Universidad Carlos iii de Madrid

Escuela Politécnica Superior
Departamento de Ingeniería de Sistemas y Automática

Ingeniería Industrial

PROYECTO FIN DE CARRERA

Algoritmo de Odometría Visual Estéreo para Sistemas de Ayuda a la Conducción. Implementación en GPU mediante CUDA.

Autor: Carlos Guindel Gómez
Tutor: Basam Musleh Lancis

Octubre de 2012

Agradecimientos

En este texto, que es símbolo de la consecución de uno de los objetivos más importantes de mi vida, no puedo dejar de expresar unas palabras de agradecimiento hacia las personas que lo han hecho posible.

En primer lugar, a mis padres y mi hermana, que sé que me admiran mucho por ser capaz de hacer cosas como este trabajo.

También a mis amigos de siempre, a los que me voy a referir con sus nombres menos comunes: Daniel y Alberto, por estar ahí siempre, haciéndome sentir que puedo contar con ellos cuando lo necesite y siendo una parte imprescindible de mi vida.

Del mismo modo, merecen mención las personas que han aparecido en mi vida durante mi etapa universitaria, entre los cuales tengo el orgullo de poder incluir a Eric, y a otros como Alejandro, Santiago, Adrián, Luis y muchos más que han contribuido a que haya podido llegar hasta aquí.

No me quiero olvidar del nombre que aparece bajo el mío en la portada: mi tutor Basam, que, pese a las adversidades, siempre ha intentado facilitar, en la medida de lo posible, la realización de este trabajo.

No obstante, hay una persona que merece una mención especial por muchas razones: porque ha sido una gran compañera de prácticas, porque le ha tocado soportarme todo este año, y porque tengo claro que, sin ella, este proyecto no habría sido posible tal y como lo es hoy. Gracias por todo, Irene.

Resumen

El objetivo del presente proyecto es el desarrollo e implementación de un sistema de localización para automóviles basado en odometría visual, consistente en el seguimiento de puntos característicos de la calzada detectados a través de un sistema de visión estéreo.

Para cumplir con las especificaciones de tiempo de cómputo requeridas, se utilizarán técnicas de computación paralela, como la arquitectura CUDA (*Compute Unified Device Architecture*, arquitectura unificada de dispositivos de cómputo), que permite ejecutar procesos en paralelo en unidades GPU (*Graphics Processing Unit*, unidad de procesamiento gráfico) de la marca NVIDIA.

El sistema está diseñado para formar parte de los sistemas de ayuda a la conducción (ADAS) equipados en el vehículo inteligente IVVI 2.0 de la Universidad Carlos III de Madrid.

Abstract

The objective of this project is the development and implementation of a locating system for cars that is based on visual odometry, consisting in tracking feature points belonging to the road, which are detected by a stereo vision system.

To accomplish the computation-time requirements, parallel computing techniques will be used, such as CUDA (Compute Unified Device Architecture), which allows to running several processes in parallel on NVIDIA GPUs (Graphics Processing Units).

The system is designed to be part of Driver Assistance Systems mounted on the Universidad Carlos III's IVVI 2.0 (Intelligent Vehicle based on Visual Information)

Índice general

Lista de Figuras

Lista de Tablas

Lista de Siglas y Acrónimos

ABS	*Anti-lock Braking System*, sistema anti-bloqueo de frenos.
ACC	*Adaptive Cruise Control*, control de crucero adaptativo.
AD	*Absolute intensity Differences*, diferencias absolutas de intensidad.
ADAS	*Advanced Driver Assistance Systems*, sistemas avanzados de asistencia al conductor.
BSD	*Berkeley Software Distribution*, distribución de *software* de Berkeley.
CCD	*Charge-Coupled Device*, dispositivo de carga acoplada.
CMOS	*Complementary Metal-Oxide-Semiconductor*, semiconductor metal-óxido complementario.
CPU	*Central Processing Unit*, unidad central de proceso.
CSIC	Centro Superior de Investigaciones Científicas.
CUDA	*Compute Unified Device Architecture*, arquitectura unificada de dispositivos de cómputo.
DoG	*Difference of Gaussians*, diferencia de gaussianas.

DGT	Dirección General de Tráfico.
ESP	*Elektronisches Stabilitätsprogramm*, programa de estabilización electrónica.
FLANN	*Fast Library for Approximate Nearest Neighbors*, biblioteca rápida para los vecinos aproximados más cercanos.
FPS	Fotogramas Por Segundo.
GPGPU	*General-Purpose computing on Graphics Processing Units*, computación de propósito general en unidades de procesamiento gráfico.
GPS	*Global Positioning System*, sistema de posicionamiento global.
GPU	*Graphics Processing Unit*, unidad de procesamiento gráfico.
HCI	*Human-Computer Interaction*, interacción hombre-máquina.
ISA	*Intelligent Speed Adaptation*, adaptación inteligente de la velocidad.
ITS	*Intelligent Transport Systems*, sistemas de transporte inteligentes.
IV	*Intelligent Vehicles*, vehículos inteligentes.
IVVI	*Intelligent Vehicle based on Visual Information*, vehículo inteligente basado en información visual.
LoG	*Laplacian of Gaussian*, laplaciana de la gaussiana.
LIDAR	*LIght Detection And Ranging*, detección y localización por luz.
LSI	Laboratorio de Sistemas Inteligentes.
MER	*Mars Exploration Rovers*, rovers de exploración de Marte.
MIL	*Matrox Imaging Library*, biblioteca de imagenes Matrox.

MIMD	*Multiple Instruction, Multiple Data*, "múltiples instrucciones, múltiples datos".
MLESAC	*Maximum LikElihood SAmple Consensus*, consenso de muestras por máxima verosimilitud.
MSAC	*M-estimator SAmple Consensus*, consenso de muestras por estimador M.
NASA	*National Aeronautics and Space Administration*, Administración Nacional de Aeronáutica y del Espacio.
NAVCAM	*NAVigation CAMeras*, cámaras de navegación.
OMS	Organización Mundial de la Salud.
PIB	Producto Interior Bruto.
PROSAC	*PROgressive SAmple Consensus*, consenso de muestras progresivo.
R-RANSAC	*Randomized* RANSAC.
RANSAC	*RANdom SAmple Consensus*, consenso de muestras aleatorias.
ROI	*Region Of Interest*, región de interés.
SD	*Squared intensity Differences*, diferencias cuadráticas de intensidad.
SIFT	*Scale-Invariant Feature Transform*, transformación de características invariante a la escala.
SIMD	*Single Instruction, Multiple Data*, "una instrucción, múltiples datos".
SSD	*Sum of Squared Differences*, suma de diferencias cuadráticas.
SURF	*Speeded Up Robust Features*, características robustas aceleradas.
UC3M	Universidad Carlos III de Madrid.

UPM	Universidad Politécnica de Madrid.
WTA	*Winner-Take-All*, "los ganadores se llevan todo".

1 Introducción

Antes de profundizar en el objetivo central de este proyecto, es necesario comprender la motivación que justifica su desarrollo. Para ello, hay que tomar consciencia de la gravedad de los accidentes de tráfico, y conocer los fundamentos en los que se basan los modernos sistemas automáticos que se han ido desarrollando para reducir su impacto, algunos de los cuales requieren información precisa sobre la localización del vehículo.

1.1 Problemática asociada a la conducción

Los accidentes de tráfico han sido una de las mayores preocupaciones de las sociedades modernas, y sus respectivos Gobiernos, desde la popularización de los vehículos a motor y el transporte por carretera a lo largo del siglo XX. Durante décadas, han permanecido como una de las principales causas de defunción en todo el mundo; ya en el siglo XXI, la OMS (Organización Mundial de la Salud) los ubicó en su informe del año 2004 en el puesto 19 de causas por numero de fallecimientos en todo el mundo, siendo responsables de 19,1 muertes por cada 100 000 habitantes anualmente [1].

Además del coste social que suponen estas muertes, los accidentes de tráfico también constituyen un enorme coste económico para los países; por ejemplo, en 1994, los accidentes de tráfico supusieron para Estados Unidos un

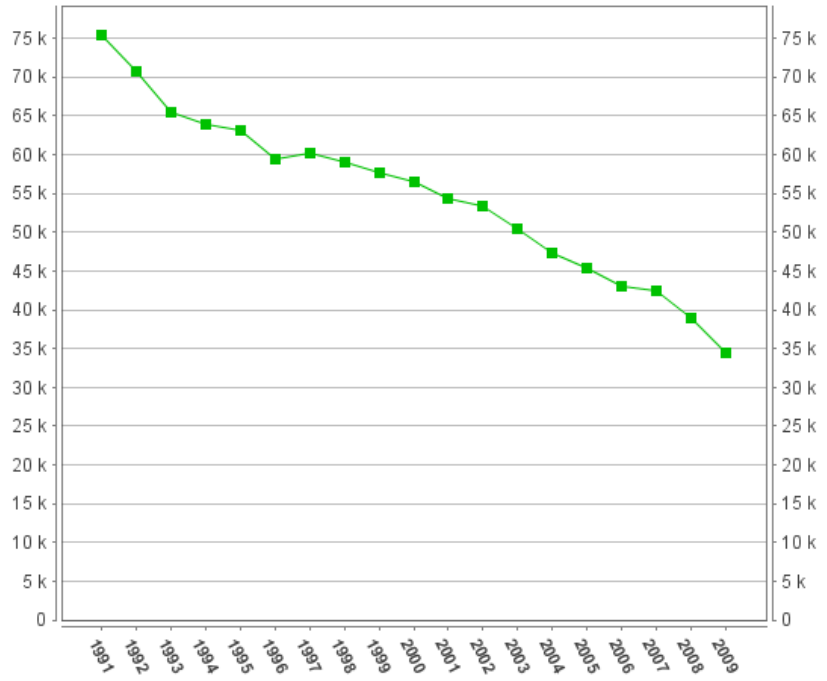


Figura 1.1. Evolución del número total de fallecidos en accidentes de tráfico en el conjunto de los 27 países de la Unión Europea.

coste de 358 022 millones de dólares, un 4,6 % de su PIB (Producto Interior Bruto) [2].

No obstante, las estadísticas recientes muestran una clara tendencia descendiente en el número de víctimas, especialmente en los países más desarrollados. Valgan como ejemplo los datos relativos a la Unión Europea para las dos últimas décadas [3], presentados en la Fig. 1.1. La cifra de defunciones debidas a accidentes de tráfico en los países de la Unión ha caído desde 29,5 hasta 22,6 por cada 100 000 habitantes, entre 2000 y 2009.

En la Fig. 1.2, por su parte, se muestra una descripción de la situación particular de España, obtenida directamente de la DGT (Dirección General de Tráfico) [4]. Los datos registrados comienzan en el año 1980, en el que se elabora el primer Plan Nacional de Seguridad Vial. Pese a la presencia de una etapa de estancamiento entre los años 1995 y 2003 que no está presente en el resto de la Unión Europea, ha tenido lugar una reducción paulatina en el número de muertos desde el máximo alcanzado en 1989 hasta nuestros días.

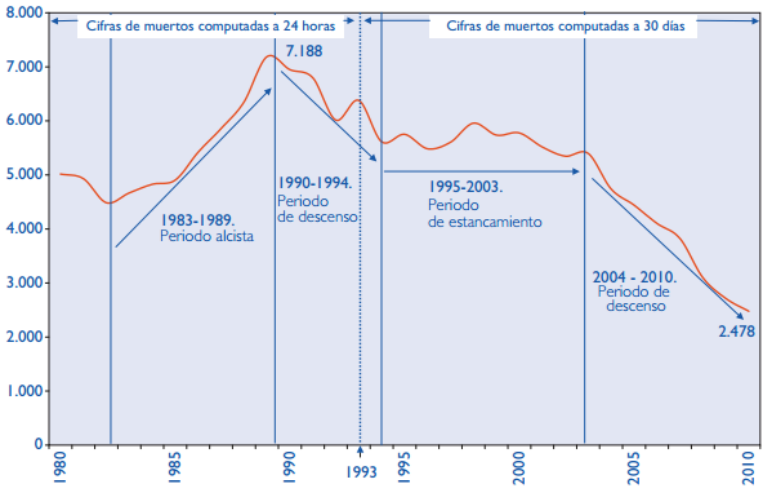


Figura 1.2. Evolución del número de fallecidos en accidentes de tráfico en España.

Hay que tener en cuenta que los accidentes de tráfico son, en su gran mayoría, responsabilidad de los conductores. Tienen su origen en excesos de velocidad, consumo de alcohol, distracciones, infracciones, etc. Esta es la razón de que el colectivo más propenso a sufrir accidentes sea el de los jóvenes, habitualmente menos conscientes de los riesgos a los que se enfrentan al volante.

Sin duda, las campañas de concienciación y el aumento de la educación vial, así como la mejora de las infraestructuras, han tenido influencia en la reducción del número de accidentes ocurridos. Sin embargo, el ser humano es un ser falible, siendo inherente a él incurrir en errores, también durante la conducción.

Por esta razón, hay que recurrir, además, al aumento de los elementos tecnológicos disponibles en los vehículos a motor si se quiere profundizar en esta tendencia. En los últimos tiempos, han proliferado en los vehículos elementos que se apoyan en fundamentos electrónicos, mecánicos, etc. con el fin de proteger a sus ocupantes y a los demás usuarios de las vías públicas (peatones, por ejemplo). Se pueden clasificar, según su filosofía, en elementos de seguridad pasiva y elementos de seguridad activa.

Figura 1.3. Elementos de seguridad pasiva en un vehículo comercial.

- **Seguridad pasiva**. Incluye todos aquellos elementos destinados a minimizar las consecuencias, sobre las personas, de un accidente. Destacan, en este sentido, los avances en la construcción de carrocerías, en busca de combinar habitáculos indeformables con elementos de deformación programada que absorban la energía de forma segura, así como la generalización de los airbag (Fig. 1.3).

- **Seguridad activa**. Se trata de todos aquellos dispositivos cuyo fin es el de evitar que se produzcan los accidentes. Incluye elementos como la suspensión del vehículo, los frenos y, sobre todo en los últimos tiempos, numerosos sistemas electrónicos, como el ABS (*Anti-lock Braking System*, sistema anti-bloqueo de frenos) o el ESP (*Elektronisches Stabilitätsprogramm*, programa de estabilización electrónica) [5].

Estos dispositivos de carácter electrónico constituyen una de las tendencias más activas en la actualidad en materia de seguridad por parte de los fabricantes de automóviles y de las entidades investigadoras. Pretenden suplir las carencias en atención, formación, concentración, etc. del conductor, reduciendo la posibilidad de que su conducta tenga como resultado la generación de un accidente.

Los más avanzados incluyen sofisticados elementos de detección del entorno, agrupándose bajo el nombre de ADAS (*Advanced Driver Assistance Systems*, sistemas avanzados de asistencia al conductor).

1.2 Sistemas de ayuda a la conducción

Los ADAS (*Advanced Driver Assistance Systems*, sistemas avanzados de asistencia al conductor)[6], son sistemas que tienen el propósito de proporcionar ayuda durante la conducción, incrementando la seguridad vial. Entre ellos, se encuentran los siguientes ejemplos:

- Sistemas de navegación embarcados.
- ACC (*Adaptive Cruise Control*, control de crucero adaptativo).
- Sistemas de alerta por abandono de carril.
- Asistencia para el cambio de carril.
- Sistemas de prevención de colisiones.
- ISA (*Intelligent Speed Adaptation*, adaptación inteligente de la velocidad).
- Sistemas de protección para los peatones.
- Reconocimiento de señales de tráfico.

Los IV (*Intelligent Vehicles*, vehículos inteligentes) [7], son vehículos que integran varios ADAS. Se encuadran dentro de los llamados ITS (*Intelligent Transport Systems*, sistemas de transporte inteligentes), nacidos hace 20 años con el aumento de los problemas de movilidad de personas y bienes. Se trata de vehículos que emplean sensores y algoritmos inteligentes para detectar su entorno inmediato, con el fin de poder realizar acciones que sean de ayuda para el conductor o, incluso, controlar totalmente el vehículo. Se pueden clasificar en varios grupos según el grado de automatización alcanzado:

1. Aconsejan o alertan al conductor.

2. Controlan parcialmente el vehículo, ya sea en forma de ayudas permanentes al conductor o interviniendo de emergencia para evitar una colisión.

3. Controlan totalmente el vehículo

Los objetivos que se persiguen con el desarrollo de estos vehículos, así como de los ADAS en general, son:

- Mejorar la seguridad vial de conductores, demás ocupantes de los vehículos y peatones, evitando las principales causas de accidentes de índole humana: imprudencias, fatiga, estados anímicos adversos, distracciones e, incluso, situaciones excepcionales que incapacitan al conductor, tales como ataques de epilepsia.

- Reducir el consumo de combustible y preservar el medio ambiente, llevando a cabo una conducción más eficiente.

- Optimizar la explotación de las redes de transporte.

Los IV requieren contar con sistemas de procesamiento que permitan tratar la información recibida del entorno mediante diversos algoritmos. Gracias a ellos, tienen la capacidad de inferir los estados futuros del entorno y tomar decisiones de forma autónoma sobre las acciones a llevar a cabo, que han de ser lo suficientemente acertadas como para permitir al vehículo circular con normalidad.

Estos sistemas de computación han de ser sistemas en tiempo real, como es habitual en aquellos procesos que requieren interacción con el entorno mediante sensores y actuadores. La razón está en la necesidad de obtener los resultados del procesamiento en un tiempo inferior a un cierto tiempo crítico, que dependerá de la aplicación.

Para conseguir el funcionamiento deseado, se suelen emplear sistemas empotrados; es decir, ordenadores de recursos limitados, específicos de la aplicación, que han de reaccionar a tiempo ante los cambios en el sistema físico. En este tipo de sistemas, el factor prioritario es el cumplimiento de los

tiempos, en detrimento del rendimiento general, que es el más habitual en el caso de los ordenadores personales.

En la actualidad, los IV se enfrentan a varias dificultades [8]:

- **Técnicas**. Los algoritmos deben tener un grado de fiabilidad del 100 %.
- **Económicas**. Su incorporación no debe suponer un gran incremento del coste actual de los vehículos.
- **Psicológicas**. Los ocupantes de los vehículos deben acostumbrarse a no controlar la marcha de éstos y ser conducidos por un ordenador.
- **Legales**. Ante un accidente no estaría claro quién sería el responsable: si el conductor o el fabricante del vehículo.

Pese a todo, resulta innegable la progresiva introducción de diversos sistemas de ayuda a la conducción (ADAS) en los vehículos comerciales disponibles en la actualidad, procedentes de los avances logrados por las marcas comerciales y por diversas entidades investigadoras. A continuación, se procede a presentar una muestra representativa de la situación en este ámbito hoy en día.

1.2.1 Aplicaciones comerciales

La fuerte competitividad existente en la industria del automóvil obliga a las empresas fabricantes a buscar la innovación en el ámbito de los ADAS, para poder utilizar estas tecnologías como elemento diferenciador frente a las demás opciones de compra. A continuación, se muestran algunos soluciones en materia de seguridad que están disponibles en vehículos a la venta en la actualidad.

Volkswagen *City Emergency Brake*

Volkswagen *City Emergency Brake* [9] es un sistema de frenado de emergencia basado en LIDAR (*LIght Detection And Ranging*, detección y

localización por luz)[1] que pretende ayudar al conductor a evitar los accidentes a baja velocidad o, en el peor de los casos, reducir su severidad. Para velocidades entre 5 km/h y 30 km/h, *City Emergency Brake* monitoriza un área de 10 metros por delante del coche en busca de vehículos que podrían presentar riesgo de colisión (Fig. 1.4).

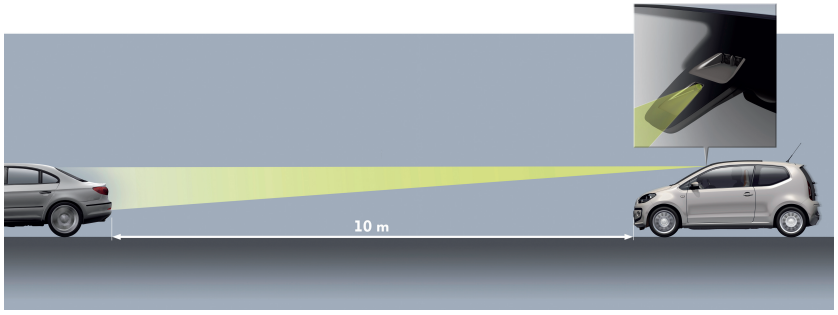


Figura 1.4. Sistema *City Emergency Brake*.

Si es probable que se produzca una colisión, *City Emergency Brake*, en primer lugar, pre-carga los frenos y aumenta su sensibilidad, de manera que el coche está preparado para responder más rápidamente a la acción de frenado del conductor. Sin embargo, si este no reacciona y la colisión es inminente, aplica los frenos fuertemente.

En condiciones ideales, el coche debería ser capaz de frenar lo suficiente como para evitar la colisión si la velocidad relativa entre él y el obstáculo es de menos de 20 km/h. Para velocidades relativas más elevadas, el sistema no será capaz de prevenir la colisión pero reducirá la velocidad de impacto.

Si el conductor interviene para tratar de evitar el accidente, ya sea acelerando fuertemente o girando el volante, el sistema se desactiva y permite al conductor completar la maniobra.

City Emergency Brake no alerta al conductor de que se aproxima una colisión y, cuando frena, lo hace a última hora y de manera muy brusca. Este comportamiento es deliberado: resulta poco confortable y los conductores no confían plenamente en el sistema para evitar los accidentes, manteniendo la adecuada atención a la conducción.

[1]El concepto de LIDAR se describirá con más detalle en la sección 1.3.

Opel *Adaptive Forward Lighting*

Adaptive Forward Lighting, de Opel [10], es un sistema que pretende ayudar a los conductores a observar objetos antes de lo que lo podrían hacerlo si entraran en una curva en total oscuridad, para prevenir accidentes nocturnos en situaciones de giro.

Normalmente, las luces delanteras del vehículo apuntan directamente hacia delante. Cuando se toma una curva, el sistema ilumina aquella parte de la carretera hacia la cual se dirige el vehículo, girando los faros hacia el interior de la curva (Fig. 1.5). Esto proporciona una mejor visión de la curva, permitiendo identificar antes los posibles peligros presentes en ella. Además, el sistema proporciona apoyo extra en curvas cerradas o maniobras en ciudad. La mayoría de las situaciones de giro son a baja velocidad, y el sistema es más efectivo evitando accidentes con peatones y ciclistas.

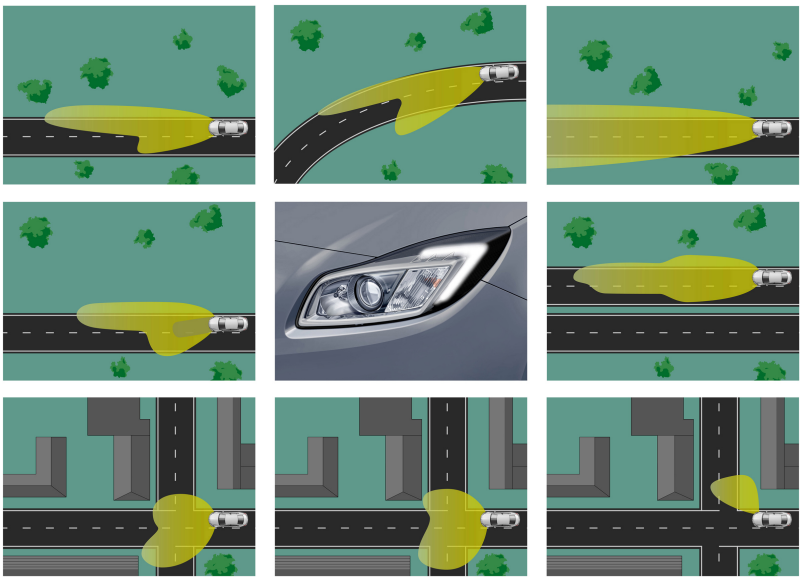


Figura 1.5. Ilustración del funcionamiento del sistema *Adaptive Forward Lighting*.

Mercedes-Benz *Attention Assist*

Attention Assist, desarrollado por Mercedes-Benz y lanzado en 2009 [11], es un sistema que trata de ayudar a los conductores reconociendo cuándo

están somnolientos o distraídos y animándoles a tomar un descanso. Cuando los conductores están alerta, toman consciencia constantemente de la posición del coche y llevan a cabo pequeños ajustes de la dirección para guiar al vehículo por un camino seguro. Sin embargo, cuando los conductores están cansados, hay periodos de desatención, durante los cuales hay pocos cambios de dirección, seguidos de correcciones bruscas y exageradas cuando se recupera la atención.

Attention Assist usa un sensor para captar los giros del volante y controlar cómo está controlando el coche el conductor. A velocidades entre 80 y 180 km/h, el sistema identifica el patrón de giro del volante, lo compara con aquellos que son característicos de una conducción bajo el influjo del sueño, y lo combina con otra información como la hora o la duración del viaje. Si se identifica un conjunto de elementos problemático, el sistema alerta al conductor para tomar un descanso mostrando una señal (una taza de café) en el cuadro de instrumentos (Fig. 1.6) y lanzando un avisador acústico.

Figura 1.6. Aviso visual del sistema *Attention Assist*.

El conductor puede atender el aviso y hacerlo desaparecer de la pantalla. Si el conductor no descansa y el estilo de conducción sigue siendo peligroso, el aviso se repetirá después de 15 minutos.

Audi *Pre Sense Front Plus*

El sistema *Pre Sense Front Plus* de Audi, introducido en el Audi A8 de 2011 [12], está diseñado para evitar o, al menos, reducir los efectos de los accidentes por alcance, ya sea en movimiento o con el vehículo detenido. Dos radares de largo alcance, ubicados en la parte delantera del vehículo, pueden detectar los vehículos por delante del coche con los que es posible colisionar a menos que se realice alguna acción. La información de los radares se combina con datos de una cámara montada en el parabrisas para calcular la probabilidad de un impacto.

El sistema funciona en cuatro fases, mostradas en la Fig. 1.7, que avanzan conforme aumenta la posibilidad de impacto sin haberse producido intervención por parte del conductor:

1. En la primera fase, se lanza un aviso óptico y acústico al conductor, y se prepara el sistema de frenado en previsión de un frenado de emergencia.

2. Si el conductor no reacciona, se utiliza una pequeña sacudida de los frenos como advertencia. Al mismo tiempo, los frenos se preparan de forma que, si el conductor pisa el pedal de freno, con cualquier intensidad, el sistema aplicará la fuerza necesaria para evitar, o minimizar, la colisión. Además, se tensan los cinturones de seguridad.

3. En caso de que el conductor siga sin responder, el sistema aplica un frenado parcial.

4. Finalmente, cuando el accidente ya no se puede evitar, el sistema aplica de forma autónoma la fueza de frenado máxima para reducir la severidad. Las luces de emergencia, asimismo, se encienden para alertar a otros usuarios.

Pre Sense Front Plus funciona a velocidades de hasta 200 km/h, y los sistemas de aviso al conductor, a velocidades incluso superiores.

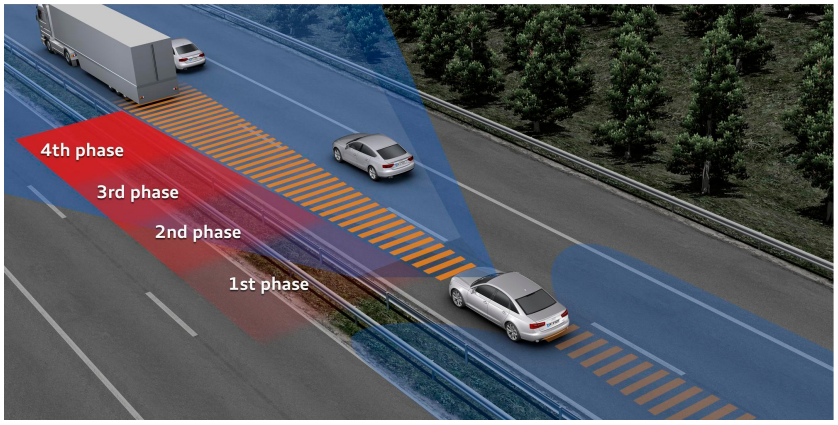


Figura 1.7. Fases del sistema *Pre Sense Front Plus*.

1.2.2 Proyectos de investigación

Además de los esfuerzos llevados a cabo por las marcas comerciales para desarrollar sistemas que se puedan aplicar con inmediatez en vehículos de calle, hay numerosos proyectos en curso en todo el mundo que tienen objetivos aún más ambiciosos, orientados a la conducción totalmente autónoma. Se describe, a continuación, el proyecto desarrollado por Google, por los logros alcanzados, así como dos de los más relevantes en territorio español.

Google

La multinacional de servicios de Internet Google está desarrollando su propio programa de vehículos autónomos. En agosto de 2012, el equipo de desarrollo anunció que habían sido capaces de circular más de 300 000 millas (482 803 km) de forma autónoma sin sufrir ningún tipo de accidente [13].

La flota está formada por, al menos, ocho vehículos: seis Toyota Prius (como el de la Fig. 1.8), un Audi TT y un Lexus RX450h.

Los vehículos de Google captan el entorno a través de los siguientes sensores [14]:

- Un LIDAR giratorio en el techo que genera un mapa tridimensional del entorno a una distancia de 60 m.

Figura 1.8. Vehículo autónomo de Google.

- Una videocámara montada cerca del retrovisor interior que detecta los semáforos y ayuda a los ordenadores a bordo a reconocer el movimiento de obstáculos como peatones o ciclistas.
- Un estimador de posición, montado en la rueda trasera izquierda, que mide pequeños movimientos realizados por el vehículo y ayuda a localizar de manera más precisa su posición en el mapa
- Cuatro radares estándar de automoción, tres en la parte delantera y uno en la trasera, para ayudar a determinar las posiciones de los objetos distantes.

AUTOPIA

El programa AUTOPIA [15] es un proyecto español promovido por el CSIC (Centro Superior de Investigaciones Científicas) y la UPM (Universidad Politécnica de Madrid) que lleva en desarrollo desde 1998. Recientemente (junio de 2012), un IV fruto de este proyecto (*Platero*) fue capaz de completar por sí mismo un recorrido de 100 km, desde San Lorenzo de El Escorial hasta Arganda del Rey, a una velocidad media de 60 km/h, circulando tanto por ciudad como por autovía [16].

El sistema se basa en el empleo de un vehículo guía que va por delante del IV y le informa de las condiciones exactas de la calzada (Fig. 1.9). Este, a su vez, utiliza sistemas de lógica difusa para guiarse sin necesidad de intervención humana.

Figura 1.9. Pruebas del proyecto AUTOPÍA en carreteras convencionales

IVVI

Otro proyecto relevante en el ámbito de los ADAS en España es el IVVI (*Intelligent Vehicle based on Visual Information*, vehículo inteligente basado en información visual) de la UC3M (Universidad Carlos III de Madrid), responsabilidad del LSI (Laboratorio de Sistemas Inteligentes). El prototipo actual es ya el segundo que se ha desarrollado en esta institución, por lo que recibe el nombre de IVVI 2.0 (Fig. 1.10). Se trata de un vehículo comercial (un Nissan Note) en el que se han montado distintos sensores basados en información visual para captar el entorno (e, incluso, el interior del propio vehículo), lo que permite alertar al conductor en situaciones de peligro.

El vehículo está equipado con:

- Un monitor que constituye la interfaz a través de la cual se informa y/o alerta al conductor de las situaciones que requieren de su atención.

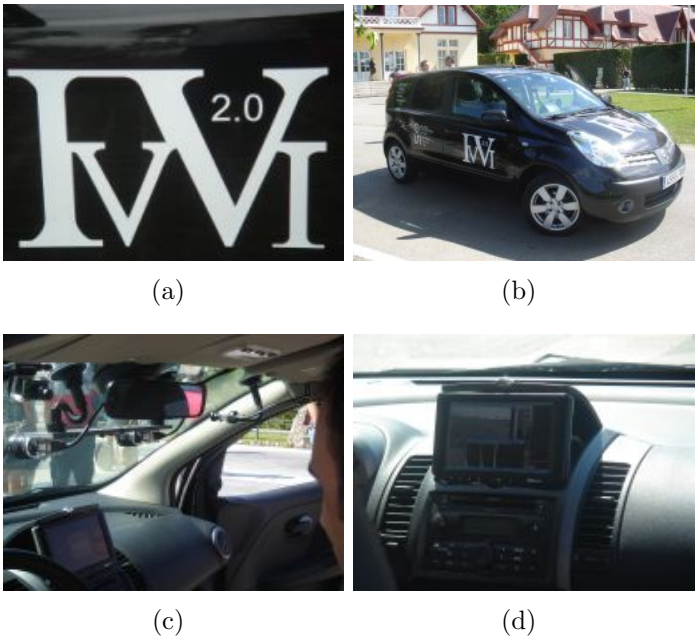


Figura 1.10. Detalles del vehículo de experimentación IVVI 2.0: Rótulo identificativo (a), vista exterior (b), parabrisas y salpicadero (c) y consola central (d).

- Dos cámaras a color que permiten detectar, respectivamente, la señalización vertical de la vía y el estado de somnolencia del conductor.
- Un sistema de visión estéreo para la detección de obstáculos durante la conducción diurna.
- Una cámara infrarroja para la detección de obstáculos en condiciones de baja iluminación.
- Un sensor láser, situado en el parachoques delantero, para la detección y clasificación de peatones y vehículos.

El algoritmo desarrollado en el presente proyecto está concebido para pasar a formar parte de este vehículo, utilizando el sistema de visión estéreo ya mencionado.

1.3 Percepción del Entorno

Los ADAS basan su funcionamiento en la recepción continua de información sobre el entorno del vehículo. Con este fin, se emplean, tal y como se ha comprobado en el epígrafe anterior, varios tipos de sensores, muchos de ellos surgidos de aplicaciones militares. Por ejemplo, es frecuente encontrar:

- Radares [17], que usan ondas de radio para determinar la distancia, la altura, la dirección o la velocidad de los objetos cercanos al vehículo.
- LIDAR (*LIght Detection And Ranging*, detección y localización por luz) capaces de medir la distancia hasta un cierto objetivo (y otras propiedades del mismo) iluminándolo con luz, frecuentemente láser. Un ejemplo de este tipo de sensor se muestra en la Fig. 1.11.
- Sensores de ultrasonidos, similares a los radares pero que utilizan ondas sonoras en lugar de ondas de radio.

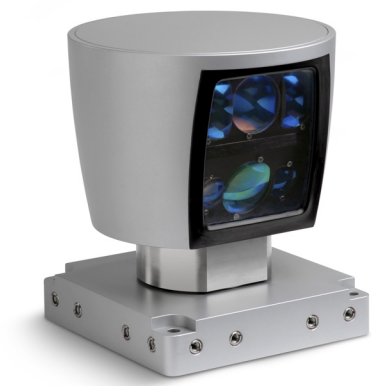


Figura 1.11. Sensor LIDAR.

Todos estos sensores se denominan *activos*, pues basan su funcionamiento en medir el tiempo que transcurre entre la emisión de una señal física y la recepción de la parte de ella que retorna tras impactar con el entorno. Nótese que, en este ámbito, no se suelen emplear sensores táctiles ni acústicos, debido a la velocidad de los vehículos, en el primer caso, y al reducido rango de detección, en el segundo.

Estos sistemas adolecen de problemas como la baja resolución espacial y la baja velocidad de escaneo. Por ello, también conviene considerar la utilidad de los sensores *pasivos*, especialmente aquellos basados en visión.

1.3.1 Percepción del entorno mediante visión

Los sistemas de visión utilizan cámaras para captar información sobre el entorno. Numerosos ADAS pueden aprovechar esta información para proporcionar al conductor ayudas como [18]:

- **Mantenimiento en el carril**. La posición relativa del vehículo con las líneas que marcan los límites del carril puede determinarse analizando imágenes capturadas por cámaras.
- **Protección de peatones**. Los sistemas de visión permiten detectar todos los peatones en torno al vehículo y analizar su actividad y movimiento para determinar y localizar aquellos casos que podrían resultar peligrosos.
- ***Adaptive Cruise Control*, control de crucero adaptativo**. Estos sistemas reconocen los vehículos precedentes y calculan su velocidad y trayectoria. Presentan una gran dificultad por la diversidad de apariencia que presentan los vehículos y por los cambios de perspectiva que sufren las cámaras.
- **Percepción de señales de tráfico**. La percepción de señales de tráfico es tan útil como complicada, debido a la presencia de otros objetos con los mismos colores, o debido a la oclusión de objetos o a las condiciones cambiantes de luz.

Frente a los sensores referidos en el epígrafe anterior, los sistemas de visión presentan las siguientes ventajas [19]:

- A diferencia de los sensores activos, las cámaras permiten adquirir datos de forma no invasiva; esto es, sin alterar el entorno. Además de evitar problemas con la contaminación ambiental y el cumplimiento de

normativas de seguridad, esto resulta importante de cara al futuro, donde es previsible una generalización en el uso de los ADAS. La utilización masiva de sistemas activos puede suponer la aparición de interferencias entre sensores del mismo tipo pertenecientes a distintos vehículos o, incluso, al mismo vehículo, impidiendo su correcto funcionamiento.

- La toma de imágenes se puede efectuar lo suficientemente rápido como para captar el movimiento de los distintos elementos del entorno del vehículo, de forma que pueden utilizarse satisfactoriamente en aplicaciones de ayuda a la conducción.

- Determinadas aplicaciones necesitan forzosamente información visual, como es el caso de la localización de líneas de carril o el reconocimiento de señales de tráfico.

En cambio, estos sensores suelen resultar menos robustos que los activos ante variaciones en la iluminación, ya sean causadas por la presencia de condiciones de niebla, noche o deslumbramiento o por sombras proyectadas por los edificios en entornos urbanos.

Los cámaras pueden clasificarse, en primer término, en:

- **Cámaras infrarrojas** o térmicas, que forman una imagen usando la radiación infrarroja. La información que reciben no es la misma que la que captan los seres humanos a través del sentido de la vista, ya que trabajan en longitudes de onda mayores, de hasta 14 000 nm. Esto permite su empleo en situaciones de escasa o nula visibilidad.

- **Cámaras de luz visible**, que utilizan la luz visible para formar la imagen. Captan las longitudes de onda a las que es sensible el ojo humano: entre 450 y 750 nm.

