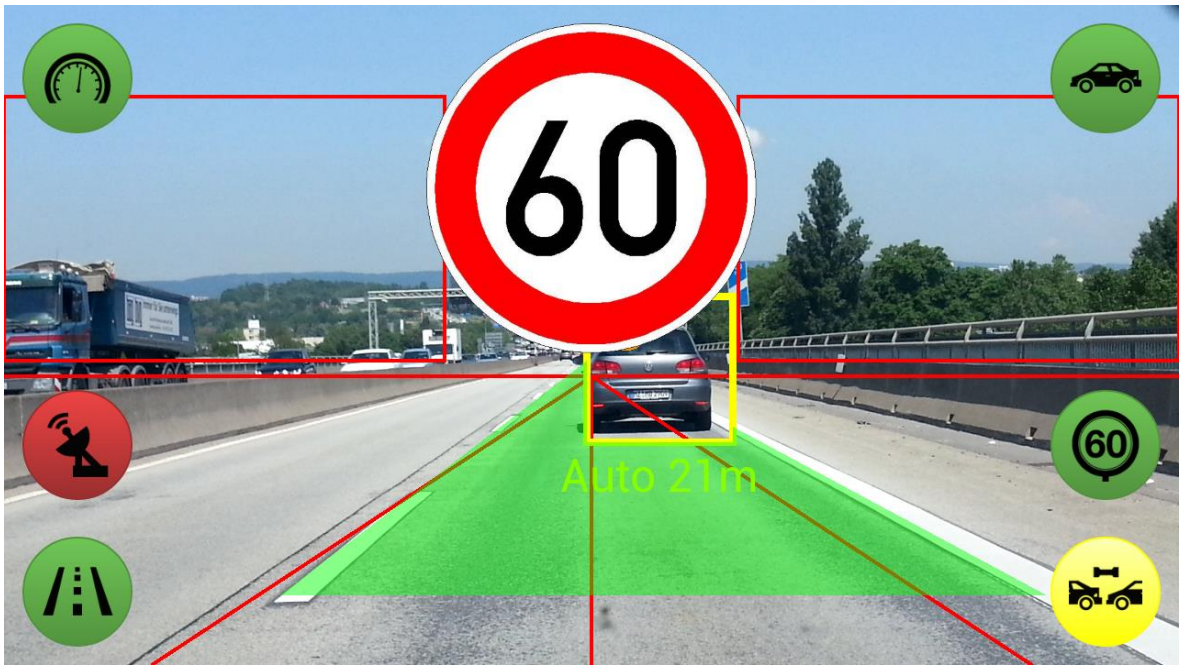


Figure 21

3. Descripción del Sistema

3.1 Propósito

Nuestra idea es realizar un programa capaz de detectar las líneas que hay en la carretera y decirnos si dichas líneas son continuas o discontinuas. Para ello hay varias consideraciones importantes que hay que tener en cuenta antes de comenzar el desarrollo de la tarea.

En primer lugar, una importante ventaja a tener en cuenta es la perspectiva que tendremos en las imágenes. Al estar colocada la cámara siempre en el mismo lugar del coche, tendremos la certeza de que las imágenes que tendremos que procesar tendrán una distribución muy parecida. Permitiéndonos así prestar mayor atención a las zonas de la imagen que sabremos que tendrán información relevante.

Figure 22

Esto será de vital importancia debido a otra de las consideraciones importantes. Y es que la capacidad de procesamiento de un móvil es mucho menor que la de un ordenador. Lo cual será un problema, dado que necesitamos que nuestro programa sea capaz de procesar todo a tiempo real. Por ello, el punto anterior será vital para reducir el procesamiento de la imagen a las zonas que sean total y absolutamente necesarias.

Esto significa quitar las zonas de la imagen correspondientes al cielo, y eliminar todas aquellas líneas detectadas que, por el ángulo que presenten, no puedan ser una línea de la carretera (líneas demasiado horizontales).

Otro aspecto a tener en cuenta es el fuerte contraste entre las líneas y la carretera. Al ser líneas de color blanco, casi siempre serán los pixeles con valores más altos de la imagen. Permitiéndonos establecer umbrales relativamente altos para quitarnos de en medio casi todas las partes de la imagen que no resulten relevantes.

La resolución de la imagen también es algo que juega a nuestro favor. Al no tener que prestar atención a pequeños detalles, por tratarse las líneas de conjuntos grandes de píxeles, podremos usar imágenes de la mínima resolución posible. Ahorrándonos así aún más capacidad de procesamiento.

Así pues, con todas estas consideraciones en mente, estamos en disposición de comenzar el desarrollo de nuestro programa con una idea bastante clara de lo que debemos hacer.

Figure 23

3.2 Aplicación inicial

Para realizar esta tarea no comenzaremos a trabajar desde cero, sino que trabajaremos sobre una aplicación ya desarrollada por el LSI. La cual incluye un menú principal con botones para las distintas funciones, y otras aplicaciones como la detección de obstáculos.

Además, también contamos con una función previa para detección de líneas de carretera incluida en la aplicación. Sin embargo, al comenzar a trabajar con la aplicación nos dimos cuenta de que no funcionaba correctamente.

Tras lograr subsanar los errores y hacer que pudiese ser compilada y ejecutada, vimos que su funcionamiento era demasiado lento para ser viable, por lo que nos dispusimos a cambiar las resoluciones de imágenes en las que trabajábamos. Así logramos que el funcionamiento fuera, por fin, fluido, pero el programa seguía sin ofrecer los resultados deseados.

Finalmente optamos por cambiar el propio algoritmo. Tomando algunas de las ideas que se usaban, pero adaptándolas a nuestro propio algoritmo que funcionase de forma correcta y más eficiente.

Figure 24

3.3 Contexto

Nuestro código se situará dentro de la aplicación desarrollada por el Laboratorio de Sistemas Inteligentes de la UC3M. Un grupo de investigación creado en el año 2000 para el desarrollo de actividades en sistemas de percepción y sistemas autónomos.

Algunos de sus miembros llevan más de 20 años dedicados a la investigación en campos relacionados con los vehículos inteligentes. Tecnologías de la percepción, sistemas inteligentes, visión por computador, transportes inteligentes o sistemas autónomos entre otros. (14)

Además de las múltiples tecnologías llevadas a cabo, el departamento ha podido desarrollar cuatro plataformas en las que controlar vehículos de forma inteligente.

Figure 25

IVVI 1.0

Sus siglas vienen de “Intelligent Vehicle based on Visual Information”. Es decir, un vehículo inteligente cuyo funcionamiento se basa en información visual. Se trata de un sistema ADAS que ayudaba a la conducción a través de varias cámaras y sensores colocados en el coche, cuya información era procesada posteriormente por dos ordenadores en su interior. Este proceso se completó en el año 2009. (15)

Figure 26

IVVI 2.0

Se trata de la segunda versión del sistema mencionado previamente. En él, además de las funciones ya desarrolladas, se incluyen otras nuevas funcionalidades. Además, el sistema entero se desarrolla de forma que esté integrado por completo en la arquitectura del coche. Sin romper la estética visual del mismo. (15)

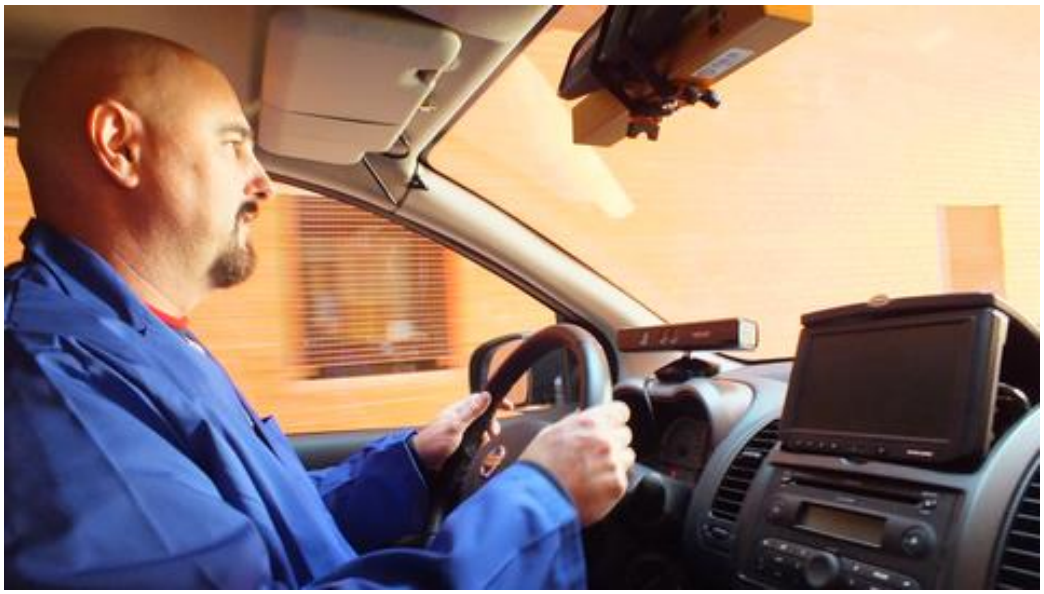


Figure 27

iCab

Se trata de dos vehículos eléctricos (carritos de golf), diseñados para llevar de forma totalmente automatizada a los visitantes de la universidad. Los vehículos los desplazarían a las localizaciones deseadas y, además, contarían con un sistema de comunicación entre ellos para poder compartir información y así trabajar de forma más eficiente. (16)

Figure 28

SkyOnyx

Se trata de un dron (vehículo aéreo no tripulado) equipado con diversas cámaras y sensores que permitan su funcionamiento de forma autónoma, en lugar de pilotado a distancia por un humano como suele ser normal. (17)

Figure 29

3.3 Herramientas de Software

Como es evidente, a la hora de programar en Android el primer paso será elegir cual va a ser el entorno de desarrollo que vamos a utilizar. En nuestro caso, la elección estaba entre Eclipse y Android Studio.

Eclipse es el entorno que lleva más tiempo utilizándose para la programación en Android, y de hecho es el entorno en el que se habían realizado, hasta ahora, los proyectos del laboratorio de sistemas inteligentes. Se trata de un entorno que, hasta la aparición de Android Studio, recibía un gran apoyo por parte de Google. Convirtiéndolo en una elección idónea. (18)

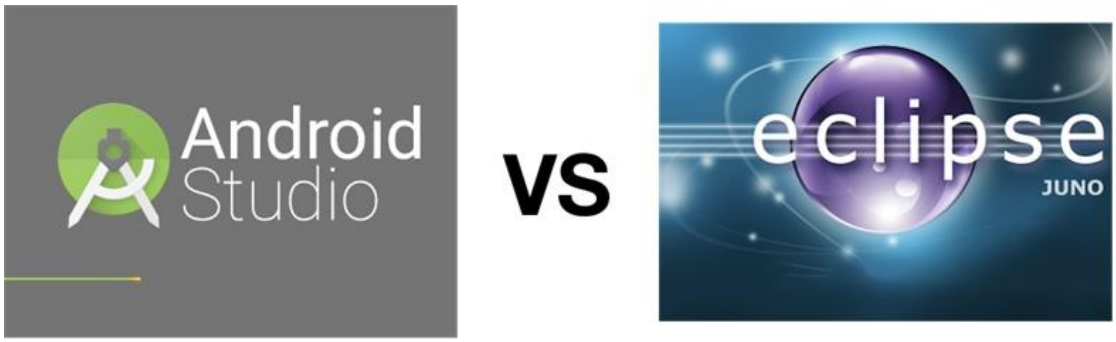


Figure 30

Por otra parte, tenemos Android Studio, un entorno iniciado por la propia Google en 2013, con el objetivo de convertirse en el principal entorno de desarrollo para su sistema operativo. (19)

En un primer momento, el departamento se decantó por Eclipse. Al ser Android Studio más reciente su funcionamiento aún no resultaba del todo óptimo. Requiriendo demasiados recursos y ofreciendo unos tiempos demasiado lentos en comparación con Eclipse. Es por esto que el departamento se había decantado por la otra opción.