Centrándose en este último grupo, también aparecen distintos tipos de cámaras:

- **Cámaras monoculares**. Están compuestas por una única lente. Su utilización para aplicaciones de percepción tiene como ventajas la facilidad

de integración y el no requerir ineludiblemente calibrar entre sensores, lo que se traduce en un menor coste de adquisición y mantenimiento. Sin embargo, permiten obtener menos información que las siguientes.

- **Cámaras omnidireccionales**. Son cámaras con un campo de visión de 360° en el plano horizontal, o con un campo visual que cubre toda la esfera. Permiten captar información en todas direcciones (Fig. 1.12), pero, a cambio, el espejo introduce una gran deformación y resultan más caras.

- **Sistemas de visión estéreo**. Tratan de imitar la configuración de los ojos en la mayoría de seres vivos: se componen de dos cámaras idénticas separadas por una distancia fija y sincronizadas para tomar imágenes de forma simultánea. La utilización de este tipo de sistemas requiere llevar a cabo un proceso de calibración entre los dos sensores. A cambio, permiten la obtención de información tridimensional del entorno.

Figura 1.12. Imágenes obtenidas con una cámara omnidireccional.

En el presente proyecto se va a requerir información tridimensional sobre el área dispuesta en frente del vehículo. Por ello, como ya se ha mencionado, se va a hacer uso del sistema estéreo montado en el IVVI 2.0, que es una cámara Bumblebee 2 de la marca Point Grey, como la que se muestra en la Fig. 1.13 [20].

Consta de dos sensores CCD (*Charge-Coupled Device*, dispositivo de carga acoplada), y es capaz de tomar imágenes de 640 × 480 a 48 FPS (Fotogramas Por Segundo), o de 1024 × 768 a 20 FPS.

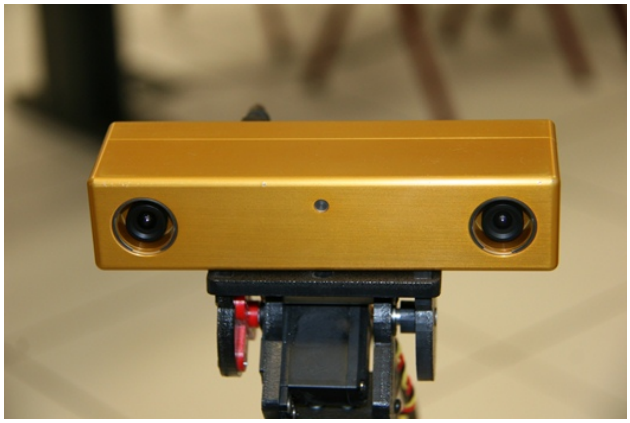


Figura 1.13. Sistema estéreo Bumblebee 2, de Point Grey.

1.4 Posicionamiento del vehículo

Los sistemas de posicionamiento constituyen en la actualidad un elemento clave para numerosos ADAS. Son la base sobre la que se construyen los sistemas de guiado de vehículos autónomos [21], aunque también hay numerosas aplicaciones dentro de los ITS que los requieren: gestión de flotas, de emergencia, información de llegada de autobuses, etc.

Pese a que el sistema GPS (*Global Positioning System*, sistema de posicionamiento global) se ha impuesto en las últimas décadas como el más usado [22], adolece de problemas que hacen recomendable seguir avanzando en la investigación de sistemas alternativos, como la odometría visual.

1.4.1 Sistema GPS

El *Global Positioning System*, sistema de posicionamiento global (GPS) es un sistema de navegación por satélite que proporciona información sobre la localización y la hora en cualquier condición meteorológica y en cualquier lugar sobre la superficie de la Tierra o cerca de ella. Se trata de un proyecto promovido por el Departamento de Defensa de los Estados Unidos, que se inició en 1973 y fue completado en 1994. Consta de, al menos, 24 satélites en órbita (Fig. 1.14) y numerosas estaciones de control repartidas por el globo terráqueo.

Figura 1.14. Ilustración del conjunto de satélites que forman parte del sistema GPS.

Un receptor GPS calcula su posición temporizando de manera precisa las señales enviadas por estos satélites. Cada uno de ellos trasmite continuamente un mensaje que incluye:

- La identificación del satélite.
- La hora exacta en que se transmitió el mensaje.
- La posición del satélite en ese momento.

El receptor usa los mensajes que recibe para determinar el tiempo de tránsito de cada mensaje y computa la distancia a cada satélite usando la velocidad de la luz. Cada una de estas distancias y las posiciones de los satélites definen una esfera, sobre la cual se encuentra el receptor. Es posible, entonces, calcular la posición del receptor usando las ecuaciones de navegación. Esta posición, y otros datos derivados de ella (velocidad, dirección, etc.) puede ser utilizada entonces por el usuario.

Se requieren cuatro satélites para determinar la posición en el espacio del receptor; no obstante, si una de las coordenadas es ya conocida, se puede obtener a partir de solo tres de ellos.

Actualmente, el sistema sigue siendo responsabilidad del Gobierno de Estados Unidos, aunque se permite el libre acceso a cualquiera con un receptor GPS. Esto, unido a que los receptores GPS comerciales estiman la posición con

un margen de error de tan solo unos 15 metros [23], ha permitido que su uso se haya extendido en todo tipo de ámbitos, ya que la fiabilidad del sistema se adecua a las necesidades habituales en aplicaciones no críticas.

Sin embargo, hay numerosas fuentes de error que afectan a las medidas que proporciona el sistema [24]:

- **Geometría de los satélites**. Una determinada posición relativa de los satélites entre sí desde el punto de vista del receptor puede impedir obtener una medida correcta.
- **Órbita de los satélites**. Pese a que los satélites están posicionados en órbitas muy precisas, es posible que tengan lugar pequeños cambios en las órbitas por fuerzas gravitatorias, que obligan a actualizar los datos enviados a los receptores regularmente. Puede suponer errores de en torno a 2 m.
- **Efecto multi-ruta**. Está causado por la reflexión de las señales de los satélites en los objetos. Tiene especial importancia en entornos urbanos, con edificios altos (Fig. 1.15).
- **Efectos atmosféricos**. La propagación de las ondas a través de la troposfera y la ionosfera es inferior a la velocidad de la luz.
- **Inexactitudes de reloj y errores de redondeo**. A pesar de la sincronización del reloj del receptor con el satélite durante la determinación de la posición, hay una imprecisión remanente.
- **Efectos relativísticos**. El tiempo es un factor relevante en el GPS, requiriendo precisiones de entre 20 y 30 nanosegundos. Sin embargo, los satélites se mueven a velocidades de unos 12 000 km/h, haciendo que el tiempo transcurra ligeramente más despacio de acuerdo a la Teoría de la Relatividad.

Son problemas que se agravan en entornos urbanos, debido a las constantes pérdidas de cobertura de señal, a la gran cantidad de calles y ramificaciones presentes e, incluso, a la presencia de túneles.

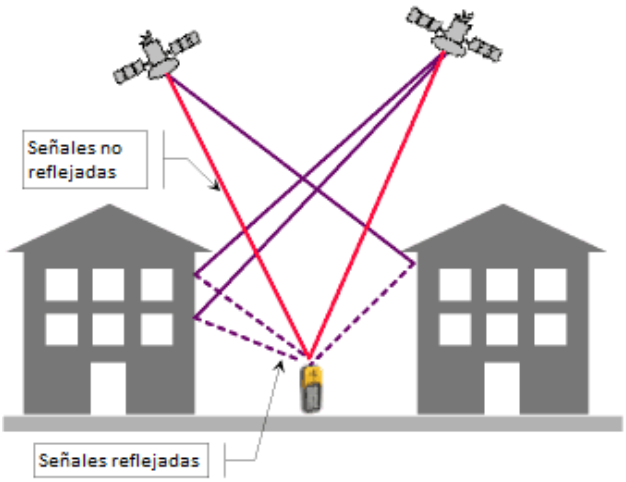


Figura 1.15. Problema del GPS en entornos urbanos.

Por eso, la utilización del GPS en determinados sistemas de ayuda a la conducción de carácter crítico resulta insatisfactoria. Los sistemas basados en visión por computador proporcionan una alternativa útil, a través de la llamada odometría visual.

1.4.2 Odometría Visual

Se puede definir la odometría visual como el proceso mediante el cual se determina la posición y la orientación de una cámara o de un sistema de cámaras mediante el análisis de una secuencia de imágenes adquiridas, sin ningún conocimiento previo del entorno [25]. En inglés, a este movimiento propio que se pretende estimar se le conoce como *ego-motion*.

Los algoritmos de odometría visual suelen basarse en procesar las imágenes recibidas a través de las cámaras para detectar una serie de puntos pertenecientes a objetos estáticos, e ir siguiendo su movimiento en sucesivos *frames* o fotogramas, tal y como se muestra en la Fig. 1.16.

Esto presenta ciertos problemas en entornos urbanos, donde muchos objetos están en movimiento y además, provocan oclusiones parciales sobre otros presentes en la escena. Además, en estas condiciones, los vehículos se

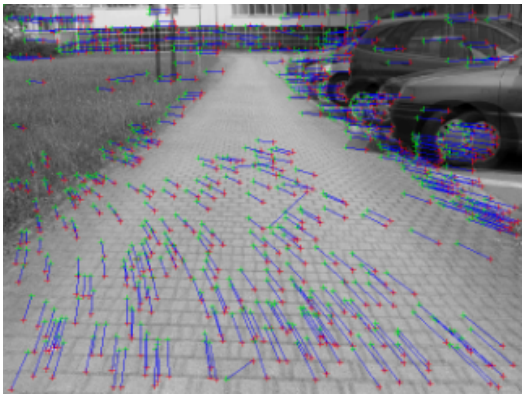


Figura 1.16. Detección y seguimiento de puntos característicos en un sistema de odometría visual.

desplazan a bajas velocidades, lo que puede llevar a considerar como estáticos puntos que, en realidad, forman parte de otros vehículos en movimiento.

El objetivo del presente proyecto es implementar un algoritmo de odometría visual que permita obtener la posición de un vehículo mediante técnicas de visión por computador y que pueda utilizarse en entornos urbanos.

Para conocer la situación de partida en este ámbito, en el siguiente capítulo se hará un breve repaso de los trabajos ya existentes basados en esta tecnología, usando los distintos tipos de cámaras anteriormente descritos: monoculares (sección 2.1), omnidireccionales (sección 2.2) y estéreo (sección 2.3).

Procesamiento de imágenes en sistemas de odometría visual

Los sistemas de odometría visual suelen requerir operaciones de procesamiento de imágenes que resultan altamente costosas desde un punto de vista computacional, tales como la ya descrita detección de puntos característicos. Por ello, al igual que sucede en general en el ámbito de la visión por computador, es frecuente el uso de sistemas *hardware* equipados con tarjetas de procesamiento de imágenes.

Estas tarjetas están equipadas con un procesador propio, la GPU (*Graphics Processing Unit*, unidad de procesamiento gráfico), dedicado en exclusiva al

procesamiento de imágenes para liberar de estas tareas al procesador central, la CPU (*Central Processing Unit*, unidad central de proceso).

El algoritmo de odometría visual que constituye el objeto del presente proyecto se va a implementar sobre una tarjeta gráfica que utiliza SIMD (*Single Instruction, Multiple Data*, "una instrucción, múltiples datos"). Esta filosofía consiste en llevar a cabo la misma instrucción sobre múltiples datos al mismo tiempo, lo cual resulta especialmente útil para acelerar el procesamiento de imágenes.

Más adelante (sección 4.1) se realizará una descripción más profusa de las ventajas que supone la utilización de estos dispositivos.

2 Trabajos previos

En este capítulo se pretenden repasar los desarrollos ya existentes para la localización de vehículos mediante sistemas basados en visión por computador. Se mostrarán trabajos que utilizan cada uno de los tipos de cámaras de luz visible mostradas en el capítulo anterior (sección 1.3.1).

2.1 Sistemas de visión monoculares

Pese a las ventajas de simplicidad y coste que suponen las cámaras monoculares, estimar el movimiento del vehículo a partir de ellas resulta problemático al ser necesario estimar un factor de escala para recuperar las dimensiones reales de la escena.

El trabajo más representativo en este ámbito es el de Stein et ál. [26]. En él, se desarrolla un método para estimar la posición del vehículo respecto a la carretera usando una única cámara montada junto al espejo retrovisor, que sirve también para su utilización en otros sistemas basados en visión, como un detector de obstáculos y otro de abandono involuntario de carril.

El trabajo parte de dos suposiciones:

- Se considera que la carretera es una estructura plana, y las medidas se toman sobre ella; los puntos exteriores no se tienen en cuenta.

- El movimiento se puede describir únicamente con tres parámetros: traslación hacia delante y ángulos de guiñada y cabeceo. Esto permite mejorar la robustez y la rapidez de procesamiento del sistema.

Además, los datos se combinan en funciones de probabilidad que mejoran la robustez al ignorar un gran número de datos erróneos, similares a los que aparecen en situaciones de tráfico real. Los autores aseguran que el método presenta un buen comportamiento en entornos reales, incluso bajo condiciones de deslumbramiento, lluvia y objetos en movimiento.

Otro estudio relevante es el llevado a cabo en Francia por Mouragnon et ál [27], en el que, aparte de estimarse el movimiento, se pretende obtener la geometría tridimensional del entorno. Los puntos que sirven de referencia se obtienen con el detector de esquinas de Harris [28], y los datos erróneos se descartan con RANSAC, método que se empleará en el presente proyecto y será descrito profusamente en la sección 3.4.

2.2 Sistemas de visión omnidireccionales

Los algoritmos basados en sistemas de visión omnidireccionales resultan idóneos para estimar movimientos en cualquier dirección, si bien hay que manejar con cuidado la distorsión introducida por el espejo.

Scaramuzza et ál desarrollan en [29] un algoritmo de tiempo real para computar la estimación del movimiento del vehículo respecto a la carretera. Se usan para ello imágenes procedentes de una única cámara omnidireccional montada en el techo del vehículo (Fig. 2.1).

La traslación y la rotación se estiman por separado: la primera, mediante un método que detecta y sigue puntos obtenidos mediante SIFT[1]; la segunda, a través del método de aproximación propuesto en [30]. El método obtiene resultados satisfactorios en una trayectoria de 400 m (Fig. 2.2).

[1]Más adelante, en la sección 3.3, se describirá con detalle el detector SIFT, junto al utilizado en este trabajo, SURF.

Figura 2.1. Vehículo utilizado en las pruebas del método propuesto por Scaramuzza et ál.

Figura 2.2. Estimación de una trayectoria cerrada con el método propuesto por Scaramuzza et ál.

2.3 Sistemas de visión estéreo

En general, la estimación de la posición basada en la utilización de sistemas estéreo proporciona resultados más estables y presenta un mejor comportamiento que los métodos anteriores, gracias a poder contar directamente con información tridimensional del entorno. Esta información suele representarse en los llamados *mapas de disparidad*, que proporcionan la profundidad respecto a la cámara de los objetos que aparecen en una escena. Más adelante, en la sección 3.2, se dará una descripción más detallada de los mismos, al constituir un elemento fundamental en este proyecto.

Habitualmente, los pasos que siguen estos sistemas son:

1. Obtención de puntos estáticos en la escena que puedan usarse como referencia.

2. Emparejamiento de estos puntos en cada par de imágenes izquierda/derecha obtenido con el sistema estéreo y posterior obtención de su posición en el espacio mediante triangulación.

3. Seguimiento de los puntos a lo largo de varias capturas de la cámara.

Pese a las ventajas que suponen estos sistemas, aparece una gran incertidumbre asociada a la estimación de la profundidad. Por ello, hay autores [31] que trabajan directamente en el espacio de disparidades aprovechando los parámetros de calibración del par estéreo. Otro punto problemático es el emparejamiento de puntos, que frecuentemente requiere utilizar métodos de rechazo como RANSAC[2] [32].

Uno de los trabajos más representativos es el de Nistér et ál. [33]. En él, se desarrolla un sistema de odometría visual que puede utilizarse con una sola cámara o con un sistema estéreo. No requiere conocimiento previo de la escena ni del movimiento, puede computarse en tiempo real y está destinado a utilizarse en la navegación tanto de plataformas aéreas, como de dispositivos de mano, y, especialmente, de vehículos terrestres autónomos.

La versión estéreo se basa en la obtención de puntos de referencia mediante el detector de esquinas de Harris [28], su emparejamiento en cada par de imágenes tomado por la cámara y su seguimiento a lo largo de la secuencia (Fig. 2.3). Para rechazar puntos erróneos se utiliza una variación *preventiva* de RANSAC.

Resultados obtenidos con este método se observan en la Fig. 2.4. Es importante resaltar que este método realiza una estimación tridimensional del movimiento, por lo que, según los autores, estos resultados obtenidos en movimientos en el plano demuestran la robustez y precisión del método.

Otro trabajo que conviene citar es el de A. Howard en [34], que presenta como principales características el no requerir de asunciones previas sobre el movimiento de la cámara y el operar sobre mapas densos de disparidad

[2]En la sección 3.4 se verá la forma en que RANSAC permite descartar datos erróneos.

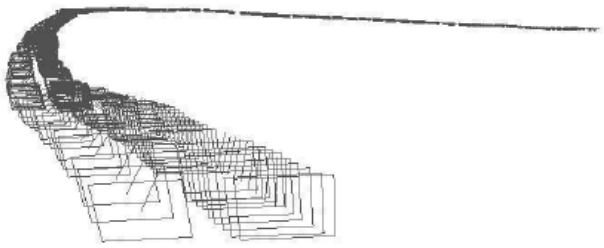
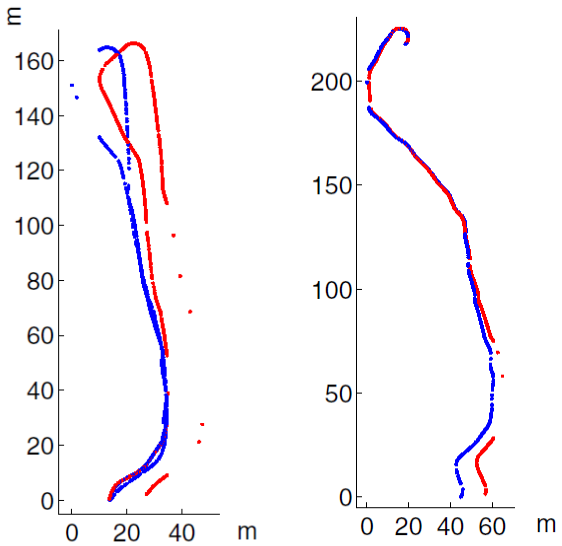


Figura 2.3. Representación de la secuencia de imágenes utilizadas en la odometría visual de Nistér et ál.



(a) (b)

Figura 2.4. Resultados obtenidos con el método propuesto por Nistér et ál. (en rojo) frente a los procedentes de un GPS diferencial (en azul), en terreno montañoso (a) y llano (b).

computados por un algoritmo separado. El autor afirma conseguir errores inferiores a 1 m para recorridos de 400 m de longitud (Fig. 2.5).

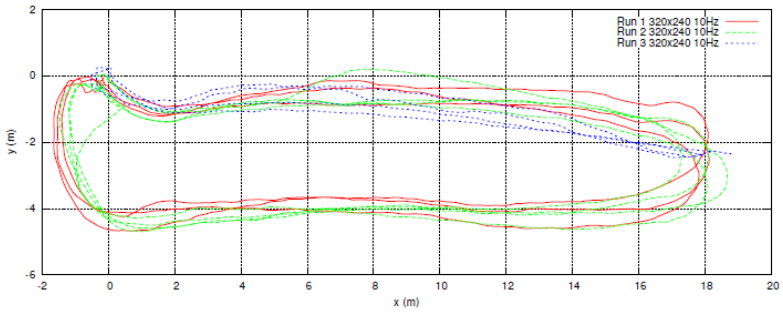


Figura 2.5. Distintas estimaciones de una misma trayectoria cerrada proporcionadas por el sistema propuesto por Howard.

También en España se han desarrollado proyectos en este ámbito; por ejemplo, el trabajo de Parra et ál, de la Universidad de Alcalá de Henares [35]. Estos autores introducen la utilización del detector de características SIFT (*Scale-Invariant Feature Transform*, transformación de características invariante a la escala), ya mencionado con anterioridad, que permite obtener mejores resultados en entornos complejos como los urbanos.

Odometría visual en Marte

Un buen ejemplo de la aplicación práctica de los algoritmos de odometría visual se encuentra en las *rovers* o MER (*Mars Exploration Rovers*, rovers de exploración de Marte) enviadas por la NASA (*National Aeronautics and Space Administration*, Administración Nacional de Aeronáutica y del Espacio) para la exploración de Marte: *Spirit*, *Opportunity* [36] y la más reciente *Curiosity* [37].

Los sistemas de odometría visual dotan a estos vehículos de un conocimiento preciso de su posición, lo que le permite detectar y compensar de forma autónoma los deslizamientos que ocurren durante el movimiento. Esto permite que se trasladen de forma más segura y efectiva en terrenos arenosos con grandes pendientes, lo que supone una reducción del tiempo necesario para alcanzar las zonas de interés de Marte.

Figura 2.6. Ilustración del funcionamiento del sistema de odometría visual montado en *Curiosity*.

Este sistema consta de *software* embarcado capaz de comparar los pares estéreo tomados por las cámaras de las que están dotadas estos vehículos, denominadas NAVCAM (*NAVigation CAMeras*, cámaras de navegación), con una resolución de 256 × 256. A partir de ellas, se actualizan los seis grados de libertad del movimiento de las *rovers*, siguiendo el movimiento de puntos seleccionados en el terreno de forma autónoma.

Este sistema demostró funcionar con éxito en las dos primeros *rovers* enviados, consiguiendo una efectividad del 97 % en *Spirit* y del 95 % en *Opportunity*, y permitiendo detectar correctamente deslizamientos de hasta 125 % y cambios de posición de tan solo 2 mm, incluso sobre pendientes de 31°.

En *Curiosity*, el sistema se ha convertido en una funcionalidad clave. Cuando este *rover* se desplaza por la superficie de Marte, deja huellas con letras codificadas en código Morse. Esto permite que los puntos tomados como referencia queden inequívocamente identificados, y se puedan efectuar mediciones absolutas, y no solo relativas, de los desplazamientos (Fig. 2.6).

3 Fundamentos Teóricos

Este capítulo está destinado a mostrar las nociones teóricas sobre las que se apoya el desarrollo del presente proyecto. Se trata, en su mayoría, de conceptos pertenecientes al ámbito de la visión por computador.

3.1 Principios ópticos

3.1.1 Modelos de lente

Las cámaras, encargadas de captar la información visual del entorno, suelen estar dotadas de una óptica, cuya misión es la de captar los rayos luminosos y concentrarlos sobre el sensor, habitualmente, de tecnología CCD o CMOS (*Complementary Metal-Oxide-Semiconductor*, semiconductor metal-óxido complementario) [38].

Los rayos paralelos que pasan por una lente convexa convergen hacia un punto, llamado *punto focal* (F_1 y F_2 en las lentes de la Fig. 3.1). La distancia entre el eje de la lente y ese punto es la distancia focal (f), que es el parámetro principal para calcular la posición y el tamaño de los objetos en la imagen. El eje óptico, por su parte, es la línea perpendicular a la lente y que la atraviesa por su punto medio.

El comportamiento de las lentes se puede representar a través de dos modelos distintos:

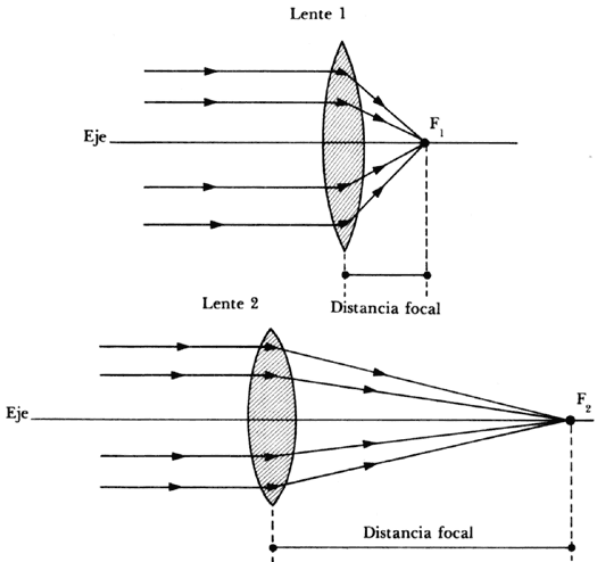


Figura 3.1. Punto focal y distancia focal de dos lentes distintas.

- El modelo de **lente fina**, que considera que los rayos que inciden de forma perpendicular a la lente convergen en el punto focal (Fig. 3.2).

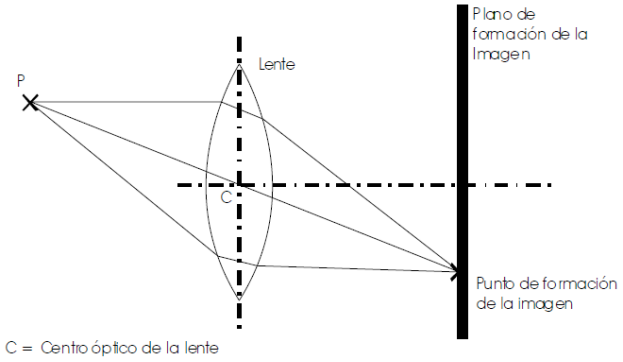


Figura 3.2. Modelo de lente fina.

- El modelo ***pin-hole***, que se basa en reducir la óptica a un punto situado a la distancia focal de la imagen. De todos los rayos luminosos procedentes de un objeto, solamente se consideran los que pasan directamente por la distancia focal. Se puede considerar que el punto focal está por delante del plano de la imagen, en cuyo caso ésta se obtiene invertida, o que está por detrás, como sucede en la Fig. 3.3.

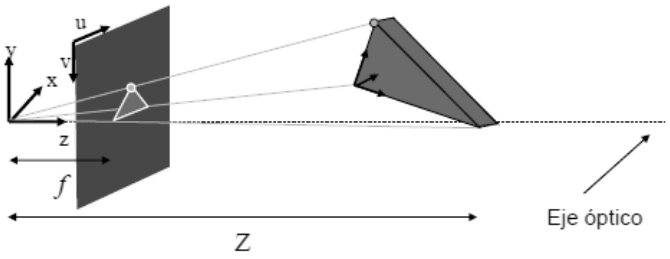


Figura 3.3. Modelo *pin-hole*.

Siguiendo este modelo, las ecuaciones que relacionan un punto del mundo, de coordenadas (X, Y, Z), con su proyección en la imagen, con coordenadas (u, v), son 3.1 y 3.2.

$$u = \frac{f}{Z}X \tag{3.1}$$

$$v = \frac{f}{Z}Y \tag{3.2}$$

Esas ecuaciones evidencian la pérdida de información que supone la captura de una imagen: se hace imposible conocer la profundidad de los puntos, Z. Este es el problema que intenta solventar la visión estéreo.

3.1.2 Sistema estéreo ideal

Un sistema estéreo ideal [39] es aquel compuesto por dos cámaras de igual distancia focal f, paralelas y separadas una distancia B, que recibe el nombre de *baseline*. Un mismo punto P se ve proyectado en cada una de las dos imágenes en los puntos $p_1(u_1, v_1)$ y $p_2(u_2, v_2)$. En general, u_1 y u_2 no serán iguales, sino que estarán separadas por una distancia llamada *disparidad* (d), medida en píxeles. En cambio, v_1 y v_2 sí coinciden. Estos conceptos aparecen ilustrados en la Fig. 3.4.

El *plano epipolar*, por su parte, es el formado por el punto P y los centros ópticos de ambas cámaras, C_1 y C_2. Las *líneas epipolares* son la intersección del plano de cada una de las imágenes con el plano epipolar. Todo esto constituye la llamada *geometría epipolar*, que permite relacionar los objetos tridimensionales con sus proyecciones en la imagen.

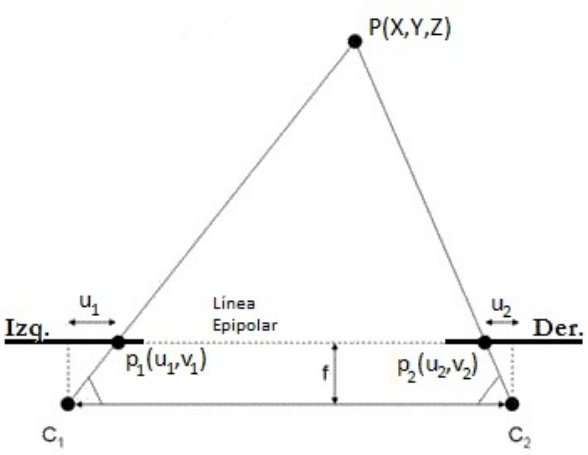


Figura 3.4. Sistema estéreo ideal.

Con esto, y el modelo *pin-hole*, se pueden obtener las coordenadas del mundo de un punto a partir de su proyección en las dos imágenes, mediante las expresiones 3.3, 3.4 y 3.5.

$$X = Z\frac{u_1}{f} - \frac{B}{2} = Z\frac{u_1}{f} + \frac{B}{2} \tag{3.3}$$

$$Y = \frac{v_1 B}{u_1 - u_2} = \frac{v_1 B}{d} \tag{3.4}$$

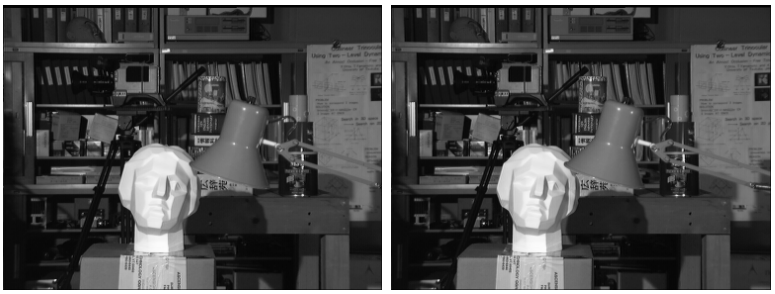
$$Z = \frac{fB}{u_1 - u_2} = \frac{fB}{d} \tag{3.5}$$

Por lo tanto, la profundidad a la que se encuentra un punto resulta ser inversamente proporcional a la distancia que separa a sus proyecciones en ambas imágenes del sistema estéreo (3.5). Esta distancia es, por ello, un elemento clave en la visión estéreo, y se representa en los llamados *mapas densos de disparidad*, objeto del siguiente epígrafe.

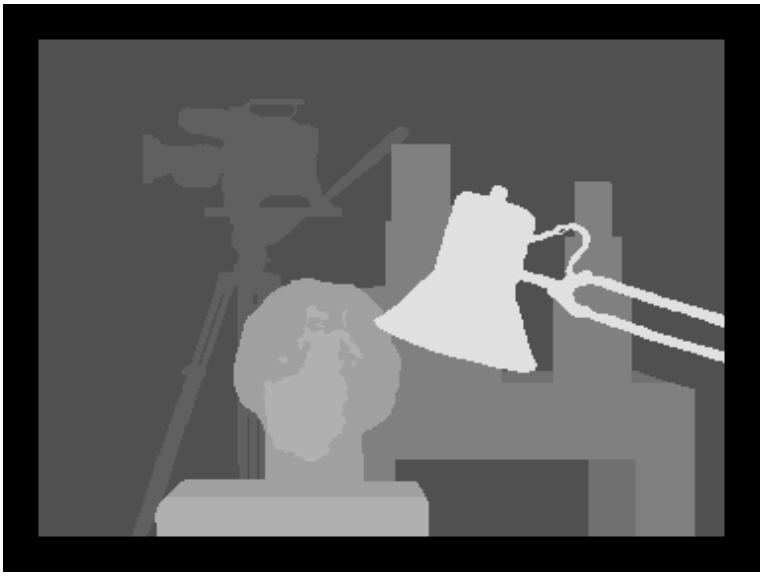
Conviene resaltar que este sistema de visión binocular constituye una imitación del sentido de la vista de una buena parte de los seres vivos, incluidos los seres humanos, que están provistos de dos ojos situados en paralelo para poder disponer, con ayuda del cerebro, de información de la profundidad a la que se encuentran los objetos.

3.2 Mapa denso de disparidad

El mapa denso de disparidad (o, simplemente, mapa de disparidad) de una imagen es una estructura que relaciona la posición de cada punto (píxel) de la imagen con su profundidad en el mundo. Para ello, adopta la forma de una imagen en la que el nivel de gris de cada punto es (o depende de) el valor de disparidad para ese punto entre las dos imágenes estéreo. Un ejemplo se muestra en la Fig. 3.5, donde también aparece el par de imágenes de partida.



(a) (b)



(c)

Figura 3.5. Imágenes estéreo izquierda (a) y derecha (b) y mapa de disparidad ideal (*ground truth*) (c) de una escena.

El cálculo de disparidades es un proceso ampliamente tratado en la literatura sobre visión por computador [40]. Habitualmente, se suele trabajar bajo las siguientes hipótesis:

- Se conocen los parámetros intrínsecos del sistema estéreo utilizado para tomar las imagenes; es decir, la distancia focal de las cámaras, el *baseline*, etc.
- Si las cámaras no están dispuestas de forma perfectamente paralela, las imágenes se someten a un proceso de rectificación para garantizar que se cumpla la geometría epipolar, de forma que las coordenadas verticales de un mismo punto en cada una de las imágenes (v_1 y v_2) son iguales.
- Un mismo punto ha de tener el mismo nivel de gris en ambas imágenes. Esto implica asumir, en primer lugar, que el aspecto de las superficies que aparecen en la imagen no varia en función del punto de vista, y en segundo lugar, que la ganancia de ambas cámaras es exactamente igual.

3.2.1 Filtrado previo de las imágenes

Para garantizar que se cumpla este último punto, es frecuente someter a las imágenes a un filtrado previo, que elimine las diferencias de ganancia entre ambas cámaras. Una posibilidad consiste en aplicar el filtro llamado LoG (*Laplacian of Gaussian*, laplaciana de la gaussiana), citado a veces como *filtro de Marr-Hildreth*.

Este filtro es equivalente a efectuar dos operaciones sucesivas sobre la imagen [38]:

1. Convolucionarla con una gaussiana (3.6), que constituye un filtro paso bajo frecuencial que elimina el *ruido* presente en ella; es decir, aquellos píxeles aislados que toman valores distintos a los de sus vecinos, debido al sensor o al medio de transmisión de la señal utilizados.
2. Obtener la laplaciana (3.7) de la imagen resultante. El operador laplaciana representa la derivada de la imagen en todas direcciones, y

proporciona como resultado una imagen en la que solo toman valores extremos aquellos píxeles que, en la imagen original, suponían cambios bruscos en el nivel de gris; esto es, los bordes de los objetos.

$$G(x,y) = e^{-\frac{(x+y)^2}{2\sigma^2}} \tag{3.6}$$

$$\nabla^2 f(x,y) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \tag{3.7}$$

Por propiedades de la gaussiana, esto es equivalente a aplicar sobre la imagen la laplaciana de la gaussiana (3.8).

$$H(x,y) = (\nabla^2 G(x,y)) * I(x,y) \tag{3.8}$$

3.2.2 Cálculo del mapa de disparidad

El cálculo del mapa de disparidad suele consistir en tomar una de las dos imágenes correspondientes al par como referencia y recorrer la otra en búsqueda de puntos que se correspondan con los presentes en la primera. Habitualmente, se divide en cuatro etapas, descritas en la taxonomía de Scharstein y Szeliski [40]:

- **Cálculo de la función de coste**. El *coste* es una medida de lo distintos que son dos píxeles correspondientes a un mismo punto en cada una de las dos imágenes del par. Hay numerosos métodos para evaluarlo; los más comunes hacen uso de la diferencia en el nivel de gris entre píxeles, ya sea absoluta, AD (*Absolute intensity Differences*, diferencias absolutas de intensidad), o cuadrática, SSD (*Sum of Squared Differences*, suma de diferencias cuadráticas).
- **Agregación del coste**. Habitualmente, la similitud de dos píxeles se evalúa teniendo en cuenta también aquellos otros pertenecientes a

su entorno cercano; esto es, una *ventana* en torno al píxel analizado. También hay gran cantidad de posibilidades en este aspecto: ventanas rectangulares, adaptativas, con pesos adaptativos, métodos iterativos, etc.

- **Cálculo de la disparidad**. En la etapa de cálculo de disparidades, hay varias posibilidades:
  - Métodos locales. Requieren concentrarse en las etapas de cálculo de costes y agregación, dado que calcular las disparidades finales es trivial; consiste en escoger en cada píxel la disparidad asociada al mínimo coste; es lo que se llama optimización WTA (*Winner-Take-All*, "los ganadores se llevan todo").
  - Optimización global. Concentran el trabajo en el cálculo de la disparidad. Tratan de encontrar el valor de la misma que minimice una *energía global* que considera toda la imagen.
  - Programación dinámica. Se basan en calcular un *camino de coste mínimo*.
  - Algoritmos cooperativos. Tienen un comportamiento similar a los métodos de optimización global.
- **Post-procesamiento**. Las etapas finales suelen consistir en refinar los resultados obtenidos, para corregir posibles discontinuidades en la disparidad. Una de las técnicas empleadas es el *cross-checking*, que compara el mapa de disparidad obtenido tomando como referencia la imagen izquierda con aquel que se construye tomando como referencia la imagen derecha. Los puntos con disparidad diferente se consideran ocluidos.

En la Fig. 3.6 se muestra el funcionamiento de un algoritmo WTA para el cálculo del mapa de disparidad. Consiste en desplazar la ventana horizontalmente en la imagen derecha para encontrar la disparidad d que minimiza la diferencia de los píxeles encuadrados en ella (coste) respecto a la referencia de la imagen izquierda.

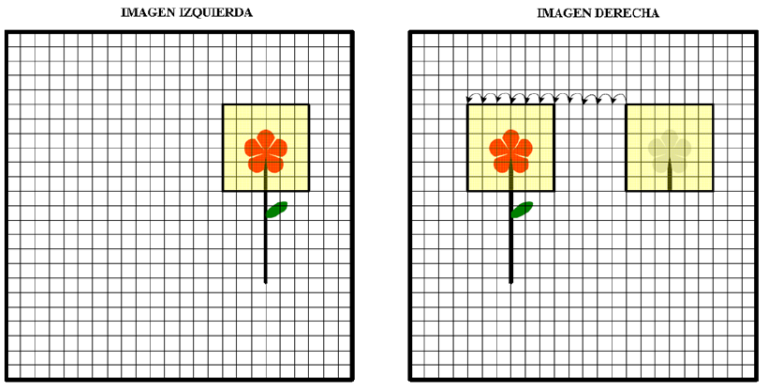


Figura 3.6. Ilustración de un proceso de obtención del mapa de disparidad.

La información tridimensional que proporciona el mapa de disparidad se puede aprovechar para construir estructuras como los u-v *disparity*.

3.2.3 U-V DISPARITY

El u-*disparity* y el v-*disparity* son dos estructuras obtenidas a partir del mapa de disparidad [41]. El v-*disparity* expresa el histograma sobre los valores de disparidad para cada fila de la imagen (coordenada v) y el u-*disparity* hace lo mismo pero, en este caso, para cada columna (coordenada u).

En resumen, el u-*disparity* se construye sumando el número de píxeles con el mismo valor de u y d, mientras que el v-*disparity* acumula los píxeles con el mismo valor de v y d. En la Fig. 3.7 se muestra un ejemplo de ambos tomada por un sistema estéreo embarcado en un vehículo.

Los u-v *disparity* permiten obtener información de gran relevancia sobre el entorno que puede ser utilizada en aplicaciones de ayuda a la conducción; especialmente en el ámbito urbano. Asi, por ejemplo, los objetos situados perpendicularmente en frente del vehículo aparecen como líneas horizontales en el u-*disparity*, mientras que, en el caso del v-*disparity*, los obstáculos aparecen como líneas verticales en las que la intensidad de los píxeles es la anchura de los obstáculos.

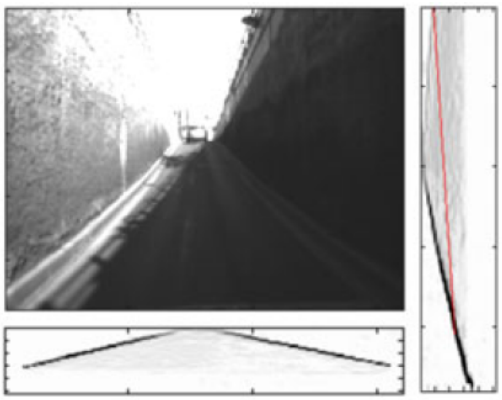


Figura 3.7. Resultados del u-v disparity en un entorno urbano.

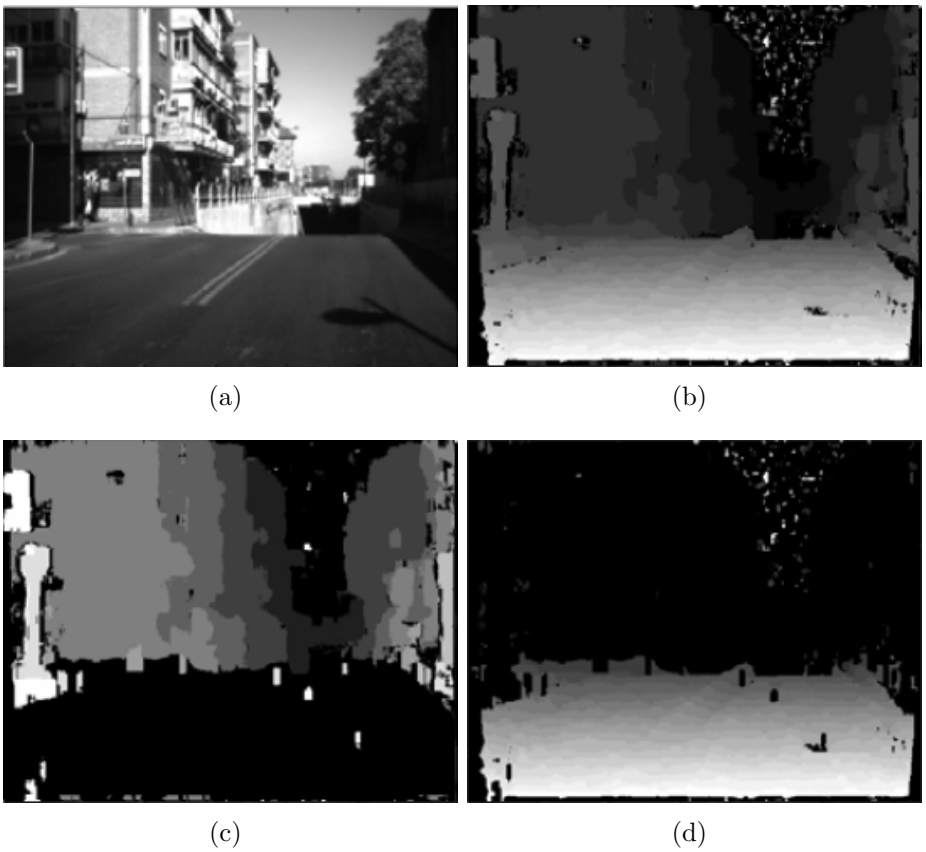
Otra característica importante es que la calzada por delante del vehículo aparece como una línea oblicua. Esto será aprovechado en el desarrollo del presente proyecto, como se verá más adelante.

A partir de la información suministrada por los u-v *disparity*, se puede segregar el mapa de disparidad en dos distintos: el mapa libre y el mapa de obstáculos, útiles en este tipo de aplicaciones.

3.2.4 Mapa de obstáculos y mapa libre

El mapa de obstáculos es un mapa de disparidad donde solo están están representados los obstáculos por delante del sistema estéreo. Por su parte, el mapa libre se obtiene eliminando del mapa de disparidad todos los píxeles pertenecientes a obstáculos, por lo que en él únicamente aparece el espacio despejado por delante de la cámara. Ambos son complementarios, y su unión da lugar al mapa de disparidad primigenio. Para su obtención, se utiliza la información relativa a los obstáculos que proporcionan los u-v *disparity*. Un ejemplo de ambos se muestra en la 3.8.

Si se calcula el v-*disparity* a partir del mapa libre, en lugar de hacerlo a partir del mapa de disparidad, se garantiza que la línea más importante que aparezca en él corresponda a la calzada, y no alguno de los obstáculos presentes por delante del vehículo.



(a) (b)

(c) (d)

Figura 3.8. Mapa de obstáculos (c) y libre (d) obtenidos a partir del mapa de disparidad (b) de una escena de tráfico urbano (a).

3.3 Detección de puntos característicos

En el capítulo 2, se resaltó la importancia que suele tener la detección, y posterior seguimiento, de puntos estáticos de referencia en los algoritmos de odometría visual. Con ese fin se emplean *detectores*, que llevan a cabo los siguientes pasos:

1. Se seleccionan *puntos de interés*, también llamados *puntos característicos*, en posiciones distintivas de la imagen, tales como esquinas, *blobs* (puntos que difieren del entorno en propiedades como el brillo o el color) o uniones en T. La propiedad más valiosa en un detector es su repetibilidad; esto

es, su capacidad para encontrar los mismos puntos de interés bajo distintas condiciones.

2. El entorno de cada punto de interés se representa por un vector de características, llamado *descriptor*. Este descriptor tiene que ser distintivo y, al mismo tiempo, robusto al ruido, errores de detección y deformaciones geométricas y fotométricas

3. Los descriptores se emparejan entre imágenes diferentes (en el caso particular de la odometría visual, izquierda y derecha de un par estéreo o pertenecientes a sucesivos momentos de tiempo). El emparejamiento se basa en la similitud entre los descriptores. El tamaño de estos tiene, por ello, un impacto directo en el tiempo que es necesario para llevar a cabo esta etapa; conviene que tengan el menor número de dimensiones posibles.

Algunos ejemplos de trabajos existentes para detectar puntos de interés son:

- Detector de esquinas de Harris [28]. Se basa en los autovalores de la matriz del segundo momento. No es invariante a la escala.

- SIFT (*Scale-Invariant Feature Transform*, transformación de características invariante a la escala) [42]. A diferencia del detector de Harris, SIFT incluye también un descriptor. Es uno de los más utilizados por ser distintivo y relativamente rápido. Además, es invariante a la escala y a la rotación. Sin embargo, las elevadas dimensiones del descriptor suponen una gran desventaja en la etapa de emparejamiento.

- SURF (*Speeded Up Robust Features*, características robustas aceleradas) [43]. Es un detector y descriptor de puntos de interés invariante a la escala y a la rotación. Sus autores afirman haber conseguido igualar, o incluso mejorar, los resultados respecto a propuestas anteriores en lo que se refiere a la repetibilidad, distintividad y robustez, requiriendo, pese a ello, mucho menos tiempo de cálculo.

Se dedicará el siguiente epígrafe a la descripción detallada de los fundamentos de este último método, que ha sido el escogido para utilizarse en el presente trabajo.

3.3.1 SURF

SURF sigue los pasos que se mencionaron anteriormente, comunes a la mayoría de los detectores. Así pues, efectúa primero una localización de los puntos de interés y, después, procede a computar los descriptores de los mismos.

Localización de puntos de interés

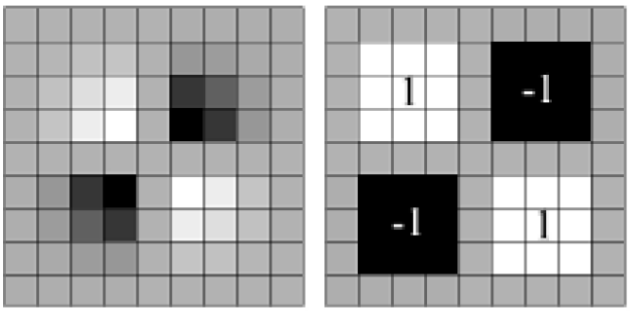
La detección de puntos de interés en SURF se basa en la matriz hessiana. Dado un punto $P = (x, y)$ en una imagen I, se define la matriz hessiana $\mathcal{H}(P, \sigma)$ en el punto P con escala σ como:

$$\mathcal{H}(P, \sigma) = \begin{bmatrix} L_{xx}(P, \sigma) & L_{xy}(P, \sigma) \\ L_{yx}(P, \sigma) & L_{yy}(P, \sigma) \end{bmatrix} \tag{3.9}$$

Donde $L_{xx}(P, \sigma)$ hace referencia a la convolución de la derivada de segundo orden de la gaussiana $g(\sigma)$ (es decir, $\frac{\delta^2}{\delta x^2} g(\sigma)$) con la imagen I en P. $L_{xy}(P, \sigma)$ y $L_{yy}(P, \sigma)$ tienen significados análogos.

A diferencia de SIFT, que aproxima la Laplaciana de la Gaussiana (LoG) por una DoG (*Difference of Gaussians*, diferencia de gaussianas), SURF aproxima las derivadas de segundo orden de la gaussiana con filtros caja. Un ejemplo se muestra en la Fig. 3.9. La convolución de las imágenes con estos filtros caja se puede calcular más rápidamente que en el caso de la gaussiana.

Los puntos de interés se seleccionan a través del determinante de la hessiana aproximada. Para ello, se eliminan todos aquellos puntos en los que este no adopte un máximo local en un entorno de $3 \times 3 \times 3$ píxeles. Finalmente, los máximos locales encontrados para el determinante se interpolan en el espacio de escala y en el de la imagen.



(a) (b)

Figura 3.9. Comparativa de la derivada de segundo orden de una gaussiana en dirección xy (a) frente a la correspondiente aproximación en forma de filtro caja (b).

Descriptor de puntos de interés

En un primer paso, SURF construye una región circular alrededor de los puntos de interés detectados, con el fin de asignarles una orientación única y conseguir así invarianza a la rotación de la imagen. La orientación se calcula usando *wavelets de Haar* en las direcciones x e y y se incluye en la información del punto de interés.

En la siguiente etapa, se construyen los descriptores SURF extrayendo las regiones cuadradas alrededor de los puntos de interés. Estas se encuentran orientadas en las direcciones asignadas por el paso anterior. Las ventanas se dividen en subregiones de 4×4 píxeles para retener cierta información espacial. En cada subregión, se extraen las *wavelets* de Haar en puntos regularmente espaciados entre si. Las respuestas de la *wavelet* en las direcciones horizontal y vertical (dx y dy) se suman en cada subregión. Además, los valores absolutos $|dx|$ y $|dy|$ también se suman para obtener información sobre la polaridad de los cambios en la intensidad de la imagen. Por lo tanto, el patrón de intensidad subyacente en cada subregión se describe por el siguiente vector:

$$\mathbf{V} = \begin{bmatrix} \Sigma d_x \\ \Sigma d_y \\ \Sigma |dx| \\ \Sigma |dy| \end{bmatrix} \tag{3.10}$$

El vector descriptor resultante para cada región 4 × 4 es de longitud 64, constituyendo el descriptor estándar SURF-64. Es posible usar regiones de 3 × 3 en su lugar, obteniendo una versión más pequeña del descriptor, SURF-36. Las *wavelets* de Haar son invariantes a cambios en la iluminación y se consigue una invarianza adicional al contraste normalizando el vector de descriptores a la longitud unidad.

Una característica importante del SURF es el rápido proceso de extracción, que aprovecha los beneficios de las imágenes integrales y del algoritmo de supresión de puntos no máximos. También es muy interesante la rápida velocidad de emparejamiento que permite, conseguida principalmente al considerar el signo de la laplaciana (la traza de la matriz hessiana) del punto de interés. El signo de la laplaciana permite distinguir la presencia de *blobs* brillantes en un fondo oscuro frente a la situación inversa. Los puntos de interés brillantes solo se emparejan con otros que también lo sean, y lo mismo sucede con los oscuros. Esta mínima información permite prácticamente duplicar la velocidad de emparejamiento y no tiene coste computacional, dado que ya ha sido calculada en el paso de localización de los puntos [44].

Un ejemplo de puntos de interés detectados en una imagen se presenta en la Fig. 3.10.

3.4 RANSAC

Es frecuente que la correspondencia de los puntos obtenidos mediante los detectores de características esté sujeta a grandes errores, que han de ser detectados y rechazados a la hora de efectuar el cálculo de la estimación del movimiento. Lo mismo sucede en otros procesos como el cálculo del perfil de

Figura 3.10. Puntos característicos detectados por SURF en una escena.

la calzada. Uno de los procedimientos que se emplean con ese fin es RANSAC (*RANdom SAmple Consensus*, consenso de muestras aleatorias) [45].

RANSAC es un método diseñado para ajustar una serie de datos obtenidos experimentalmente a un modelo conocido. Esto requiere, además de la selección del modelo adecuado, la estimación de los parámetros de dicho modelo. Un caso frecuente es el que supone el ajuste de datos a una recta, cuyos parámetros hay que estimar.

La mayor parte de las técnicas empleadas para ello, como el conocido método de los mínimos cuadrados, tratan de encontrar los parámetros del modelo que consigan el mejor ajuste de *todos* los datos de los que se dispone. Parten de la asunción de que la máxima desviación esperable de un dato respecto al modelo es proporcional al tamaño del conjunto, de tal forma que siempre habrá suficientes valores correctos como para compensar los datos más desviados.

Sin embargo, esta hipótesis no resulta siempre válida en los casos prácticos, especialmente en el ámbito de la visión por computador. Existen determinadas variaciones de estos métodos que, en sucesivas iteraciones, van eliminando los datos más alejados de los modelos obtenidos, pero el resultado obtenido sigue difiriendo, en muchas ocasiones, del deseado (Fig. 3.11).

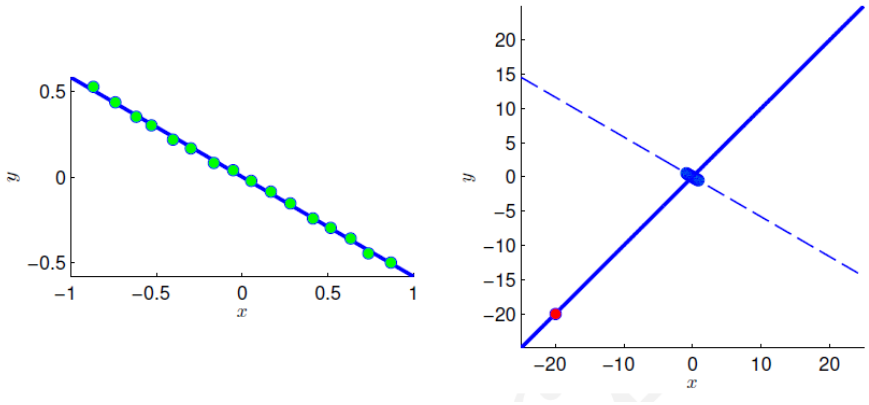


Figura 3.11. Distorsión introducida por un punto erróneo en la estimación del ajuste lineal de unos datos mediante mínimos cuadrados.

Por esta razón, Martin A. Fischler y Robert C. Bolles propusieron, en 1981, RANSAC como nuevo paradigma para el ajuste de datos a un modelo. Los autores pensaron desde el principio en la detección de puntos característicos como campo de aplicación de su método. Sin embargo, su uso se extendió rápidamente en distintas aplicaciones de visión por computador y procesamiento de imágenes, debido a la sencillez de su implementación y a su robustez.

Al contrario que los métodos tradicionalmente empleados, en los que se busca la solución inicial a partir de tantos datos como sea posible, RANSAC utiliza subconjuntos, seleccionados aleatoriamente, con el menor número de datos necesario para estimar los parámetros del modelo. A partir de esos subconjuntos, genera distintas hipótesis de los parámetros del modelo. El resto de los datos presentarán un cierto error respecto a cada una de ellas. El método escoge como la mejor solución aquella que maximiza el número de datos cuyo error está por debajo de una cierta tolerancia; esto es, el número de *inliers*. Los datos con un error mayor se denominan *outliers* y corresponden a muestras contaminadas; es decir, errores graves que tienden a alterar los resultados y no han de tenerse en cuenta (Fig. 3.12). La secuencia básica, tal cual se resume en en su versión original [45], aparece en el Algoritmo 1.

El algoritmo consiste, básicamente, en una sucesión iterativa de dos pasos: en el primero, se genera una hipótesis y en el segundo, se evalúa comparándola

Algoritmo 1 RANSAC.

Entrada: Umbral ϵ, umbral t, número máximo de iteraciones k.

1: Dado un modelo tal que se requieren n datos para estimar sus parámetros libres, y un conjunto de datos P tal que el número de elementos en P es mayor que n:
2: Seleccionar aleatoriamente un subconjunto $S1$ de n puntos de P.
3: Con $S1$, generar el modelo $M1$.
4: Usar el modelo generado $M1$ para determinar el subconjunto de $S1^*$ de puntos en P cuyo error respecto a $M1$ está dentro de una cierta tolerancia ϵ.
5: **si** el número de elementos en $S1^*$ es mayor que un umbral t **entonces:**
6: Usar $S1^*$ para generar un nuevo modelo $M1^*$ (usando mínimos cuadrados, por ejemplo)
7: **devolver** $M1^*$
8: **si no**
9: **si** el número de iteraciones es menor que el máximo fijado, k **entonces:**
10: Seleccionar aleatoriamente un nuevo subconjunto $S2$ y repetir desde 3.
11: **si no**
12: Resolver el modelo con el subconjunto más grande de los encontrados, o terminar en fallo.
13: **fin si**
14: **fin si**

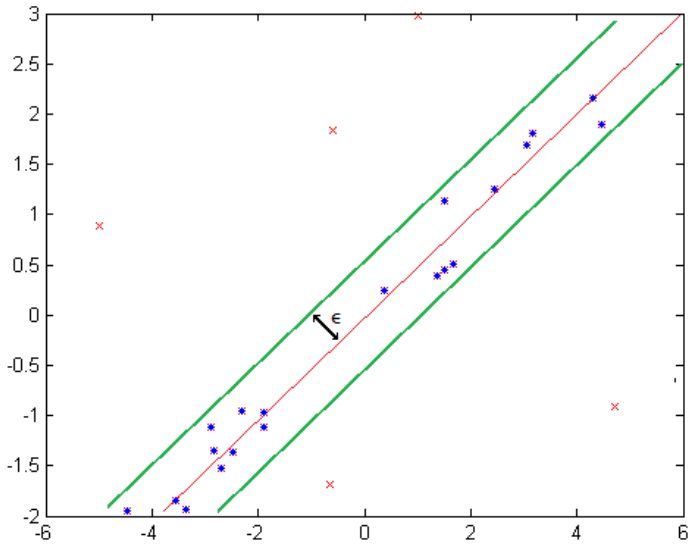


Figura 3.12. *Inliers* (puntos en azul) y *outliers* (cruces rojas) en una aplicación de RANSAC para el ajuste de datos a una recta.

con los datos. Los aspectos a tener en cuenta en cada uno de ellos son los siguientes:

Generación de hipótesis

El número de iteraciones, k, ha de escogerse lo suficientemente alto como para garantizar con una probabilidad z elevada que al menos uno de los subconjuntos elegidos aleatoriamente está libre de errores. Si n es el mínimo número de datos para generar los parámetros del modelo y w la probabilidad de que un dato presente un error respecto al modelo menor que la tolerancia, ϵ, se cumple:

$$k = \frac{\log(1-z)}{\log(1-w^n)} \tag{3.11}$$

No obstante, hay que tener en cuenta que, en muchas aplicaciones, el valor de la probabilidad w es desconocido y es necesario estimar su valor o, directamente, el valor de k.

Evaluación de hipótesis

El caso de la tolerancia ϵ, que divide los datos en *inliers* y *outliers*, es similar al de k: en muchas ocasiones, su valor ha de obtenerse experimentalmente, al no ser posible una determinación analítica de la misma.

Es importante resaltar que RANSAC puede ser visto como un algoritmo de optimización que busca el modelo $\mathcal{M}$ que minimiza la función de coste [46]:

$$C_{\mathcal{M}}(D;\boldsymbol{\theta}) = \sum_{\boldsymbol{d}\in D} \rho(\boldsymbol{d}, \mathcal{M}(\boldsymbol{\theta})) \tag{3.12}$$

En esta ecuación, D representa el conjunto total de datos de los que se dispone, $\boldsymbol{\theta}$ el vector con los parámetros del modelo $\mathcal{M}$, $\boldsymbol{d}$ cada uno de los conjuntos mínimos de datos extraído de D y ρ es la *función de pérdida*:

$$\rho(\boldsymbol{d}, \mathcal{M}(\boldsymbol{\theta})) = \begin{cases} 0 & \text{si } |e_{\mathcal{M}}(\boldsymbol{d},\boldsymbol{\theta})| \leq \epsilon \\ 1 & \text{en otro caso} \end{cases} \tag{3.13}$$

A su vez, $e_{\mathcal{M}}(\boldsymbol{d},\boldsymbol{\theta})$ es una función que proporciona el error entre el dato $\boldsymbol{d}$ y el modelo con parámetros $\boldsymbol{\theta}$; por ejemplo, la distancia geométrica.

Variaciones de RANSAC

El planteamiento inicial de RANSAC ha dado lugar a múltiples variaciones a lo largo del tiempo, principalmente para mejorar la velocidad del algoritmo, la robustez y precisión de la solución estimada y para reducir la dependencia de esta de las distintas constantes elegidas por el usuario (ϵ, k, etc.) [47].

En concreto, RANSAC puede ser muy sensible a la elección del umbral ϵ: si es demasiado grande, todas las hipótesis tienden a ser evaluadas como correctas, y si es demasiado pequeño, los parámetros estimados tienden a ser inestables. Para compensar este efecto, Torr et ál. desarrollaron MSAC (*M-estimator SAmple Consensus*, consenso de muestras por estimador M) y MLESAC (*Maximum LikElihood SAmple Consensus*, consenso de muestras por máxima verosimilitud) [48]. La idea en la que se basan es la de evaluar la calidad del conjunto de consenso (es decir, los datos que se ajustan al modelo)

calculando su probabilidad (en el modelo original, su calidad únicamente se evaluaba según el número de elementos presentes en ellos).

Existe una extensión de MLESAC que tiene en cuenta las probabilidades previas asociadas a los datos de entrada, llamada *Guided*-MLESAC [49]. Otra propuesta similar es PROSAC (*PROgressive SAmple Consensus*, consenso de muestras progresivo) [50], que propone guiar el procedimiento de muestreo si se tiene algún conocimiento a priori de la información.

De los mismos autores es R-RANSAC (*Randomized* RANSAC) [51], que reduce la carga computacional al evaluar inicialmente la bondad de los modelos escogidos empleando solo un conjunto reducido de puntos, en lugar de todos ellos. Esta estrategia de prevención obtendrá mejores resultados cuanto mayor sea el porcentaje de *inliers*. En este sentido, D. Nistér propuso un RANSAC preventivo [52] pensado para obtener estimaciones robustas de movimiento en sus trabajos de odometria visual [33]. La idea básica consiste en generar un número fijo de hipótesis de manera que la comparación se establece respecto a la calidad de las hipótesis generadas y no sobre una medida absoluta.

Otros investigadores trataron de enfrentarse a situaciones difíciles, con ruido desconocido o perteneciente a múltiples modelos. Es el caso de *J-Linkage* [53], que no requiere especificar de manera manual ningún parámetro.

4 Herramientas

En el desarrollo del proyecto se han utilizado múltiples herramientas para implementar el algoritmo de odometría visual en el sistema *hardware* existente. En líneas generales, se ha programado en lenguaje C++ mediante Visual C++ 2008 [54], con ayuda de las bibliotecas MIL [55] y OpenCV [56]. Para paralelizar los procesos, se utilizan las utilidades de programación concurrente existentes en Windows, así como la plataforma CUDA, que se describe a continuación.

4.1 CUDA

4.1.1 Unidades de procesamiento gráfico GPU

Las GPU (*Graphics Processing Unit*, unidad de procesamiento gráfico) son circuitos electrónicos especializados en manipular y modificar rápidamente la memoria con el fin de acelerar la construcción de imágenes dentro de un *buffer* pensado, generalmente, para su posterior visualización en una pantalla. En la actualidad, las GPU forman parte de sistemas tan dispares como computadores empotrados, teléfonos móviles, ordenadores personales, estaciones de trabajo o consolas de videojuegos. En la Fig. 4.1 se muestra un ejemplo de tarjeta gráfica diseñada para ser insertada en un bus PCI Express de un ordenador de sobremesa.

Figura 4.1. Tarjeta gráfica Quadro FX 380 de NVIDIA.

Las GPU están construidas sobre una filosofía radicalmente distinta a la de las CPU. Como se muestra en la Fig. 4.2, las GPU destinan más transistores al procesamiento de datos, mientras que se reduce la lógica de control, por lo que solo se permite un número limitado de instrucciones de control de flujo.

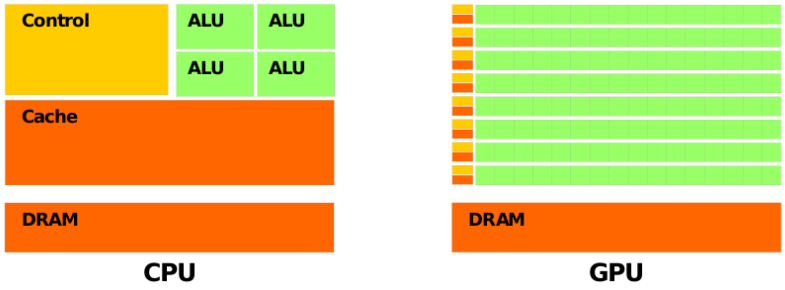


Figura 4.2. Comparativa entre las estructuras de una CPU y de una GPU.

En general, las CPU están diseñadas para una gran variedad de aplicaciones y para proporcionar tiempos de respuesta rápida ante una única tarea. Por ello, resultan interesantes para ejecutar programas cuyas instrucciones están destinadas a llevarse a cabo secuencialmente.

En cambio, las GPU están construidas específicamente para procesos gráficos que suponen un alto grado de paralelismo de datos, al poderse procesar cada píxel de una imagen independientemente. Implementan la

Ventajas	Inconvenientes
Rápido procesamiento en paralelo	Excesiva especialización
Precio por capacidad de procesamiento	Cambio en la mentalidad para programar
Bajo consumo energético	Latencia en la comunicación con la CPU
Alto ritmo de desarrollo (Fig. 4.3)	Obsolescencia rápida

Tabla 4.1. Ventajas e inconvenientes de las GPU.

arquitectura SIMD comentada en la sección 1.4.2. Las ventajas y desventajas de estos dispositivos se presentan en la Tabla 4.1.

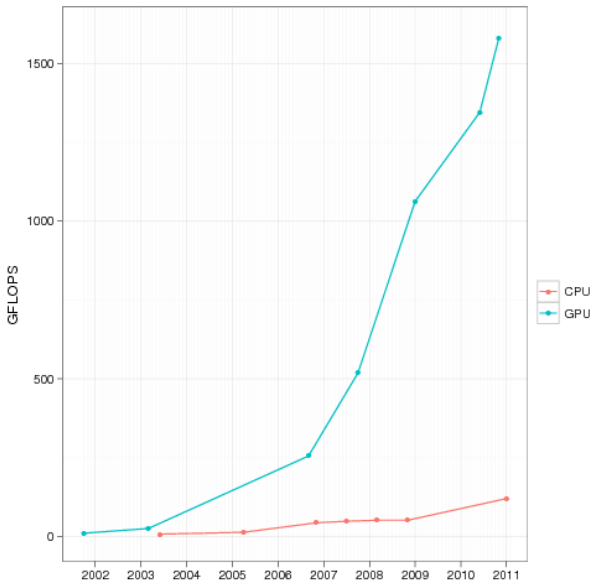


Figura 4.3. Evolución en la tasa de GFLOPS (*Giga Floating Point Operations Per Second*, operaciones de coma flotante por segundo) capaces de ser procesados por las CPU (rojo) y las GPU (azul).

Debido a estas ventajas, las GPU están siendo cada vez más utilizadas como procesadores SIMD de ámbito general, para aplicaciones que son distintas de las puramente gráficas, pero que también pueden desarrollarse paralelamente en varios hilos de ejecución. Es lo que se ha dado en llamar GPGPU (*General-Purpose computing on Graphics Processing Units*, computación de propósito general en unidades de procesamiento gráfico). Gracias a esto, multitud de procesos relacionados con la ciencia y la ingeniería están consiguiendo mejoras de hasta dos órdenes de magnitud en los tiempos de cómputo [57].

Actualmente, el mercado de las GPU está copado por dos compañías: AMD y NVIDIA. Esta última lanzó, en noviembre de 2006, CUDA (*Compute Unified Device Architecture*, arquitectura unificada de dispositivos de cómputo) para facilitar el desarrollo de este tipo de aplicaciones.

4.1.2 Conceptos generales

CUDA [58] es una plataforma de computación paralela y un modelo de programación desarrollados por NVIDIA. Está diseñado para conseguir grandes incrementos en el rendimiento de cómputo aprovechando la potencia de las GPU de esta marca.

Se caracteriza por:

- Poderse utilizar a través de extensiones mínimas de C/C++, de forma que resulta familiar para los programadores que se enfrentan a él por primera vez. Esto permite centrarse en los algoritmos de paralelización, en lugar de tener que dedicar tiempo a aprender el lenguaje.
- Implementar un modelo de programación heterogéneo serie-paralelo, basado en la interacción CPU-GPU. Esta interacción también funciona para CPU multinúcleo.

4.1.3 Modelo de programación

CUDA se basa en la programación de *kernels* destinados a procesarse en muchos hilos de ejecución paralelos (*threads*) en la GPU, que constituye lo que, en CUDA, recibe el nombre de *device*. Todos los hilos del *device* ejecutan el mismo *kernel* a la vez. Los *kernels*, por su parte, están constituidos por una secuencia de instrucciones.

La ejecución de los *kernels* es ordenada por un hilo de la CPU; *host*, en CUDA. Un esquema de esta filosofía heterogénea de programación se muestra en la Fig. 4.4, siendo cada una de las flechas blancas un hilo de ejecución.

La única diferencia entre los distintos hilos de ejecución es su identificación propia; esto es, a cada hilo la corresponde un `threadID` distinto.

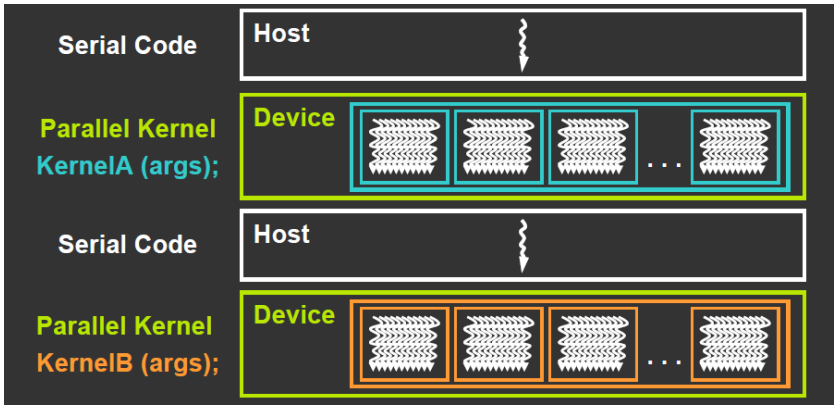


Figura 4.4. Filosofía heterogénea de programación en CUDA.

Los *threads* se agrupan en *bloques*, que se pueden sincronizar mediante barreras como la instrucción `__syncthreads()`. Cada bloque consta, igualmente, de un `blockID` único.

Los bloques de *threads* no se pueden sincronizar entre si, de manera que, en la práctica, la GPU puede decidir cuáles de ellos se ejecutan realmente de manera concurrente. Esta independencia proporciona escalabilidad, dado que permite la ejecución de los *kernels* en tarjetas gráficas con distinto número de núcleos.

Distintos bloques componen el *grid*. A cada *kernel* se le puede asignar una configuración de *grid* y de bloque distinta. Es decir, el numero de *threads* por bloque y el número de bloques totales en el *grid* puede ser modificado, y además, ambos pueden ser distribuidos de forma distinta de acuerdo a las tres coordenadas tridimensionales x, y, z. Esto último afecta, fundamentalmente, a los `threadID` y a los `blockID`.

4.1.4 Jerarquía de memoria

CUDA consta de varios espacios de memoria con ámbitos de aplicación diferenciados. De acuerdo a estos, se tiene:

- Cada ***thread*** tiene acceso a sus propios registros y a una memoria local (Fig. 4.5).

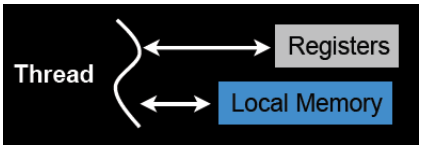


Figura 4.5. Memoria por *thread*.

- Los *threads* pertenecientes a un mismo **bloque** acceden a una memoria compartida, de tamaño definido por el usuario (Fig. 4.6).

Figura 4.6. Memoria por bloque.

- Todos los *threads* del **dispositivo** tienen acceso a una misma memoria global, más lenta que la anterior (Fig. 4.7).

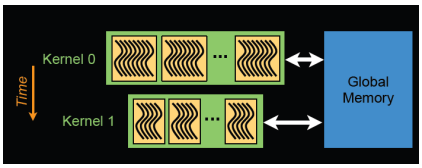


Figura 4.7. Memoria por *device*.

La disposición física de estos espacios de memoria se muestra en la Fig. 4.8. Además, hay dos memorias de solo lectura a la que pueden acceder todos los *threads*: la memoria de constantes y la memoria de texturas. El *host* puede interaccionar con ellas y con la memoria global.

Es necesario proporcionar a los *kernel* los argumentos sobre los que van a trabajar. Asimismo, los resultados obtenidos han de pasar a formar parte del hilo de ejecución principal del programa en la CPU. Esto requiere un proceso de intercambio de memoria entre *host* y *device*, de acuerdo al diagrama mostrado en la Fig. 4.9. La copia de memoria entre ambos se lleva a cabo mediante la instrucción `cudaMemcpy()`.

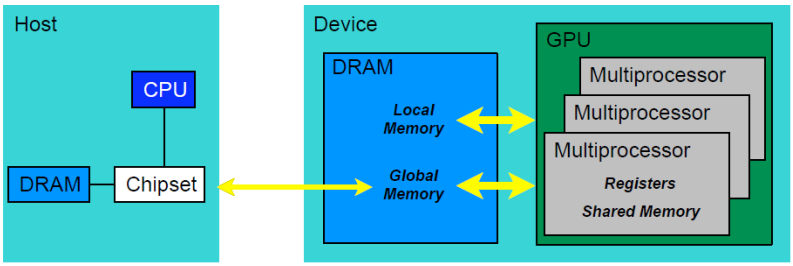


Figura 4.8. Resumen de los espacios de memoria en CUDA.

4.1.5 Implementación en la GPU

El *device* es un conjunto de multiprocesadores, cada uno de los cuales es un conjunto de procesadores de 32 bits SIMD. En cada ciclo de reloj, el multiprocesador ejecuta la misma instrucción sobre un grupo de *threads* llamado *warp*. El número de *threads* en un *warp* es el *tamaño del warp*.

Cada multiprocesador procesa conjuntos de bloques uno tras otro. Los bloques de *threads* que están siendo procesados en uno de estos conjuntos se llaman *activos*.

4.1.6 Estrategias de optimización

Para conseguir el mejor rendimiento, NVIDIA recomienda:

- **Maximizar la ejecución en paralelo**, lo que conlleva estructurar los algoritmos de una forma que permita el máximo paralelismo posible. Además, hay que elegir cuidadosamente la configuración del *grid* en cada llamada a los *kernel*; por ejemplo, conviene que el número de threads por bloque sea múltiplo de 32, al proporcionar mejor eficiencia computacional.
- **Optimizar el uso de memoria para conseguir el máximo ancho de banda**. Entre otras cosas, conviene reducir las transferencias de datos entre el *host* y el *device*, que tienen mucho menos ancho de banda que las transferencias de datos internas en el *device*. También hay que evitar los accesos del *kernel* a la memoria global.