Sin embargo, tras varios años en el mercado, el funcionamiento de Android Studio se ha ido mejorando enormemente. Y dado que es la opción apoyada por Google, a la hora de realizar este proyecto se decidió a dar el paso de un entorno al otro. Es por ello que, antes de comenzar a realizar ninguna labor, fue necesario adaptar la aplicación ya realizada al entorno de Android Studio. Y subsanar todos los errores que surgieron a raíz del cambio.

Otro aspecto importante es la elección de librerías. La aplicación sobre la que trabajábamos estaba desarrollada con las librerías de la versión 2.4.9 de OpenCV. Sin embargo, la versión 3.0.0 se encontraba ya completamente operativa, por lo que se decidió también cambiar esto, y adaptar el trabajo existente a las nuevas librerías.

Figure 31

Las versiones de Android capaces de soportar nuestra aplicación son todas aquellas que vayan de la 2.2 en adelante. Siendo las 4.0 y superiores las más idóneas para su funcionamiento. Sin embargo, debido al alto coste computacional de la aplicación, resulta poco realista pensar en su funcionamiento en móviles tan antiguos como para usar versiones tan bajas.

4. Implementación

4.1 Estructura de la aplicación

Nuestra aplicación se dividirá en tres apartados distintos, que se realizaran cada uno a continuación del siguiente.

4.1.1 Tratamiento previo de la imagen

En primer lugar, nos aseguraremos de aplicar a nuestra imagen los tratamientos suficientes para que se pueda analizar con facilidad. Este es el apartado que conllevará más trabajo y mayor coste computacional, por lo que debemos tener especial cuidado en su desarrollo.

Para empezar, ajustaremos la imagen obtenida por la cámara a la menor resolución posible, para disminuir en gran medida el coste de procesamiento. Esto resulta vital, pues cualquier aumento en la resolución por encima de lo necesario supone un aumento enorme de la carga de trabajo. Pues al tener que recorrer la matriz imagen pixel a pixel el aumento en la resolución aumenta los cálculos de forma exponencial.

Una vez hecho esto estableceremos una ROI en las ¾ partes inferiores de la imagen, para quitarnos así la parte de la imagen correspondiente al cielo, cuya información no nos interesa. La ROI seleccionada podría ser aún menor, pero se decidió dejarla así para no correr riesgos. En una versión final implementada en un vehículo podría calibrarse mejor y elegir exactamente la parte de la imagen justa y necesaria.

Tras esto, binarizaremos la imagen, y le aplicaremos un filtro canny para eliminar cualquier posible ruido que influya en nuestros cálculos. El resultado será una imagen completamente negra que tendrá en blanco casi exclusivamente las líneas de carretera.

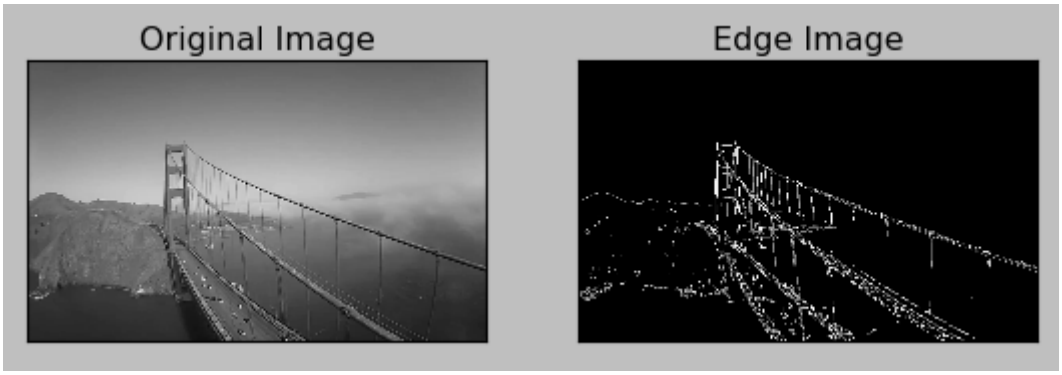


Figure 32: ejemplo del efecto del filtro canny

Utilizaremos ahora el algoritmo de detección de líneas HoughLinesP para detectar las líneas de la imagen. Calculando el ángulo y la distancia de las mismas para eliminar aquellas que, por ser demasiado horizontales o demasiado cortas, resulte más probable que sean una interferencia y no una línea de interés. En total, nos quedaremos con 3 lineas, una a cada lado y otra en medio.

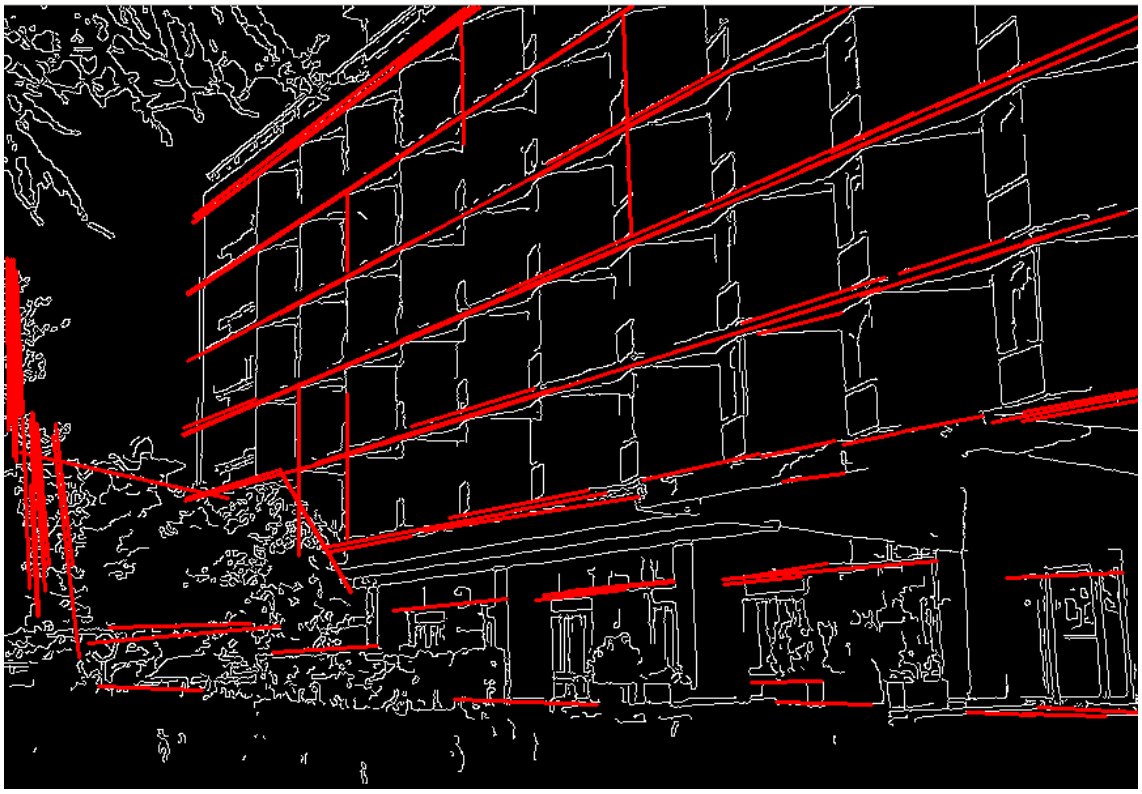


Figure 33: ejemplo houghlinesp

Una vez hallados las líneas principales, veremos donde se cortan y utilizaremos ese dato como punto de fuga para aplicar una transformación a la imagen. De esta forma el resultado que obtendremos será una imagen a vista de pájaro de la carretera.

En dicha imagen la posición y orientación de las líneas, al haber sido la transformación hecha en función de estas, será siempre la misma. Por tanto, nuestro proceso termina aquí. Pues ya hemos conseguido convertir cualquier posible imagen que nos llegase a un formato que sea siempre igual, totalmente idóneo para su análisis.

4.1.2 Análisis de la imagen

Con la imagen obtenida previamente nos resultara muy fácil estudiar las posibles líneas. En nuestro caso, por la naturaleza de nuestro algoritmo, tendremos las 3 posibles líneas de la carretera situadas siempre a la misma altura.

Por tanto, lo que haremos será contar la cantidad de pixeles blancos a distintas alturas de la imagen, en las zonas en las que sabemos que hay línea. Si encontramos a pixeles blanco a todas las alturas, sabremos que se trata de una línea continua. Si encontramos solo en algunos tramos, será una discontinua. Si no encontramos ninguno, entonces es que no hay línea (esto solo será posible para el caso de la línea de en medio, en el caso de carriles únicos).

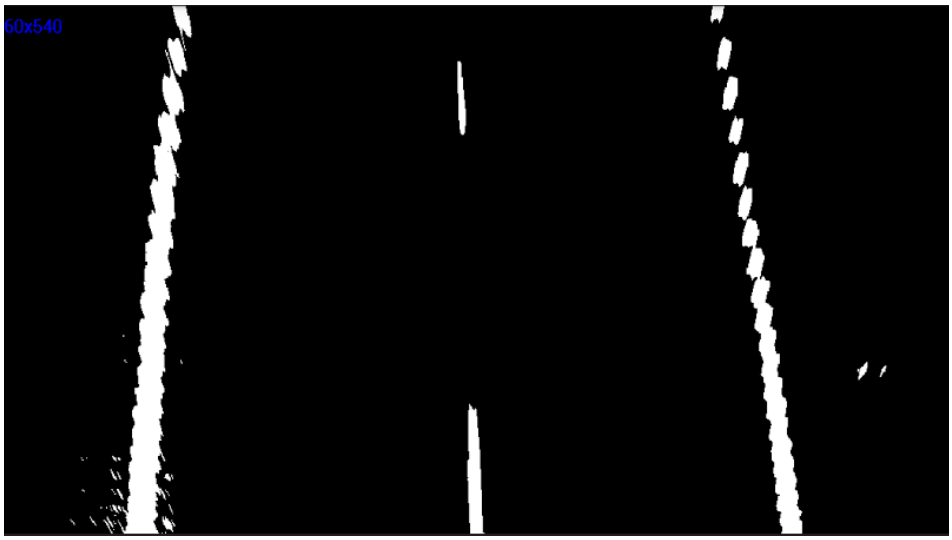


Figure 34: ejemplo de carretera con el cambio de perspectiva

Tras esto ya tendremos identificadas las líneas que hay, y de qué tipo son. Lo cual era nuestro objetivo inicial. Solo nos quedará mostrar esta información de la manera adecuada.

4.1.3 Presentación de los resultados

Este apartado está desarrollado de manera algo más libre, dado que, en principio, dependiendo de la función que queramos dar a nuestro programa, haremos una cosa u otra con los datos obtenidos.

Si, por ejemplo, quisiésemos implementar nuestro programa en un sistema de conducción automatizado, tendríamos que devolver los datos de forma que pudiesen ser usados por las siguientes funciones del programa.

En nuestro caso, al ser una aplicación para el móvil, nos limitaremos a mostrárselo por pantalla al usuario de una manera sencilla y adecuada.

Para ello nos decantamos por un sistema de colores sobreimpresionado sobre la imagen original obtenida por la cámara. Para lo cual cogeremos las líneas principales que teníamos detectadas mediante HoughLinesP, y las pintaremos sobre la imagen original a color.

El código de colores será verde para líneas continuas, azul para discontinuas, y rojo si no hay línea.

Además, se incluirán 3 círculos en la esquina superior derecha, que seguirán el mismo código de colores, para facilitar aún más la comprensión de la información por parte del usuario.