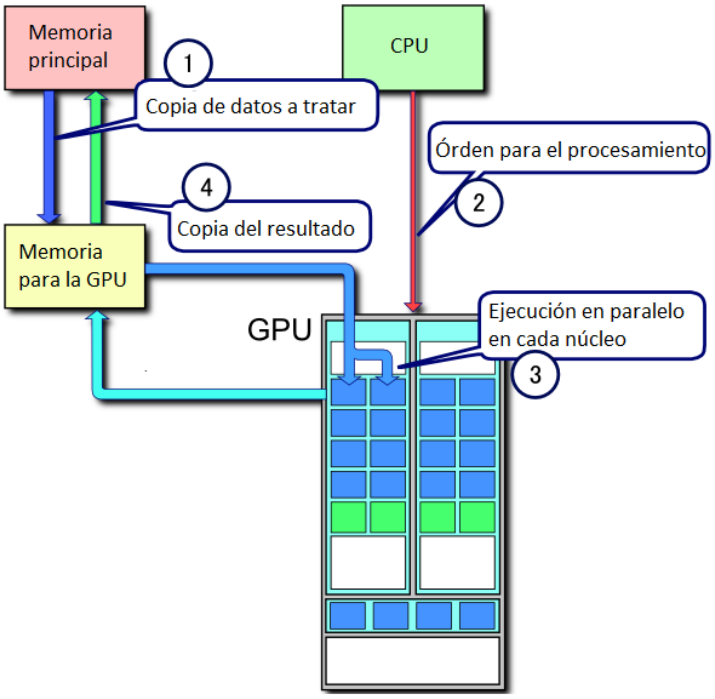


Figura 4.9. Flujo de datos en CUDA.

- **Optimizar el uso de instrucciones para conseguir el máximo rendimiento**. Se recomienda sacrificar la precisión en aras de lograr una mayor velocidad siempre que eso sea posible (p. ej., usar precisión simple en vez de doble). Además, hay que tener cuidado con las instrucciones de flujo de control dentro del *kernel*: no conviene crear múltiples caminos de ejecución en el mismo *warp* pues podría provocar grandes cambios en el tiempo de procesamiento de los diferentes hilos.

4.2 OpenCV

OpenCV [56] es una biblioteca de funciones de programación principalmente orientada a la visión por computador en tiempo real. Tiene licencia BSD (*Berkeley Software Distribution*, distribución de *software* de Berkeley), por lo que es gratis para usos comerciales o de investigación. OpenCV fue originalmente escrita en C, pero ahora tiene una interfaz C++ completa y

todos los nuevos desarrollos son en C++. Hay también una interfaz completa de Python de la biblioteca.

OpenCV fue diseñada para conseguir eficiencia computacional y centrada en aplicaciones de tiempo real [59], pudiendo ejecutarse en procesadores multinúcleo e, incluso, en sus últimas versiones, sobre unidades GPU mediante CUDA.

Las aplicaciones de las OpenCV abarcan campos tan variados como: HCI (*Human-Computer Interaction*, interacción hombre-máquina), identificación de objetos, segmentación y reconocimiento, reconocimiento de caras, reconocimiento de gestos, seguimiento del movimiento, comprensión del movimiento, calibración de sistemas estéreo y multi-cámara y computación de la profundidad, robots móviles y, por supuesto, *ego-motion* u odometria visual.

En la Fig. 4.10 se muestran algunas de las herramientas que es posible encontrar entre las más de 500 funciones de la biblioteca, entre las cuales se halla la que se usará en este trabajo: la detección de características.

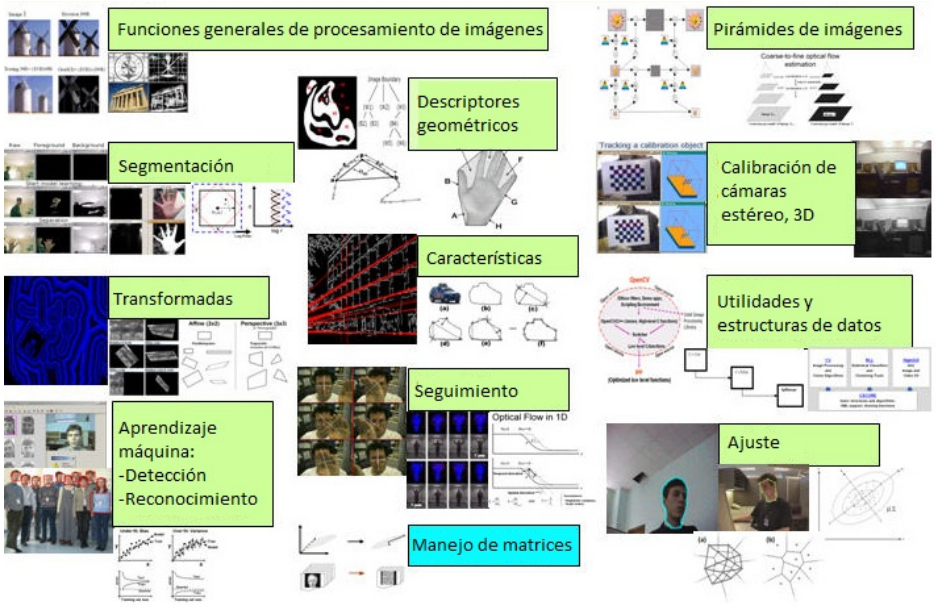


Figura 4.10. Algunas de las herramientas que proporcionan las OpenCV.

5 Desarrollo del Proyecto

Como ya se ha mencionado, en este proyecto se pretende implementar un sistema de localización mediante odometria visual para el vehículo inteligente IVVI 2.0 (sección 1.2.2), empleando el sistema estéreo montado en el vehículo [60]. Para ello, se seguirán las líneas generales planteadas en otros trabajos, como los mostrados en la sección 2.3 (fijación de puntos de referencia, seguimiento de los mismos en varios *frames*, etc.), con ciertas aportaciones para mejorar el tiempo de cómputo y la calidad de los resultados.

Tal y como se muestra en la Fig. 5.1, los procesos que es necesario aplicar sobre cada par de imágenes capturado son:

1. **Cálculo del mapa de disparidad** (sección 3.2). A partir de él, se obtiene el v-*disparity* (sección 3.2.3), que será fundamental en los siguientes procesos.

2. **Estimación del perfil de la calzada**. El v-*disparity* permite conocer la posición de la calzada respecto a la cámara y mejorar la estimación del movimiento. Para descartar los datos erróneos, se empleará RANSAC (sección 3.4).

3. **Detección de puntos característicos**. En esta etapa, se toman puntos estáticos de referencia a partir de los cuales determinar el movimiento. En este trabajo, estos puntos pertenecen únicamente a la calzada. Se

utilizará SURF (sección 3.3.1), en su implementación de las OpenCV (sección 4.2), para detectarlos y emparejarlos.

4. **Cálculo de la estimación del movimiento**. La determinación del movimiento requiere de cálculos geométricos que permitan la conversión entre el desplazamiento de los puntos de referencia y el del vehículo. Además, es necesario rechazar los datos erróneos obtenidos durante el emparejamiento, para lo cual se volverá a utilizar RANSAC (sección 3.4).

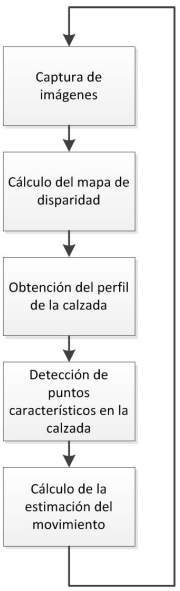


Figura 5.1. Visión general del algoritmo implementado.

La implementación del algoritmo se llevará a cabo mediante su programación en C++, con el fin de ser ejecutado en los sistemas *hardware* del IVVI 2.0. Además, el manejo de imágenes distinto de la detección de puntos característicos se lleva a cabo con las bibliotecas MIL (*Matrox Imaging Library*, biblioteca de imagenes Matrox) [55].

Debido a la alta carga computacional que supone el proceso, se ha intentado paralelizar las distintas etapas tanto como fuese posible. En el caso de los pasos 2 y 4, se ha optado por utilizar la arquitectura CUDA (sección 4.1) para ello. Esto ha supuesto el desarrollo de múltiples versiones que intentan alcanzar, sucesivamente, un grado de paralelismo mayor. El esquema general

de las mismas se muestra en la Fig. 5.2, estando resaltadas en gris las versiones que se basan en el procesamiento paralelo.

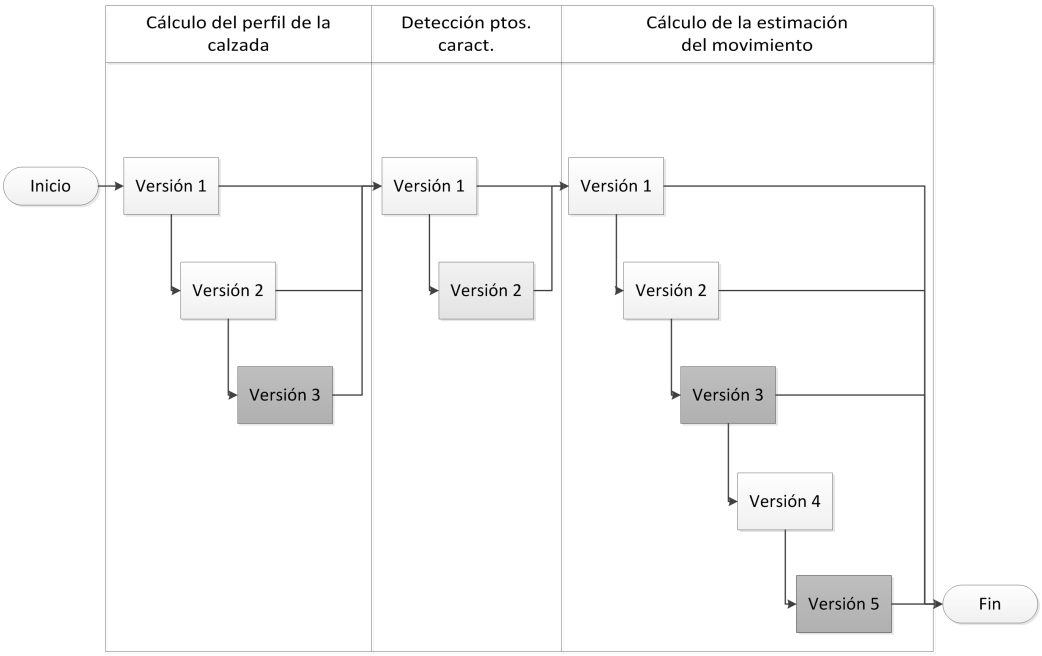


Figura 5.2. Esquema general de versiones para cada una de las etapas del algoritmo. En gris oscuro, las implementadas con CUDA y en gris claro, las que conllevan paralelización entre procesos.

5.1 Cálculo del mapa de disparidad y del u-v disparity

Para la obtención del mapa denso de disparidad y las u-v *disparity*, se ha recurrido a los algoritmos descritos en [61] y [62]. De forma muy sucinta, en esta fase inicial se llevan a cabo los procesos citados a continuación.

Filtrado del par de imágenes

Antes de calcular el mapa de disparidad, con el fin de minimizar las diferencias de ganancia de ambas cámaras del sistema estéreo, se elimina el ruido y se detectan los bordes de las dos imágenes obtenidas con la cámara

estéreo. Para ello, se emplea la laplaciana de la gausiana (LoG). Este paso resulta especialmente importante porque en las etapas posteriores se asumirá que un mismo punto presenta el mismo valor de gris en ambas imágenes del par.

Para su implementación, se lanza un *thread* por píxel, encargado de realizar todas las operaciones necesarias para obtener el valor final que adoptará dicho píxel

Cálculo del mapa de disparidad

Las características del algoritmo empleado para el cálculo del mapa de disparidad, en cada una de las etapas correspondientes a la taxonomía de Scharstein y Szeliski [40], son:

1. **Cálculo de la función de coste**. Aunque hay funciones de coste más precisas, se utiliza SD (*Squared intensity Differences*, diferencias cuadráticas de intensidad) por su rapidez y sencillez de implementación en la GPU.

2. **Agregación del coste**. Se utilizan ventanas cuadradas para la agregación de coste por tener mejor rendimiento en la GPU y proporcionar unos resultados lo suficientemente precisos.

3. **Cálculo de la disparidad**. Se ha optado por el método local WTA. El procedimiento empleado permite la obtención simultánea del mapa de disparidad de la imagen izquierda con respecto a la derecha y del de la imagen derecha con respecto a la izquierda.

4. **Post-procesamiento**. En este paso, se tratan de minimizar los errores en el mapa de disparidad, que suelen producirse en áreas donde no existe textura (como la calzada), cerca de fronteras de discontinuidades o en zonas en las que hay patrones repetidos. Para ello, se efectúa un *cross-checking* entre los dos mapas de disparidad calculados.

La implementación del algoritmo en CUDA se basa en hacer que cada *thread* calcule y agregue el coste correspondiente a los píxeles de una columna

de ellos. Por otro lado, el *cross-checking* se ejecuta lanzando un *thread* por píxel.

Obtención del v-*disparity*

El v-*disparity* se calcula a partir del mapa libre. Para ello, la secuencia de pasos que se efectúan es:

1. **Cálculo del u-*disparity***. El u-*disparity* está formado por los histogramas de cada columna del mapa de disparidad. Su ejecución en la GPU se lleva a cabo de forma que cada *thread* se encarga de una columna. Posteriormente, ese *thread* umbraliza toda su columna con un valor umbral de 25, escogido para una óptima detección de los obstáculos.

2. **Cálculo del mapa de obstáculos**. El cálculo del mapa de obstáculos se basa en el u-*disparity* umbralizado que se ha obtenido en la etapa anterior. De nuevo, cada *thread* se encarga de computar una columna distinta.

3. **Cálculo del mapa libre**. El mapa libre se construye eliminando del mapa de disparidad los píxeles que tienen valor distinto de cero en el mapa de obstáculos. La tarea es llevada a cabo mediante un *thread* por píxel.

4. **Cálculo del v-*disparity***. Finalmente, se puede calcular el v-*disparity* del mapa libre. De ello se encarga un *thread* por cada fila del mapa de disparidad, encargado de realizar el histograma de la misma.

5.2 Estimación y uso del perfil de la calzada

Como se mencionó en la sección 3.2.3, el v-*disparity* representa el histograma sobre los valores de disparidad de cada fila. Su utilidad radica, habitualmente, en que permite detectar los obstáculos situados en frente del

vehículo, que aparecen como líneas verticales en sus correspondientes valores de disparidad [63]. Sin embargo, en este caso, el v-*disparity* va a ser utilizado con el fin contrario: obtener información sobre la calzada por la que circula el vehículo; es decir, aquello que *no es* un obstáculo. Este es el motivo por el cual el v-*disparity* se ha procesado a partir del mapa libre, en lugar de obtenerse directamente del mapa de disparidad: en el v-*disparity* que se emplea en este caso no aparecen los obstáculos, únicamente la información referida a la calzada.

Si se considera la hipótesis de que el suelo es plano, la calzada aparece en el v-*disparity* como una línea recta oblicua (5.1), que recibe el nombre de *perfil de la calzada*, *road profile* en inglés [64].

$$v = m \cdot d + b \tag{5.1}$$

Esta recta tiene una pendiente m y una ordenada en el origen b, que es el valor teórico del horizonte del perfil de la calzada. Relaciona el valor de la coordenada vertical en la imagen de los puntos de la calzada (v) con su valor de disparidad (d), el cual, por supuesto, viene dado por la distancia a la que estos se encuentran del sistema estéreo (Z''), como expresa la ecuación 5.2.

$$Z'' = \frac{f \cdot B}{u_L - u_R} = \frac{f \cdot B}{d} \tag{5.2}$$

Siendo d la disparidad, f la distancia focal y B la separación entre las dos cámaras o *baseline*. Si se conoce la coordenada vertical del centro óptico (C_v), la estimación del perfil de la calzada permite conocer el ángulo *pitch* (α) que forma el sistema estéreo con la calzada (Fig. 5.3), a través de 5.3.

$$\alpha = \arctan \frac{b - C_v}{f} \tag{5.3}$$

Uniendo las ecuaciones 5.1 y 5.2, se obtiene la relación entre la profundidad de los puntos pertenecientes a la calzada (Z'') y su coordenada vertical en la imagen (5.4).

$$Z'' = \frac{m \cdot f \cdot B}{v - b} \tag{5.4}$$

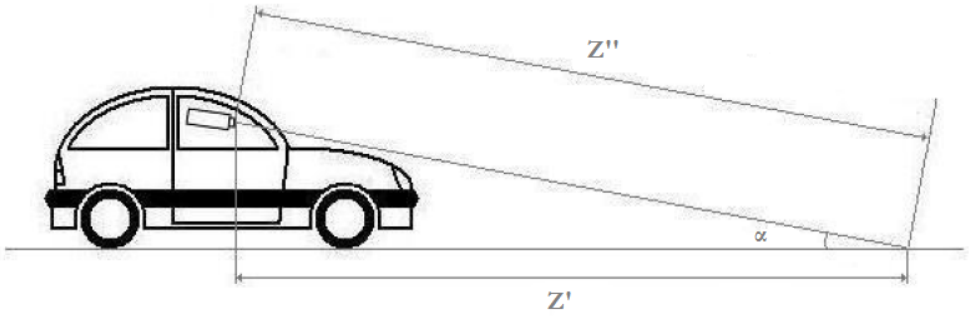


Figura 5.3. Diferencia entre las dos medidas de profundidad, Z'' y Z'.

Hay que tener en cuenta que Z'' designa la profundidad medida desde la cámara. Sin embargo, la distancia que resulta interesante conocer es la profundidad de los puntos respecto al vehículo, Z', tal y como se muestra en la Fig. 5.3. Para ello, se introduce la información del *pitch* (α) dando lugar a 5.5.

$$Z' = \frac{m \cdot f \cdot B}{v - b} \cdot \cos\alpha \tag{5.5}$$

Esta nueva expresión, al depender de v en lugar de hacerlo de d, supone un aumento significativo de la resolución, como se aprecia en la Fig. 5.4. Hay que tener en cuenta que la disparidad d puede tomar valores entre 0 y 30, mientras que la coordenada v del perfil de la calzada tiene variabilidades del orden de la mitad de la altura de las imágenes estéreo; en este caso, 240 píxeles.

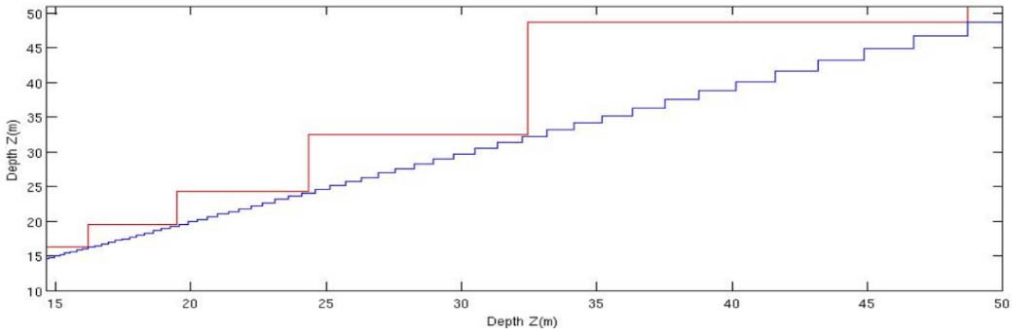


Figura 5.4. Comparativa de la resolución obtenida en el cálculo de la profundidad usando el perfil de la calzada (línea inferior) frente a la que proporciona el empleo de los valores de la disparidad (línea superior).

Si se denota por C_u la coordenada horizontal del centro óptico, la coordenada X' (paralela a la línea epipolar del sistema estéreo) de un punto de la calzada se obtiene a partir de sus coordenadas en la imagen (u, v) mediante 5.6.

$$X' = \frac{Z' \cdot (u - C_u)}{f} = \frac{m \cdot B \cdot (u - C_u)}{v - b} \tag{5.6}$$

Para mejorar el cálculo de las coordenadas en el mundo, se puede emplear el dato de calibración de la cámara que proporciona la desviación del ángulo *yaw* (ϕ). Se obtienen, entonces, las coordenadas (X, Z) (Fig. 5.5) dadas por 5.7.

$$\begin{bmatrix} X \\ Z \end{bmatrix} = \begin{bmatrix} \cos\phi & -\operatorname{sen}\phi \\ \operatorname{sen}\phi & \cos\phi \end{bmatrix} \begin{bmatrix} X' \\ Z' \end{bmatrix} \tag{5.7}$$

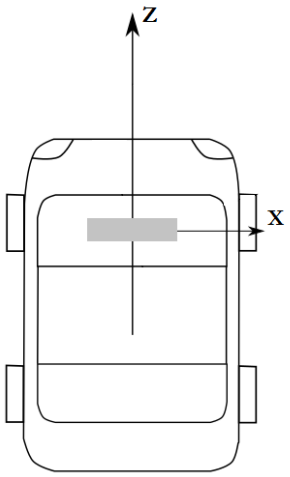


Figura 5.5. Ubicación de las coordenadas X y Z respecto al sistema estéreo montado en el vehículo.

5.2.1 Implementación del algoritmo

A la vista de los resultados anteriores, resulta evidente la necesidad de obtener los parámetros de la recta que define el perfil de la calzada, dado que se requiere obtener las coordenadas en el mundo de los puntos de la calzada.

Se han desarrollado tres versiones distintas del algoritmo que lleva a cabo esta tarea. La primera, de la que se partía, utiliza la transformada de Hough

para buscar la línea recta en el v-*disparity*, una vez binarizado este. Las dos siguientes, en cambio, se basan en el enfoque RANSAC, con el fin de conseguir mejores tiempos de procesamiento sin que ello conlleve una gran pérdida de robustez ante los probables *outliers* que aparecerán en el v-*dispariry* como consecuencia de los errores mencionados anteriormente.

Estas dos últimas versiones difieren entre si en que la primera se ejecuta en la CPU, mientras que la segunda lo hace en la GPU, con la intención de aprovechar la capacidad de paralelización de la misma en aras de conseguir mejores tiempos de cómputo.

En todos los casos, como ya se ha mencionado, se obtienen los parámetros de la recta que define el perfil de la calzada, llevándose a cabo posteriormente, en los tres casos, el cálculo del *pitch* (α) mediante 5.3.

Versión 1

La transformada de Hough es un método que busca formas geométricas de un tipo determinado en una imagen, encontrando los parámetros de aquella que contenga más puntos en la imagen. Su aplicación a este algoritmo se basa en binarizar la imagen y hacer que cada punto a nivel alto *vote* por una familia de rectas, obteniendo, en último término, la recta más votada. Cabe destacar que, en este método, las rectas se modelizan con dos parámetros llamados ρ y θ, de acuerdo a la ecuación 5.8.

$$x \cos\theta + y \operatorname{sen}\theta = \rho \tag{5.8}$$

En esta primera versión, se hace uso de las bibliotecas OpenCV para binarizar la imagen y llevar a cabo, posteriormente, la transformada de Hough. En concreto, se utiliza la transformada de Hough probabilística, que es más eficiente para casos en que la imagen contiene pocos segmentos lineales, pero muy largos.

La llamada a la función encargada de binarizar se realiza con:

```
cvThreshold(v,v,200,255,CV_THRESH_BINARY);
```

Estando la imagen almacenada en `v`. Con esto, se ponen a nivel alto todos los píxeles cuyo nivel de gris sea superior a 200. Posteriormente, se aplica la transformada de Hough a través de:

```
cvHoughLines2(v,datos,CV_HOUGH_PROBABILISTIC,1,CV_PI/7200,50,75,1500);
```

Habiéndose reservado anteriormente la estructura de almacenamiento de datos `CvMemStorage datos`. De esta manera se aplica la transformada de Hough probabilistica con una resolución para ρ de 1 píxel y para θ de $\pi/7200$ radianes. Además, se especifica que la línea más votada tiene que haber recibido, al menos, 50 votos.

Mediante este procedimiento se obtienen varias rectas; la más importante de ellas es la que corresponde al perfil de la calzada.

Versión 2

En este caso, y en el posterior, el v-*disparity*, a partir del que se obtiene la información, no recibe ningún tratamiento (umbralización, ecualización,...) previo. Sus píxeles presentan niveles de gris que oscilan entre 0 y 255, puesto que, durante su construcción, se limita el valor máximo a este último; es decir, si un valor de disparidad aparece un número de veces igual o superior a 255 en una fila del mapa libre, el punto del v-*disparity* correspondiente a esa fila y a ese valor de disparidad tendrá valor 255.

El primer paso, en esta versión, para la obtención del perfil de la calzada consiste en obtener las coordenadas de los puntos que superan un cierto umbral en el v-*disparity*. Estos puntos son, fundamentalmente, los correspondientes a la calzada, salvo errores como los ya mencionados. Para obtener estas coordenadas, se recorre el v-*disparity* completamente, tal y como se muestra en el Algoritmo 2.

Una vez obtenidos los puntos, hay que buscar la recta que mejor se ajusta a ellos. Como ya se ha mencionado, el procedimiento utilizado para ello está basado en la filosofía RANSAC. Como se explicó en la sección 3.4, se trata de una técnica de ajuste de datos a un modelo especialmente útil en aquellos casos en los que esté presente un cierto número errores importantes

Algoritmo 2 Algoritmo para la obtención de los puntos a nivel alto en el v-*disparity*

Parámetro(s): `RP_UMBRAL`

```
1: para y recorriendo todas las filas hacer:
2:   para x recorriendo todas las columnas hacer:
3:     si el nivel de gris de (x, y) >RP_UMBRAL entonces:
4:       Almacenar coordenadas (x, y) en un array.
5:       Incrementar contador.
6:     fin si
7:   fin para
8: fin para
```

en los datos, como es probable que suceda en este caso con los puntos en el v-*disparity*.

En primer lugar, se comprueba si, efectivamente, se han hallado puntos en el v-*disparity* que superen el umbral. En caso negativo, no es posible calcular el perfil.

Durante un número preestablecido de iteraciones, se seleccionan aleatoriamente dos puntos distintos de entre todos aquellos seleccionados en la etapa anterior, y se obtienen los parámetros (m, b) de la recta definida por ellos. Posteriormente, se toma de nuevo el conjunto de puntos a nivel alto del v-*disparity* y se calcula el número de ellos que se encuentran a una distancia de la recta modelo inferior a un cierto umbral; esto es, el número de *inliers*. Una vez acabadas todas las iteraciones, se busca la recta que ha obtenido la mayor puntuación de *inliers*, que será la elegida como perfil de la calzada. El procedimiento se describe con mayor detalle en el Algoritmo 3.

En la línea 4 del Algoritmo 3, se puede comprobar la imposición de una exigencia para los dos puntos destinados a originar cada una de las rectas: han de tener distinto valor de coordenada x, para evitar la presencia de rectas con pendiente infinita. Este problema se podría solventar utilizando la ecuación de la recta dada por 5.8, pero no resulta necesario, al no tener sentido la aparición de un perfil de la calzada de pendiente infinita: en tal caso, se estaría manifestando la presencia de un obstáculo.

Algoritmo 3 Aplicación de RANSAC para la obtención de los parámetros del perfil de la calzada

Parámetro(s): `RP_NUM_THREADS`, `RP_RANGO`

1: **para** i incrementándose hasta un número límite de iteraciones `RP_NUM_THREADS` {el porqué de este nombre se comprenderá en el siguiente epígrafe} **hacer:**
2: Generar dos números aleatorios distintos
3: Acceder a dos posiciones, dadas por los números aleatorios anteriormente generados, del *array* de coordenadas de puntos a nivel alto del v-*disparity* para obtener dos puntos (x_1, y_1) y (x_2, y_2)
4: **si** $(x_1 \neq x_2)$ **entonces:**
5: $m_i \leftarrow \dfrac{y_1 - y_2}{x_1 - x_2}$
6: $b_i \leftarrow y_1 - m \cdot x_1$
7: **para** todos los puntos a nivel alto en el v-*disparity*, $(x, y)_i$ **hacer:**
8: **si** $|y_i - (m \cdot x_i + b)| <$`RP_RANGO` **entonces:**
9: Incrementar en 1 el número de *inliers* correspondiente al par (m_i, b_i)
10: **fin si**
11: **fin para**
12: **si no**
13: Descartar (m_i, b_i)
14: **fin si**
15: **fin para**
16: Buscar posición n correspondiente a la recta con más *inliers*.
17: **devolver** (m_n, b_n)

En caso de no cumplirse esta condición, se le asigna a la recta correspondiente puntuación -1, que, evidentemente, será la menor entre todas las presentes y no será escogida como la correcta. Es, por tanto, equivalente a descartarla.

Por otro lado, en la línea 8 del Algoritmo 3 se observa el método usado para calcular la distancia entre la recta y los puntos candidatos a ser *inliers*: el valor absoluto de la diferencia entre la coordenada y de cada punto y la que correspondería a la recta en la coordenada x del punto. Se trata por tanto, de la distancia vertical entre ambos elementos. En este caso, al tratarse de una recta (y no de un segmento) y estar los puntos contenidos en un cierto rango de x, el espacio en torno a la recta que se abarca al usar esta distancia

es el mismo que si se hallara la distancia euclídea, siendo mucho más simple su cálculo.

Los valores de los parámetros necesarios en el proceso se han obtenido experimentalmente, y están definidos como macros del preprocesador en un archivo cabecera destinado a tal efecto, resultando fácilmente modificables. En el capítulo 6 se justificarán los valores más recomendables para ellos.

Versión 3

La tercera y última versión para el cálculo del perfil de la calzada es análoga a la segunda, siendo válido el Algoritmo 3, con la única diferencia de que, en este caso, se realizan todas las iteraciones del RANSAC (línea 1) simultáneamente, aprovechando la capacidad para computar en paralelo de la GPU. Cada una de estas iteraciones es llevada a cabo por un hilo (*thread*), lanzado al principio del proceso, que ejecuta un *kernel* creado para tal fin.

Siguiendo ese *kernel*, cada hilo se encarga de generar su propio par de números aleatorios, usando la biblioteca de generación de números aleatorios de CUDA: cuRAND. Todos los hilos acceden a los puntos encontrados en el v-*disparity*, que se han copiado a la memoria global del *device*, y cada uno devuelve, también a través de esa memoria, los parámetros m y b de la recta que ha calculado, así como su puntuación (número de *inliers*). La búsqueda de la recta con mayor puntuación tiene lugar posteriormente, ya en la CPU, partiendo de los datos generados por todos los *threads*.

Por tanto, no resulta necesario sincronizar los hilos hasta el final del proceso, lo cual tiene lugar implícitamente en la instrucción `cudaMemcpy` que se utiliza para efectuar la copia de los datos a la memoria de la CPU.

En la Fig. 5.6 se presenta un resumen en forma de diagrama de flujo de las dos variaciones del algoritmo desarrolladas. En el capítulo 6, se podrá comprobar que esta última es la versión que requiere menor tiempo de cálculo y, por lo tanto, la más recomendable.

Por último, en la Fig. 5.7 se muestra un ejemplo del perfil de la calzada que se obtiene a partir del v-*disparity* mediante este último método.

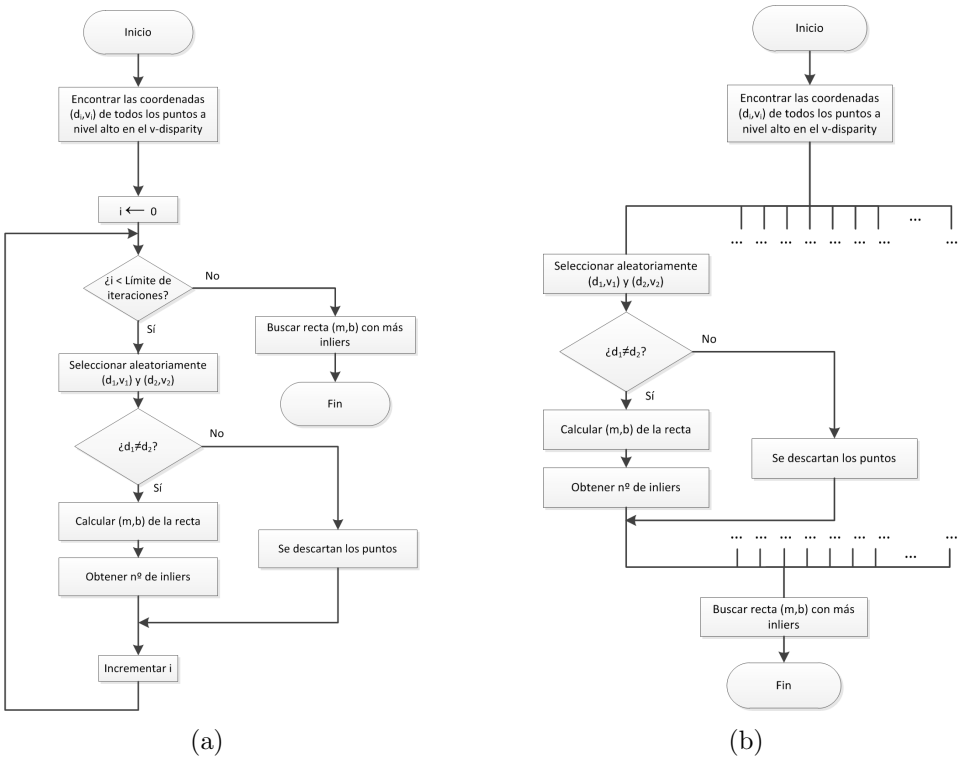


Figura 5.6. Diagramas de flujo de las versiones 2 (a) y 3 (b) para el cálculo del perfil de la calzada.

5.3 Detección de puntos característicos

El sistema de odometría visual desarrollado en este proyecto se basa en el seguimiento de puntos característicos entre *frames* consecutivos (t y $t+1$) capturados por la cámara izquierda, únicamente. Esto es posible porque las expresiones que se han desarrollado anteriormente permiten obtener la posición de puntos del mundo (X, Z) a partir de las coordenadas de su proyección en la imagen (u, v), con la condición de que dichos puntos pertenezcan a la calzada. Esto lo diferencia de otros métodos [33], en los que es necesario emparejar los puntos encontrados en cada una de las dos imágenes estéreo para obtener sus coordenadas en el mundo (X, Z).

Figura 5.7. Estimación del perfil de la calzada (línea roja) sobre el v-disparity.

A partir de las coordenadas en la imagen de los puntos característicos de la calzada, sus coordenadas en el mundo pueden obtenerse mediante las ecuaciones 5.5 y 5.6, corregidas con 5.7.

Para detectar los puntos característicos, se utiliza el detector y descriptor SURF, cuyos aspectos fundamentales ya han sido descritos en la sección 3.3.1. En concreto, se emplea la implementación del mismo presente en las bibliotecas OpenCV, dentro del conjunto de funcionalidades no libres. Por otro lado, el emparejamiento de los puntos de *frames* sucesivos se realiza mediante la biblioteca FLANN (*Fast Library for Approximate Nearest Neighbors*, biblioteca rápida para los vecinos aproximados más cercanos).

5.3.1 Implementación del algoritmo

Para llevar a cabo esta etapa del algoritmo, se han desarrollado dos versiones. Ambas llevan a cabo la misma secuencia de operaciones, con la única diferencia de que, en la primera, se lleva a cabo como parte del programa principal que implementa el algoritmo, mientras que, en la segunda, se ejecuta en paralelo a este, en un proceso del sistema distinto, transfiriéndose datos

entre ambos mediante zonas de memoria compartida cuyo acceso se gestiona mediante semáforos. Esto permite desarrollar la detección de características en paralelo con el cálculo del mapa de disparidad (y derivados) y el perfil de la calzada, como se verá más adelante.

Como se pretenden utilizar puntos característicos pertenecientes a la calzada, la búsqueda se restringe, en ambos casos, al tercio inferior de la imagen izquierda. Además de reducir el tiempo de computación necesario, usar solo los puntos más cercanos al vehículo permite asumir, con menos error, la hipótesis de suelo plano.

Versión 1

Las bibliotecas OpenCV se han utilizado a través de la interfaz clásica de C, al estar altamente probada y ser, por tanto, la más robusta.

El proceso comienza con la conversión de la imagen izquierda, contenida en un array común de C, al formato propio para cabeceras de imagen en OpenCV, `IplImage`. Posteriormente, se selecciona el tercio inferior de la misma como ROI (*Region Of Interest*, región de interés), de manera que será esta la parte de la imagen sobre las que se efectúen las operaciones posteriores. Para ello, se efectúa la siguiente llamada:

```
cvSetImageROI(mapa_libre,cvRect(0,ALTURA_SURF,WIDTH_IMAGEN,HEIGHT_IMAGEN));
```

El tamaño de la región sobre la que se trabaja se puede cambiar modificando `ALTURA_SURF`, definida en el archivo de cabecera de parámetros.

Los puntos característicos (keypoints) y los descriptores (descriptors) quedan almacenados durante el proceso en estructuras de datos incrementables `CvSeq`, almacenadas en un mismo espacio de memoria definido con `CvMemStorage`.

Antes de efectuar la búsqueda, se almacenan los *keypoints* y los descriptores del *frame* anterior (t) en un espacio de memoria distinto, con el fin de poder utilizar el mismo espacio en el nuevo *frame* ($t + 1$) preservando los

datos del anterior para su uso posterior. Se utiliza para ello la instrucción `CvCloneSeq`, seguida de `cvClearMemStorage`.

La detección de los puntos característicos y sus descriptores se efectúa con la función `cvExtractSURF`:

```
cvExtractSURF(izq_visible_1,0,&Keypoints,&Descriptors,storage,params);
```

De esta forma, los puntos quedan almacenados en `Keypoints`, junto con sus descriptores, en `Descriptors`, dentro del espacio de memoria `storage`.

Para ello se usan los dos parámetros dados por `params`, una estructura de tipo `cvSURFParams`: el umbral hessiano y el tipo de descriptores a obtener. Pese a que la documentación de las OpenCV recomienda usar valores del umbral entre 300 y 500 [65], se ha decidido utilizar un valor más bajo, 100, debido a que la calzada presenta poco contraste y escasa textura. Por otro lado, los descriptores son básicos, de 64 elementos cada uno, para acelerar el procedimiento de emparejamiento.

Empleando los keypoints y los descriptores de este *frame* $(t+1)$ y del anterior (t), se emparejan los puntos de ambos mediante FLANN:

```
flannFindPairs(anteriorKeypoints,anteriorDescriptors,Keypoints,Descriptors,
    ptpairs);
```

El vector `ptpairs` recibe, de forma secuencial, todos los pares hallados. Posteriormente, de cada pareja, se copia un elemento a un vector y el otro a otro vector distinto, obteniendo unos vectores `pt1` y `pt2` tales que el punto ubicado en la posición i de `pt1` pertenece al *frame* t y está emparejado (esto es, debería corresponder al mismo punto del mundo) con el de la posición i de `pt2`, que pertenece al *frame* $t+1$.

El proceso aparece descrito detalladamente en el Algoritmo 4.

En la Fig. 5.8, se muestra un ejemplo de los resultados obtenidos. Presenta los puntos característicos correspondientes a dos *frames* consecutivos, unidos entre sí por líneas rojas, mostradas sobre el *frame* más reciente de los dos.

Figura 5.8. Resultados de la detección y emparejamiento de puntos característicos.

Versión 2

Mediciones preliminares de los tiempos de computación (que se ampliarán en el capítulo 6) mostraron que la versión 1 del algoritmo de detección y emparejamiento de puntos característicos, mostrada en el epígrafe anterior, era el proceso más lento de todos cuantos se llevaban a cabo en la CPU en el cálculo de la odometría visual.

Con el fin de acelerar el proceso, se decidió que esta etapa fuese ejecutada en paralelo con el resto del algoritmo. Para ello, se ejecuta en un proceso aparte, e intercambia información con el programa principal mediante memorias compartidas ubicadas en el archivo de paginación de Windows. Hay dos factores que justifican esta decisión:

- La búsqueda de puntos característicos no requiere de ningún resultado intermedio obtenido en el algoritmo; únicamente necesita disponer de la imagen izquierda, tal cual es capturada por la cámara estéreo.

Algoritmo 4 Algoritmo para la detección y emparejamiento de puntos característicos

Parámetro(s): Umbral Hessiano

1: Conversión de la imagen izquierda a formato de OpenCV.
2: Selección del tercio inferior.
3: Almacenamiento de los puntos característicos y los descriptores del *frame* anterior, para preservarlos.
4: Extracción de puntos característicos y descriptores del nuevo *frame*.
5: Emparejamiento de los puntos del *frame* actual y el anterior.
6: Distribución de los pares en dos vectores de puntos tales que el punto en la posición i del primer vector está emparejado con el punto en la posición i del segundo vector.

- Las primeras etapas del algoritmo comprenden, básicamente, el cálculo del mapa de disparidad (y estructuras que se derivan de él: mapa libre, u-v *disparity*, etc.) y la obtención del perfil de la calzada. Ambas operaciones concentran la carga computacional en la GPU, por lo que, en la versión 1, la CPU permanecía a un nivel bajo de utilización durante la realización de estos procesos. Sin embargo, la detección de puntos característicos se produce únicamente en la CPU, por lo que puede utilizar esta capacidad no aprovechada.

Además, la tendencia actual en el desarrollo de microprocesadores está orientada a la presencia de varios núcleos de ejecución en la CPU, lo que se conoce como MIMD (*Multiple Instruction, Multiple Data*, "múltiples instrucciones, múltiples datos")[1], algo que aumenta las ventajas de esta paralelización en términos de tiempo de cómputo.

En adelante, en aras de una mayor brevedad, se denotará el proceso que ejecuta la detección de puntos característicos como *auxiliar*, mientras que el que lleva a cabo el resto del algoritmo será el *principal*.

El proceso auxiliar se encarga, fundamentalmente, de llevar a cabo el Algoritmo 4, que requiere, para iniciarse, la imagen izquierda. Esta imagen es capturada por el proceso principal al comienzo de cada iteración, y enviada

[1] Valgan como ejemplo los microprocesadores basados en las arquitecturas *Nehalem* y *Sandy Bridge* de Intel, con nombres comerciales Core i3, i5 e i7, que constan de dos, cuatro o, incluso, seis núcleos de ejecución.

al proceso auxiliar mediante una memoria compartida. Este genera como resultado los ya mencionados dos vectores (`pt1` y `pt2`) con los pares de puntos característicos, que son enviados del mismo modo al proceso principal para que los reciba al finalizar el cálculo del perfil de la calzada.

Son necesarias, por lo tanto, tres espacios de memoria compartida: uno para la imagen, y otros dos para los vectores. Para su creación en Windows se ha recurrido a la siguiente secuencia:

```
HANDLE hMapFile;
TIPO* pBuf;
hMapFile = CreateFileMapping(INVALID_HANDLE_VALUE,NULL,PAGE_READWRITE,0,
    BUF_SIZE,szName);
pBuf = (TIPO*) MapViewOfFile(hMapFile,FILE_MAP_ALL_ACCESS,0,0,BUF_SIZE);
```

Donde la palabra `TIPO` hace referencia al tipo de dato que se ha de almacenar en la memoria, en cada caso. De esta forma, se crea una memoria, con permisos de lectura y escritura y de tamaño `BUF_SIZE` en el archivo de paginación del sistema, y se vincula al puntero `pBuf`. En este caso, se ha optado por llevar a cabo la creación de todas las memorias necesarias en el proceso principal. El proceso auxiliar accede a ellas mediante:

```
HANDLE hMapFile;
TIPO* pBuf;
hMapFile = OpenFileMapping(FILE_MAP_ALL_ACCESS,FALSE,szName);
pBuf = (TIPO*) MapViewOfFile(hMapFile,FILE_MAP_ALL_ACCESS,0,0,BUF_SIZE);
```

Hay que tener en cuenta que las zonas de memoria se identifican por su nombre, dado por el string `szName`. Su tamaño se ha escogido de forma que la primera pueda albergar la imagen, y las otras dos, hasta 2048 pares de puntos, un número ampliamente superior al máximo alcanzado durante las pruebas del programa y suficiente para los cálculos posteriores.

La sincronización entre ambos procesos para la gestión del acceso a las memorias compartidas se lleva a cabo mediante semáforos. Los semáforos son variables que controlan el acceso de múltiples procesos a un recurso común, como es el caso. Se utilizan cuatro, con las siguientes funciones:

- Semáforo 1: Gestión de la escritura de la imagen

- Semáforo 2: Gestión de la lectura de la imagen
- Semáforo 3: Gestión de la escritura de los vectores
- Semáforo 4: Gestión de la lectura de los vectores

Los semáforos también son creados en el proceso principal, usando:

```
HANDLE ghSemaphore;
ghSemaphore = CreateSemaphore(NULL, CUENTA_INICIAL, 1, semName);
```

Siendo `CUENTA_INICIAL` el valor inicial del semáforo y 1 el valor máximo del mismo. El proceso auxiliar accede a ellos con:

```
HANDLE ghSemaphore;
ghSemaphore = OpenSemaphore(SYNCHRONIZE|SEMAPHORE_MODIFY_STATE, FALSE,
    semName);
```

De nuevo, los semáforos se identifican a través de su nombre, `semName`.

El proceso principal intenta, una vez capturada la imagen, decrementar el Semáforo 1 para copiarla a la memoria compartida. Una vez efectuada esta operación, incrementa el Semáforo 2. El proceso auxiliar, que estaba esperando para decrementarlo, lo hace y, una vez leída la imagen, incrementa el Semáforo 1. Se deduce, por tanto, que el estado inicial del Semáforo 1 ha de ser 1, y el del Semáforo 2, 0.

Posteriormente, una vez detectados los puntos característicos, el proceso auxiliar intenta decrementar el semáforo 3 para escribir los resultados en la memoria compartida. Una vez lo consigue, incrementa el semáforo 4, que será decrementado por el proceso principal para leer los datos e incrementar, posteriormente, el semáforo 3. El estado inicial del semáforo 3 es, por tanto, 1, mientras que el del 4 es 0.

En la Fig. 5.9 se presenta la diferencia que suponen cada una de estas dos versiones en el algoritmo.

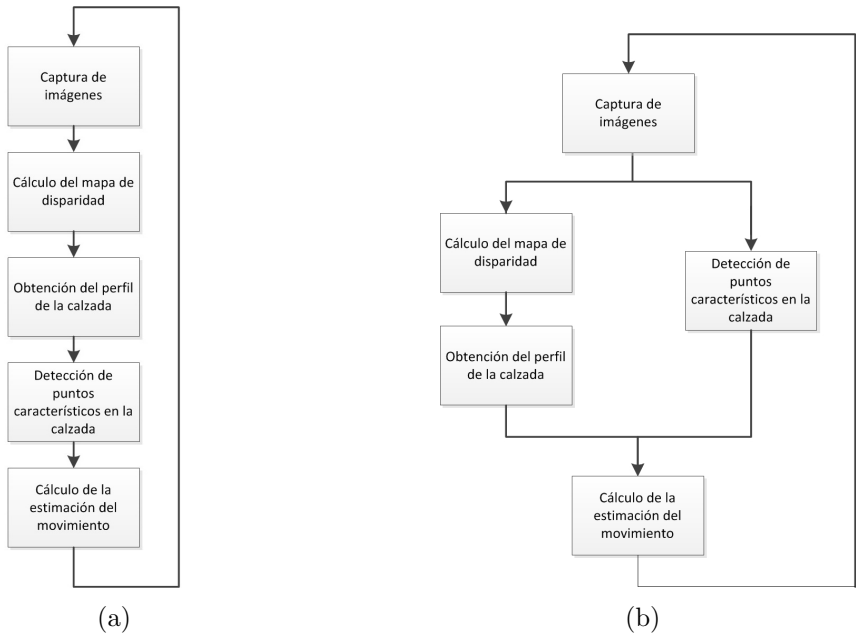


Figura 5.9. Diagramas de flujo de las versiones 1 (a) y 2 (b) para la detección de puntos característicos en la calzada.

5.4 Cálculo de la estimación de movimiento entre frames consecutivos

El modelo escogido para representar el movimiento del vehículo hace uso de la geometría de Ackermann [66]. Es un diseño que trata de solucionar el problema que surge cuando un vehículo toma una curva: las ruedas del lado interior a la misma siguen una circunferencia de menor radio que las del exterior.

La geometria de Ackermann se basa en provocar que la rueda delantera[2] situada en el lado interior de la curva gire un ángulo ligeramente mayor que la que está situada en el exterior, eliminando el deslizamiento inducido por causas geométricas. De esta manera, los ejes perpendiculares a cada una de las cuatro ruedas se intersecan en un mismo punto, como muestra la Figura 5.10.

[2]Se considera que las ruedas directrices del vehículo son las del eje delantero.

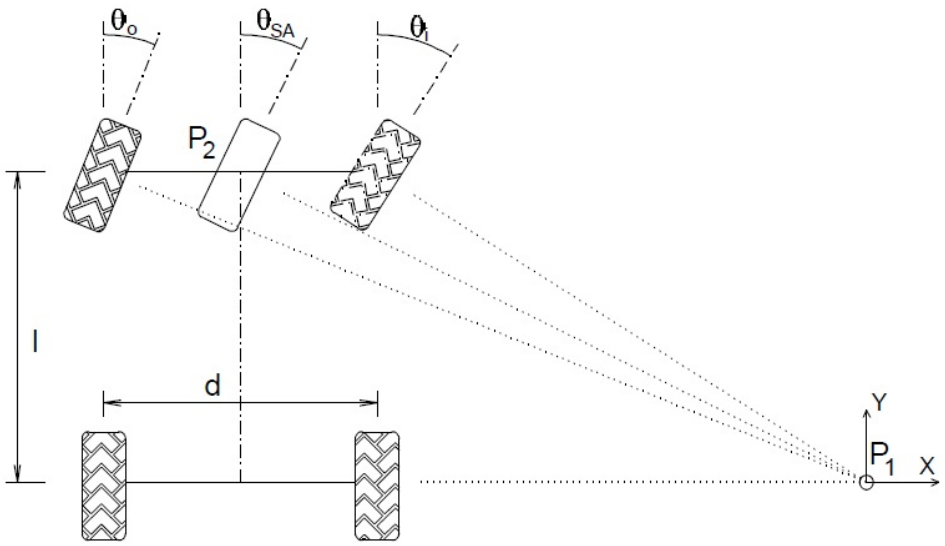


Figura 5.10. Geometria de Ackermann.

Se trata de una solución ampliamente extendida en la industria del automóvil, como demuestra el empleo del *trapecio de Jeantaud* en el mecanismo de dirección de los coches o la presencia del elemento llamado *diferencial*, que compensa la diferencia en la velocidad de giro de ambas ruedas del eje delantero.

Para simplificar la estimación del movimiento del vehículo, se considerarán dos hipótesis:

1. El movimiento del vehículo entre dos *frames* sucesivos se puede dividir en dos etapas de velocidad constante, tal y como se muestra en la Fig. 5.11:

   a) Una rotación en torno al centro del eje trasero, de ángulo θ.

   b) Una traslación hacia delante después de la rotación, descrita de acuerdo a su proyección en los dos ejes cartesianos del plano de la calzada, $(\Delta Z, \Delta X)$.

2. No hay deslizamiento en ninguna dirección

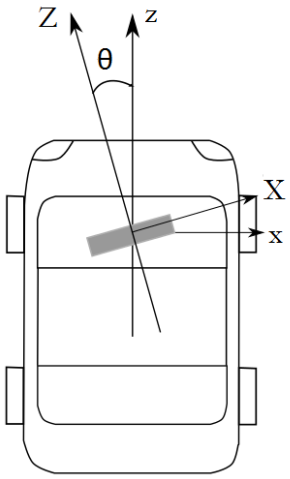


Figura 5.11. Esquema de las coordenadas respecto al vehículo.

Cuando se llega a esta etapa, se tienen las coordenadas en la imagen (u, v) de los puntos característicos presentes en la calzada en el *frame* actual, $t + 1$, y de los mismos en el anterior, t, emparejadas. Es posible obtener sus coordenadas en el mundo (X, Z) mediante las ecuaciones presentadas al comienzo de este capítulo (5.5 y 5.6, corregidas con 5.7). Conviene recordar que la coordenada Z hace referencia a la distancia al punto desde el vehículo, en el plano de la calzada, mientras que la X se mide en el eje que es perpendicular a la longitudinal del vehículo y que está contenido, igualmente, en el plano de la calzada.

La estimación del movimiento del vehículo se puede llevar a cabo usando el cambio de posición entre t y $t + 1$ de los puntos detectados en el paso anterior. Se desglosará de acuerdo a las dos etapas con las que se ha modelado tal movimiento:

Ángulo de rotación θ

El ángulo de rotación se puede obtener, haciendo uso de principios geométricos, mediante 5.9

$$\theta = \arctan \frac{\Delta X}{\Delta Z} \tag{5.9}$$

Desplazamiento longitudinal $(\Delta Z, \Delta X)$

Es posible expresar ΔZ y ΔX en función de θ y las posiciones de un mismo punto de la calzada en los instantes t y $t+1$ (representadas con (Z^t, X^t) y (Z^{t+1}, X^{t+1}), respectivamente) usando 5.10.

$$\begin{bmatrix} X^t \\ Z^t \end{bmatrix} = \begin{bmatrix} \cos\theta & \operatorname{sen}\theta \\ -\operatorname{sen}\theta & \cos\theta \end{bmatrix} \begin{bmatrix} X^{t+1} \\ Z^{t+1} \end{bmatrix} + \begin{bmatrix} \Delta X \\ \Delta Z \end{bmatrix} \tag{5.10}$$

Despejando, se obtienen las expresiones 5.11 y 5.12, con las que se puede calcular el desplazamiento del vehículo a partir del desplazamiento de los puntos conociendo el ángulo de rotación θ.

$$\Delta X = X^t - X^{t+1}\cos\theta - Z^{t+1}\operatorname{sen}\theta \tag{5.11}$$

$$\Delta Z = Z^t - Z^{t+1}\cos\theta + X^{t+1}\operatorname{sen}\theta \tag{5.12}$$

El único dato desconocido en las anteriores ecuaciones es θ, por lo que se requiere llevar a cabo su cálculo como primer paso. Para ello, se hace uso de 5.13.

$$\tan\theta = \frac{\operatorname{sen}\theta}{\cos\theta} = \frac{\Delta X}{\Delta Z} \tag{5.13}$$

De 5.11, 5.12 y 5.13, se obtiene 5.14.

$$\frac{\Delta X}{\Delta Z} = \frac{\operatorname{sen}\theta}{\cos\theta} = \frac{X^t - X^{t+1}\cos\theta - Z^{t+1}\operatorname{sen}\theta}{Z^t - Z^{t+1}\cos\theta + X^{t+1}\operatorname{sen}\theta} \tag{5.14}$$

Y con ella, se puede obtener 5.15, que es una ecuación de segundo grado que permite calcular θ.

$$((X^t)^2 + (Z^t)^2)\operatorname{sen}^2\theta + (2 \cdot X^{t+1} \cdot Z^t)\operatorname{sen}\theta + ((X^{t+1})^2 - (X^t)^2) = 0 \tag{5.15}$$

A partir de θ, como se indicó con anterioridad, bastaría con aplicar 5.11 y 5.12 para obtener todos los parámetros del movimiento.

Es decir, a partir de las posiciones de un punto de la calzada en un *frame* (Z^t, X^t) y en el siguiente (Z^{t+1}, X^{t+1}), se ha obtenido una terna $\{\theta, \Delta Z, \Delta X\}$,

que describe el movimiento del vehículo. Es de esperar que se tengan, por lo tanto, tantas soluciones $\{\theta, \Delta Z, \Delta X\}$ como pares de puntos se han encontrado en la calzada.

Para elegir una solución única, se pueden emplear varios métodos. El más simple podría ser la media; sin embargo, no se ha tomado en consideración debido a su baja robustez ante posibles errores. En su lugar, se empleará la mediana y, en último término, RANSAC.

Con el fin de describir cómo se ha llevado a cabo la programación de esta sección, se distinguirá entre la parte en la que se efectúa el cálculo de los desplazamientos, mediante las fórmulas anteriormente indicadas, y aquella otra en la que se busca la mejor entre todo el conjunto de soluciones encontradas.

5.4.1 Implementación del algoritmo de cálculo de resultados

Se dispone de dos versiones para el cálculo de los distintos conjuntos de soluciones $\{\theta, \Delta Z, \Delta X\}$. En la primera de ellas, el procesamiento se realiza de forma iterativa en la CPU; en la segunda, se modifica para adaptarlo a la arquitectura orientada al paralelismo de la GPU.

Versión 1

La obtención de las diferentes soluciones $\theta, \Delta Z, \Delta X$ de la odometría visual a partir de los pares de puntos característicos en la imagen (u, v) se lleva a cabo empleando la secuencia de fórmulas anteriormente mostrada, de acuerdo al Algoritmo 5. Hay que resaltar que para ello se hace uso de los parámetros de calibración del sistema estéreo (distancia focal, *baseline* y coordenadas del centro óptico), de los parámetros del perfil de la calzada (pendiente y horizonte) y del ángulo *pitch* (α) que se obtiene a partir de ellos.

Los resultados que se obtendrán en último término no incluirán los desplazamientos $(\Delta Z, \Delta X)$, sino las coordenadas (Y_T, X_T) que va alcanzando el vehículo durante su movimiento, referidas al punto inicial del movimiento, para poder analizar con más facilidad los resultados obtenidos. Para ello, hay

que efectuar una rotación (5.16), obteniéndose las soluciones finales con 5.17 y 5.18.

$$\begin{bmatrix} X_T \\ Y_T \end{bmatrix} = \begin{bmatrix} \cos\theta & \operatorname{sen}\theta \\ -\operatorname{sen}\theta & \cos\theta \end{bmatrix} \begin{bmatrix} \Delta X \\ \Delta Z \end{bmatrix} \tag{5.16}$$

$$X_T = \Delta X \cos\theta + \Delta Z \operatorname{sen}\theta \tag{5.17}$$

$$Y_T = \Delta Z \cos\theta - \Delta X \operatorname{sen}\theta \tag{5.18}$$

Algoritmo 5 Algoritmo para el cálculo de los parámetros del movimiento

Parámetro(s): Desviación de ϕ, distancia focal, baseline, coordenadas del centro óptico.

1: **para** todos los pares de puntos (u^t, v^t) y (u^{t+1}, v^{t+1}) **hacer:**
2: Calcular coordenadas en el mundo (Z^t, X^t) y (Z^{t+1}, X^{t+1}) (5.5, 5.6).
3: Corregir coordenadas en el mundo con la desviación del ϕ (5.7).
4: Obtener θ (5.15) y almacenarla.
5: Obtener ΔZ y ΔX (5.11, 5.12).
6: Obtener X_T y Y_T (5.17, 5.18) y almacenarlas.
7: **fin para**

Como se puede comprobar, esta implementación obliga a realizar tantas iteraciones como pares de puntos caracteristicos hayan sido encontrados. Por ello, los tiempos de procesamiento serán altamente sensibles a variaciones en su numero.

Versión 2

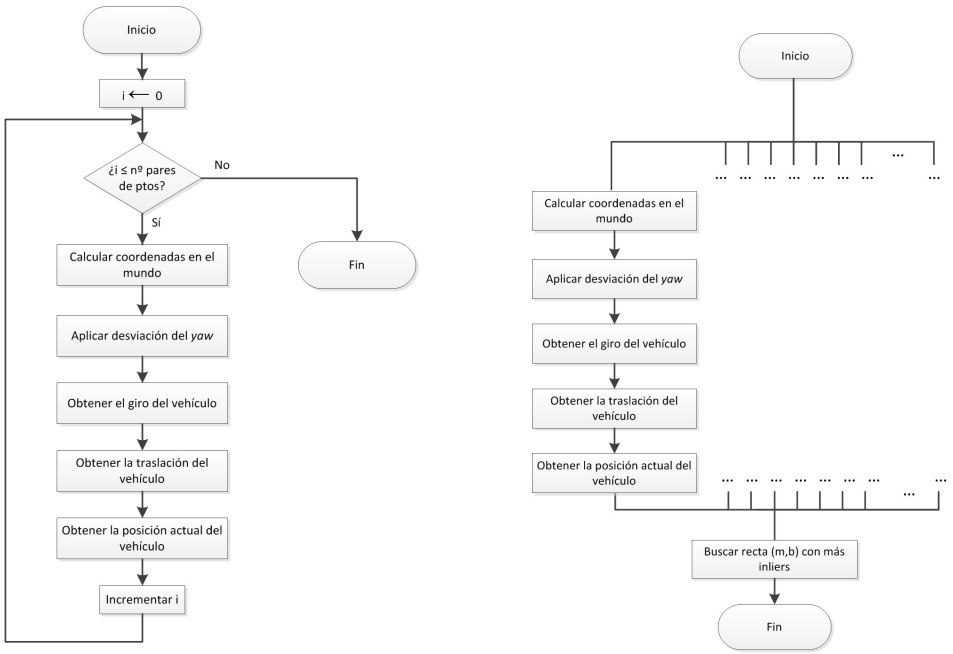
En esta evolución, se ha adaptado el cálculo de la odometría para su procesamiento en la GPU. Se realizan las mismas operaciones que se describieron en la Versión 1, por lo sigue siendo válido el Algoritmo 5, con la única diferencia de que todas las iteraciones tienen lugar en paralelo.

Para ello, se lanzan tantos hilos de ejecución como pares de puntos característicos se hayan encontrado en la calzada. Cada uno de estos hilos se encarga de calcular sus coordenadas en el mundo y el desplazamiento que se interpreta a través de ellas.

Para ello, es necesario trasladar a la memoria de la GPU los dos vectores que contienen los pares de puntos de la calzada. Los resultados (θ, Y_T, X_T), por su parte, quedan almacenados sucesivamente en tres vectores en la memoria global de la GPU, y han de ser trasladados a la memoria de la CPU.

En este caso, tampoco existe interacción entre los distintos hilos, por lo que la sincronización se efectúa al final del proceso.

En la figura 5.12 se comparan las dos versiones mediante sus respectivos diagramas de flujo.



(a) (b)

Figura 5.12. Diagramas de flujo de las versiones 1 (a) y 2 (b) para el cálculo de los resultados que caracterizan el movimiento del vehículo.

5.4.2 Implementación del algoritmo de búsqueda de una solución única

Una vez se dispone de las ternas (θ, Y_T, X_T) correspondientes a cada uno de los pares de puntos de la calzada, es necesario hallar cuál de ellas representa mejor el movimiento del vehículo. Como se mencionó anteriormente, se ha

descartado la media por su alta sensibilidad ante la presencia de grandes errores, en favor de la mediana y RANSAC.

A continuación se describen las cinco versiones que se han desarrollado con este propósito. La primera de ellas, asume el resultado formado por la mediana de cada uno de los elementos como la mejor solución. La segunda, por su parte, utiliza RANSAC, en cada uno de los parámetros, para alcanzar una solución única. De ella se deriva la tercera versión, que aprovecha la capacidad de computar en paralelo de la GPU para ejecutar RANSAC.

Las dos últimas versiones suponen un intento por mejorar la robustez del algoritmo. Implementan una variante de las versiones 2 y 3, consistente en utilizar la mediana y RANSAC de forma sucesiva. La versión 4 realiza el procesamiento en la CPU y la 5, lo paraleliza en la GPU.

Versión 1

Si se simboliza la mediana de una variable X como $\tilde{X}$, la solución final obtenida en esta versión es $(\tilde{\theta}, \tilde{Y_T}, \tilde{X_T})$. Se pretende obtener un valor que dé una idea de los elementos intermedios de entre todos los que se han recopilado, evitando al mismo tiempo que los errores graves causen grandes desviaciones.

Versión 2

Una evolución respecto a lo anterior es llevar a cabo RANSAC para obtener la mejor solución de cada uno de los parámetros. El procedimiento es similar al utilizado en 5.2.1 para obtener el perfil de la calzada, y se detalla en el Algoritmo 6.

Es importante resaltar que cada uno de los tres elementos del desplazamiento, θ, Y_T y X_T, se tratan por separado. Se ha optado por esta solución al no haber resultado satisfactorias las pruebas iniciales realizadas usando la estrategia alternativa, consistente en manejar los conjuntos (θ, Y_T y X_T) obtenidos por un mismo par de puntos como indivisibles.

Este procedimiento supone llevar a cabo RANSAC sobre cada parámetro tratándolo como un ente independiente. Eso es precisamente lo que se refleja

Algoritmo 6 Aplicación de RANSAC para la obtención de la solución única del movimiento

Parámetro(s): `OD_NUM_THREADS`, `RANGO_THETA`, `RANGO_X`, `RANGO_Y`.

1: **para** i incrementándose hasta un número límite de iteraciones `OD_NUM_THREADS` {al igual que sucedía en el algoritmo 3, el nombre tiene su explicación en la siguiente versión} **hacer:**
2: Asignar a cada iteración i uno de los conjuntos de parámetros existentes (θ, Y_T o X_T)
3: Generar un número aleatorio.
4: Usar ese número para elegir un elemento de entre el conjunto de parámetros asignado en el paso 2.
5: **para** j recorriendo todos los elementos del conjunto **hacer:**
6: **si** distancia entre el elemento seleccionado y el elemento j es menor que el rango correspondiente (`RANGO_THETA`, `RANGO_X` o `RANGO_Y`) **entonces:**
7: Incrementar en 1 el número de *inliers* correspondiente al elemento seleccionado.
8: **fin si**
9: **fin para**
10: **fin para**
11: Buscar, para cada uno de los conjuntos, el elemento con mayor número de *inliers*.
12: **devolver** (θ, Y_T, X_T) única.

en el Algoritmo 6: en las primeras iteraciones, se escoge un parámetro θ al azar y se calcula el número de *inliers* del mismo: esto es, el número de puntos cercano. En iteraciones más tardías, se realiza el mismo procedimiento para las X_T y, en las últimas, para las Y_T.

Conviene tener en cuenta que, para cada uno de los parámetros del movimiento, existe una distancia límite para la consideración de los *inliers* distinta. Si bien esta ha sido escogida experimentalmente y presenta, finalmente, el mismo valor en los tres casos, es posible efectuar cualquier modificación fácilmente con solo cambiar sus valores en el fichero cabecera destinado a definir los parámetros del programa.

Por su parte, el número de iteraciones a realizar se ha alcanzado de manera experimental. Como se deduce de lo descrito con anterioridad, el

número de iteraciones para cada uno de los tres RANSAC que se aplican no es `OD_NUM_THREADS`, sino un tercio de este valor.

Versión 3

Es posible observar que el algoritmo de la versión anterior presenta grandes posibilidades de ser paralelizado con ayuda de la GPU. Eso es, precisamente, lo que se ha llevado a cabo en esta otra. Se trata del Algoritmo 6 modificado de forma que todas las iteraciones del primer bucle (línea 1) se ejecutan en hilos de ejecución paralelos.

Nótese que ello supone aplicar RANSAC simultáneamente sobre cada uno de los tres parámetros del movimiento. Al comienzo del *kernel* que ejecuta esta tarea, a cada hilo se le asigna uno de los tres conjuntos de parámetros (θ, Y_T o X_T), y sobre él escoge un valor aleatorio y calcula el número de *inliers*.

Para ello, es necesario copiar los tres conjuntos a la memoria global de la GPU al inicio del proceso. Las posiciones aleatorias a las que accede cada hilo, así como la puntuación obtenida por el valor que ha seleccionado, también quedan almacenadas en memoria global y es necesario trasladarlas a la memoria de la CPU para, en último término, realizar en esta la búsqueda de los valores con mayor numero de *inliers* (línea 11 del Algoritmo 6).

Versiones 4 y 5

Las dos últimas versiones son un intento por mejorar la robustez del método ante la presencia de errores. Para ello, los valores de los parámetros que se utilizan en cada una de las iteraciones de RANSAC no se obtienen directamente mediante una selección aleatoria entre todo el conjunto de ellos, sino que se escogen como la mediana de un conjunto de tres puntos elegidos, ahora si, aleatoriamente.

De esta forma, se garantiza que los puntos más extremos, correspondientes a errores gruesos, ni siquiera lleguen a ser candidatos en RANSAC.

La diferencia entre estas dos versiones reside en que la primera de ellas (Versión 4) parte de la Versión 2, estando destinada a ejecutarse secuencial-

mente en la CPU, mientras que la segunda (Versión 5) se basa en la Versión 3 para desarrollar la ejecución en paralelo en la GPU mediante CUDA.

El proceso es el que aparece descrito en el Algoritmo 6, con la diferencia de que se seleccionan tres números aleatorios en el paso 3, siendo necesario calcular posteriormente la mediana de los mismos. Este será el número que se utilice en el paso descrito en la línea 4 para efectuar la selección.

En la figura 5.13 se muestran diagramas de flujo de las últimas cuatro versiones para esta etapa del algoritmo.

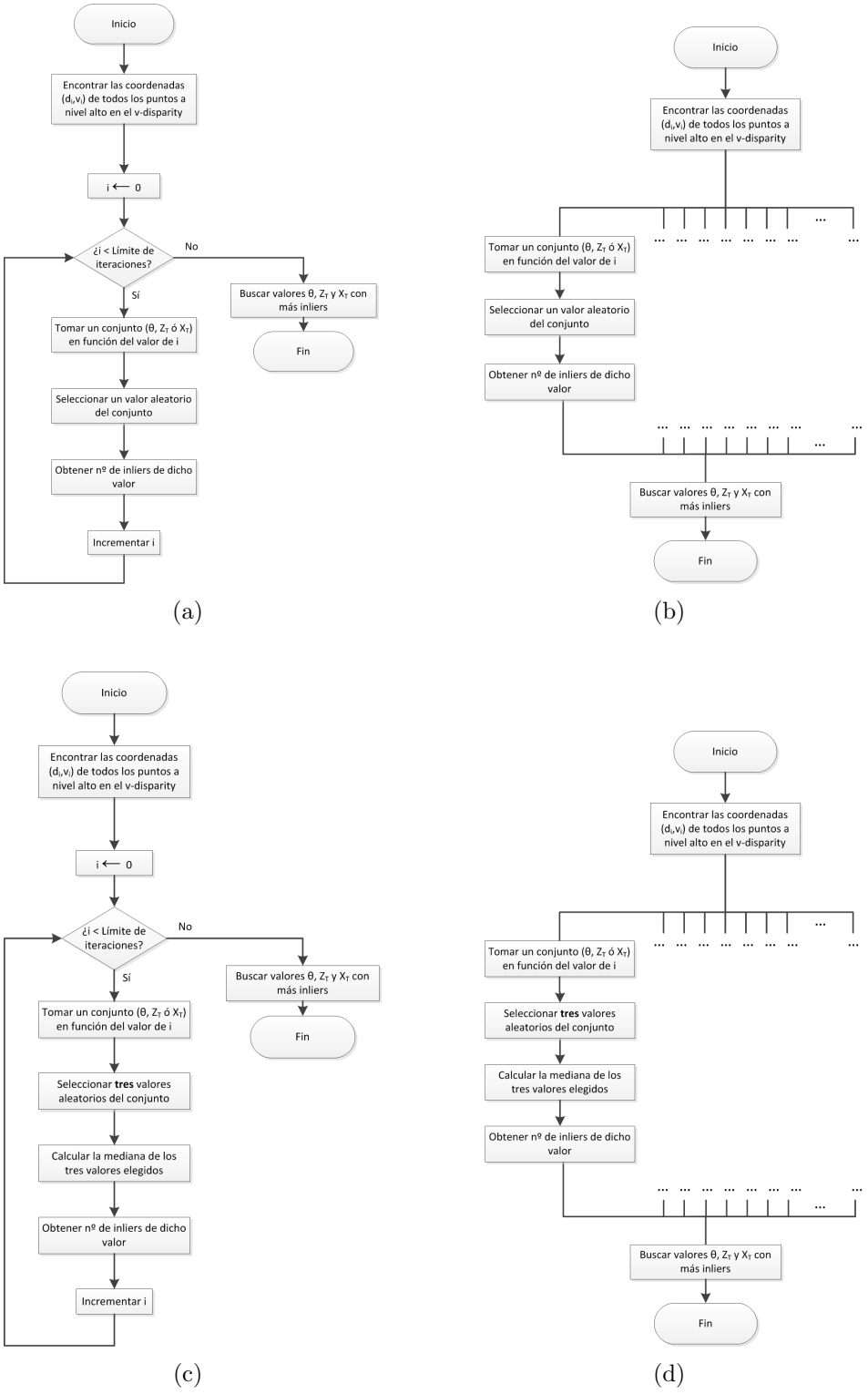


Figura 5.13. Diagramas de flujo de las últimas cinco versiones para el cálculo de una solución única.

6 Resultados

En este capítulo, se va a analizar la adecuación del algoritmo desarrollado, así como de su posterior implementación, a los requisitos impuestos para su correcto funcionamiento. Básicamente, son los siguientes:

- Se pretende integrar el desarrollo propuesto dentro del conjunto de sistemas que forman parte del vehículo IVVI 2.0 [8], mostrado en el epígrafe 1.2.2. Para ello, es necesario garantizar un adecuado funcionamiento del mismo sobre los sistemas en él instalados, tanto *hardware* como de captura de imágenes (el sistema estéreo Bumblebee [20]).
- La velocidad a la que se ejecuta el algoritmo debe ser la mayor posible. De esta forma, se consigue el procesamiento de un número mayor de *frames* por segundo, evitando que se produzcan pérdidas de información entre dos capturas sucesivas de la cámara.

Para ello, en este capitulo se presentarán resultados de tiempos de cómputo del algoritmo ejecutándose sobre el computador presente en el IVVI para el procesamiento de imágenes estéreo, con las características técnicas mostradas en la Tabla 6.1.

Dicho sistema está provisto de una tarjeta gráfica NVIDIA, preparada para el procesamiento de instrucciones en la GPU mediante CUDA. Se trata del modelo NVIDIA FX 380 LP, que presenta los parámetros técnicos mostrados en la Tabla 6.2.

Microprocesador	Intel Core i5 660 a 3,33 GHz
Memoria RAM	3 GB
Sistema Operativo	Windows XP Service Pack 2

Tabla 6.1. Especificaciones técnicas del ordenador utilizado en la toma de resultados.

Núcleos de CUDA	16
Memoria compartida global	512 MB GDDR3
Memoria compartida por bloque	16 KB
Registros por bloque	16384
Ancho de banda de la memoria	12,8 GB/s
Número máximo de hilos por bloque	512
Tamaño máx. de las dim. 0 y 1 de los bloques	512
Tamaño máx. de la dim. 2 de los bloques	64
Tamaño máx. de las dim. 0 y 1 del *grid*	65535
Tamaño máx. de las dim. 2 del *grid*	1

Tabla 6.2. Especificaciones técnicas de la GPU empleada en la toma de resultados.

Hay que destacar que, en todas las pruebas, se ha utilizado la versión 2.4.2. de las OpenCV, lanzada en julio de 2012.

Para realizar las distintas pruebas garantizando la repetibilidad de los resultados, se han utilizado como imágenes de entrada del algoritmo una secuencia de capturas efectuadas por la cámara estéreo en el entorno del campus de Leganés de la Universidad. La secuencia, que comprende 982 imágenes, muestra la visión de la cámara estéreo mientras el IVVI describe una trayectoria cerrada de unos 500 m.

Para lograr una mayor precisión, se ha tenido en cuenta el valor de la desviación en torno al ángulo *yaw* de la cámara durante esa secuencia, $0,0547\ rad$.

Otro aspecto a destacar es que, en la toma de tiempos, no se tiene en cuenta la primera de las iteraciones del algoritmo. Esto es debido a que en ella se ejecutan operaciones de reserva de espacio de memoria e inicialización de determinados parámetros necesarias en el arranque del sistema. Como se

trata de un algoritmo destinado a ejecutarse de manera continua durante la conducción, estas operaciones no resultan significativas.

Los tiempos de cómputo han sido tomados con la función `QueryPerformanceCounter`, el contador de alta resolución de Windows. Para ello, se han habilitado varias clases, tal y como se muestra en el Anexo A.2.

A continuación, se va a justificar cuál de las versiones presenta un mejor funcionamiento de entre todas las desarrolladas, atendiendo a los resultados obtenidos y a los tiempos de computo empleados por cada una de ellas. Una vez seleccionada, se realizará un análisis más detallado de los resultados obtenidos con ella.

6.1 Análisis de las distintas versiones

6.1.1 Versión inicial

El programa en su primera versión se construyó con todas las implementaciones marcadas como *Versión 1* en el capítulo anterior. Es decir, con el algoritmo ejecutándose en la CPU, salvo aquellos procesos relacionados con la construcción del mapa de disparidad, y empleando la Transformada de Hough para la obtención del perfil de la calzada, la mediana para la obtención de la solución final y SURF en el mismo proceso que el resto del algoritmo. La trayectoria estimada es la que se presenta en la Fig. 6.1.