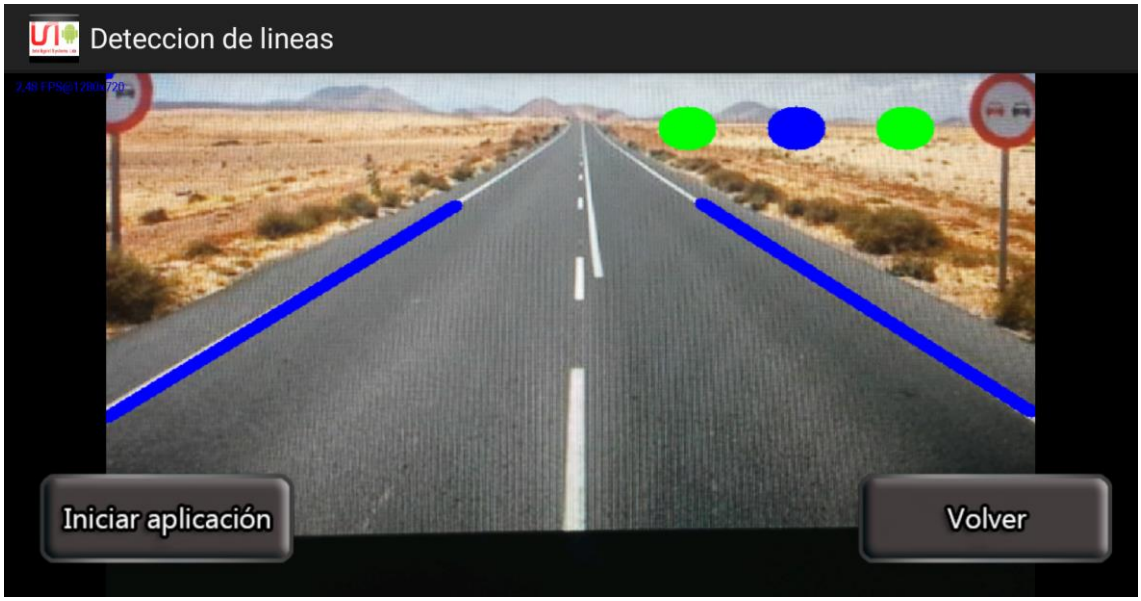


Figure 35: ejemplo del formato en el que se muestran los resultados

4.2 Preprocesamiento de la imagen

Nuestro algoritmo de detección de líneas irá colocado dentro de la función *onCameraFrame*. Ésta es una función de *Android* que recibe una matriz, con la información del frame que acaba de capturar la cámara del dispositivo, y devuelve otra matriz, que en nuestro caso será la imagen una vez le hayamos aplicado todo nuestro código.

Esto significa que debemos asegurarnos de que nuestro código sea lo suficientemente ligero. De lo contrario, el tiempo entre frames será demasiado elevado, por lo que la aplicación no se ejecutará con la fluidez adecuada para que su funcionamiento sea viable.

Una vez comprendido el ciclo del algoritmo y la función en la que se ejecuta, pasamos a explicar cada uno de los pasos que seguirá nuestro programa:

4.2.1 Ajuste del tamaño de la imagen

Como explicamos anteriormente, nuestro programa no requiere de una alta resolución para su ejecución. Las líneas a detectar son suficientemente grandes como para detectarse sin problema a la mínima resolución posible, por lo que reduciendo el tamaño de la imagen no solo disminuimos enormemente el tiempo de ejecución, sino que la única información que perdemos sería posible ruido.

Por tanto, nada más comenzar el código lo primero que haremos será usar la función *Resize* para cambiar la resolución del frame recibido a 640x480. No obstante, deberemos volver a utilizar la función para devolver la imagen a su tamaño original antes de que finalice el código. Pues la función *onCameraFrame* solo puede devolver matrices del mismo tamaño que la que recibió.

4.2.2 Binarización y región de interés

Una vez en el tamaño adecuado, utilizaremos la función *Threshold* con un valor de umbral de 200. De esta forma, obtendremos completamente negra salvo por los pixeles de la imagen que fueran prácticamente blancos. Facilitándonos enormemente la detección de líneas.

Cabe destacar que la elección del valor exacto se puede ajustar en función de las condiciones de uso. Por tanto, una vez implementado el sistema en un vehículo, sería de gran utilidad hacer este valor seleccionable en una labor de calibración previa al funcionamiento. En nuestro caso, al ser una versión simplificada para móvil, donde no podemos permitirnos aumentar la capacidad de procesamiento, hemos dejado el valor fijo.

Dado que no toda la imagen posee información que nos resulte útil, vamos a eliminar aquellas partes que no nos sirve, que en este caso será las correspondientes a la parte superior de la imagen recibida.

Para ello estableceremos una región de interés. Con la función *submat* extraeremos una submatriz de nuestra imagen, quedándonos con el espacio situado entre los puntos (0, 0, 640, 120). Esto significa que nos quedaremos con las ¾ partes inferiores de la imagen.

Nuevamente, la zona exacta que nos quedemos también podría variarse. Por tanto, una vez instalado el sistema, también podría recalibrarse este valor para asegurarnos de que nos quedamos con la región estrictamente necesaria.

4.2.3 Filtro Canny

Se trata de un algoritmo de detección de bordes, mediante el cual reduciremos aún más la información de la imagen para quedarnos solamente con lo necesario.

Dicho algoritmo aplica, en primer lugar, un filtro para eliminar el ruido de la imagen. Ya que, al estar basado en la primera derivada gaussiana, cualquier pixel de ruido podría afectarle significativamente.

$$\mathbf{B} = \frac{1}{159}\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix} * \mathbf{A}$$

Figure 36

Tras esto, el algoritmo de canny utilizará cuatro filtros para detectar bordes en las direcciones horizontal, vertical y ambas diagonales.

$$\mathbf{G} = \sqrt{\mathbf{G}_x{}^2 + \mathbf{G}_y{}^2}$$
$$\mathbf{\Theta} = \arctan\left(\frac{\mathbf{G}_y}{\mathbf{G}_x}\right)$$

Figure 37

4.2.4 aplicación de HoughlinesP

Ahora nos dispondremos a comenzar con la detección de líneas propiamente dicha. Para ello utilizaremos la función de hough lines probabilística. Una función basada en la transformada de hough, una técnica utilizada para la detección de formas geométricas en una imagen. En este caso, líneas.

Utilizamos la versión HoughLinesP, es decir, la probabilística, en lugar de la estándar. Esto nos ayudará a ahorrar cálculos, pues la versión probabilística no examina todos los puntos de la imagen, sino que solo necesitar examinar un conjunto de puntos al azar para detectar las líneas. Ofreciéndonos resultados similares a la vez que optimizamos.

Dicha función nos devolverá los puntos de inicio y de fin de todas las rectas detectadas en nuestra imagen. Sin embargo, no necesitamos todas ellas, sino tan solo las necesarias para la detección de carriles. En nuestro caso, 3 líneas, para así detectar 2 carriles.

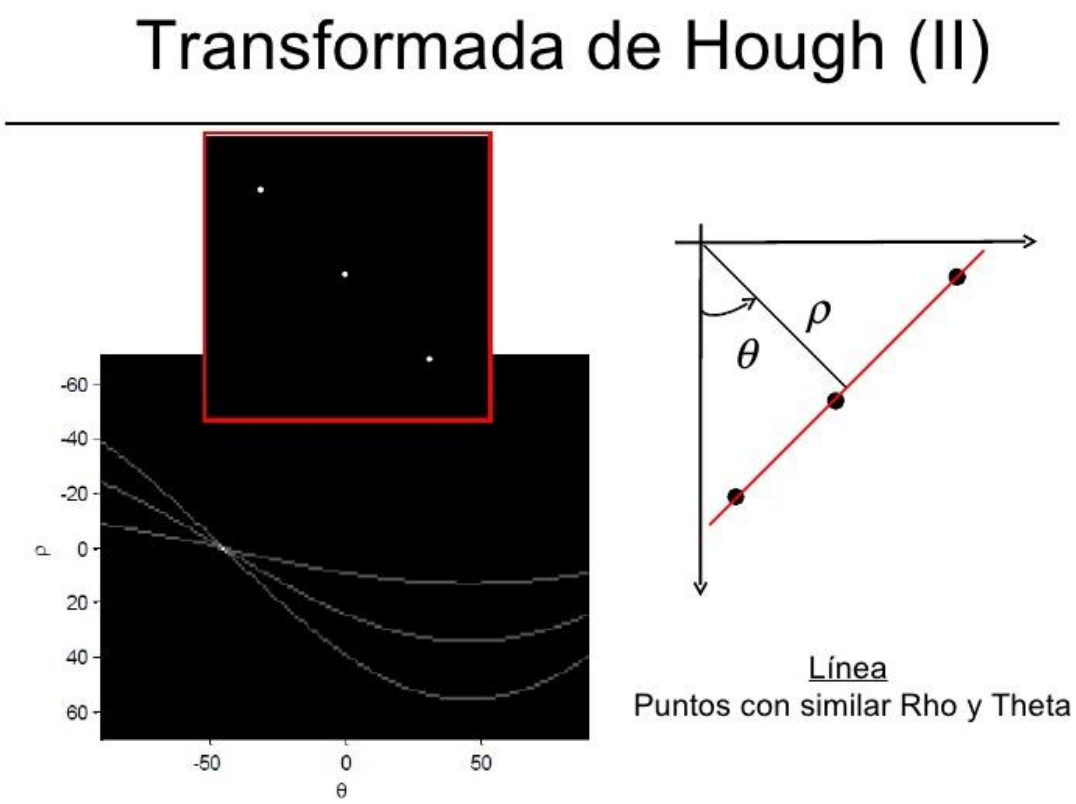


Figure 38

4.2.5 Selección de las 3 líneas principales

Para la elección nos basaremos en las condiciones que sabemos que deberán cumplir dichas líneas. En primer lugar, con el umbral que establecimos previamente en la transformada de Hough, nos aseguramos quitarnos todas aquellas líneas que, por su corta distancia, no podían formar parte de un carril.

De las restantes, nos basaremos en el ángulo que sabemos que deben formar para poder elegirlas. Así pues, sabemos que la línea central será prácticamente vertical, mientras que las laterales presentaran una ligera inclinación hacia dentro, debido a la perspectiva de la cámara.

Por tanto, estableceremos 3 bucles que recojan todas las líneas que cumplan las condiciones angulares para pertenecer al carril izquierdo, central y derecho respectivamente. Y, de entre todas ellas, nos quedaremos con la que tenga mayor longitud.

De esta forma ya tendremos seleccionadas las 3 líneas de nuestro carril. Ahora tendremos que detectar si son continuas o discontinuas.

4.2.6 Cambio de perspectiva

A continuación, cambiaremos la perspectiva de nuestra imagen, de forma que las líneas de la carretera queden completamente perpendiculares. De esta manera, resultara mucho más sencillo analizar si son continuas o discontinuas.

Para ello nos serviremos de la función *Imgproc.WarpPerspective*. Dicha función aplicará una transformación geométrica a la matriz de la imagen, basándose en unos parámetros que le hayamos introducido previamente.

Dichos parámetros consistirán en la posición de 4 puntos en la matriz original, y la posición que tendrán dichos puntos en la nueva matriz. En base al cambio de tales puntos, la función recolocará todos los demás puntos de la imagen.

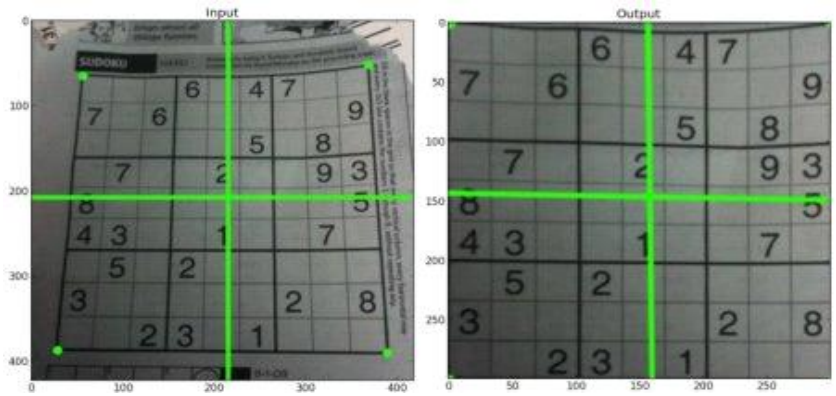


Figure 39

En nuestro caso, los 4 puntos originales que tomaremos serán los correspondientes al inicio y final de las 2 líneas laterales detectadas. Como puntos de destino, elegimos (100,0) (100 480) (540 480) (540,0). De esta forma las líneas nos quedaran a 100 pixeles de distancia de los bordes laterales, e irán de arriba a abajo hasta los bordes superior e inferior.

4.3 Análisis de la imagen y presentación de los resultados

La parte más compleja del proceso ya está realizado, hemos procesado la imagen por completo hasta tenerla en un formato ideal para su análisis. Ya solo nos queda realizar dicho análisis, extraer los resultados, y mostrarlos al usuario de una forma adecuada.

4.3.1 Análisis

Como sabemos la posición de las 3 líneas, solo nos faltaría saber si con continuas o discontinuas. Para ello lo que haremos será, para cada una de ellas, contar la cantidad de píxeles blancos que hay a distintas alturas.

De esta forma, si en alguna de ellas no encontramos píxeles blancos en una o varias alturas, significará que dicha línea es una línea discontinua. De encontrarlos a todas las alturas posibles, significará que es continua.

Cabe destacar que, como el algoritmo funciona en base a las 2 líneas laterales, la presencia de ellas esta asegurada (de lo contrario no habría carril). Sin embargo, podría darse el caso de ser un carril único. Por tanto, también incluimos la opción de, en caso de no encontrar píxeles blancos a ninguna altura, mostrar como resultado que no hay línea en medio. Y por tanto es un solo carril.

Ya tenemos, pues, toda la información extraída de la imagen. Solo nos quedaría mostrarla por pantalla de forma adecuada para el usuario.

4.3.2 Presentación de los resultados

Para mostrar los resultados, nos decidimos por sobre impresionar los datos en la imagen original capturada por la cámara. Es decir, cogeremos la matriz que guardaba el frame en color (recordemos que, hasta ahora, habíamos trabajado sobre la matriz del frame en blanco y negro) y mostraremos la información sobre ella.

Dado que las posiciones de los puntos de inicio y final de las líneas son las mismas, sea la matriz en color o blanco y negro, bastará con usar la función *Imgproc.line* para pintar las 3 líneas detectadas.

Además, usaremos un código de color para mostrar de forma aún más sencilla si la línea es continua y discontinua. Pintando la línea de verde en caso de ser continua, y de azul si se trata de una línea discontinua, en cuyo caso pintaremos únicamente el tramo más largo de la misma. Al considerar que no merece la pena el coste operacional de detectar todos los tramos discontinuos y pintarlos.

Además, en la esquina superior derecha de la imagen, incluiremos 3 círculos, que colorearemos siguiendo el mismo código usado previamente. De esta forma nos aseguramos de que la información queda totalmente clara. Además de permitirnos pintar un círculo rojo en caso de no haber línea detectada en el tramo de en medio.

Ya solo nos queda volver a usar un resize para devolver la matriz a su tamaño original, y la función está completa.

4.3.3 Otras funciones

Además del algoritmo principal, que ya hemos descrito, nuestro programa requiere de algunos aspectos que también hemos debido programar.

Por una parte, está la integración de nuestro algoritmo en la aplicación del departamento del laboratorio de sistemas. Más adelante entraremos más en detalle sobre este proceso, simplemente mencionaremos aquí el hecho de que, al haber sido programado sobre dicha aplicación, no fue necesaria la parte de programación del *AndroidManifest* destinada a permitirnos operar con la cámara, al menú inicial, etc. Simplemente nos aseguramos de que funcionará, y comenzamos a programar sobre el mismo.

El otro aspecto a destacar es la inclusión de botones, para permitirnos ver 2 imágenes distintas. Por un lado, una versión en blanco y negro, con los puntos de las líneas detectados, y por otro la imagen final con toda la información a color mostrada por pantalla.

Esto se hizo así para facilitar su utilización desde el punto de vista del programador. Al no ser una versión final, instalada directamente sobre el coche, con la calibración realizada. Hay muchos aspectos de perspectiva que variaran el funcionamiento de la aplicación.

Es por eso que el modo alternativo de visión, nos permitirá ajustar el ángulo del móvil hasta dejarlo en uno adecuado para su funcionamiento. Sustituyendo así la labor de calibración que habría de hacerse en una hipotética versión final.

4.4 Consistencia temporal

Una vez realizada esta primera versión del programa, pensamos en una posibilidad de mejorar aún más su funcionamiento. Por eso realizamos una segunda versión, para posteriormente comprobar los resultados de ambas y ver si la mejoría merece la pena.

Esta nueva versión añade un apartado de consistencia temporal. Básicamente, lo que hacemos es tener en cuenta que es poco probable que ocurran cambios bruscos en una línea. Y que, si es continua, lo más probable es que en el siguiente frame siga siéndolo.

Lo que haremos será, en lugar de definir si la línea es continua o discontinua solo por el frame actual, hacerlo teniendo en cuenta el actual y los inmediatamente anteriores. De forma que la decisión de si es continua o no, se haga teniendo en cuenta varios resultados y no solo uno.

Con este método, lo que hacemos es cubrirnos las espaldas para que, en caso de un error puntual en un frame concreto, no se produzcan cambios bruscos. Sin embargo, hay que encontrar un equilibrio adecuado, para asegurarnos de que, si una línea cambia realmente de estado, el programa no tarde demasiado en registrarlo.

El mayor problema que encontramos a la hora de afrontar esta nueva función, es que, por la naturaleza del programa de android, nuestro algoritmo se ejecuta desde 0 cada vez que llega un nuevo frame. Haciendo imposible guardar las variables del mismo de una ejecución para otra.

Deberemos, por tanto, crear un sistema de variables globales para *Android*. Dado que el sistema operativo tiene por sí mismo la posibilidad de usarlas. Lo que haremos será crear una nueva clase java en la que crear estas variables. Y, en dicha clase, crear unas funciones *get* y *set*, con las que poder leer y escribir las variables allí creadas.

De esta forma lo que haremos será, en cada ejecución de nuestro algoritmo, utilizar el *get* para cargar los resultados previos. Entonces, procederemos a analizar el resultado del *frame* actual, y guardarlo junto a los previos, a la vez que eliminamos el resultado más antiguo que tuviéramos almacenado para hacer sitio al nuevo.

Tras esto, seguiremos con la ejecución estándar de nuestro código, con la salvedad de que, al llegar al apartado de presentación de resultados, pintaremos las líneas de un color o de otro en base al resultado actual y a los previos.

Finalmente, antes de que acabe el ciclo del algoritmo, usaremos el *set* para guardar los resultados de las líneas, y así poder cargarlos en el siguiente ciclo.

4.5 Integración en la app

A la hora de realizar nuestro programa teníamos dos opciones: realizar el programa por separado y posteriormente integrarlo en la aplicación del laboratorio de la universidad, o comenzar a programarlo directamente sobre la aplicación.

En un principio, la idea de programarlo por separado parece más simple, pues nos permite comenzar a trabajar directamente sobre el algoritmo con una versión más simple. Ahorrándonos posibles problemas que surgieran al tratar el problema de forma global.

Sin embargo, hay que recordar que la aplicación original estaba realizada en eclipse, y usaba una versión distinta de las librerías de Open CV. Eso nos llevó a decidir. que la primera labor que haríamos sería la adaptación de la aplicación al nuevo formato.

Por tanto, una vez esta labor estaba concluida, parecía más sencilla trabajar directamente sobre ella. Pues de todas formas ya habíamos comenzado a tratarla y habíamos tenido que comprobar que funcionase correctamente.

Así pues, una vez adaptamos el código previo que disponíamos a las versiones y programas que íbamos a usar, nos dispusimos a programar sobre el mismo hasta implementar por completo nuestra función.

5. Resultados

5.1 Instantáneo

En primer lugar, vamos a comentar los resultados correspondientes al funcionamiento del algoritmo sin el añadido posterior que tenía en cuenta los resultados previos.

5.1.1 Casos

Tras hacer las pruebas sobre un conjunto de 50 imágenes de carreteras, hemos seleccionado un total de 15 sobre las que hablaremos aquí, para que sirvan como muestra del conjunto completo:

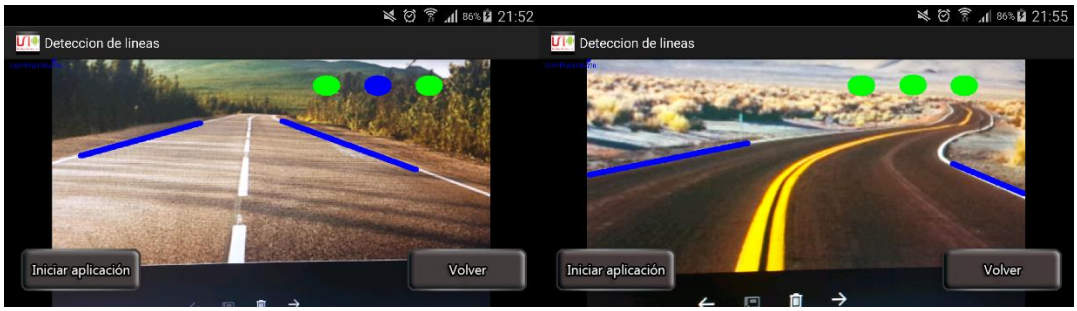


Figure 40

Aquí tenemos un buen ejemplo de la aplicación en funcionamiento. Vemos como la detección es correcta, pese a la presencia de sombras que podrían interferir en su funcionamiento.

En este caso, podemos ver como la aplicación detecta el tipo de las líneas de forma correcta. Pese a ser la línea central doble, y de color amarillo, es capaz de detectar que se trata de una línea continua.

Sin embargo, vemos como a la hora de pintarla el programa falla. La explicación más lógica es que el programa ha podido detectarla y analizar su continuidad una vez realizado el cambio de perspectiva. Pero no ha podido detectar correctamente la línea mediante houghlines antes de dicha transformación.