Por su parte, los tiempos de cómputo invertidos en la ejecución de cada una de las operaciones aparecen en la Tabla 6.3 y en la Fig. 6.2. Se han separado los procesos necesarios para llevar a cabo el proceso de aquellos que únicamente tienen como fin mostrar la información al usuario.

Los resultados muestran la importancia de reducir el tiempo de computación del cálculo del perfil de la calzada y de la obtención de puntos característicos mediante SURF, al tratarse de los dos procesos más costosos tras la obtención del mapa de disparidad.

Figura 6.1. Trayectoria obtenida con la primera versión del algoritmo.

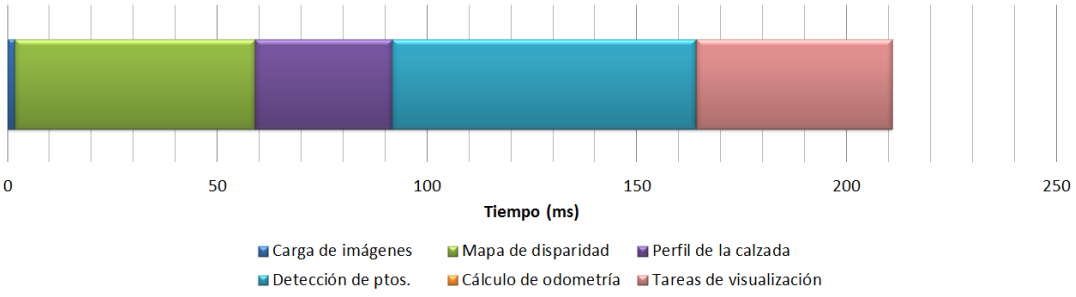


Figura 6.2. Tiempos de ejecución de las distintas operaciones en la primera versión del algoritmo.

Operación	Tiempo (ms)
Carga de imágenes	1,81
Mapa de disparidad	57,09
Perfil de la calzada	32,73
Detección de ptos.	72,46
Cálculo de la odometria	0,15
Total sin visualización	164,24
Presentación de imágenes	14,81
Mostrar ptos. car.	31,03
Mostrar mapa	0,76
Tareas visualización	46,59
Total	**210,83**

Tabla 6.3. Tiempos de cómputo de las distintas operaciones en la primera versión del programa.

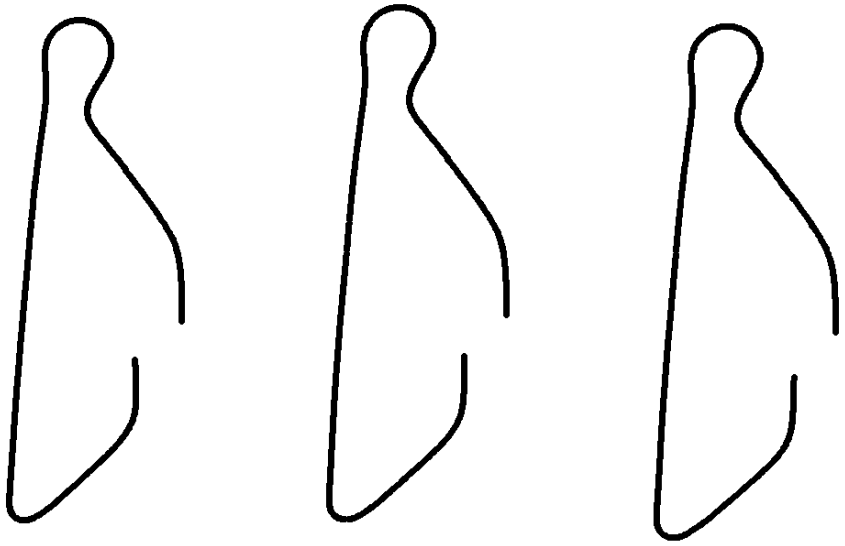
6.1.2 Versiones para el cálculo del perfil de la calzada

Para el cálculo del perfil de la calzada, se han desarrollado tres versiones: la primera de ellas, utiliza la Transformada de Hough, y las otras dos, RANSAC. Estas dos últimas versiones requieren elegir una serie de parámetros, cuya determinación se va a llevar a cabo de manera empírica.

Se han realizado distintas ejecuciones del algoritmo, variando la versión de esta etapa y dejando el resto de ellas en su versión 1. Los resultados se muestran en la Fig. 6.3.

Es posible apreciar que la variación es mínima. De hecho, hay que tener en cuenta que las versiones 2 y 3 realizan, básicamente, el mismo proceso, diferenciándose únicamente en la forma de llevarlo a cabo. Sin embargo, hay que destacar que, mientras que, en la versión que usa la Transformada de Hough, el resultado obtenido no varía en cada ejecución del programa, en las otras dos versiones (que usan RANSAC) sí que se experimenta una pequeña variabilidad, debido al carácter aleatorio que introduce RANSAC.

Esta variabilidad no es lo suficientemente grande como para restar validez a los resultados, por lo que la característica determinante para elegir la mejor



(a) (b) (c)

Figura 6.3. Trayectoria estimada usando cada una de las tres versiones del algoritmo de cálculo del perfil de la calzada: versión 1 (a), versión 2 (b) y versión 3 (c).

versión serán los tiempos de cómputo. Éstos se muestran en la Tabla 6.4 y se comparan en la Fig. 6.4.

	Versión 1	**Versión 2**	**Versión 3**
Tiempo (ms)	32,73	41,67	2,03
Variación vs v.1 (ms)		27,3 %	-93,8 %

Tabla 6.4. Tiempos de cómputo de las distintas versiones para el cálculo del perfil de la calzada.

Los resultados muestran que la versión 3 resulta mucho más eficiente que las otras dos (un 93,8 % más rápida que la primera y un 95,1 % más que la segunda), y habrá de ser, por tanto, el método a utilizar para obtener buenos resultados con el mejor rendimiento.

Para estas medidas se han utilizado parámetros seleccionados de acuerdo al proceso descrito a continuación.

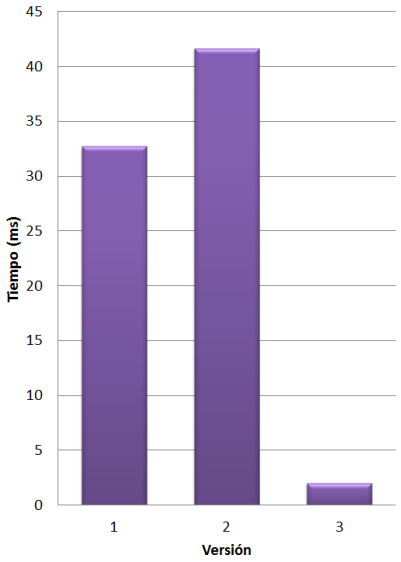
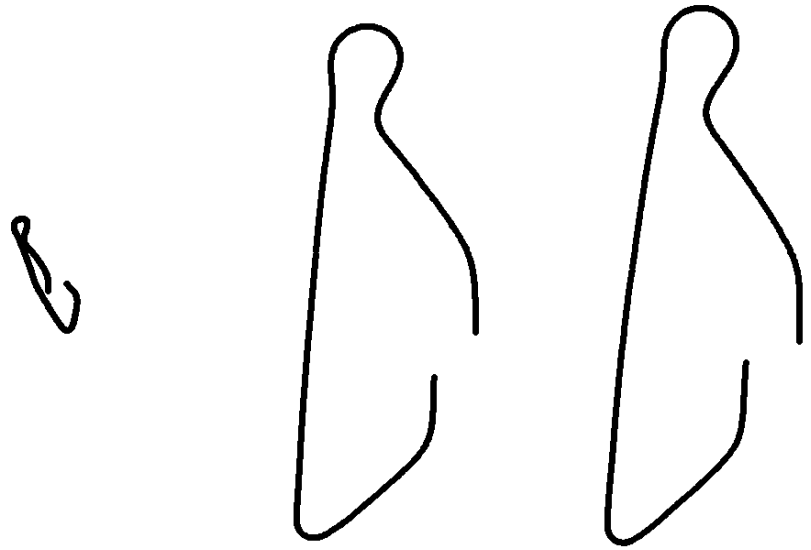


Figura 6.4. Comparativa de los tiempos de cómputo de las distintas versiones para el cálculo del perfil de la calzada.

Número de iteraciones

El número de iteraciones (`RP_NUM_THREADS`) que se ha escogido al ejecutar el RANSAC es de 8192. Responde a la necesidad de encontrar una situación de compromiso entre el incremento de los tiempos de cómputo que supone aumentar las iteraciones, y las imprecisiones que se cometen con un número pequeño de ellas. En la Fig. 6.5 se muestran los resultados obtenidos con varios valores.

Se comprueba como la utilización de un bajo número de iteraciones supone una degradación importante de los resultados; en cambio, valores grandes no mejoran sustancialmente los resultados, incluso cuando el número de iteraciones es tan grande que garantiza escoger todo el conjunto de puntos disponibles, como sucede en la Fig. 6.5(c). En cambio, el tiempo de cómputo sí que aumenta considerablemente, como se muestra en la Tab. 6.5.



(a) (b) (c)

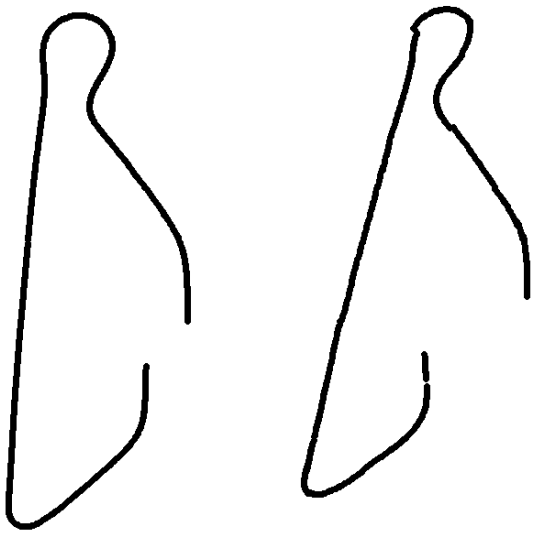
Figura 6.5. Trayectoria estimada usando distintos valores para el límite de iteraciones: 64 (a), 8192 (b) y 32768 (c).

N°iteraciones	**Tiempo (ms)**
64	0,69
8192	2,05
32768	7,72

Tabla 6.5. Comparativa de los tiempos de cómputo requeridos en el cálculo del perfil de la calzada para varios valores del número límite de iteraciones.

Umbral de detección de puntos a nivel alto en el *v-disparity*

El umbral escogido para la detección de puntos a nivel alto en el v-*disparity* (`RP_UMBRAL`) es 128. En general, este valor no tiene excesiva influencia en los resultados, puesto que el perfil de la calzada suele aparecer en él con valores saturados (255), pero se ha decidido reducir la exigencia a que tan solo un 20 % (128/640) de los puntos de una misma fila tengan el mismo valor de disparidad, para obtener mejores resultados en distintas situaciones.



(a) (b)

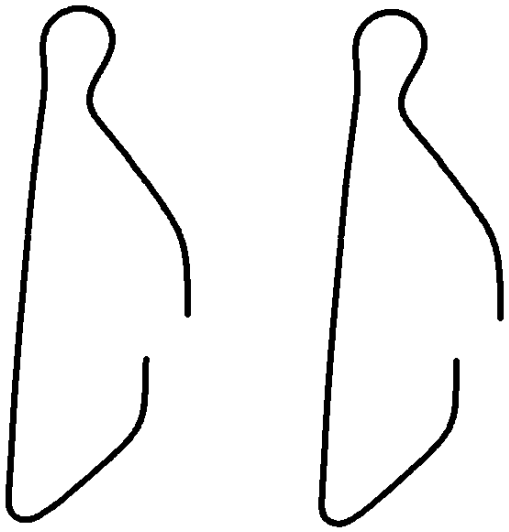
Figura 6.6. Trayectoria obtenida variando el rango de admisión de *inliers* para el RANSAC correspondiente al cálculo del perfil de la calzada, con valores de 1 (a) y de 100 (b).

Tolerancia para la consideración de *inliers*

El rango en torno a la recta dentro del que se consideran los *inliers* (`RP_RANGO`) debe ser lo suficientemente grande como para que siempre haya alguno, pero lo suficientemente pequeño como para no tomar todos los puntos como tales. Los resultados obtenidos en este último caso se ilustran en la Fig. 6.6(b), comparándolos con el valor finalmente escogido de 1, en la Fig. 6.6(a).

6.1.3 Versiones para la detección de puntos característicos

Los datos correspondientes a la versión inicial (Fig. 6.2) muestran que la detección de puntos característicos mediante SURF es la operación más costosa de todas cuantas hay que llevar a cabo en la CPU. Reducir el tiempo de cómputo asociado a ella se presenta, por tanto, como un aspecto vital de cara a optimizar el algoritmo.



(a) (b)

Figura 6.7. Trayectoria estimada usando cada una de las dos versiones del algoritmo de detección de puntos característicos: versión 1 (a) y versión 2 (b).

Como se ha visto en el capítulo 5, se han desarrollado dos posibilidades para su implementación: ejecutar la detección secuencialmente después de obtener el perfil de la calzada y antes de calcular las soluciones de la estimación del movimiento, o ejecutarlo en un proceso individual que comienza a procesar la imagen desde el comienzo del algortimo.

Puede parecer que la segunda versión supone una solución visiblemente más óptima que la primera; no obstante, es necesario valorar los resultados, pues en ella se introducen operaciones de copia de datos que no están presentes en la primera.

Las trayectorias estimadas con cada una de las dos versiones se muestran en la Fig 6.7. Hay que tener en cuenta que, en estas pruebas, ya se utilizó la versión 3 para el cálculo del perfil de la calzada.

En cuanto a los tiempos de cómputo, las pruebas han mostrado que el proceso que detecta los puntos característicos resulta más lento que el resto del algoritmo que se ejecuta en paralelo; es decir, en cada iteración, el proceso principal tiene que esperar a que finalice el proceso auxiliar para poder obtener

los puntos característicos a partir de él. La espera se produce en el semáforo 3 (ver sección 5.3.1).

Por lo tanto, mientras que los tiempos de cómputo de la versión 1 se han podido tomar directamente, los de la versión 2 corresponden a la suma de:

- El tiempo invertido en trasvasar la imagen izquierda al espacio de memoria compartida.
- El tiempo que el proceso principal permanece a la espera de que el proceso auxilar termine su procesamiento.
- El tiempo dedicado a recibir los vectores de pares de puntos a través del espacio de memoria compartida.

Los tiempos medidos en cada una de las versiones se presentan en la Tabla 6.6 y en la Fig. 6.8.

	Versión 1	**Versión 2**
Tiempo (ms)	71,87	10,23
Variación vs v.1 (ms)		-85,8 %

Tabla 6.6. Comparativa de tiempos de cómputo de las distintas versiones para la detección de puntos característicos.

La versión 2 resulta, por lo tanto, más eficiente en la obtención de los puntos característicos. Un desglose más detallado de los tiempos para cada una de las operaciones que tiene lugar en esta etapa se muestra en la Tabla 6.7.

6.1.4 Versiones para el cálculo de soluciones de la estimación del movimiento y para la obtención de la solución única

El análisis de las dos últimas operaciones del algoritmo se va a efectuar en paralelo, debido a que la interacción entre ambas determina los resultados obtenidos. Por otro lado, a diferencia de lo que sucedía en los procesos anteriores, en este caso la reducción del tiempo de cómputo no constituye un

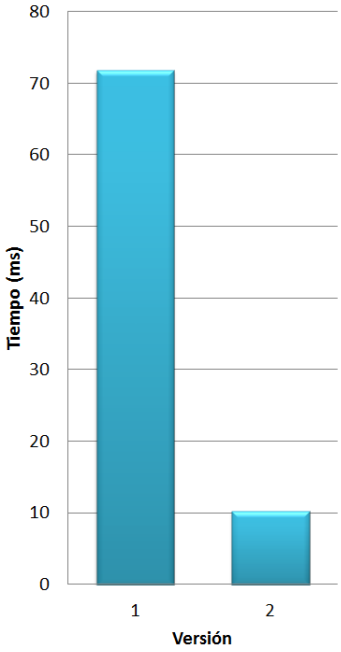


Figura 6.8. Comparativa de los tiempos de cómputo de las distintas versiones para la detección de puntos característicos.

objetivo crítico, puesto que supone un porcentaje muy pequeño respecto al tiempo total del algoritmo (inferior 0,1 % en la versión inicial).

Además, hay que tener en cuenta que, pese a que se tienen 10 combinaciones posibles entre las distintas versiones de los dos procesos, solo hay tres alternativas atendiendo a variaciones en el procedimiento de cálculo, y estas se concentran en la obtención de la solución final: puede hacerse mediante la mediana (versión 1), mediante RANSAC (versiones 2 y 3) o mediante RANSAC modificado de forma que los candidatos se obtengan a partir de la mediana (versiones 4 y 5).

En la Fig. 6.9, se da una muestra del resultado obtenido en sucesivas ejecuciones de prueba con cada una de estas posibilidades. Para los procesos anteriores se ha utilizado la versión 3 del cálculo del perfil de la calzada y la 2 de la detección de puntos característicos. Esa es la razón de que, de nuevo, exista una cierta variabilidad en los resultados (más aún cuando en las versiones de la 2 a la 5 se usa RANSAC por duplicado).

Operación	**Tiempo (ms)**
Copia de la imagen	0,05
Conversión a formato OpenCV	1,41
Extracción de puntos y descriptores	44,99
Emparejamiento de puntos	25,38
Traspaso de vectores	0,04
Total	**71,87**

Tabla 6.7. Tiempos de cómputo de las distintas operaciones para la detección de puntos característicos.

6.1.5 Versiones para la obtención de una solución única

Se puede ver una cierta mejora de los resultados (la trayectoria está más próxima a cerrarse) al usar RANSAC (Fig. 6.9(b)), así como al añadir la mediana en la selección de resultados (Fig. 6.9(c)). Conviene, por tanto, usar las versiones 4 ó 5 y, entre ellas dos, la más óptima, por ejecutar el proceso en paralelo, es la versión 5 (RANSAC en la GPU).

Utilizar la versión 5 para el hallazgo de la solución única supone que la versión 2 del cálculo de resultados (en paralelo en la GPU) pasa a ser la más conveniente de entre las dos que existen, por ejecutar el cálculo en paralelo y, adicionalmente, por evitar operaciones de copia de memoria desde la CPU a la GPU que serían necesarias en caso de usar la versión 1.

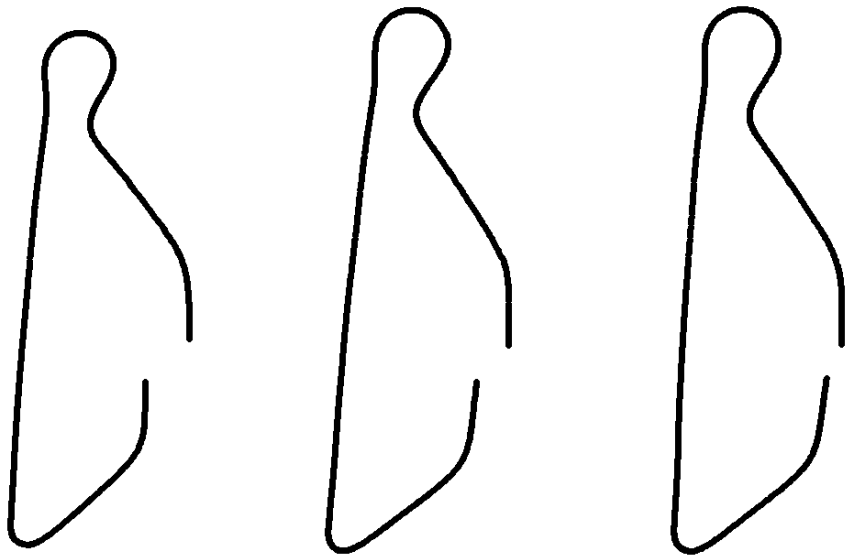
Con esto, los tiempos de cómputo del conjunto de operaciones resultan los de la Tabla 6.8.

	Versión inicial	**Versión final**
Tiempo (ms)	0,15	0,67

Tabla 6.8. Comparativa de tiempos de cómputo para las distintas versiones para el cálculo de resultados y la obtención de una solución única.

Se ha obtenido una mejora en los resultados a cambio de un cierto incremento del tiempo de cómputo. No obstante, este sigue siendo notablemente inferior al que requieren los procesos anteriores.

El proceso de selección de los parámetros idóneos en RANSAC se detalla a continuación:



(a) (b) (c)

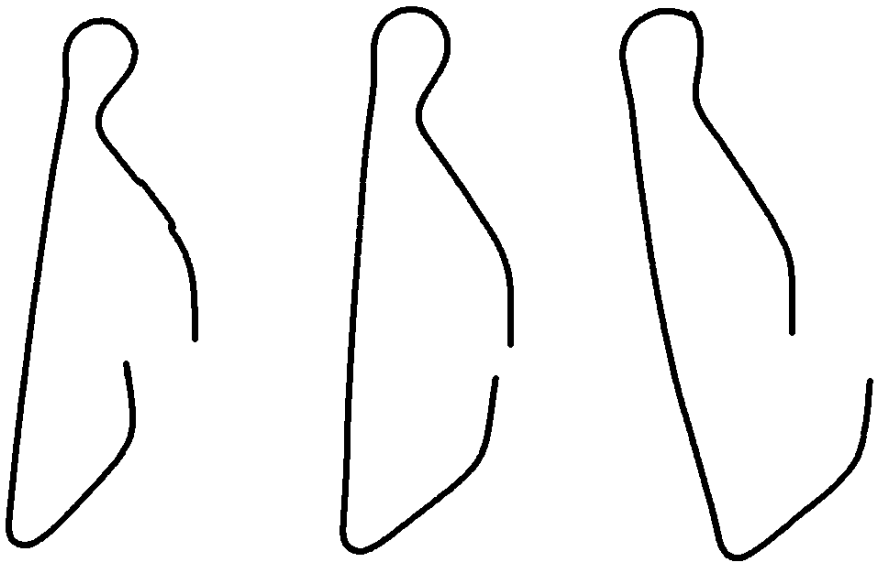
Figura 6.9. Trayectoria estimada usando cada una de las cinco versiones del algoritmo de obtención de una solución única: versión 1 (a), versiones 2 y 3 (b) y versiones 4 y 5 (c).

Selección de la tolerancia para RANSAC

Las pruebas se han efectuado admitiendo *inliers* en un rango de 0,01 rad (para θ) y de 0,01 m (para X_T e Y_T) en torno a los candidatos seleccionados aleatoriamente en RANSAC (parámetros `RANGO_THETA`, `RANGO_X` y `RANGO_Y`). Se trata de un valor lo suficientemente grande como para abarcar un número razonable de puntos alrededor; pero no tan grande como para aceptarlos todos o casi todos. Al alejarse del valor propuesto, se obtienen los resultados de la Fig. 6.10, visiblemente peores.

Selección del número de iteraciones para RANSAC

El valor elegido para el número de *threads* a lanzar (`OD_NUM_THREADS`) ha sido de 512. Esto supone que, para cada uno de los tres parámetros (θ, Z_T y X_T), se evaluarán 170 modelos (lanzando 512 hilos en la GPU en el caso de las versiones 3 y 5 de la búsqueda de la solución única). Hay que tener en



(a) (b) (c)

Figura 6.10. Trayectoria estimada variando el rango de admisión de *inliers* para el RANSAC en la elección de la mejor solución, con valores de 0,001 rad/m (a), de 0,01 rad/m (b) y 0,02 rad/m (c).

cuenta que el número de pares de puntos característicos que se han detectado en las pruebas es inferior al millar. Además, lanzar 512 hilos resulta acorde con las recomendaciones de CUDA en cuanto al número de *threads* por bloque, al ser múltiplo de 32 e igual, además, al número máximo de hilos por bloque que soporta la tarjeta del equipo de pruebas.

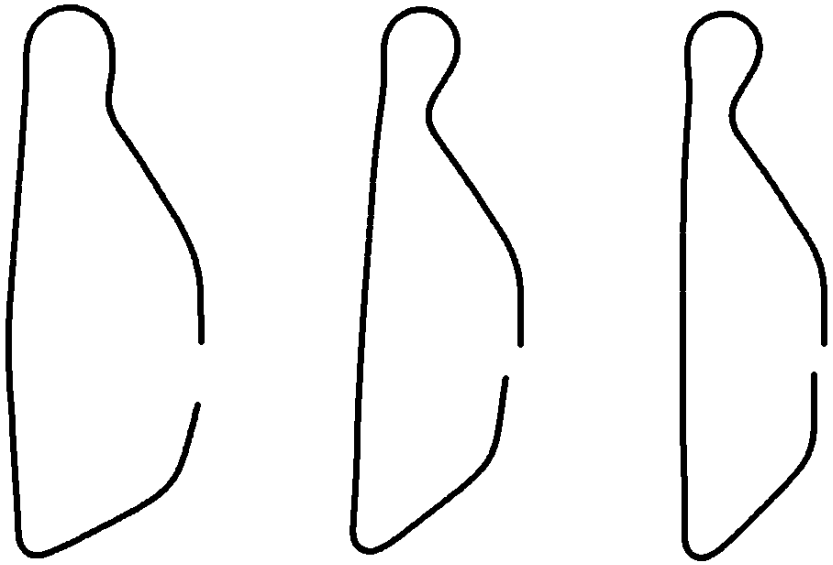
Valores mayores proporcionan resultados similares requiriendo mayor tiempo de cómputo, mientras que valores inferiores no consideran el número suficiente de puntos como para obtener una solución satisfactoria. Las anteriores aseveraciones se pueden comprobar en los datos de la tabla 6.9 y de la Fig. 6.11, referidos a la versión 5.

6.1.6 Versión final

Las versiones elegidas para componer el algoritmo final se muestran recuadradas en la Fig. 6.12. A continuación, se ofrece un análisis detallado del comportamiento general del algoritmo compuesto por ellas.

N°iteraciones	Tiempo (ms)
64	0,66
512	0,67
2048	0,87

Tabla 6.9. Comparativa de los tiempos de cómputos requeridos en el cálculo de la odometría para distintos valores del número de hilos de procesamiento lanzados.



(a) (b) (c)

Figura 6.11. Trayectoria estimada para distintos valores del número de hilos de procesamiento lanzados: 64 (a), 512 (b) y 2048 (c).

Tiempo de computo

La versión final del algoritmo ha conseguido reducir de manera importante el tiempo de cómputo necesario para calcular la estimación del movimiento del vehículo, como se muestra en la Tabla 6.10[1] y en la Fig. 6.13.

El tiempo de procesamiento del programa final se ha reducido en un 43,71 % respecto al original. Sin tener en cuenta el tiempo necesario para la captura de las imágenes por parte del sistema estéreo, se ha dotado al sistema

[1]En la tabla solo se presentan las variaciones en el tiempo de cómputo de aquellas secciones que han sido modificadas en el presente proyecto

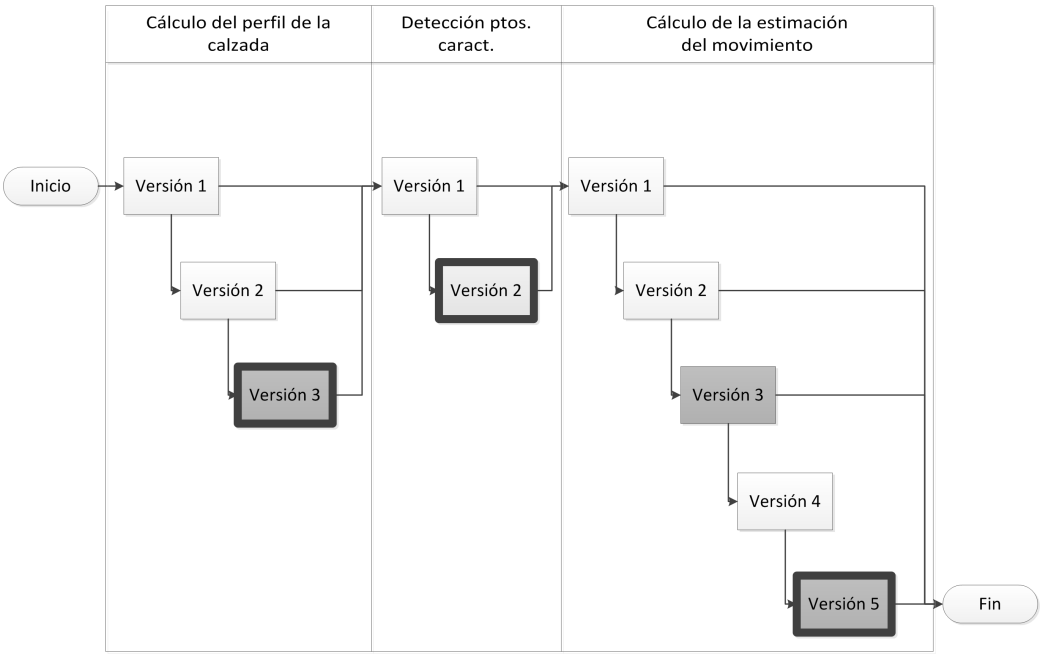


Figura 6.12. Esquema general de versiones para cada una de las etapas del algoritmo. En gris oscuro, las implementadas con CUDA y en gris claro, las que conllevan paralelización en Windows. Remarcadas con línea gruesa las versiones seleccionadas.

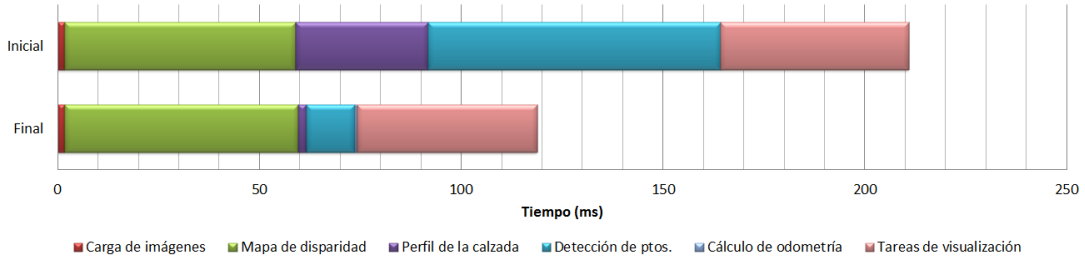


Figura 6.13. Diagrama comparativo entre los tiempos de cómputo de la versión inicial y los de la final.

Operación	Tiempos(ms)		Variación
	Ver. Inicial	**Ver. Final**	
Carga de imágenes	1,81	1,79	
Mapa de disparidad	57,09	57,65	
Perfil de la calzada	32,73	2,05	-93,74 %
Detección de ptos.	72,46	11,99	-83,45 %
Cálculo de la odometria	0,15	0,67	355,84 %
Total sin visualización	164,24	86,14	-47,55 %
Procesamiento de imágenes	14,81	13,02	
Mostrar ptos. car.	31,03	30,61	
Mostrar mapa	0,76	0,90	
Tareas visualización	46,59	44,53	
Total	**210,83**	**130,66**	**-43,71 %**

Tabla 6.10. Comparativa entre los tiempos de cómputo de las distintas operaciones en versión final frente a los correspondientes a la versión inicial.

de la capacidad de procesar 11,6 imágenes por segundo (7,7 si se quieren visualizar los resultados) frente a las 7 imágenes por segundo que se lograban al comienzo (4,7 mostrando resultados).

Hay que tener en cuenta que, en los cerca de 80 ms ahorrados, un vehículo circulando a 120 km/h puede recorrer 2,67 metros. Poder contar con una información más actualizada del entorno es clave en determinados sistemas de ayuda a la conducción; especialmente aquellos que forman parte de los sistemas de seguridad del vehículo.

Precisión

La Fig. 6.14 permite valorar la precisión del resultado obtenido con el algoritmo. El error promedio cometido en 10 muestras, medido como la distancia entre el punto de inicio de la trayectoria estimada y su punto final, se muestra en la tabla 6.11, incluyendo en la última columna el error respecto a la estimación de la distancia total recorrida. Asimismo, se aprecia que dicha estimación no se ciñe con total exactitud al trazado de las calles, pero permanece en un margen de error de unos 10 metros.

Figura 6.14. Trayectoria estimada por el algoritmo sobre una imagen de satélite del recorrido.

Error absoluto (m)	Distancia total estimada (%)	Error relativo (%)
14,88 m	441,01 m	3,37 %

Tabla 6.11. Error cometido con el algoritmo.

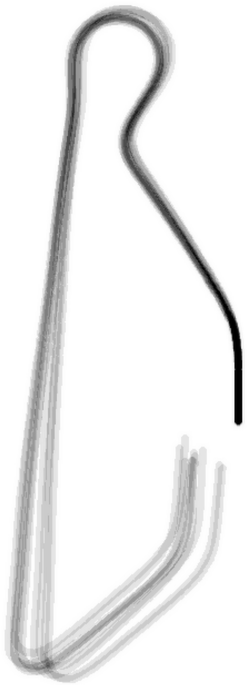


Figura 6.15. Superposición de varias trayectorias obtenidas en distintas ejecuciones del algoritmo.

Repetibilidad

La variabilidad de los resultados obtenidos con el sistema se plasma en la Fig. 6.15, donde se ponen en común los resultados de diez ejecuciones sucesivas.

La desviación típica del error cometido en dichas iteraciones es de 2,5 m. Esta variabilidad se explica por el uso de la filosofía RANSAC que se hace en el algoritmo, por el cual el cálculo del perfil de la calzada y la elección de la mejor solución no tienen en cuenta todas las posibilidades, sino tan solo un conjunto aleatorio de ellas.

Hay que tener en cuenta que el algoritmo únicamente es capaz de inferir desplazamientos relativos, sin contar con la posición absoluta. Por ello, el error se va acumulando a lo largo de la trayectoria.

7 Conclusiones y Trabajos Futuros

A continuación se valora el grado de adecuación del presente trabajo a los objetivos que se perseguían al comienzo del mismo, así como varias propuestas de desarrollo para mejorar los resultados obtenidos.

7.1 Cumplimiento de objetivos

Con el algoritmo desarrollado e implementado, se pretendía conseguir una estimación del movimiento bidimensional de un vehículo utilizando un sistema estéreo de visión. Uno de los objetivos era que el sistema fuese capaz de funcionar en entornos urbanos.

Frente a la localización basada en el uso de GPS, que sufre pérdidas de la información debidas a la presencia de edificios altos, árboles, túneles y otros elementos sólidos alrededor del vehículo, el sistema desarrollado proporciona información de forma continua sobre la trayectoria del vehículo.

El error cometido en la estimación de un camino cerrado oscila alrededor del 3 %, lo que demuestra que el sistema funciona con una precisión lo suficientemente alta como para poder ser tenido en cuenta en aplicaciones de ayuda a la conducción.

Además, usar el perfil de la calzada y el tercio inferior de la calzada, permite que solo se usen como referencia puntos pertenecientes a la calzada,

reduciendo la incertidumbre en la estimación de la profundidad al tratarse de puntos cercanos al vehículo y evitando que sean considerados puntos pertenecientes a otros vehículos en movimiento.

Conviene resaltar, por último, que la utilización de técnicas de computación paralela y procesamiento en GPU ha conseguido acelerar la ejecución del programa en más del 40 % respecto a su alternativa inicial, consistente en su ejecución secuencial en la CPU.

7.2 Trabajos futuros

El sistema aquí expuesto tiene un amplio margen de desarrollo en futuros trabajos. Una de las posibles mejoras, que permitiría reducir en gran medida el error y la variabilidad de los resultados, consiste en implementar un método para autocalibrar los parámetros extrínsecos del sistema estéreo; esto es, su orientación respecto a la calzada, dada por los ángulos *pitch*, *yaw* y *roll* [67]. Esto permitiría estimar fácilmente la variación de la posición del sistema estéreo, a diferencia de lo que ocurre en el sistema actual, que necesita un conocimiento a priori de estos ángulos al ser extremadamente sensible a cambios en ellos.

Otra aportación interesante podría ser fusionar la estimación de posición proporcionada por el sistema de odometría visual con los datos obtenidos por un receptor GPS. Esto permitiría evitar los problemas de ambos sistemas:

- Las pérdidas de señal que sufre el GPS en entornos urbanos.
- La falta de información sobre la posición absoluta de la que adolece la odometría visual, que lleva a acumular errores durante el movimiento.

8 Costes del Proyecto

En este capítulo se presenta una relación aproximada de los costes en los que se ha incurrido durante la elaboración del proyecto. Se incluyen tanto costes de personal como de materiales.

Para lo primero, es necesario realizar una estimación del tiempo que ha sido necesario dedicar al proyecto. Este se adjunta en la siguiente tabla (8.1):

Etapa	**Tiempo**
Estudio y comprensión del problema a resolver	50 h
Familiarización con las herramientas (CUDA, OpenCV,...)	30 h
Implementación del código	400 h
Pruebas y corrección de errores	50 h
Ajuste de parámetros y mejoras	100 h
Obtención de resultados	20 h
Memoria del proyecto	100 h
Total	**750 h**

Tabla 8.1. Tiempo estimado para cada fase del proyecto

En cuanto al coste de personal, si se le asigna un salario bruto de $30€/h$ al ingeniero responsable del proyecto, resulta:

$$750\,h \cdot 30\,€/h = 22\,500\,€ \tag{8.1}$$

Por otro lado, es necesario contar con los materiales descritos en la siguiente tabla (8.2):

Material	Coste
Ordenador de sobremesa (con GPU soportando CUDA)	1 200 €
Sistema estéreo (*Point Grey Bumblebee*)	3 000 €
Bibliotecas MIL	500 €
Bibliotecas OpenCV, Visual C++	Gratuito
Otro material (cables, soportes, etc.)	100 €
Total	**4 800 €**

Tabla 8.2. Estimación de los costes materiales del proyecto

El presupuesto total destinado al proyecto será (tabla 8.3):

Coste de personal	22 500 €
Coste de material	4 800 €
Total	**27 300 €**

Tabla 8.3. Estimación de los costes materiales del proyecto

Bibliografía

[1] W. H. Organization, "The world health report 2004 - changing history," 2004.

[2] D. Mohan, "Social cost of road traffic crashes in India," in *Proceedings First Safe Community Conference on Cost of Injury. Viborg, Denmark*, pp. 33–38, 2002.