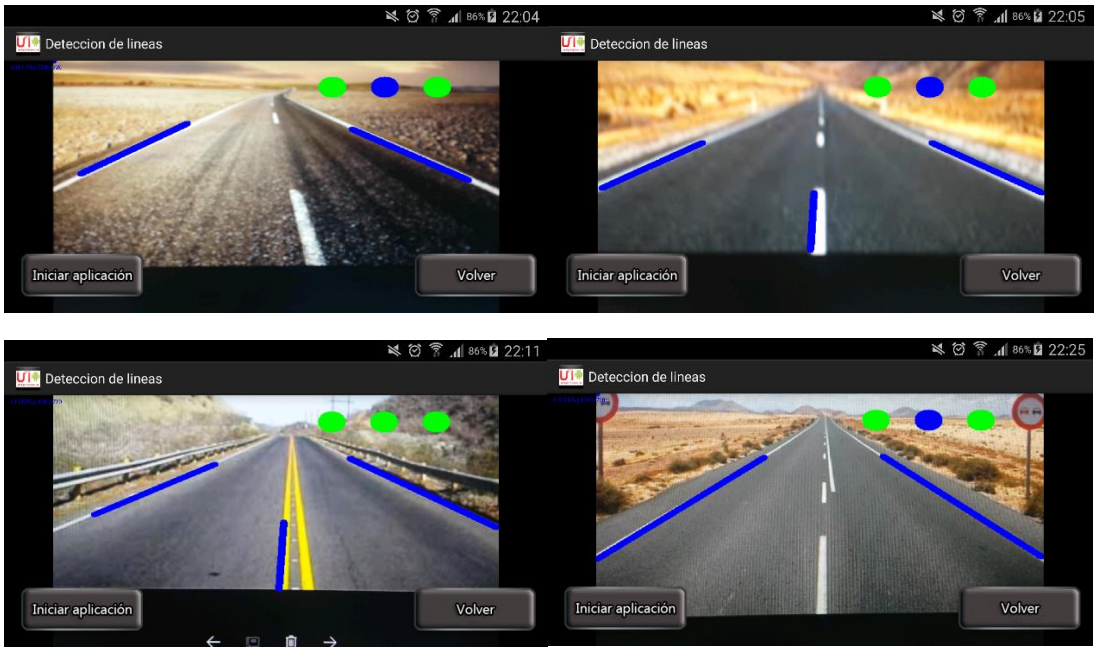


Figure 41

Nuevamente nos encontramos con un acierto. La posición y el tipo de cada línea está perfectamente identificado. Cabe destacar, además, que nuevamente estamos ante un caso donde la iluminación no es la idónea. Pues, como se puede observar, la forma en la que llega la luz hace que el carril izquierdo tenga un tono casi blanco.

Otro caso de acierto del programa. En esta ocasión era bastante fácil, pues las condiciones de la carretera y de la imagen eran óptimas. Sin embargo, destacamos el hecho de que, al ser el primer tramo de la línea discontinua lo bastante largo, el programa también la ha pintado.

Nuevamente un caso de acierto. Este lo traemos como contrapartida de uno de los ejemplos anteriores, en los que, al haber dos líneas amarillas centrales, las detectaba como continuas, pero no era capaz de pintarlas.

En este caso, podemos observar que la línea si ha sido pintada. Así que, por comparativa con la anterior, deducimos que el fallo previo pudo deberse a una excesiva curvatura en la línea, que impidiera que la función Houghlines la detectase antes de la transformación de perspectiva.

Vemos como en este caso el programa también ha acertado. Pero se produce un hecho curioso. Se trata de un cambio de línea discontinua a línea continua y, como es lógico, el programa lo detecta como una discontinua. Esto se debe a que, mientras haya algún hueco, el programa la clasificará de este modo. Por tanto, a la hora de usar el programa hay que tener en cuenta que, en los cambios de tipo de línea, no se detectará hasta que el cambio sea completo.

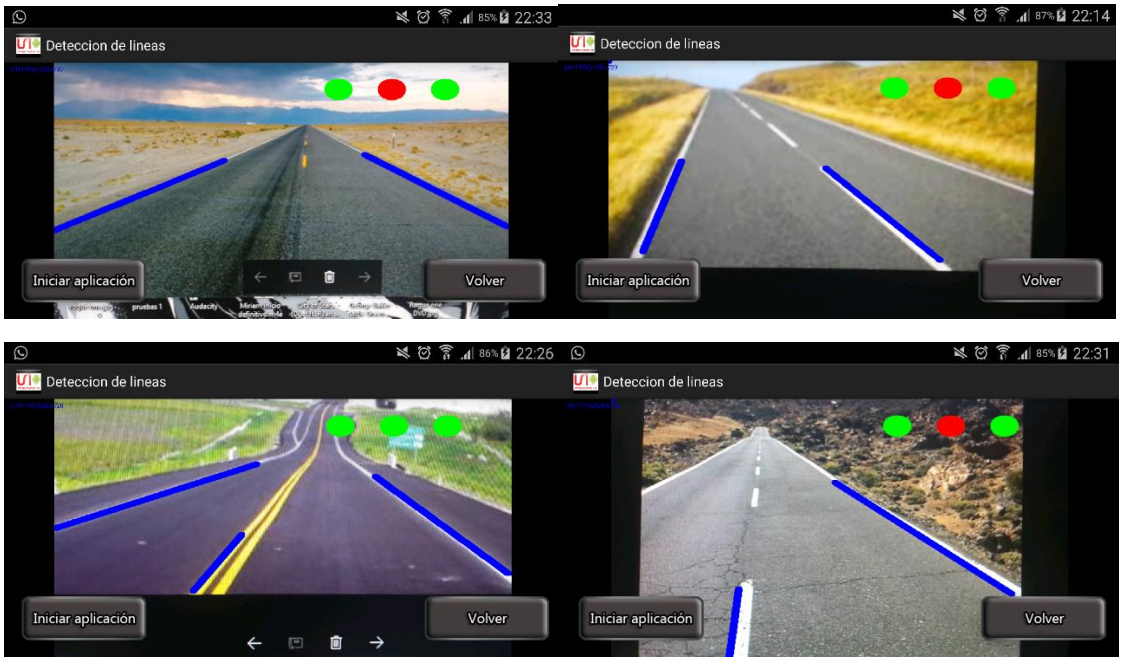


Figure 42

Aquí podemos comprobar como el programa detecta una ausencia de línea de en medio, pese a que podemos ver unos pocos tramos discontinuos al fondo. Lo más lógico es que, al estar dichos tramos demasiado alejados, no produzcan una cantidad suficiente de pixeles para ser detectados. Por lo que el programa lo ha identificado como una ausencia.

Un clarísimo caso de error. El programa ha fallado a la hora de detectar la línea de la derecha. Esto se debe a que la extraña perspectiva ha forzado que la línea de en medio cumpla las características angulares que nuestro algoritmo pedía para clasificar una línea como lateral derecha. Y, al ser además la de en medio suficientemente larga, el programa la ha clasificado como la línea principal en el lateral derecho.

Posteriormente, al hacer el cambio de perspectiva entre esta y la de la izquierda, es evidente que en medio no quedará ninguna línea por detectar. Así que el programa asume que está ante una carretera de un único carril.

Otro caso con doble línea amarilla que el programa logra acertar. Aunque cabe añadir que el tramo de línea que ha pintado ha sido muy corto, pese a ser una línea continua. Podemos deducir, por tanto, que las líneas amarillas son más susceptibles a presentar error, seguramente debido a que, al no ser blancas, el valor de sus píxeles estará más cerca del umbral que seleccionamos.

Nuevamente un tramo erróneo en el que el programa falla a la hora de detectar una línea lateral, y acaba clasificando la carretera como vía de carril único. En este caso, resulta obvio que la extraña perspectiva, que deja fuera de la imagen gran parte del carril izquierdo, es la que ha provocado el error.

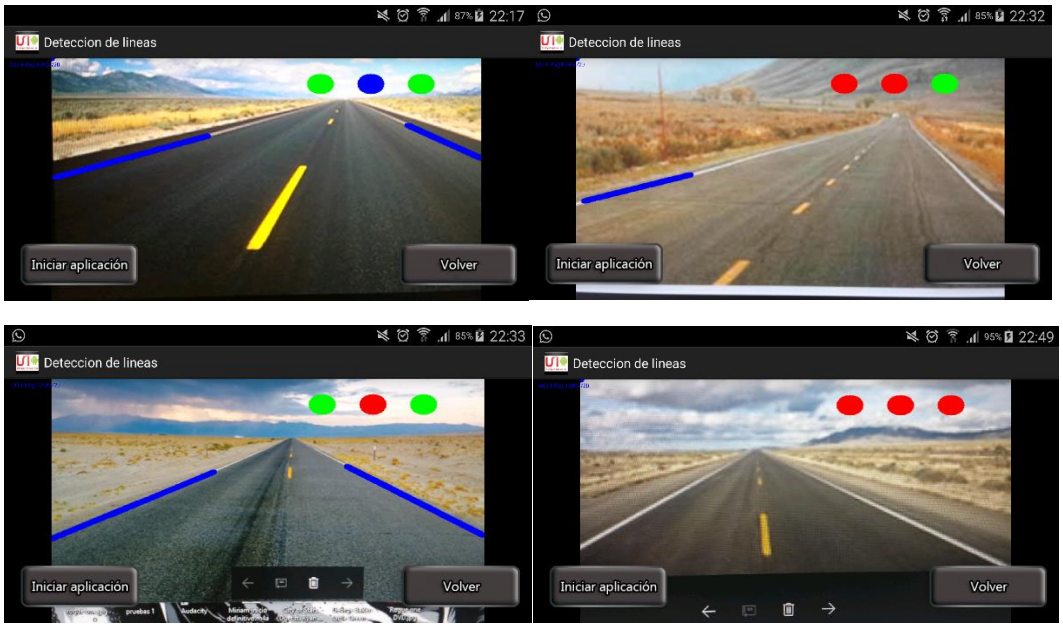


Figure 43

Nuevamente un caso complejo, debido a la línea amarilla que, sumada a la desigual iluminación del asfalto, podría dar lugar a errores. Sin embargo, podemos ver como el programa acierta. Aunque quizás el primer tramo discontinuo que aparece en la imagen era lo suficientemente largo para que hubiera sido pintado.

Un caso de error claro. El programa falla a la hora de detectar el carril derecho, resultando así imposible hacer el cambio de perspectiva con el que detectar cada tipo de línea. En este caso, podemos deducir que la baja calidad de la imagen, unida a su perspectiva, que se come parte del tramo derecho, ha debido influir a la hora de provocar el error.

Nuevamente estamos ante un caso de línea discontinua no detectada. En esta ocasión, podemos deducir que la amplia separación entre tramos ha hecho que quedasen todos demasiado alejados. Lo cual unido a su color amarillo, ha debido dificultar su detección.

Error total del programa, que ha sido incapaz de detectar ninguna línea. Es posible que el cambio de tonalidad en el asfalto, más claro a los lados que por dentro, haya provocado errores a la hora de detectar las líneas, por haber demasiado píxeles blancos y no solo una línea.

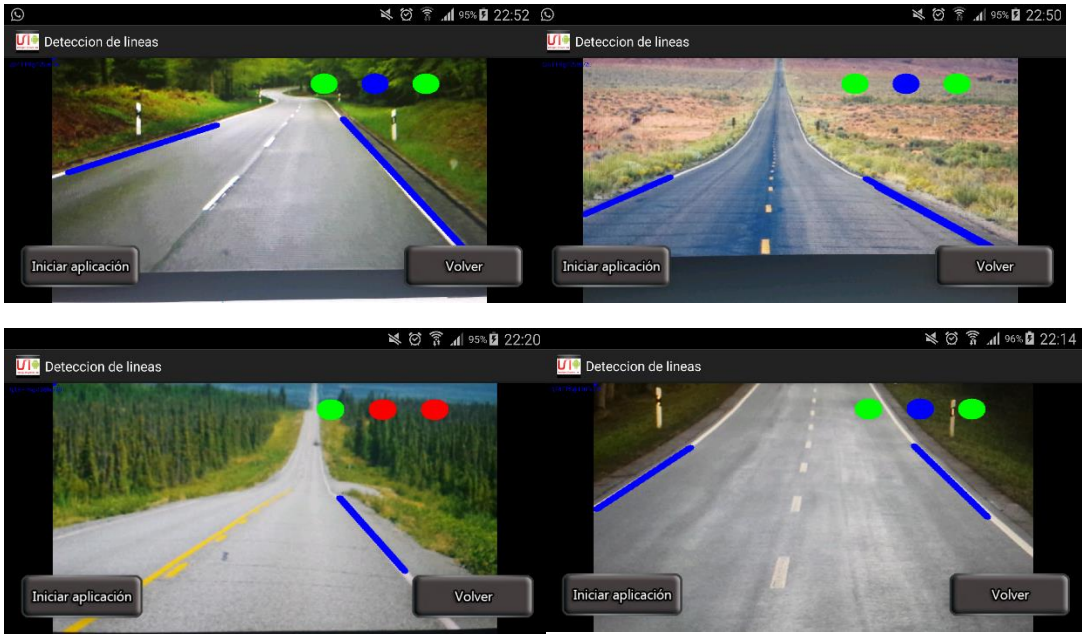


Figure 44

Un nuevo caso de acierto, este además en circunstancias bastante adversas. Como se puede comprobar, la línea de en medio resulta muy difícil de diferenciar. Pero seguramente, el primer tramo de la imagen (el de la parte inferior), sea lo suficientemente claro para hacer una clasificación correcta.

Otro acierto complicado. Al ser la línea de en medio amarilla, y con tramos tan cortos, podría haber sido de esperar algún tipo de error. Además, el color del asfalto también resultaba bastante irregular. Sin embargo, el programa ha podido solventar las dificultades con acierto.

Nuevamente un error debido a la no detección de una de las líneas laterales. En este caso, la perspectiva y la baja calidad hacen imposible la detección de la línea de la izquierda. Que resulta casi imposible de ver incluso para un humano.

Sin embargo, cabría esperar que el programa hubiese identificado la línea central como una lateral, en ausencia de la propiamente dicha. Su no detección como tal, hace pensar que tal vez ese tipo de líneas, que juntan una continua con una discontinua, presenten problemas a la hora de ser identificadas como una línea lateral principal.

Vamos con otro caso de acierto. En este caso, de una cierta complejidad debido a la desigual tonalidad del asfalto, y al desgaste de las líneas, que las vuelve grisáceas.

Figure 45

Otro caso en el que el programa no detecta una línea lateral. En este caso, la línea central si ha sido correctamente detectada, y por tanto ha identificado el tramo como un carril de único sentido.

Terminamos con otro acierto. En este caso, con colores claros tanto en el asfalto como en el arcén. Esta situación ya la vimos provocar errores en un caso previo. Sin embargo, en este caso el color es lo suficientemente distinto como para que la detección sea correcta.

5.1.2 Conclusiones

Como hemos podido ver, el programa presenta una tasa de éxito bastante aceptable. Además, los errores que presenta suelen seguir patrones comunes, por lo que sería relativamente sencillo resolverlos en una hipotética versión final.

En primer lugar, las condiciones lumínicas son un factor bastante importante. Vemos que cuando la diferencia de iluminación entre unas zonas y otras es bastante amplia, se puede producir ciertas zonas que se confundan con líneas. Esto sería subsanable si, al estar integrado en el coche, tuviéramos más información sobre el estado de funcionamiento de los faros, e hiciésemos que el programa actuase en consecuencia.

El estado del asfalto también resulta importante, y puede hacer que alguna zona de distinto color influya en los resultados. Esto podría solucionarse en gran medida con una calibración más exacta. Al estar la cámara fijada al coche y siempre en la misma posición, podríamos determinar con mayor facilidad cual es la región de interés de la imagen, y limitarlo solo a dos zonas laterales y una central, en la que podrían aparecer las líneas. De esta manera nos evitaríamos bastantes errores.

Otra opción que tendríamos en la versión final, sería aumentar el número de cálculos que se harían, al disponer de una mayor capacidad que en un simple móvil. De esta manera, sería relativamente sencillo incluir pequeños apartados de código destinados a chequear los posibles errores más comunes.

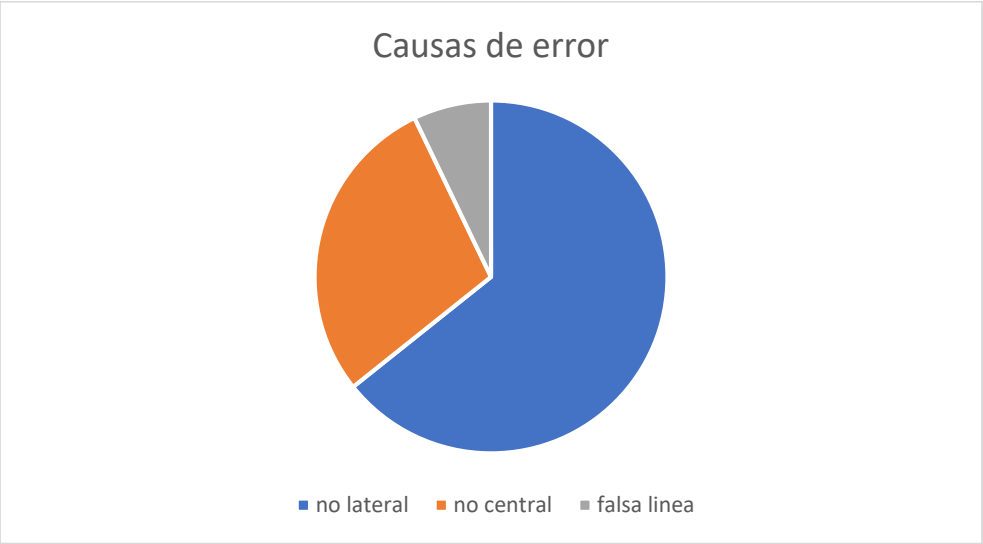
Por último, existe un error ante el que poco podemos hacer. Se trata del error debido al posible mal estado de la pintura de las líneas. En el caso de que la pintura este demasiado desgastada, podría confundirse un tramo desgastado en una línea continua con una línea discontinua. O incluso directamente no llegar a detectar ninguna línea. En la mayoría de casos, un desgaste normal no supondrá un problema. Pero en casos extremos, el programa fallaría sin que nosotros pudiéramos hacer nada por evitarlo.

Número	Resultado	Tipo de error	Causa
4	Error	único carril	no detección central
5	Error	No carril	no detección lateral
10	Error	No carril	no detección lateral
14	Error	único carril	no detección lateral
18	Error	fallo completo	no detección laterales
19	Error	fallo completo	falsa línea
23	Error	no carril	no detección lateral
27	Error	no carril	no detección lateral
29	Error	único carril	no detección central
34	Error	no carril	no detección lateral
40	Error	único carril	no detección central
42	Error	fallo completo	ninguna detección
45	Error	único carril	no detección lateral
47	Error	no carril	no detección lateral

Table 1

Graph 1

- Único carril: la no detección de una línea provoca que se detecte un carril doble como uno solo
- No carril: se detecta una sola línea y por tanto no puede detectarse el carril
- Fallo completo: no se detecta ninguna línea en la imagen



Graph 2

- No lateral: no se detecta una de las líneas laterales
- No central: no se detecta la línea central
- Falsa línea: se identifica como línea algo que realmente no lo es

En estas tablas y gráficas se muestra las características exactas de los fallos que tuvimos en nuestro análisis. Como se puede comprobar, los tipos de error están bastante repartidos entre los tres tipos, aunque cabe añadir que el error de “Carril único” es de mucha menor importancia que los otros. Pues identifica el carril actual del vehículo que, al fin y al cabo, es lo más importante.

En las causas de error, en cambio, sí que vemos una gran disparidad. La deducción es obvia, la causa de error más común es el fallo al detectar una de las líneas laterales. Lo cual resulta bastante lógico, pues son las líneas más alejadas de la cámara y, por tanto, las más propensas a tener poca representación en la imagen o a ser bloqueadas por algo.

5.2 Integración Temporal

Tras el análisis previo, decidimos realizar una segunda prueba. Esta vez añadiendo una función que tuviese en cuenta el estado previo de las líneas. Puesto que, si en el fotograma previo una línea era discontinua, resulta más probable que siga siéndolo en este. Aquí os mostramos algunos de los resultados.

5.2.1 Casos

Figure 46

Aquí tenemos un ejemplo típico de algo que ya veíamos en el análisis previo. Debido a la perspectiva de la cámara, no llega a captar las 3 líneas. Por lo que el programa solo llega a detectar un carril. Sin embargo, podemos comprobar que pese a ser discontinuo el carril izquierdo, ha podido realizar bien la detección.

En este caso la integración temporal no tiene ningún efecto. El mismo error se produce a lo largo del tiempo, pues en ningún momento el programa es capaz de detectar una línea lateral para detectar el segundo carril.

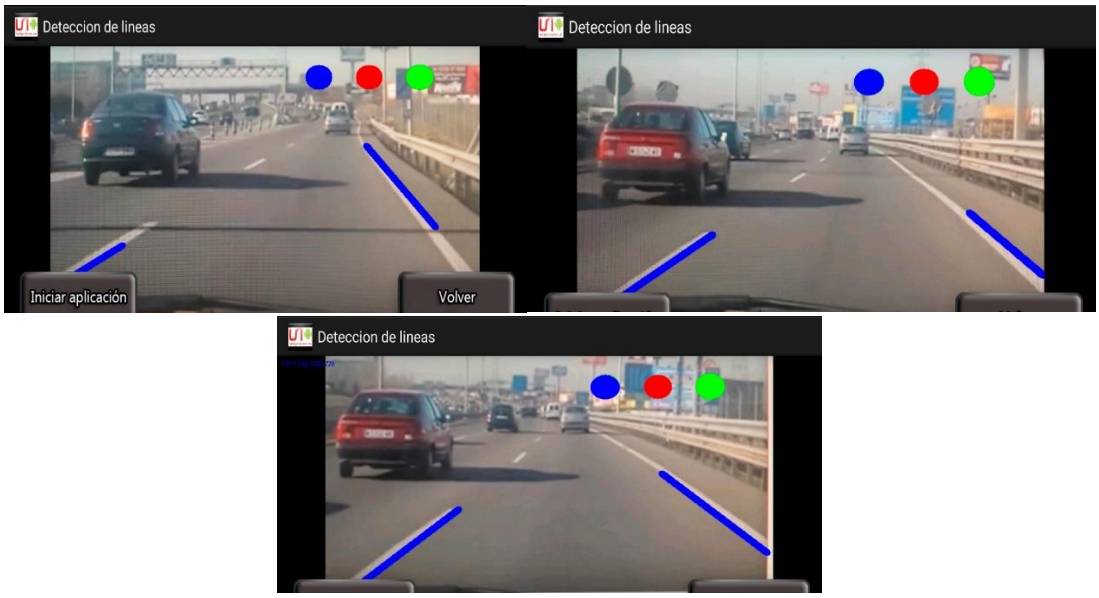


Figure 47

Mismo caso que en el ejemplo anterior. La mala perspectiva provoca que solo se identifique un carril. Pero nuevamente comprobamos que el carril discontinuo no impide su buena detección.

De igual manera, la integración temporal no logra arreglar el error, al no detectarse en ningún momento la línea lateral que sería necesaria para el acierto.

Figure 48

Aquí tenemos un caso en el que el análisis temporal ha variado el resultado. En un primer momento el programa podía fallar en determinados instantes, y no detectar la línea de en medio. Sin embargo, al incluir la versión temporal, esos fallos puntuales se ven corregidos, dando un resultado correcto.

Figure 49

En este caso la detección de las líneas laterales es correcta, pero no la de la línea central. Ni en un primer momento, ni con el paso del tiempo. Podemos concluir que, al contrario que en el caso anterior, aquí la imagen tiene muy baja calidad. Por lo que la línea central no se llega a ver con la suficiente claridad para ser detectada.

La integración temporal no logra solucionarlo. Pues, aunque la línea central se podría detectar de forma intermitente, nunca llegaba a detectarse de forma mayoritaria. Por lo que al final el programa se estabilizaba en el resultado de línea central no detectada, y por tanto de carril único.