[3] Eurostat, "People killed in road accidents," 2012.

[4] D. G. de Tráfico, "Las principales cifras de la siniestralidad vial en España," 2010.

[5] G. Leen and D. Heffernan, "Expanding automotive electronic systems," *Computer*, vol. 35, no. 1, pp. 88–93, 2002.

[6] O. Carsten and L. Nilsson, "Safety assessment of driver assistance systems," *European Journal of Transport and Infrastructure Research*, vol. 1, no. 3, pp. 225–243, 2001.

[7] R. Bishop, "Intelligent vehicle applications worldwide," *Intelligent Systems and their Applications, IEEE*, vol. 15, no. 1, pp. 78–81, 2000.

[8] J. Collado, C. Hilario, J. Armingol, and A. De la Escalera, "Visión por computador para vehículos inteligentes," *XXIV Jornadas de Automática, León, España*, 2003.

[9] Volkswagen, "City Emergency Braking." http://www.volkswagen.co.uk/technology/braking-and-stability-systems/city-emergency-braking. [Online; disponible el 1 de octubre de 2012].

[10] Opel, "Advanced driving assistance and active safety systems." http://media.opel.com/media/intl/en/opel/vehicles/AFL/2009.html. [Online; disponible el 1 de octubre de 2012].

[11] Mercedes-Benz, "Mercedes-Benz to introduce attention assist into series production in spring 2009." http://www.emercedesbenz.com/Aug08/11_001331_Mercedes_Benz_To_Introduce_Attention_Assist_Into_Series_Production_In_Spring_2009.html. [Online; disponible el 1 de octubre de 2012].

[12] Bosch, "Bosch predictive emergency braking system goes into series production." http://www.bosch-presse.de/presseforum/details.htm?txtID=4570&locale=en. [Online; disponible el 1 de octubre de 2012].

[13] Google, "The self-driving car logs more miles on new wheels." http://googleblog.blogspot.hu/2012/08/the-self-driving-car-logs-more-miles-on.html. [Online; disponible el 1 de octubre de 2012].

[14] N. Y. Times, "Google cars drive themselves, in traffic." http://www.nytimes.com/2010/10/10/science/10google.html?_r=3. [Online; disponible el 1 de octubre de 2012].

[15] UPM-CSIC, "Autopia."

[16] E. Mundo, "Un vehículo recorre 100 km sin conductor." www.elmundo.es/elmundo/2012/06/10/ciencia/1339335540.html. [Online; disponible el 1 de octubre de 2012].

[17] M. Skolnik, "Introduction to radar," *Radar Handbook*, 1962.

[18] J. Armingol, A. de la Escalera, C. Hilario, J. Collado, J. Carrasco, M. Flores, J. Pastor, and F. Rodríguez, "Ivvi: Intelligent vehicle based on visual information," *Robotics and Autonomous Systems*, vol. 55, no. 12, pp. 904–916, 2007.

[19] M. Bertozzi, A. Broggi, and A. Fascioli, "Vision-based intelligent vehicles: State of the art and perspectives," *Robotics and Autonomous systems*, vol. 32, no. 1, pp. 1–16, 2000.

[20] P. Grey, "Bumblebee." `http://www.ptgrey.com/products/bumblebee2/bumblebee2_stereo_camera.asp`. [Online; disponible el 1 de octubre de 2012].

[21] M. Quddus, W. Ochieng, and R. Noland, "Current map-matching algorithms for transport applications: State-of-the art and future research directions," *Transportation Research Part C: Emerging Technologies*, vol. 15, no. 5, pp. 312–328, 2007.

[22] C. Wu, Y. Meng, L. Zhi-lin, C. Yong-qi, and J. Chao, "Tight integration of digital map and in-vehicle positioning unit for car navigation in urban areas," *Wuhan University Journal of Natural Sciences*, vol. 8, no. 2, pp. 551–556, 2003.

[23] Garmin, "What is GPS?." `http://www8.garmin.com/aboutGPS/`. [Online; disponible el 1 de octubre de 2012].

[24] Kowoma.de, "Sources of errors in GPS." `http://www.kowoma.de/en/gps/errors.htm`. [Online; disponible el 1 de octubre de 2012].

[25] R. García and M. Vázquez, *Sistema de Odometría Visual para la Mejora del Posicionamiento Global de un Vehículo*. 2007.

[26] G. Stein, O. Mano, and A. Shashua, "A robust method for computing vehicle ego-motion," in *Intelligent Vehicles Symposium, 2000. IV 2000. Proceedings of the IEEE*, pp. 362–368, IEEE, 2000.

[27] E. Mouragnon, M. Lhuillier, M. Dhome, F. Dekeyser, and P. Sayd, "Real time localization and 3D reconstruction," in *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, vol. 1, pp. 363–370, IEEE, 2006.

[28] C. Harris and M. Stephens, "A combined corner and edge detector," in *Alvey vision conference*, vol. 15, p. 50, Manchester, UK, 1988.

[29] D. Scaramuzza and R. Siegwart, "Appearance-guided monocular omnidirectional visual odometry for outdoor ground vehicles," *Robotics, IEEE Transactions on*, vol. 24, no. 5, pp. 1015–1026, 2008.

[30] F. Labrosse, "The visual compass: Performance and limitations of an appearance-based method," *Journal of Field Robotics*, vol. 23, no. 10, pp. 913–941, 2006.

[31] D. Demirdjian and T. Darrell, "Motion estimation from disparity images," in *Computer Vision, 2001. ICCV 2001. Proceedings. Eighth IEEE International Conference on*, vol. 1, pp. 213–218, IEEE, 2001.

[32] R. Hartley and A. Zisserman, *Multiple view geometry in computer vision*, vol. 2. Cambridge Univ Press, 2000.

[33] D. Nistér, O. Naroditsky, and J. Bergen, "Visual odometry," in *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, vol. 1, pp. I–652, IEEE, 2004.

[34] A. Howard, "Real-time stereo visual odometry for autonomous ground vehicles," in *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pp. 3946–3952, Ieee, 2008.

[35] I. Parra, M. Sotelo, D. Llorca, and M. Ocana, "Robust visual odometry for vehicle localization in urban environments," *Robotica*, vol. 28, no. 3, p. 441, 2010.

[36] M. Maimone, Y. Cheng, and L. Matthies, "Two years of visual odometry on the Mars exploration rovers," *Journal of Field Robotics*, vol. 24, no. 3, pp. 169–186, 2007.

[37] NASA, "Curiosity visual odometry," 2012. [Online; disponible el 20 septiembre de 2012].

[38] A. De La Escalera, *Visión por computador: Fundamentos y métodos*. Prentice Hall, 2001.

[39] J. González, *Visión por computador*. Paraninfo, 2000.

[40] D. Scharstein and R. Szeliski, "A taxonomy and evaluation of dense two-frame stereo correspondence algorithms," *International journal of computer vision*, vol. 47, no. 1, pp. 7–42, 2002.

[41] B. Musleh, A. de la Escalera, and J. Armingol, "UV disparity analysis in urban environments," *Computer Aided Systems Theory–EUROCAST 2011*, pp. 426–432, 2012.

[42] D. Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[43] H. Bay, T. Tuytelaars, and L. Van Gool, "SURF: Speeded Up Robust Features," *Computer Vision–ECCV 2006*, pp. 404–417, 2006.

[44] A. Murillo, J. Guerrero, and C. Sagues, "Surf features for efficient robot localization with omnidirectional images," in *Robotics and Automation, 2007 IEEE International Conference on*, pp. 3901–3907, IEEE, 2007.

[45] M. Fischler and R. Bolles, "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography," *Communications of the ACM*, vol. 24, no. 6, pp. 381–395, 1981.

[46] M. Zuliani, "RANSAC for dummies," *With examples using the RANSAC toolbox for Matlab and more*, 2009.

[47] S. Choi, T. Kim, and W. Yu, "Performance evaluation of RANSAC family," in *Proceedings of the British Machine Vision Conference*, 2009.

[48] P. Torr and A. Zisserman, "MLESAC: A new robust estimator with application to estimating image geometry," *Computer Vision and Image Understanding*, vol. 78, pp. 138–156, 2000.

[49] B. Tordoff and D. Murray, "Guided-MLESAC: Faster image transform estimation by using matching priors," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 27, no. 10, pp. 1523–1535, 2005.

[50] O. Chum and J. Matas, "Matching with PROSAC-progressive sample consensus," in *Computer Vision and Pattern Recognition, 2005. CVPR*

2005. IEEE Computer Society Conference on, vol. 1, pp. 220–226, IEEE, 2005.

[51] O. Chum and J. Matas, "Randomized RANSAC with Td, d test," in *Proc. British Machine Vision Conference*, vol. 2, pp. 448–457, 2002.

[52] D. Nistér, "Preemptive RANSAC for live structure and motion estimation," *Machine Vision and Applications*, vol. 16, no. 5, pp. 321–329, 2005.

[53] R. Toldo and A. Fusiello, "Robust multiple structures estimation with J-Linkage," *Computer Vision–ECCV 2008*, pp. 537–547, 2008.

[54] Microsoft, "Visual Studio." `http://www.microsoft.com/visualstudio/eng/products/visual-studio-overview`. [Online; disponible el 1 de octubre de 2012].

[55] M. Imaging, "Matrox Imaging Library." `http://www.matrox.com/imaging/en/products/software/`. [Online; disponible el 1 de octubre de 2012].

[56] W. Garage, "OpenCV Wiki." `http://opencv.willowgarage.com/wiki/FullOpenCVWiki`. [Online; disponible el 1 de octubre de 2012].

[57] C. Zeller, "NVIDIA Tutorial CUDA," 2008.

[58] NVIDIA, "CUDA parallel computing platform." `http://www.nvidia.com/object/cuda_home_new.html`. [Online; disponible el 1 de octubre de 2012].

[59] G. Bradski and A. Kaehler, *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Incorporated, 2008.

[60] B. Musleh, D. Martin, A. de la Escalera, and J. Armingol, "Visual ego motion estimation in urban environments based on uv disparity," in *Intelligent Vehicles Symposium (IV), 2012 IEEE*, pp. 444–449, IEEE, 2012.

[61] A. Martín Clemente *et al.*, "Generación de mapas de disparidad utilizando CUDA," 2009.

[62] V. Jiménez Monje, "Detección y localización de obstáculos en entornos urbanos mediante visión estéreo," 2011.

[63] A. Broggi, C. Caraffi, R. Fedriga, and P. Grisleri, "Obstacle detection with stereo vision for off-road vehicle navigation," in *Computer Vision and Pattern Recognition-Workshops, 2005. CVPR Workshops. IEEE Computer Society Conference on*, pp. 65–65, IEEE, 2005.

[64] R. Labayrade, D. Aubert, and J. Tarel, "Real time obstacle detection in stereovision on non flat road geometry through v-disparity representation," in *Intelligent Vehicle Symposium, 2002. IEEE*, vol. 2, pp. 646–651, Ieee, 2002.

[65] W. Garage, "OpenCV 2.0 c reference. feature detection." `http://opencv.willowgarage.com/documentation/feature_detection.html`. [Online; disponible el 1 de octubre de 2012].

[66] J. Borenstein, H. Everett, and L. Feng, "Where am i? sensors and methods for mobile robot positioning," *University of Michigan*, vol. 119, p. 120, 1996.

[67] I. Vázquez García, "Desarrollo de un algoritmo de auto-calibración de sistemas estéreo basado en el uso del uv-disparity. Aplicación a la odometria visual.," 2012.

APÉNDICES

A

Código del programa.

A.1 Parámetros

En este apéndice, se ofrece el código fuente del programa que implementa el algoritmo de odometría visual mostrado en este proyecto. El código mostrado incluye todas las versiones del algoritmo, pudiéndose elegir entre una u otra mediante definiciones para el preprocesador (`#define`).

En este primer apartado, se ofrecen los archivos de cabecera modificables para cambiar los parámetros de funcionamiento del programa. El primero corresponde al programa principal; el segundo, al programa auxiliar que detecta los puntos.

```
/**********************************/
/* PARÁMETROS DEL PROG. PRINCIPAL  */
/**********************************/

//Para usar la secuencia de prueba
#define OFF_LINE

//Para usar FLANN en el emparejamiento de puntos
#define USE_FLANN

//Para usar la versión 3 del cálculo de perfil de la calzada
#define PERFIL_V3
//Alternativa: PERFIL_V2. Para v.1, no poner nada.

//Descomentar para visualizar el perfil de la calzada
//#define MOSTRAR_PERFIL
```

```
//Para usar la versión 2 de la detección de puntos característicos
#define SURF_V2
//Para v.1, no poner nada

//Parámetros para las memorias compartidas
#define MAX_PTOS_SURF 2048  //Máx. número de puntos
//Tamaño para la transmisión de imagen
#define BUF_SIZE WIDTH_IMAGEN*HEIGHT_IMAGEN*sizeof(unsigned char)
//Tamaño para los vectores
#define BUF_SIZE2 (MAX_PTOS_SURF+1)*sizeof(Point2f)

//Para visualizar los puntos detectados en la calzada
#define MOSTRAR_SURF

//Para usar la versión 2 del cálculo de soluciones
#define CALPTOS_V2
//Para v.1, no poner nada

//Para usar la versión 5 de la obtención de la solución única
#define CALSOL_V5
//Alternativas: CALSOL_V2, CALSOL_V3, CALSOL_V4. Para v.1, no poner nada.

//Para descartar grandes desplazamientos
#define FILTRO
#define UMBRAL_FILTRO 2 //Umbral

//NO está implementado CALPTOS_V2 sin alguna CALSOL distinta de V1

//Parámetros de las imágenes
#define WIDTH_IMAGEN 640
#define HEIGHT_IMAGEN 480
#define NUM_IMAGENES 982

//Parámetros para el mapa de disparidad
#define ROWSperTHREAD 40
#define BLOCK_W 128
#define ANCHO_BLOQUE 64
#define RADIUS_H 8
#define RADIUS_V 8
#define MIN_SSD ((RADIUS_H*2+1)*(RADIUS_V*2+1)*255)
#define STEREO_MIND 0.0f
#define STEREO_MAXD 30.0f
#define SHARED_MEM_SIZE ((BLOCK_W + 2*RADIUS_H)*sizeof(int) )
#define VALOR_INICIAL 0
#define U_DISP_UMBRALIZADO
#define UMBRAL_AREA_ESTUDIO 1

//Parámetros para el perfil de la calzada
//Número máximo de píxeles a nivel alto en el v-disparity
//Está ajustado al tamaño total del mismo
```

```
#define MAX_PTOS_BLANCOS 14400

//Número de iteraciones en el RANSAC
#define RP_NUM_THREADS 8192

//Distancia de inclusión de inliers en torno a la recta
#define RP_RANGO 1

//Umbral para la consideración de puntos a nivel alto en el v-disparity.
#define RP_UMBRAL 128

//Parámetros para la estimación del movimiento
//Parámetros intrínsecos de la cámara
#define B 0.119915f   //Baseline
#define F 811.9104f   //Distancia focal
#define CX 322.0614f  //Coord. x del centro óptico
#define CY 247.6637f  //Coord. y del centro óptico

#define DESVIACION_YAW 0.0547f  //Desviación del yaw (par. extrínseco)

//Distancia a partir de la que se detectan puntos car.
#define ALTURA_SURF (HEIGHT_IMAGEN-(int)(HEIGHT_IMAGEN/3))

//Parámetros para el RANSAC de la obtención de la mejor solución
#define OD_NUM_THREADS 2048 //Iteraciones
//Distancias de consideración de inliers
#define RANGO_THETA 0.01f
#define RANGO_X 0.01f
#define RANGO_Y 0.01f

//Visualización de la trayectoria
#define RESOLUCION_MAPA 3 //Resolución
#define ALTO_MAPA 800   //Tamaño
#define ANCHO_MAPA 1000
```

```
/**********************************/
/* PARÁMETROS DEL PROG. AUXILIAR   */
/**********************************/
//Dimensiones de las imágenes
#define WIDTH_IMAGEN 640
#define HEIGHT_IMAGEN 480
//Distancia a partir de la que se detectan puntos característicos
//Debe coincidir con la especificada en el programa principal.
#define ALTURA_SURF (HEIGHT_IMAGEN-(int)(HEIGHT_IMAGEN/3))

//Parámetros para la memoria compartida
//Máx. número de ptos.
#define MAX_PTOS_SURF 2048
//Tamaños
#define BUF_SIZE WIDTH_IMAGEN*HEIGHT_IMAGEN*sizeof(unsigned char) //Imagen
```

```
#define BUF_SIZE2 (MAX_PTOS_SURF+1)*sizeof(Point2f)          //Vectores
```

A.2 Código del main del programa principal

A continuación, se ofrece el código correspondiente al `main` del programa principal, así como de las funciones que no hacen uso de la GPU. También aparecen los temporizadores utilizados para realizar las pruebas. Este programa se ejecuta íntegramente sobre la CPU.

```
////////////////////
// INCLUSIONES
////////////////////
//Inclusión de las bibliotecas del sistema
#include <iostream>
#include <windows.h>
#include <tchar.h>
#include <fstream>  //para el manejo de ficheros
using namespace std;

//Inclusiones de CUDA
#include <vector_types.h>
#include "cutil.h"

//Inclusión de las bibliotecas de visión
//Matrox Imaging Libraries
#include <mil.h>
//OpenCV
#include <opencv2/features2d/features2d.hpp>
#include <opencv2/nonfree/nonfree.hpp>
#include <cv.h>
#include <cxcore.h>
#include <highgui.h>
using namespace cv;

//Inclusión de la biblioteca de cámaras point_grey
#include "point_grey.h"
#include "bumblebee.h"

//Inclusión de las funciones para trabajar off-line
#include "off_line.h"
#include "deteccion_obstaculos.h"
#include "graficos_overlay.h"
```

```
//Inclusión de los parametros de control
#include "parametros.h"

//Inclusión del funciones complementarias para el SURF
#include "funcionesSURF.cpp"

////////////////////
// CLASES, TIPOS
////////////////////
//Clase para el perfil de la calzada y el pitch
class PerfilCalzada{
public:
  float m;
  float horizonte;
  float pitch;
  PerfilCalzada(){m=0;horizonte=0;pitch=0;}
};

struct myclass{
  bool  operator() (float i,float j) {return (i<j);}
}myobject;

//Temporizador
class Timer{
public:

  double PCFreq;
  __int64 CounterStart;
  string nombre;

  Timer(string nombre){
    this->nombre=nombre;

    LARGE_INTEGER li;

    if (!QueryPerformanceFrequency(&li))
      cout << "QPF failed\n";

    PCFreq = double (li.QuadPart)/1000.0;

    QueryPerformanceCounter(&li);
    CounterStart = li.QuadPart;

  }
  void parar(){
    LARGE_INTEGER li;

    QueryPerformanceCounter(&li);
    double tiempo=(li.QuadPart-CounterStart)/PCFreq;
    cout << nombre << " processing time: " << tiempo << " <ms>" << endl;
  }
```

```
  void parar(ofstream &f){
    LARGE_INTEGER li;

    QueryPerformanceCounter(&li);
    double tiempo=(li.QuadPart-CounterStart)/PCFreq;
    f<<nombre<<" - "<<tiempo<<" ms"<<endl;
  }
};

class TimerMedia{
public:

  double PCFreq;
  __int64 CounterStart;
  string nombre;
  double acTiempo;
  int veces;
  LARGE_INTEGER li;

  TimerMedia(string nombre){
    this->nombre=nombre;

    if (!QueryPerformanceFrequency(&li))
      cout << "QPF failed\n";

    PCFreq = double (li.QuadPart)/1000.0;

    acTiempo=0;
    veces=0;

  }

  void Iniciar(){
    QueryPerformanceCounter(&li);
    CounterStart = li.QuadPart;
  }

  void Parar(){
    QueryPerformanceCounter(&li);
    double tiempo=(li.QuadPart-CounterStart)/PCFreq;
    veces++;
    if (veces>1){
      acTiempo+=tiempo;
    }
  }
  void Escribir(ofstream &f){
    f<<nombre<<" - "<<acTiempo/(veces-1)<<" ms"<<endl;
  }
};

//////////////////
```

```
// PROTOTIPOS
//////////////////
//Funciones cppintegration.cu
extern "C" void runTest(int argc, char** argv,unsigned char* h_LG_izq,
    unsigned char* h_LG_der,unsigned char* h_u,unsigned char* h_v,unsigned
    char* h_izq,unsigned char* h_der,int width,int height);
extern "C" void ransac(unsigned char * h_v, PerfilCalzada &perfil,float
    rango, unsigned int umbral,int ancho, int alto);
void ransacCPU(unsigned char * h_v, PerfilCalzada &perfil,float rango,
    unsigned int umbral,int ancho, int alto);
void Calculo_perfil(IplImage* v,float* m,float* horizonte,float* pitch,bool
    imprimir,Overlay* dibujo);

extern "C" void odometria(PerfilCalzada perfil, PerfilCalzada
    perfil_anterior, Point2f* v_pt1, Point2f* v_pt2, vector<float> &theta_ac
    , vector<float> &x_ac, vector<float> &y_ac, FILE*f, int i);
void odometriaCPU(  PerfilCalzada perfil, PerfilCalzada perfil_anterior,
    Point2f* v_pt1, Point2f* v_pt2, vector<float> &theta_ac, vector<float> &
    x_ac, vector<float> &y_ac, FILE*f, int i);

//Funciones main.cpp
void surf (unsigned char* h_der, unsigned char *h_izq, int i, Point2f* pt1,
    Point2f* pt2);

//////////////////
// PROGRAMA
//////////////////
int
main(int argc, char** argv)
{
  //
  //Declaración variables MIL
  //
  MIL_ID  MilApplication,
      MilSystem,
      MilDisplay[2],
      MilImageEntrada[2],
      MilDisparitylibre,
      MilColor,
      MilLibre,
      MilMapa;
#ifdef MOSTRAR_PERFIL
  MIL_ID MilImageVDisparity;
#endif
  //
  //Reseva de memoria para las MIL
  //
  MappAllocDefault(M_SETUP, &MilApplication, &MilSystem,M_NULL,M_NULL,
      M_NULL);

  MdispAlloc(MilSystem,M_DEFAULT ,"M_DEFAULT",M_DEFAULT ,&MilDisplay[0]);
```

```
  MdispAlloc(MilSystem,M_DEFAULT ,"M_DEFAULT",M_DEFAULT ,&MilDisplay[1]);

  MbufAlloc2d(MilSystem, WIDTH_IMAGEN, HEIGHT_IMAGEN,8+M_UNSIGNED,M_IMAGE+
      M_DISP+M_PROC,&MilImageEntrada[0]);
  MbufAlloc2d(MilSystem, WIDTH_IMAGEN, HEIGHT_IMAGEN,8+M_UNSIGNED,M_IMAGE+
      M_DISP+M_PROC,&MilImageEntrada[1]);
  MbufAlloc2d(MilSystem, WIDTH_IMAGEN, HEIGHT_IMAGEN,8+M_UNSIGNED,M_IMAGE+
      M_PROC+M_DISP,&MilDisparitylibre);
  MbufAlloc2d(MilSystem, WIDTH_IMAGEN, HEIGHT_IMAGEN,8+M_UNSIGNED,M_IMAGE+
      M_PROC,&MilLibre);
  MbufAlloc2d(MilSystem, ANCHO_MAPA, ALTO_MAPA,8+M_UNSIGNED,M_IMAGE+M_DISP,&
      MilMapa);

  MbufAllocColor(MilSystem,3,640,480,8+M_UNSIGNED,M_IMAGE+M_DISP,&MilColor);

#ifdef MOSTRAR_PERFIL
  MIL_ID MilPerfil;
  MdispAlloc(MilSystem,M_DEFAULT ,"M_DEFAULT",M_DEFAULT ,&MilPerfil);
  MbufAlloc2d(MilSystem, (long)STEREO_MAXD, HEIGHT_IMAGEN,8+M_UNSIGNED,
      M_IMAGE+M_DISP+M_PROC,&MilImageVDisparity);
#endif

  //
  //Visualización
  //
  MdispSelect(MilDisplay[0],MilColor);
  MdispSelect(MilDisplay[1],MilMapa);
#ifdef MOSTRAR_PERFIL
  MdispSelect(MilPerfil,MilImageVDisparity);
#endif

  //Dibujos
  Overlay *dibujo = new Overlay(MilDisplay[0]);
  Overlay *mapa = new Overlay(MilDisplay[1]);
#ifdef MOSTRAR_PERFIL
  Overlay *dibujoPerfil = new Overlay(MilPerfil);
#endif
  dibujo->Cambiar_Color(M_RGB888(255,0,0));

  //
  //Constantes
  //
  unsigned int const size = WIDTH_IMAGEN*HEIGHT_IMAGEN*sizeof(unsigned char)
      ;

  //
  //Reserva de memoria
  //
    //Reserva de memoria para los resultados en el host
    unsigned char* h_LG_izq = (unsigned char*) malloc(size);
  unsigned char* h_LG_der = (unsigned char*) malloc(size);
```

```
  unsigned char* h_u = (unsigned char*) malloc((size_t)STEREO_MAXD*
      WIDTH_IMAGEN*sizeof(unsigned char));
  unsigned char* h_v = (unsigned char*) malloc((size_t)STEREO_MAXD*
      HEIGHT_IMAGEN*sizeof(unsigned char));
  //Reserva de memoria para las imágenes de entrada
  unsigned char* h_izq = (unsigned char*) malloc(size);
  unsigned char* h_der = (unsigned char*) malloc(size);

#if (!defined PERFIL_V2) && (!defined PERFIL_V3)
  char* aux_v =  (char*)calloc((int)((STEREO_MAXD+2)*HEIGHT_IMAGEN),sizeof(
      char));
  IplImage *v = cvCreateImageHeader(cvSize((int)STEREO_MAXD,HEIGHT_IMAGEN)
      ,8,1);
#endif

  //Perfil de la calzada
  PerfilCalzada perfil, perfil_anterior, perfil_roll,perfil_rollant;

  //Detección de puntos característicos
  Point2f* v_pt1=(Point2f*)malloc((MAX_PTOS_SURF+1)*sizeof(Point2f));
  Point2f* v_pt2=(Point2f*)malloc((MAX_PTOS_SURF+1)*sizeof(Point2f));

  //Odometría
  vector<float> theta_ac, x_ac, y_ac;
  theta_ac.push_back(0.0);
  x_ac.push_back(0.0);
  y_ac.push_back(0.0);
  punto p_1 = {0,0};

  //Inicialización de las bibliotecas no-libres de OpenCV
  initModule_nonfree();

  //
  //Inicializar las cámaras
  //
#ifndef OFF_LINE
  bumblebee *camara = new bumblebee;
#endif

#ifdef SURF_V2
  //Memoria compartida
  TCHAR szName[]=TEXT("Global\\ShMemImage");
  TCHAR szName2[]=TEXT("Global\\ShMemV1");
  TCHAR szName3[]=TEXT("Global\\ShMemV2");

  HANDLE hMapFile, hMapFile2, hMapFile3;

  unsigned char* pBuf;
  Point2f* pBuf2, *pBuf3;

  //Semáforos
```

```
HANDLE ghSemaphoreWIm, ghSemaphoreRIm, ghSemaphoreWVec, ghSemaphoreRVec;
TCHAR semNameWIm[]=TEXT("Global\\SemWriteImage");
TCHAR semNameRIm[]=TEXT("Global\\SemReadImage");
TCHAR semNameWVec[]=TEXT("Global\\SemWriteVectors");
TCHAR semNameRVec[]=TEXT("Global\\SemReadVectors");

//MEMORIA COMPARTIDA
//-Para pasar la imagen
hMapFile = CreateFileMapping( INVALID_HANDLE_VALUE, // usar archivo de
    paginación
                NULL,          // seguridad por defecto
                PAGE_READWRITE,      // acceso lectura/escritura
                0,             // tamaño máximo de objeto (DWORD alta)
                BUF_SIZE,        // tamaño máximo de objeto (DWORD baja)
                szName);         // nombre

if (hMapFile == NULL)
{
  _tprintf(TEXT("Could not create file mapping object (%d).\n"),
  GetLastError());
}else{
  pBuf = (unsigned char*) MapViewOfFile(  hMapFile, // handle to map
      object
                    FILE_MAP_ALL_ACCESS,  // read/write permission
                    0,
                    0,
                    BUF_SIZE);
}

//-Para pasar los vectores
hMapFile2 = CreateFileMapping(  INVALID_HANDLE_VALUE, // usar archivo de
    paginación
                NULL,          // seguridad por defecto
                PAGE_READWRITE,      // acceso lectura/escritura
                0,             // tamaño máximo de objeto (DWORD alta)
                BUF_SIZE2,         // tamaño máximo de objeto (DWORD baja)
                szName2);       // nombre

hMapFile3 = CreateFileMapping(  INVALID_HANDLE_VALUE, // usar archivo de
    paginación
                NULL,          // seguridad por defecto
                PAGE_READWRITE,      // acceso lectura/escritura
                0,             // tamaño máximo de objeto (DWORD alta)
                BUF_SIZE2,         // tamaño máximo de objeto (DWORD baja)
                szName3);       // nombre


if (hMapFile2 == NULL || hMapFile3 == NULL)
{
  _tprintf(TEXT("Could not create file mapping object (%d).\n"),
  GetLastError());
```

```
  }else{
    pBuf2 = (Point2f*) MapViewOfFile( hMapFile2,  // puntero al espacio de
        memoria
                      FILE_MAP_ALL_ACCESS,  // permisos de lectura/escritura
                      0,
                      0,
                      BUF_SIZE2);

    pBuf3 = (Point2f*) MapViewOfFile( hMapFile3,  // puntero al espacio de
        memoria
                      FILE_MAP_ALL_ACCESS,  // permiso de lectura/escritura
                      0,
                      0,
                      BUF_SIZE2);
  }

  //SEMÁFOROS
  //-Para controlar la escritura de la imagen
  ghSemaphoreWIm = CreateSemaphore(NULL, 1, 1, semNameWIm);

  if(ghSemaphoreWIm == NULL){
    printf("CreateSemaphore error: %d\n", GetLastError());
  }
  //-Para controlar la lectura de la imagen
  ghSemaphoreRIm = CreateSemaphore(NULL, 0, 1, semNameRIm);

  if(ghSemaphoreRIm == NULL){
    printf("CreateSemaphore error: %d\n", GetLastError());
  }
  //-Para controlar la escritura de los vectores de puntos
  ghSemaphoreWVec = CreateSemaphore(NULL, 1, 1, semNameWVec);

  if(ghSemaphoreWVec == NULL){
    printf("CreateSemaphore error: %d\n", GetLastError());
  }
  //-Para controlar la lectura de los vectores de puntos
  ghSemaphoreRVec = CreateSemaphore(NULL, 0, 1, semNameRVec);

  if(ghSemaphoreRVec == NULL)
    printf("CreateSemaphore error: %d\n", GetLastError());

  cout << "Configuraci\242n completada" << endl;

if ((pBuf!=NULL)&&(pBuf2!=NULL)&&(pBuf3!=NULL)&&(ghSemaphoreWIm != NULL)&&(
    ghSemaphoreRIm != NULL)&&(ghSemaphoreWVec != NULL)&&(ghSemaphoreRVec !=
    NULL)){
#endif

for(int i=1;i<NUM_IMAGENES;i++)
#ifndef OFF_LINE
int i=1;
```

```
for(;;)
  i++;
#endif
  {

//Captura de imágenes
#ifdef OFF_LINE
  if(!Cargar_imagenes(argc,argv,WIDTH_IMAGEN,HEIGHT_IMAGEN,MilImageEntrada,i
      )){
    CUT_EXIT(argc, argv);
  }
#else
  camara->Capturar(MilImageEntrada);
#endif

  MbufGet(MilImageEntrada[0],h_izq);
  MbufGet(MilImageEntrada[1],h_der);

  //Envío de la imagen izquierda al programa auxiliar
#ifdef SURF_V2

  WaitForSingleObject(ghSemaphoreWIm, INFINITE);

  CopyMemory((PVOID)pBuf, h_izq, size);
  if(!ReleaseSemaphore(ghSemaphoreRIm,1,NULL)) printf("ReleaseSemaphore
      error: %d\n", GetLastError());

#endif

  //Mapa de disparidad y derivados
  runTest(argc,argv,h_LG_izq,h_LG_der,h_u,h_v,h_izq,h_der,WIDTH_IMAGEN,
      HEIGHT_IMAGEN);

#ifdef MOSTRAR_PERFIL
  MbufPut(MilImageVDisparity,h_v);
#endif

  //Perfil de la calzada
  perfil_anterior=perfil;

#ifdef PERFIL_V3
  ransac(h_v, perfil,RP_RANGO, RP_UMBRAL, (int)STEREO_MAXD,HEIGHT_IMAGEN);
#elif defined(PERFIL_V2)
  ransacCPU(h_v, perfil,RP_RANGO, RP_UMBRAL, (int)STEREO_MAXD,HEIGHT_IMAGEN)
      ;
#else
  for(int y=0; y<HEIGHT_IMAGEN; y++)
  {
    for(int x=0; x<STEREO_MAXD; x++)
    {
      aux_v[v->widthStep*y+x*v->nChannels]=h_v[y*(int)STEREO_MAXD+x];
```

```
    }
  }

  //Se cargan los datos de la nueva matriz calculada a la imagen OpenCv.
  v->imageData = aux_v;
  perfil_anterior=perfil;

  Calculo_perfil(v,&perfil.m,&perfil.horizonte,&perfil.pitch,0,dibujo);
#endif

  perfil.pitch=(CY-perfil.horizonte)/F;

#ifdef MOSTRAR_PERFIL
  dibujoPerfil->Iniciar();
  punto pp0 = {0, (int)perfil.horizonte};
  punto pp1 = {(int)(STEREO_MAXD-1), (int)(perfil.m*(STEREO_MAXD-1)+perfil.
      horizonte)};
  dibujoPerfil->Linea(pp0,pp1);
  dibujoPerfil->Finalizar();
#endif

  //Procesamiento de imágenes
  MbufPut(MilDisparitylibre,h_LG_der);
  MimArith(MilDisparitylibre,10,MilLibre, M_MULT_CONST);
  MimArith(MilImageEntrada[0],MilLibre,MilLibre, M_ADD+M_SATURATION);
  MbufCopyColor(MilImageEntrada[0],MilColor,M_RED);
  MbufCopyColor(MilImageEntrada[0],MilColor,M_BLUE);
  MbufCopyColor(MilLibre,MilColor,M_GREEN);
  MbufGet(MilImageEntrada[0],h_izq);
  MbufGet(MilDisparitylibre,h_der);

  //Puntos característicos

#ifdef SURF_V2

  //Recibir de programa auxiliar
  WaitForSingleObject(ghSemaphoreRVec, INFINITE);

  memcpy(v_pt1, pBuf2, ((int)(pBuf2[0].x)+1)*sizeof(Point2f));
  memcpy(v_pt2, pBuf3, ((int)(pBuf3[0].x)+1)*sizeof(Point2f));

  if(!ReleaseSemaphore(ghSemaphoreWVec,1,NULL)) printf("ReleaseSemaphore
      error: %d\n", GetLastError());
#else
  surf (h_der, h_izq, i, v_pt1, v_pt2);
#endif

#ifdef MOSTRAR_SURF
  //Visualizar puntos característicos
  dibujo->Iniciar();
  dibujo->Cambiar_Color(M_RGB888(255,0,0));
```

```
  punto izq_sup_1={0,0}, der_inf_1={0,0};
  for(int pos = 1; pos <= (int)(v_pt1[0].x); pos++ )
    {
    izq_sup_1.x = (int)v_pt1[pos].x;
    izq_sup_1.y = (int)v_pt1[pos].y+ALTURA_SURF;
    der_inf_1.x = (int)v_pt2[pos].x;
    der_inf_1.y = (int)v_pt2[pos].y+ALTURA_SURF;
    dibujo->Linea(izq_sup_1,der_inf_1);
  }
  dibujo->Finalizar();
#endif

  //Cálculo de la odometría
  if((int)(v_pt1[0].x)>0)
  {

#if (defined(CALPTOS_V2)||defined(CALSOL_V3)||defined(CALSOL_V5))
    odometria(perfil, perfil_anterior, v_pt1, v_pt2, theta_ac, x_ac, y_ac,
        file,i);
#else
    odometriaCPU(perfil, perfil_anterior, v_pt1, v_pt2, theta_ac, x_ac, y_ac
        , file, i);
#endif

    p_1.x = ALTO_MAPA/2+(int)(x_ac.back()*RESOLUCION_MAPA);
    p_1.y = ANCHO_MAPA/2-(int)(y_ac.back()*RESOLUCION_MAPA);

    mapa->Punto_relleno(p_1);

  }
  mapa->Finalizar();

}

#ifdef SURF_V2
}
else
{
  printf("Fallo creando memorias/semaforos\n");
}
#endif

cout << "\nENTER para cerrar" << endl;
getchar();

#ifndef OFF_LINE
  camara->Liberar();
  delete [] camara;
#endif

  delete dibujo;
```

```
  free(h_LG_izq);
  free(h_izq);
  free(h_LG_der);
  free(h_der);
  free(h_u);
  free(h_v);
  MbufFree(MilImageEntrada[0]);
  MbufFree(MilImageEntrada[1]);

  MbufFree(MilDisparitylibre);
  MbufFree(MilLibre);
  MdispFree(MilDisplay[0]);
  MdispFree(MilDisplay[1]);
  MbufFree(MilColor);
  MbufFree(MilMapa);

#ifdef MOSTRAR_PERFIL
  MbufFree(MilImageVDisparity);
  MdispFree(MilPerfil);
#endif

    MappFreeDefault(MilApplication, MilSystem,M_NULL,M_NULL,M_NULL);

#ifdef SURF_V2
  UnmapViewOfFile(pBuf);
  UnmapViewOfFile(pBuf2);
  UnmapViewOfFile(pBuf3);

  CloseHandle(hMapFile);
  CloseHandle(hMapFile2);
  CloseHandle(hMapFile3);

  CloseHandle(ghSemaphoreWIm);
  CloseHandle(ghSemaphoreRIm);
  CloseHandle(ghSemaphoreWVec);
  CloseHandle(ghSemaphoreRVec);
#endif

  f.close();
  fclose(file);

  CUT_EXIT(argc, argv);
}

void surf(unsigned char * h_der,
      unsigned char * h_izq,
      int i,
      Point2f *pt1,
      Point2f *pt2)
{
```

```
#ifdef SURF_V1_MLIBRE
  static IplImage *mapa_libre = cvCreateImageHeader(cvSize(WIDTH_IMAGEN,
      HEIGHT_IMAGEN),8,1);
  static char* mapa_libre_aux =  (char*)calloc(((WIDTH_IMAGEN+2)*
      HEIGHT_IMAGEN),sizeof(char));
#endif

  static bool hayPuntos = FALSE;

  static IplImage *izq_visible_1 = cvCreateImageHeader(cvSize(WIDTH_IMAGEN,
      HEIGHT_IMAGEN),8,1);
  static char* visible_aux =  (char*)calloc(((WIDTH_IMAGEN+2)*HEIGHT_IMAGEN)
      ,sizeof(char));

  static CvSeq *anteriorKeypoints = 0, *anteriorDescriptors = 0;
  static CvSeq *Keypoints = 0, *Descriptors = 0;
  static CvMemStorage* storage = cvCreateMemStorage(0);
  static CvMemStorage* storage_anterior = cvCreateMemStorage(0);

  vector<int> ptpairs;

  static const CvSURFParams params = cvSURFParams(100, 0);

#ifdef SURF_V1_MLIBRE
  for(int y=0; y<HEIGHT_IMAGEN; y++)
  {
    for(int x=0; x<WIDTH_IMAGEN; x++)
    {
      mapa_libre_aux[mapa_libre->widthStep*y+x*mapa_libre->nChannels]=h_der[
          y*(int)WIDTH_IMAGEN+x];
    }
  }
  mapa_libre->imageData = mapa_libre_aux;
  cvSetImageROI(mapa_libre,cvRect(0,ALTURA_SURF,WIDTH_IMAGEN,HEIGHT_IMAGEN))
      ;
#endif

  for(int y=0; y<HEIGHT_IMAGEN; y++)
  {
    for(int x=0; x<WIDTH_IMAGEN; x++)
    {
      visible_aux[izq_visible_1->widthStep*y+x*izq_visible_1->nChannels]=
          h_izq[y*(int)WIDTH_IMAGEN+x];
    }
  }
  izq_visible_1->imageData = visible_aux;
  cvEqualizeHist(izq_visible_1, izq_visible_1);
  cvSetImageROI(izq_visible_1,cvRect(0,ALTURA_SURF,WIDTH_IMAGEN,
      HEIGHT_IMAGEN));
  cvClearMemStorage(storage_anterior);
```

```
  if(hayPuntos)
  {
    anteriorKeypoints = cvCloneSeq((const CvSeq*)Keypoints,storage_anterior)
        ;
    anteriorDescriptors = cvCloneSeq((const CvSeq*)Descriptors,
        storage_anterior);
  }
  cvClearMemStorage(storage);

#ifdef SURF_V1_MLIBRE
  cvExtractSURF(izq_visible_1, mapa_libre, &Keypoints, &Descriptors, storage
      , params);
#else
  cvExtractSURF(izq_visible_1, NULL, &Keypoints, &Descriptors, storage,
      params);
#endif


  if(hayPuntos)
  {
  #ifndef USE_FLANN
    findPairs( anteriorKeypoints, anteriorDescriptors, Keypoints,
        Descriptors, ptpairs);
  #else
    flannFindPairs( anteriorKeypoints, anteriorDescriptors, Keypoints,
        Descriptors, ptpairs);
  #endif
  }

  hayPuntos=((Keypoints->total)!=0);
  int n = (int)(ptpairs.size()/2);

  pt1[0].x=(float)n;
  pt2[0].x=(float)n;

  for(int pos = 0; pos < n; pos++ )
    {
        pt1[pos+1] = ((CvSURFPoint*)cvGetSeqElem(anteriorKeypoints,ptpairs[
            pos*2]))->pt;
        pt2[pos+1] = ((CvSURFPoint*)cvGetSeqElem(Keypoints,ptpairs[pos*2+1])
            )->pt;
    }

  if (i==NUM_IMAGENES){
#ifdef SURF_V1_MLIBRE
  free(mapa_libre_aux);
  cvReleaseImage(&mapa_libre);
#endif
  free(visible_aux);
  cvClearSeq(Keypoints);
  cvClearSeq(anteriorKeypoints);
```

```
  cvClearSeq(Descriptors);
  cvClearSeq(anteriorDescriptors);
  cvClearMemStorage(storage);
  cvClearMemStorage(storage_anterior);
  cvReleaseMemStorage(&storage);
  cvReleaseMemStorage(&storage_anterior);
  cvReleaseImage(&izq_visible_1);}
}


void ransacCPU(unsigned char * h_v, PerfilCalzada &perfil, float rango,
    unsigned int umbral, int ancho, int alto){

  //Declaración de variables, reserva de memoria
  static bool iniciado = 0;

  static float * puntos_seleccionados =(float*)malloc(RP_NUM_THREADS*2*
      sizeof(float));
  static unsigned int * puntos_blancos = (unsigned int *)malloc(
      MAX_PTOS_BLANCOS*2*sizeof(unsigned int));
  static int * puntuacion = (int *)malloc(RP_NUM_THREADS*sizeof(int));

  unsigned int num_ptos_blancos = 0;
  unsigned int posicion=0;

  int x,y,n;

  //Búsqueda de puntos a nivel alto
  for (y=0;y<alto;y++){
    for (x=0;x<ancho;x++){
      if (*(h_v+x+y*ancho)>umbral){
        *(puntos_blancos+num_ptos_blancos)=x;
        *(puntos_blancos+num_ptos_blancos+MAX_PTOS_BLANCOS)=y;
        num_ptos_blancos++;
      }
    }
  }

  //RANSAC
  if (num_ptos_blancos!=0){

    srand((unsigned int)time(NULL));

    for(int x=0;x<RP_NUM_THREADS;x++){

      unsigned int RANDOM1, RANDOM2;

      RANDOM1=rand()%num_ptos_blancos;
      do{
        RANDOM2=rand()%num_ptos_blancos;
      }while (RANDOM1==RANDOM2);
```

```
      float m=0,b=0;
      float x1,y1,x2,y2;

      int num_inliers=0;
      unsigned int n;

      x1=(float)*(puntos_blancos+RANDOM1);
      y1=(float)*(puntos_blancos+RANDOM1+MAX_PTOS_BLANCOS);
      x2=(float)*(puntos_blancos+RANDOM2);
      y2=(float)*(puntos_blancos+RANDOM2+MAX_PTOS_BLANCOS);

      unsigned int const y_thres=Y_THRESHOLD;
      if ((x1!=x2)&&(abs(y1-y2)>y_thres)){
        m=(y1-y2)/(x1-x2);
        b=y1-m*x1;

        for (n=0;n<num_ptos_blancos;n++){
          if ((abs(*(puntos_blancos+n+MAX_PTOS_BLANCOS)-m*(*(puntos_blancos+
              n))-b))<RP_RANGO){
            num_inliers++;
          }
        }
      }else{
        num_inliers=-1;
      }

      *(puntuacion+x)=num_inliers;
      *(puntos_seleccionados+x)=m;
      *(puntos_seleccionados+x+RP_NUM_THREADS)=b;
    }

    //Búsqueda de la solución más votada
    for (n=0;n<RP_NUM_THREADS;n++){
      if (*(puntuacion+n)>*(puntuacion+posicion)){
        posicion=n;
      }
    }

    perfil.m=*(puntos_seleccionados+posicion);
    perfil.horizonte=*(puntos_seleccionados+posicion+RP_NUM_THREADS);

  }else{printf("Perfil de la calzada no encontrado\n");}
}

void Calculo_perfil(IplImage* v,float* m,float* horizonte,float* pitch,bool
    imprimir,Overlay* dibujo)
{
  cvThreshold(v,v,200,255,CV_THRESH_BINARY);
  // Se define una estructura para poder almacenar datos genéricos de forma
      dinámica en formato OpenCv.
```

```
CvMemStorage* datos = cvCreateMemStorage(0);

// Se define una estructura para poder almacenar líneas de forma dinámica
    en formato OpenCv.
  CvSeq* lineas = 0;

// Se calcula la transformada de Hough de la imagen v, almacenando los
    resultados en lineas.
lineas = cvHoughLines2(v,datos,CV_HOUGH_PROBABILISTIC,1,CV_PI
    /7200,50,75,1500);

// CvSeq* cvHoughLines2(CvArr* image, void* storage, int method, double
    rho, double theta, int threshold,
//            double param1=0, double param2=0);
//    image   -> Imagen de origen.
//    storage -> Almacenamiento de las líneas detectadas.
//    method  -> Método de cálculo de la transformada de Hough:
//               CV_HOUGH_STANDARD    -> Método clásico.
//               CV_HOUGH_PROBABILISTIC -> Método probabilístico.
//               CV_HOUGH_MULTI_SCALE   -> Variable multiescala del método
    clásico.
//    rho   -> Distancia de resolución (en píxeles).
//    theta   -> Ángulo de resolución (en radianes).
//    threshold -> Parámetro umbral.
//    param1  -> Para la transformada de Hough probabilística, longitud
    mínima de la línea.
//    param2  -> Máxima distancia entre dos segmentos para ser
    considerados pertenecientes a la misma línea.

// Extracción de la primera línea de todas las obtenidas en la
    transformada de Hough.

CvPoint* linea = (CvPoint*)cvGetSeqElem(lineas,0);

// char* cvGetSeqElem(const CvSeq* seq, int index)
//    seq   -> Secuencia.
//    index -> Índice del elemento.
if(linea)
{
*m = (float)(linea[0].y-linea[1].y)/(float)(linea[0].x-linea[1].x);
*horizonte = linea[0].y - linea[0].x*(*m);
*pitch = (CY-(*horizonte))/F;
}

if(imprimir)
{
printf("Coordenadas de la línea resultante");
printf("(%d,%d)(%d,%d), pendiente = %f,horizonte = %f,pitch = %f",linea
    [1].y,linea[1].x,linea[0].y,linea[0].x,*m,*horizonte,*pitch);
}
if(dibujo != M_NULL)
```

```
  {
  dibujo->Iniciar();
  dibujo->Cambiar_Color(M_RGB888(255,0,0));
  punto izq_sup_1 = {linea[1].x,linea[1].y};
  punto der_inf_1 = {linea[0].x,linea[0].y};
  dibujo->Iniciar();
  dibujo->Linea(izq_sup_1,der_inf_1);
  dibujo->Finalizar();
  }

  cvReleaseMemStorage(&datos);
}

void odometriaCPU(  PerfilCalzada perfil,
              PerfilCalzada perfil_anterior,
              Point2f* v_pt1,
              Point2f* v_pt2,
              vector<float> &theta_ac,
              vector<float> &x_ac,
              vector<float> &y_ac,
              FILE *f,
              int i)
{

  //Declaración de variables, reserva de memoria
  static bool iniciado=0;

  static float r_ac[3];
  static float r_ac_anterior[3];
  float * r_ac_sel=NULL;

  int nelementos=(int)(v_pt1[0].x);
  int tam=nelementos*sizeof(float);

#if (defined(CALSOL_V2)||defined(CALSOL_V4))
  float * v_theta=(float*)malloc(tam);
  float * v_x_transl=(float*)malloc(tam);
  float * v_y_transl=(float*)malloc(tam);
#else
  vector<float> v_theta;
  vector<float> v_x_transl;
  vector<float> v_y_transl;

  v_theta.resize(nelementos);
  v_x_transl.resize(nelementos);
  v_y_transl.resize(nelementos);
#endif

  static unsigned int * v_seleccion;
  static unsigned int * v_puntuacion;
```

```
  float theta_ac_back=theta_ac.back();

  float m=perfil.m;
  float m_anterior=perfil_anterior.m;
  float horizonte=perfil.horizonte;
  float horizonte_anterior=perfil_anterior.horizonte;
  float pitch=perfil.pitch;
  float pitch_anterior=perfil_anterior.pitch;

  if(!iniciado){

    v_seleccion=(unsigned int*)malloc(OD_NUM_THREADS*sizeof(unsigned int));
    v_puntuacion=(unsigned int*)malloc(OD_NUM_THREADS*sizeof(unsigned int));

  }

  for (int pos=0; pos<nelementos; pos++){

    Point2f mundo_0, mundo_1,trans;
    Point2f pt_mundo_1;
    Point2f pt_mundo_0;
    float a,b,c,raiz,angulo1,angulo2,angulo;

    mundo_0.y = cos(pitch_anterior)*(m_anterior*F*B/(v_pt1[pos].y+
        ALTURA_SURF-horizonte_anterior));
    mundo_0.x = m_anterior*B*(v_pt1[pos].x-CX)/(v_pt1[pos].y+ALTURA_SURF-
        horizonte_anterior);

    mundo_1.y = cos(pitch)*(m*F*B/(v_pt2[pos].y+ALTURA_SURF-horizonte));
    mundo_1.x = m*B*(v_pt2[pos].x-CX)/(v_pt2[pos].y+ALTURA_SURF-horizonte);

    pt_mundo_0.x = cos(DESVIACION_YAW)*mundo_0.x + sin(DESVIACION_YAW)*
        mundo_0.y;
    pt_mundo_0.y = cos(DESVIACION_YAW)*mundo_0.y - sin(DESVIACION_YAW)*
        mundo_0.x + 1.4f;

    pt_mundo_1.x = cos(DESVIACION_YAW)*mundo_1.x + sin(DESVIACION_YAW)*
        mundo_1.y;
    pt_mundo_1.y = cos(DESVIACION_YAW)*mundo_1.y - sin(DESVIACION_YAW)*
        mundo_1.x + 1.4f;

    a = (pt_mundo_0.x)*(pt_mundo_0.x) + (pt_mundo_0.y)*(pt_mundo_0.y);
    b = 2*pt_mundo_1.x*pt_mundo_0.y;
    c = pt_mundo_1.x*pt_mundo_1.x - pt_mundo_0.x*pt_mundo_0.x;
    raiz = sqrt(b*b-4*a*c);

    angulo1 = asin((-b-raiz)/(2*a));
    angulo2 = asin((-b+raiz)/(2*a));
    angulo = (abs(angulo1)<abs(angulo2)) ? angulo1 : angulo2;

    v_theta[pos]=angulo;
```

```
    trans.x = pt_mundo_0.x - pt_mundo_1.x*cos(v_theta[pos])-pt_mundo_1.y*sin
        (v_theta[pos]);
    trans.y = pt_mundo_0.y + pt_mundo_1.x*sin(v_theta[pos])-pt_mundo_1.y*cos
        (v_theta[pos]);
    v_x_transl[pos]=cos(theta_ac_back)*trans.x + sin(theta_ac_back)*trans.y;
    v_y_transl[pos]=cos(theta_ac_back)*trans.y - sin(theta_ac_back)*trans.x;

  }

#if (defined(CALSOL_V2)||defined(CALSOL_V4))
#ifdef CALSOL_V4
  int j;

  int inliers;

  float rango;
  float sel[3], sel_aux;
  float * vector;

  unsigned int RANDOM[3], RANDOM_aux;

  srand((unsigned int)time(NULL));

  for (int x=0; x<OD_NUM_THREADS; x++){

    inliers=0;

    for (int i=0;i<3;i++){
      RANDOM[i]=rand()%nelementos;
    }

    if (x<((OD_NUM_THREADS)/3)){
      for (int n=0; n<3; n++){
        sel[n]=v_theta[RANDOM[n]];
      }
      vector=v_theta;
      rango=RANGO_THETA;
    }else if (x<(2*((OD_NUM_THREADS)/3))){
      for (int n=0; n<3; n++){
        sel[n]=v_x_transl[RANDOM[n]];
      }
      vector=v_x_transl;
      rango=RANGO_X;
    }else{
      for (int n=0; n<3; n++){
        sel[n]=v_y_transl[RANDOM[n]];
      }
      vector=v_y_transl;
      rango=RANGO_Y;
    }
    for(int i=1; i<3; i++){
```

```
      for(int j=0; j<=2-i;j++){
        if(sel[j]>sel[j+1]){
          sel_aux=sel[j];
          sel[j]=sel[j+1];
          sel[j+1]=sel_aux;
          RANDOM_aux=RANDOM[j];
          RANDOM[j]=RANDOM[j+1];
          RANDOM[j+1]=RANDOM_aux;
        }
      }
    }

    v_seleccion[x]=RANDOM[1];

    for (j=0; j<nelementos; j++){
      if(abs(sel[1]-vector[j]) < rango){
        inliers++;
      }
    }
    v_puntuacion[x]=inliers;
  }
#else

  int j;

  int inliers;

  float rango;
  float sel;
  float * vector;

  unsigned int RANDOM;

  srand((unsigned int)time(NULL));

  for (int x=0; x<OD_NUM_THREADS; x++){

    inliers=0;

    RANDOM=rand()%nelementos;

    if (x<((OD_NUM_THREADS)/3)){
      sel=v_theta[RANDOM];
      vector=v_theta;
      rango=RANGO_THETA;
    }else if (x<(2*((OD_NUM_THREADS)/3))){
      sel=v_x_transl[RANDOM];
      vector=v_x_transl;
      rango=RANGO_X;
    }else{
      sel=v_y_transl[RANDOM];
```

```
        vector=v_y_transl;
        rango=RANGO_Y;
      }

      v_seleccion[x]=RANDOM;

      for (j=0; j<nelementos; j++){
        if(abs(sel-vector[j]) < rango){
          inliers++;
        }
      }
      v_puntuacion[x]=inliers;
    }

#endif

    memcpy(r_ac_anterior, r_ac, 3*sizeof(float));

    for (int x=0; x<3; x++){
      unsigned int pos=(unsigned int)(x*((OD_NUM_THREADS)/3));

      int limite_inf=(unsigned int)(x*((OD_NUM_THREADS)/3));
      int limite_sup=(unsigned int)((x+1)*((OD_NUM_THREADS)/3));

      for (int i=limite_inf; i<limite_sup; i++){
        if (v_puntuacion[i]>v_puntuacion[pos]){
          pos=i;
        }
      }

      if (x==0)
        r_ac[0]=v_theta[v_seleccion[pos]];
      else if (x==1)
        r_ac[1]=v_x_transl[v_seleccion[pos]];
      else
        r_ac[2]=v_y_transl[v_seleccion[pos]];
    }
#else

    if(v_theta.size()>0)
    {
      sort(v_theta.begin(),v_theta.end(),myobject);
      sort(v_x_transl.begin(),v_x_transl.end(),myobject);
      sort(v_y_transl.begin(),v_y_transl.end(),myobject);

      memcpy(r_ac_anterior, r_ac, 3*sizeof(float));

      r_ac[0]=v_theta[(int)(v_theta.size()/2)];
      r_ac[1]=v_x_transl[(int)(v_x_transl.size()/2)];
      r_ac[2]=v_y_transl[(int)(v_y_transl.size()/2)];
```

```
  }
#endif


#ifdef FILTRO
  if ((r_ac[1]*r_ac[1]+r_ac[2]*r_ac[2])<UMBRAL_FILTRO){
    r_ac_sel=r_ac;
  }else{
    r_ac_sel=r_ac_anterior;
  }
#else
  r_ac_sel=r_ac;
#endif

  //Liberado de memoria
  theta_ac.push_back(r_ac_sel[0]+theta_ac.back());
  x_ac.push_back(r_ac_sel[1]+x_ac.back());
  y_ac.push_back(r_ac_sel[2]+y_ac.back());

#if (defined(CALSOL_V2)||defined(CALSOL_V4))
  free(v_theta);
  free(v_x_transl);
  free(v_y_transl);
#endif

}
```