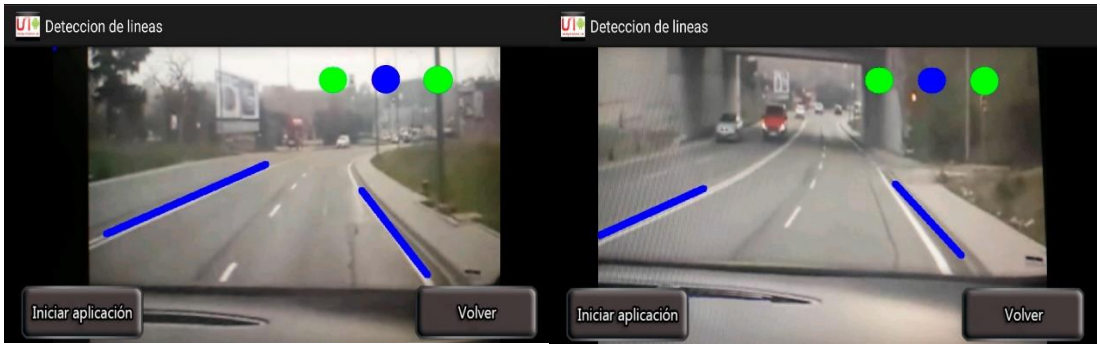


Figure 50

En este caso vemos que el resultado es correcto. Con la pequeña salvedad de que, en la segunda imagen, ha confundido una línea del bordillo con la del carril. Debido a la naturaleza de dicha línea, se trata de un error bastante difícil de evitar. No obstante, no sería un error demasiado grave, puesto que la posición de ambas líneas es casi idéntica.

La integración temporal no cambia el resultado. Aunque es cierto que el bordillo no aparecía demasiado tiempo. Igual en un mayor intervalo temporal se habría podido corregir esa falsa detección.

5.2.2 Conclusiones

La inclusión de información previa para el análisis ha ofrecido unos resultados divididos, que nos hacen dudar sobre si su inclusión sería idónea o no en el caso de nuestra aplicación.

Por una parte, vemos que en algunos casos los resultados mejoran con la inclusión de esta parte. Recibimos unos resultados mucho más estables, pues varios errores puntuales que podían darse en la otra versión, se ven corregidos en esta.

Así, si en un momento dado, por un cambio de luz, o un error en el fotograma, se podía provocar un error en un instante. En este caso resulta mucho menos probable que eso ocurra.

Sin embargo, no evitamos al 100% que esto siga sucediendo. Además, tampoco ayuda a subsanar algunos de los errores más comunes que teníamos en la versión normal, como las diferencias de iluminación o el desgaste del asfalto. Por eso, la mejoría, aunque existente, tampoco es demasiado importante.

Esto nos lleva a plantearnos si merece la pena su inclusión. Al fin y al cabo, nuestra capacidad operacional es limitada, por tratarse de un dispositivo móvil. Y hemos podido comprobar que la inclusión de estos apartados empeoraba ligeramente la fluidez del programa. Sigue siendo funcional, pero puede quedarse colgado con más frecuencia en dispositivos de menor potencia.

Seguramente, en una versión posterior en la que contásemos con un móvil más avanzado, el procesador que tuviéramos manejando las operaciones sería suficientemente potente para poder ejecutar el programa sin problema, incluso con su añadido. Por lo que la duda, en principio, no tiene tanto sentido de cara a una versión final.

Sin embargo, de cara a dicha versión, son muchas otras las funciones que se podrían incluir. Haciendo que la inclusión de esta mejora en concreto no resulte completamente óptima.

Como conclusión final, diremos que una versión del programa que tenga cuenta el estado previo de la carretera a la hora de hacer el análisis actual, será, casi sin duda, superior a una versión que no lo haga. Pero dicho efecto sería más útil a la hora de posicionar las líneas de los carriles, que a la hora de detectar si son continuas o no.

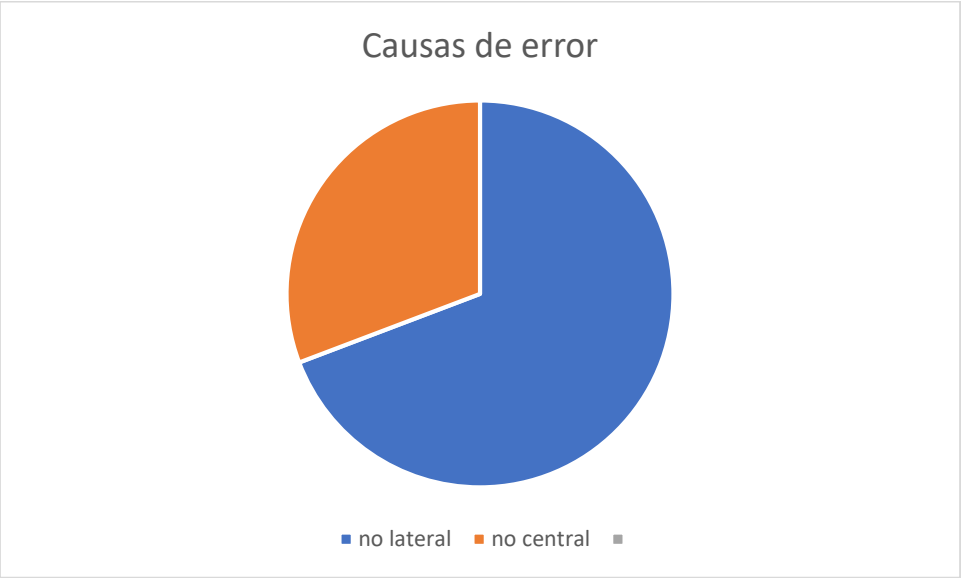
Por tanto, de cara a una versión final, creemos que lo más importante sería conseguir utilizar la información de análisis previos a la hora de posicionar correctamente el carril. Siendo de menor importancia, aunque igualmente viable y útil, el definir si son continuas o discontinuas en base a análisis previos.

Número	Resultado	Tipo de error	Causa
2	Error	único carril	no detección central
3	Error	fallo completo	no detección latera
8	Error	único carril	no detección central
12	Error	No carril	no detección lateral
18	Error	único carril	no detección central
20	Error	no carril	no detección lateral
26	Error	único carril	no detección lateral
28	Error	fallo completo	no detección laterales
33	Error	no carril	no detección lateral
37	Error	único carril	no detección central
39	Error	no carril	no detección lateral
42	Error	no carril	no detección lateral
49	Error	único carril	no detección lateral

Table 2

Graph 3

- Único carril: la no detección de una línea provoca que se detecte un carril doble como uno solo
- No carril: se detecta una sola línea y por tanto no puede detectarse el carril
- Fallo completo: no se detecta ninguna línea en la imagen



Graph 4

- No lateral: no se detecta una de las líneas laterales
- No central: no se detecta la línea central
- Falsa línea: se identifica como línea algo que realmente no lo es (en este caso no se ha producido nunca este error)

Como podemos ver, en el caso de la integración temporal se repite la dinámica de los anteriores casos. El error en las líneas laterales sigue siendo, lógicamente, el más común de todos ellos. Siendo en este caso aún más común.

Esto, unido al hecho de que el error de falsa línea (el menos común en el caso anterior) también ha desaparecido, nos hace pensar que la integración temporal ayuda a subsanar, sobre todo, errores de naturaleza puntual. Mientras que aquellos errores debidos al estado de la imagen o del entorno siguen causando problemas. Lo cual tiene sentido sin pensamos en la forma en la que funciona el programa.

6. Presupuesto

Vamos a proceder a realizar los cálculos para el coste total que supondría la realización de este proyecto. Para ello calcularemos por separado los gastos materiales y los de personal.

Materiales

La lista de materiales necesarios para el proyecto se reduce, simplemente, a dos dispositivos. Por un lado, el móvil en el que realizar la aplicación, y por el otro, el ordenador en el que programar con android studio.

Para el móvil utilizamos un Samsung Galaxy S4, cuyo precio de adquisición fue de 250€. En cuanto al ordenador utilizamos una torre, comprada por piezas y cuyo montaje realizamos nosotros mismos. Para ahorrar lo máximo posible en costes. El precio total de la misma fue de 850€

Sin embargo, no debemos considerar el precio total de los componentes. Sino que debemos calcular cual ha sido su amortización. Según la última ley de impuesto sobre sociedades (20), el coeficiente de amortización en el caso de aparatos electrónicos es de un 25% por año. Teniendo en cuenta que el tiempo de utilización de los mismos fue de 9 meses, el coste total será el calculado será el calculado en la siguiente tabla. Obtenido de hallar el 25% del coste total, dividirlo entre 12 meses y multiplicarlo por 9.

Equipo	Coste total	Amortización
Ordenador	850 €	159 €
Móvil	250 €	47 €
	Total	206 €

Table 3

Personal

Para calcular el sueldo nos basaremos en lo que cobra un ingeniero industrial de media en España, sin experiencia profesional previa. Lo cual, según las estadísticas, asciende a 2174€ brutos al mes. (21)

Sin embargo, vamos a suponer que para la realización del proyecto se ha cogido a un estudiante a través de un contrato de prácticas, por el que se le pagará un 60% de lo previamente dicho.

Sueldo Medio	Meses	Porcentaje	Total
2.174 €	9	60%	11.740 €

Table 4

Total

Sumando lo obtenido en los apartados previos, nos queda que el presupuesto total del proyecto asciende a 11.946 €

	Cantidad
Materiales	206 €
Personal	11.740 €
Total	11.946 €

Table 5

7. Conclusiones y trabajos futuros

7.1 Conclusiones

Tras haber trabajado en profundidad en nuestro proyecto, hay varias conclusiones que consideramos de especial relevancia en el desarrollo de esta aplicación:

Gran variación de la dificultad

La enorme cantidad de posibles errores, elementos a tener en cuenta o funciones implementables, hace que la dificultad de una aplicación de este tipo pueda aumentar de forma exponencial hasta alcanzar niveles altísimos.

Realizar una aplicación básica para detectar líneas en circunstancias normales es una labor sencilla. Un algoritmo no demasiado complejo podría realizarlo, y cualquier smartphone de gama media o alta podría ejecutarlo sin ningún tipo de complicación. Convirtiendo este tipo de programas en algo extremadamente sencillo de implementar.

Sin embargo, conforme vamos añadiendo casos y tenemos en cuenta más posibilidades distintas, aumenta enormemente la complejidad del sistema. Al considerar distintos números de carriles, objetos que puedan estar bloqueando la carretera, o la enorme lista de posibles fuentes de error, el algoritmo necesario aumenta en complejidad.

De ahí que hablemos de una complejidad tan elevada. Un sistema muy sencillo requerirá pocos pasos, pero para tener en cuenta todos los factores harán falta operaciones cada vez más y más complejas. Es por eso que, según avanzamos en nuestro programa, cada vez necesitaremos un mayor trabajo y una mayor capacidad de procesamiento, para producir el más mínimo aumento en el porcentaje de acierto de nuestra aplicación.

Múltiples fuentes de error

Las posibles fuentes de error a la hora de detectar líneas en una carretera son tremendamente elevadas. Lo cual es, en gran medida, responsable de lo que comentábamos en el apartado previo de nuestras conclusiones.

Las diferentes condiciones lumínicas son un factor importante. El estado de la carretera lo es más aún si cabe. Tanto por sus condiciones, como color de asfalto o de las líneas; como por su estado, pues el desgaste puede inducir a multitud de errores.

Las situaciones de tráfico elevado también pueden complicar mucho la detección de líneas. Aunque, en este caso, seguramente no influya tanto. Puesto que en este tipo de situaciones de todas formas no tendría mucha utilidad la aplicación.

Condiciones meteorológicas, obstáculos en la vía, reflejos, etc. La lista se extiende enormemente, y nos deja una cosa clara: una implementación final completamente a prueba de fallos de este tipo de algoritmos no es, ni mucho menos, sencilla.

Aplicaciones viables en la conducción automatizada

Los dos hechos que acabamos de señalar nos llevan a una conclusión bastante sencilla. Un automóvil completamente automatizado y que pueda moverse libremente en cualquier circunstancia resulta extremadamente difícil de realizar. Sin embargo, las aplicaciones de asistencia a la conducción, al contar con la supervisión de un ser humano al volante, resultan mucho más fáciles de llevar a cabo.

Como hemos dicho, el problema es la enorme posibilidad de errores, que aumenta más aún conforme más desconozcamos el entorno. Una complicación difícil de vitar, pues resulta extremadamente difícil preparar de antemano a un algoritmo para posibles errores que no sabemos ni que pueden existir. De ahí que la opción más sensata sea buscar la forma de reducir a 0 la posibilidad de que surjan esas posibilidades extremas.

Una posibilidad, compleja pero completamente automática, sería la utilización de vehículos automatizados en entornos concretos. Autobuses que operen realizando un mismo recorrido continuamente, o taxis adaptados específicamente para la ciudad en la que se encuentran serían las utilidades más obvias.

De esta forma, al limitar el campo de actuación a un espacio geográfico concreto, resultaría mucho más sencillo para el programador identificar todas las posibles fuentes de error que se podrían dar. Y optimizar el programa para funcionar en dicho espacio.

Esta aplicación parece una de las más viables en el corto plazo. Y, de hecho, empresas como Uber han producido ya coches completamente automatizados en ciudades como San Francisco. Su viabilidad depende, como es obvio, de las condiciones de cada ciudad. Pero todo hace indicar que, de aquí a no demasiado tiempo, los vehículos automatizados serán bastante comunes en las grandes ciudades.

Sin embargo, fuera de este tipo de entornos la aplicación es más compleja, de ahí que contar con un ser humano al mando siga siendo la opción más viable. Todo esto nos conduce, como dijimos en el primer párrafo, a la confirmación de que la aplicación que hemos desarrollado, y en general las aplicaciones de asistencia a la conducción, son las opciones más viables para sacar al mercado en el corto plazo.

Aplicaciones en otros ámbitos

Como hemos comentado previamente, la aplicación en la conducción resulta una de las más complejas de realizar. Al tratarse de un ámbito en el que cualquier mínimo error puede traducirse en la perdida de vida. Lo cual unido a la gran dificultad que hemos señalado para conseguir un programa cien por cien a prueba de errores, hace que sea uno de los campos donde más difícil sea su implementación.

Sin embargo, si nos vamos a otros campos en los que un error sea mucho más asumible, sería tremendamente sencillo realizar tareas automatizadas, con un coste de aplicación bastante baje y un índice de éxito suficiente alto para resultar viable.

La utilización de autómatas que puedan desplazarse por sí mismo en entornos controlados, como almacenes, hospitales, centros residenciales o talleres, resulta un opción útil y no demasiado compleja de implementar.

Igualmente, la implementación de pequeños o medianos vehículos de transporte para ayudar a trabajadores o clientes a moverse a través de este tipo de instalaciones también sería muy útil.

Todo este tipo de usos serían tan sencillos de implementar, como incluir algún tipo de línea en el suelo que guie a los vehículos por sus distintas trayectorias. Algo tan sencillo como líneas de distintos colores para ir a distintas localizaciones, permitiría una implementación tremendamente sencilla.

Muchos de estos usos ya han sido llevados a cabo. Otros, es posible que lo hagan pronto, o quizás nunca se lleven a cabo por no merecer la pena. Pero, lo que está claro, es que este tipo de tecnología está, cada vez más, al alcance de todos. Por lo que no cabe duda de que iremos viendo cada vez más vehículos y objetos que hagan uso de ella.

7.2 Trabajos futuros

El proceso de desarrollo de nuestra aplicación nos ha servido, sobre todo, para entender que aspectos resultan más importantes para el correcto funcionamiento del programa, cuales son los puntos que dan más fallos, y que funciones merecería la pena implementar de cara al futuro.

Número de carriles

En primer lugar, y la más obvia de todas, sería aumentar el número de carriles detectables. No limitándolo solo a dos, como es el caso de nuestro programa, sino hacer uno que, en función de la cantidad de líneas detectadas, de sus características, y de su ángulo, pueda determinar cuántos carriles y de qué tipo tiene la vía que estamos transitando.

Sin embargo, ya hemos comprobado lo difícil que puede ser detectar las líneas. Sobre todo, si las condiciones no son óptimas, o si el tráfico es demasiado elevado. Es por eso que esta función no la consideramos prioritaria. Y creemos que lo ideal sería centrarnos en la correcta detección de nuestro carril actual y, como mucho, de los adyacentes.

Calibración

Otra función muy interesante, y que ayudaría a subsanar gran parte de los errores que el programa comete, sería incluir un apartado de calibración. Existen dos apartados principales que vamos a querer que sean calibrados en nuestro programa. Uno es el ángulo de la cámara, y el otro el umbral que tomaremos a la hora de binarizar nuestra imagen.

El primer apartado es bastante sencillo, pero mejorará enormemente el funcionamiento de nuestro programa. Tanto en porcentaje de éxitos, como en coste operacional. Y es que, una vez tengamos certeza total de la posición de la cámara y su orientación, podremos saber con total exactitud en que partes de la imagen estará la información deseada.

Esto no solo nos evitara muchos errores, sino que lo hará, además, ahorrándonos cálculos, al permitirnos elegir una ROI que se limite a solo unas pocas partes relevantes del fotograma.

El segundo es algo más difícil de realizar, pero que también ayudará enormemente en los resultados. En este caso, al contrario que en el de la cámara, no se trata de algo que podamos calibrar una vez instalado y ya está. Sino que la posición óptima del umbral dependerá, en cada momento, de las condiciones de nuestro entorno.

Lo más sencillo seria incluir una pantalla en la que el usuario pudiera, manualmente, calibrar el umbral hasta que las líneas de la carretera apareciesen perfectamente definidas en la versión binarizada de la imagen, con la mínima cantidad de ruido.

Sin embargo, si nuestro objetivo es realizar un programa lo más automatizado posible, la opción que pensamos para la calibración consiste en un análisis inverso de la imagen previo al arranque del automóvil.

La idea es que, antes de comenzar a funcionar, el programa realizase una calibración automática. Sabiendo en qué posición lo tenemos aparcado (podemos guardar esa información cada vez que el coche se aparque), podemos saber qué tipo de lectura tenemos que obtener.

Así pues, si, por ejemplo, sabemos que nuestro coche al momento de aparcar, podía detectar dos líneas discontinuas. Una vez arranquemos, podemos ejecutar rápidamente nuestro programa en todo el espectro de umbrales posibles (de 0 a 255), y quedarnos con aquellos valores en los que el programa logre detectar las mismas líneas y del mismo tipo que debería (en nuestro ejemplo, dos líneas discontinuas). Así será fácil utilizar un umbral adecuado para las condiciones de nuestro entorno.

Esto resultará especialmente útil para solucionar los problemas que puedan provocar las distintas iluminaciones. Pues según conduzcamos de día o de noche, harán falta distintos valores para hacer una correcta detección. Con este mecanismo, se podría realizar esta labor de forma completamente automática. E ir ajustándose en tiempo real para adaptarse a los cambios en el entorno.

Pareja de cámaras

Otra opción que ayudaría, sería la utilización de 2 móviles en nuestro vehículo. De esta manera, en lugar de tener que realizar cambios de perspectivas para la detección de las líneas, podríamos hacer hacerlo simplemente basándonos en la distancia entre ambas cámaras, y la posición de una determinada línea en cada una de ellas.

Esta técnica, además, nos permite implementar muchas otras funciones, que harán el funcionamiento de nuestra aplicación mucho más efectivo y fiable que si lo hacemos simplemente con la cámara de un móvil.

Integración completa

El último trabajo sería el más evidente de todos. Una vez tuviésemos la aplicación completa, la idea sería unir todas las distintas funciones desarrolladas por el laboratorio en una única aplicación.

Así podríamos tener un programa que detectase a la vez las señales de tráfico, las líneas de la carretera y los obstáculos. Información suficiente para desarrollar una aplicación que mostrase, simultáneamente, una enorme cantidad de información de gran utilidad para el usuario.

Esto no sería una labor sencilla. En primer lugar, la capacidad requerida para unificar estos procesos seria extremadamente alta. Ya hemos visto que un móvil tiene problemas a la hora de ejecutar una sola de ellas. Para ejecutarlas todas simultáneamente haría falta una Smartphone con bastante capacidad de procesamiento.

Para ello sería de gran utilidad intentar optimizar las funciones a la hora de unificarlas. Viendo que procesamiento de la imagen tiene cada una, para ver si sería posible extraer toda esta información de la misma imagen procesada, en lugar de tener que hacer 3 procesados distintos para cada función.

Por último, habría que encontrar la forma de mostrar toda esta información de la manera que resultase más útil para él usuario. En lugar de como hasta ahora, que al tratarse solo de unas pocas líneas no resultaba demasiado complicado mostrarlo todo de forma clara.

Si seguimos más allá, con aún más capacidad de procesamiento, se podría utilizar toda esta información para llevar a cabo maniobras concretas. Búsqueda de aparcamiento, ayuda en maniobras de adelantamiento, guiado más claro con el GPS, ayuda en intersecciones o incorporaciones, etc.

8. Bibliografía

Bibliografía

(1). Android: el SO más usado en dispositivos móviles

http://www.expansion.com/economia-digital/companias/2015/12/09/56684be1ca474151018b4590.html

(2). Funcionamiento de Android

https://developer.android.com/guide/platform/index.html

(3). Historia de OpenCV

http://opencv.org/

(4). Origenes de la visión por computador

https://dspace.mit.edu/handle/1721.1/6125

(5). Origenes de la visión por computador

https://books.google.es/books?id=bXzAlkODwa8C&printsec=frontcover&hl=es&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=false

(6). Últimos avances en la visión por computador

https://books.google.es/books?id=t5FygksotoQC&printsec=frontcover&hl=es#v=onepage&q&f=false

(7). niveles de conducción automatizada según SAE internacional

https://www.sae.org/misc/pdfs/automated_driving.pdf

(8). Vehículo automatizado Uber

https://elpais.com/tecnologia/2016/12/14/actualidad/1481680772_942809.html

(9). Vehículo automatizado Tesla

https://www.xataka.com/vehiculos/los-tesla-ya-disponen-de-piloto-automatico-y-los-primeros-videos-muestran-que-no-estamos-preparados

(10). Vehículo automatizado Google

http://www.elmundo.es/tecnologia/2014/05/28/538565f0268e3e58098b456c.html

(11). Sistemas de asistencia avanzada a la conducción

http://www.autoconnectedcar.com/adas-advanced-driver-assistance-sytems-definition-auto-connected-car/

(12). Ejemplos de sistemas de asistencia avanzada a la conducción

https://ec.europa.eu/transport/road_safety/specialist/knowledge/old/what_can_be_done_about_it/adas_en

(13). Apps disponibles en la play store

https://play.google.com/

(14). Presentación LSI

http://portal.uc3m.es/portal/page/portal/dpto_ing_sistemas_automatica/investigacion/IntelligentSystemsLab/about

(15). LSI: IVVI 1.0 e IVVI 2.0

http://portal.uc3m.es/portal/page/portal/dpto_ing_sistemas_automatica/investigacion/IntelligentSystemsLab/research/adas

(16). LSI: iCab

http://portal.uc3m.es/portal/page/portal/dpto_ing_sistemas_automatica/investigacion/IntelligentSystemsLab/research/augv

(17). LSI: SkyOnyx

http://portal.uc3m.es/portal/page/portal/dpto_ing_sistemas_automatica/investigacion/IntelligentSystemsLab/research/auav

(18). Eclipse

http://www.eclipse.org/org/

(19). Origen Android Studio

https://android-developers.googleblog.com/2013/05/android-studio-ide-built-for-android.html

(20). Ley de impuesto sobre las sociedades

https://www.boe.es/buscar/act.php?id=BOE-A-2014-12328

(21). Salario medio de un ingeniero en España

http://www.tusalario.es/main/salario/comparatusalario?job-id=2141010000000#/