A.3 Código del host del programa principal

Se muestra ahora la parte del programa principal que forma parte del *host* e interactúa con el *device*. Forma parte de un archivo con extensión `.cu`, y se accede a sus funciones a través del `main`.

```
////////////////////
// INCLUSIONES
////////////////////
//Inclusión de bibliotecas del sistema
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <vector>
using namespace std;
```

```
//Inclusiones de CUDA
#include <cutil.h>
#include "curand.h"

//Inclusión de los kernels
#include <cppIntegration_kernel.cu>

////////////////////
// CLASES
////////////////////
//Clase para el perfil de la calzada y el pitch
class PerfilCalzada{
public:
  float m;
  float horizonte;
  float pitch;
  PerfilCalzada(){m=0;horizonte=0;pitch=0;}
};

class TimerCUDA{
public:
  unsigned int timer;
  char* nombre;
  float acTiempo;
  int veces;

  TimerCUDA(char nombre[]){
    this->nombre=nombre;
    CUT_SAFE_CALL( cutCreateTimer(&timer));
    acTiempo=0;
    veces=0;
  }
  void Iniciar(){
    CUT_SAFE_CALL(cutResetTimer(timer));
    CUT_SAFE_CALL(cutStartTimer(timer));
  }
  void Parar(){
    CUT_SAFE_CALL(cutStopTimer(timer));
    if (veces>1){
      acTiempo+=cutGetTimerValue(timer);
    }
    veces++;
  }
  void Escribir(FILE *f){
    fprintf(f, "%s - %f ms\n", nombre, acTiempo/(veces-1));
    CUT_SAFE_CALL( cutDeleteTimer(timer));
  }
};

////////////////////
// PROTOTIPOS
```

```
////////////////////
void ransacOdometria(unsigned int* seleccion, unsigned int* puntuacion, int
    nelementos, float*theta, float*x_transl, float*y_transl);
void ransacOdometriaOriginal(unsigned int * seleccion, unsigned int *
    puntuacion, int num_ptos, float * theta, float * x_transl, float *
    y_transl);
void calcularPtos(int n, float m, float m_anterior, float horizonte, float
    horizonte_anterior, float pitch, float pitch_anterior, Point2f* pt1,
    Point2f* pt2, float* theta, float *y_transl, float*x_transl, float
    theta_ac_back);

////////////////////
// Cálculo del Mapa de Disparidad y derivados
////////////////////
extern "C" void
runTest(int argc,
    char** argv,
    unsigned char* h_LG_izq,
    unsigned char* h_LG_der,
    unsigned char* h_u,
    unsigned char* h_v,
    unsigned char* h_izq,
    unsigned char* h_der,
    int width,
    int height)
{
  static int iniciado = 0;
  static unsigned int size = width*height*sizeof(unsigned char);
  static unsigned char* d_izq = NULL; // d_LG almacena el resultado de la L
      de la G
  static unsigned char* d_der = NULL; // d_LG almacena el resultado de la L
      de la G
  static unsigned char* d_u = NULL; //d_u almacena el resultado del u-
      disparity
  static unsigned char* d_v = NULL; //d_v almacena el resultado del v-
      disparity
  static int *g_minSSD_izq;
  static int *g_minSSD_der;
  static cudaArray* cu_array_izq;
  static cudaArray* cu_array_der;
  static cudaChannelFormatDesc channelDesc_izq;
  static cudaChannelFormatDesc channelDesc_der;

  if(!iniciado)
  {

  //Reservas e inicialización///
    CUT_DEVICE_INIT(argc, argv);

    // d_LG almacena el resultado de la L de la G
    CUDA_SAFE_CALL( cudaMalloc( (void**) &d_izq,size));
```

```
    // d_LG almacena el resultado de la L de la G
    CUDA_SAFE_CALL( cudaMalloc( (void**) &d_der,size));
    // memoria para guardar el coste
    CUDA_SAFE_CALL(cudaMalloc((void**)&g_minSSD_izq,width*height*sizeof(int)
        ));
    CUDA_SAFE_CALL(cudaMalloc((void**)&g_minSSD_der,width*height*sizeof(int)
        ));
    // memoria para u-disparity y v_disparity
    CUDA_SAFE_CALL(cudaMalloc((void**)&d_u,width*STEREO_MAXD*sizeof(unsigned
         char)));
    CUDA_SAFE_CALL(cudaMalloc((void**)&d_v,height*STEREO_MAXD*sizeof(
        unsigned char)));

    channelDesc_izq = cudaCreateChannelDesc(8, 0, 0, 0,
        cudaChannelFormatKindUnsigned);
    CUDA_SAFE_CALL( cudaMallocArray( &cu_array_izq, &channelDesc_izq, width,
         height));
    CUDA_SAFE_CALL( cudaBindTextureToArray( tex_izq, cu_array_izq,
        channelDesc_izq));

    channelDesc_der = cudaCreateChannelDesc(8, 0, 0, 0,
        cudaChannelFormatKindUnsigned);
    CUDA_SAFE_CALL( cudaMallocArray( &cu_array_der, &channelDesc_der, width,
         height));
    CUDA_SAFE_CALL( cudaBindTextureToArray( tex_der, cu_array_der,
        channelDesc_der));

    iniciado = 1;
  }

  //Dimensionamiento de los bloques y el grid///
  dim3 dimBlock(1,height,1);
  dim3 dimGrid(width / dimBlock.x, height/ dimBlock.y, 1);

  dim3 dimBlock2(BLOCK_W,1,1);
  dim3 dimGrid2(divUp(width, BLOCK_W),divUp(height,ROWSperTHREAD),1);

  dim3 dimBlock3(16,16,1);
  dim3 dimGrid3(divUp(width, dimBlock3.x),divUp(height,dimBlock3.y),1);

  dim3 dimBlock4(320,1,1);
  dim3 dimGrid4(divUp(width, dimBlock4.x),1,1);

  dim3 dimBlock5(1,60,1);
  dim3 dimGrid5(1,divUp(height, dimBlock5.y),1);

  //Pasamos la imagenes a la GPU///
  CUDA_SAFE_CALL( cudaMemcpyToArray( cu_array_izq, 0, 0, h_izq, size,
      cudaMemcpyHostToDevice));
  CUDA_SAFE_CALL( cudaMemcpyToArray( cu_array_der, 0, 0, h_der, size,
      cudaMemcpyHostToDevice));
```

```
Laplaciana_gausiana_doble<<< dimGrid, dimBlock>>>(d_izq,d_der, width,
    height);

CUDA_SAFE_CALL( cudaThreadSynchronize() );

//Pasamos el resultado al cu_array que a su vez éste está ligado a la
    textura
CUDA_SAFE_CALL(cudaMemcpyToArray( cu_array_izq, 0, 0,d_izq, size,
    cudaMemcpyDeviceToDevice));
CUDA_SAFE_CALL(cudaMemcpyToArray( cu_array_der, 0, 0,d_der, size,
    cudaMemcpyDeviceToDevice));

disparidadizda<<<dimGrid2,dimBlock2,SHARED_MEM_SIZE>>>(d_izq,d_der,
    g_minSSD_izq,g_minSSD_der,width,height);

CUDA_SAFE_CALL(cudaThreadSynchronize());

CUDA_SAFE_CALL( cudaMemcpyToArray( cu_array_izq, 0, 0,d_izq, size,
    cudaMemcpyDeviceToDevice));
CUDA_SAFE_CALL( cudaMemcpyToArray( cu_array_der, 0, 0,d_der, size,
    cudaMemcpyDeviceToDevice));

// Realizacion del croos-cheking///
cross_checking<<<dimGrid3,dimBlock3>>>(d_izq, width);
CUDA_SAFE_CALL( cudaThreadSynchronize() );

// Calculo del u disparity ///
CUDA_SAFE_CALL( cudaMemcpyToArray( cu_array_izq, 0, 0,d_izq, size,
    cudaMemcpyDeviceToDevice));

u_disparity<<<dimGrid4,dimBlock4>>>(d_u,width,height);
CUDA_SAFE_CALL( cudaThreadSynchronize() );

//Realizacion del mapa_de_obstaculos///
inicializacion<<<dimGrid3,dimBlock3>>>(d_izq,width,height);
for(int i=0;i<=STEREO_MAXD;i++)
{
  mapa_obstaculos<<<dimGrid4,dimBlock4>>>(d_izq,d_u,i,width,height);
  CUDA_SAFE_CALL(cudaThreadSynchronize());
}
//Realizacion del mapa_libre///
CUDA_SAFE_CALL( cudaMemcpyToArray( cu_array_der, 0, 0,d_izq, size,
    cudaMemcpyDeviceToDevice));
inicializacion<<<dimGrid3,dimBlock3>>>(d_der,width,height);
mapa_libre<<<dimGrid3,dimBlock3>>>(d_der,width,height);
CUDA_SAFE_CALL(cudaThreadSynchronize());

//Calculo del V disparity ///
CUDA_SAFE_CALL( cudaMemcpyToArray( cu_array_izq, 0, 0,d_der, size,
    cudaMemcpyDeviceToDevice));
```

```
  v_disparity<<<dimGrid5,dimBlock5>>>(d_v,width,height);
  CUDA_SAFE_CALL( cudaThreadSynchronize());

  //Calculo de los bordes de los obstaculos///
  //filtradas<<<dimGrid3,dimBlock3>>>(d_izq,width,height);

  CUDA_SAFE_CALL( cudaMemcpy( h_LG_der,d_der, size, cudaMemcpyDeviceToHost))
      ;
  CUDA_SAFE_CALL( cudaMemcpy( h_u,d_u, width*STEREO_MAXD*sizeof(unsigned
      char), cudaMemcpyDeviceToHost));
  CUDA_SAFE_CALL( cudaMemcpy( h_v,d_v, height*STEREO_MAXD*sizeof(unsigned
      char), cudaMemcpyDeviceToHost));

  //CUDA_SAFE_CALL(cudaUnbindTexture(tex_izq));
  //CUDA_SAFE_CALL(cudaUnbindTexture(tex_der));

    //CUDA_SAFE_CALL(cudaFree(d_LG_izq));
    //CUDA_SAFE_CALL(cudaFreeArray(cu_array_izq));
  //CUDA_SAFE_CALL(cudaFree(d_LG_der));
    //CUDA_SAFE_CALL(cudaFreeArray(cu_array_der));
}

////////////////////
// Implementación de RANSAC
////////////////////
extern "C" void ransac(unsigned char * h_v, PerfilCalzada &perfil, float
    rango, unsigned int umbral/*float *recta_res*/, int ancho, int alto){

  //Declaración de variables, reserva de memoria
  static bool iniciado = 0;

  static float * puntos_seleccionados =(float*)malloc(RP_NUM_THREADS*2*
      sizeof(float));
  static unsigned int * puntos_blancos = (unsigned int *)malloc(
      MAX_PTOS_BLANCOS*2*sizeof(unsigned int));
  static int * puntuacion = (int *)malloc(RP_NUM_THREADS*sizeof(int));

  static float * d_puntos_seleccionados;
  static unsigned int * d_puntos_blancos;
  static int * d_puntuacion;

  static curandState * devStates;

  //Tamaño de bloque y grid
  dim3 dimBlockR4(512,1,1);
    dim3 dimGridR4(2*RP_NUM_THREADS/512, 1, 1);

  dim3 dimBlockR1(512,1,1);
  dim3 dimGridR1(RP_NUM_THREADS/512, 1, 1);

  if(!iniciado){
```

```
    CUDA_SAFE_CALL(cudaMalloc((void**) &d_puntos_seleccionados,
        RP_NUM_THREADS*2*sizeof(float)));
    CUDA_SAFE_CALL(cudaMalloc((void**) &d_puntos_blancos, MAX_PTOS_BLANCOS
        *2*sizeof(unsigned int)));
    CUDA_SAFE_CALL(cudaMalloc((void**) &d_puntuacion, RP_NUM_THREADS*sizeof(
        int)));
    iniciado=1;

    //Semilla para la generación de números aleatorios
    CUDA_SAFE_CALL(cudaMalloc((void**) &devStates, 2*RP_NUM_THREADS*sizeof(
        curandState)));
    setup_kernel<<<dimGridR4, dimBlockR4>>>(devStates, (unsigned long)time(
        NULL));

  }

  unsigned int num_ptos_blancos = 0;
  unsigned int posicion=0;

  int x,y,n;

  //Búsqueda de puntos a nivel alto
  for (y=0;y<alto;y++){
    for (x=0;x<ancho;x++){
      if (*(h_v+x+y*ancho)>umbral){
        *(puntos_blancos+num_ptos_blancos)=x;
        *(puntos_blancos+num_ptos_blancos+MAX_PTOS_BLANCOS)=y;
        num_ptos_blancos++;
      }
    }
  }

  //RANSAC
  if (num_ptos_blancos!=0){

    CUDA_SAFE_CALL( cudaMemcpy(d_puntos_blancos,puntos_blancos,
        MAX_PTOS_BLANCOS*2*sizeof(unsigned int), cudaMemcpyHostToDevice) );

    ransac_t<<<dimGridR1, dimBlockR1>>>(d_puntos_blancos, num_ptos_blancos,
        d_puntuacion, d_puntos_seleccionados, devStates, rango);

    CUDA_SAFE_CALL( cudaMemcpy(puntos_seleccionados,d_puntos_seleccionados,
        RP_NUM_THREADS*2*sizeof(float), cudaMemcpyDeviceToHost) );
    CUDA_SAFE_CALL( cudaMemcpy(puntuacion,d_puntuacion, RP_NUM_THREADS*
        sizeof(unsigned int), cudaMemcpyDeviceToHost) );

  //Búsqueda de la solución más votada
    for (n=0;n<RP_NUM_THREADS;n++){
      if (*(puntuacion+n)>*(puntuacion+posicion)){
        posicion=n;
```

```
      }
    }

    perfil.m=*(puntos_seleccionados+posicion);
    perfil.horizonte=*(puntos_seleccionados+posicion+RP_NUM_THREADS);

  }else{printf("Perfil de la calzada no encontrado\n");}
}

extern "C" void odometria(  PerfilCalzada perfil,
              PerfilCalzada perfil_anterior,
              Point2f* v_pt1,
              Point2f* v_pt2,
              vector<float> &theta_ac,
              vector<float> &x_ac,
              vector<float> &y_ac,
              FILE*f, int i)
{

  //Declaración de variables, reserva de memoria
  static bool iniciado=0;

  static float r_ac[3];
  static float r_ac_anterior[3];
  float * r_ac_sel=NULL;

  int nelementos=(int)(v_pt1[0].x);
  int tam=nelementos*sizeof(float);

  float * v_theta=(float*)malloc(tam);
  float * v_x_transl=(float*)malloc(tam);
  float * v_y_transl=(float*)malloc(tam);
  static unsigned int * v_seleccion;
  static unsigned int * v_puntuacion;

  Point2f* d_pt1;
  Point2f* d_pt2;

  float* d_theta;
  float* d_y_transl;
  float* d_x_transl;

  static unsigned int *d_puntuacion;
  static unsigned int *d_seleccion;

  float theta_ac_back=theta_ac.back();

  float m=perfil.m;
  float m_anterior=perfil_anterior.m;
  float horizonte=perfil.horizonte;
  float horizonte_anterior=perfil_anterior.horizonte;
```

```
  float pitch=perfil.pitch;
  float pitch_anterior=perfil_anterior.pitch;

#ifdef CALPTOS_V2
  CUDA_SAFE_CALL(cudaMalloc((void**) &d_pt1, (nelementos+1)*sizeof(Point2f))
      );
  CUDA_SAFE_CALL(cudaMalloc((void**) &d_pt2, (nelementos+1)*sizeof(Point2f))
      );
#endif
#if (defined(CALPTOS_V2) || defined(CALSOL_V3) || defined(CALSOL_V5))
  CUDA_SAFE_CALL(cudaMalloc((void**) &d_theta, tam));
  CUDA_SAFE_CALL(cudaMalloc((void**) &d_y_transl, tam));
  CUDA_SAFE_CALL(cudaMalloc((void**) &d_x_transl, tam));
#endif

  static curandState * devStates;

  if(!iniciado){

    CUDA_SAFE_CALL(cudaMalloc((void**) &d_seleccion, OD_NUM_THREADS*sizeof(
        unsigned int)));
    CUDA_SAFE_CALL(cudaMalloc((void**) &d_puntuacion, OD_NUM_THREADS*sizeof(
        unsigned int)));

    dim3 dimBlockRAND(512,1,1);
    dim3 dimGridRAND(OD_NUM_THREADS/512, 1, 1);
    CUDA_SAFE_CALL(cudaMalloc((void**) &devStates, OD_NUM_THREADS*sizeof(
        curandState)));
    setup_kernel<<<dimGridRAND, dimBlockRAND>>>(devStates, (unsigned long)
        time(NULL));

    v_seleccion=(unsigned int*)malloc(OD_NUM_THREADS*sizeof(unsigned int));
    v_puntuacion=(unsigned int*)malloc(OD_NUM_THREADS*sizeof(unsigned int));

    iniciado=1;
  }

  //Tamaños de bloque y grid
  dim3 dimBlockR4(512,1,1);
  dim3 dimGridR4((nelementos/512)+1, 1, 1);

  dim3 dimBlockR2(512,1,1);
    dim3 dimGridR2(OD_NUM_THREADS/512, 1, 1);

  //Cálculo de soluiones
#ifdef CALPTOS_V2
  CUDA_SAFE_CALL( cudaMemcpy(d_pt1, v_pt1, (nelementos+1)*sizeof(Point2f),
      cudaMemcpyHostToDevice) );
  CUDA_SAFE_CALL( cudaMemcpy(d_pt2, v_pt2, (nelementos+1)*sizeof(Point2f),
      cudaMemcpyHostToDevice) );
```

```
  calcularPtos_t<<<dimGridR4, dimBlockR4>>>(nelementos, m, m_anterior,
      horizonte, horizonte_anterior, pitch, pitch_anterior, d_pt1, d_pt2,
      d_theta, d_y_transl, d_x_transl, theta_ac_back);
  CUDA_SAFE_CALL( cudaMemcpy(v_theta, d_theta, tam, cudaMemcpyDeviceToHost)
      );
  CUDA_SAFE_CALL( cudaMemcpy(v_x_transl, d_x_transl, tam,
      cudaMemcpyDeviceToHost) );
  CUDA_SAFE_CALL( cudaMemcpy(v_y_transl, d_y_transl, tam,
      cudaMemcpyDeviceToHost) );
#else
  calcularPtos(nelementos, m, m_anterior, horizonte, horizonte_anterior,
      pitch, pitch_anterior, v_pt1, v_pt2, v_theta, v_y_transl, v_x_transl,
      theta_ac_back);
#endif

  //RANSAC
#ifdef CALSOL_V5
  ransacOdometria_t<<<dimGridR2, dimBlockR2>>>(d_seleccion, d_puntuacion,
      nelementos, d_theta, d_x_transl, d_y_transl,  devStates);
#elif defined(CALSOL_V4)
  ransacOdometria(v_seleccion, v_puntuacion, nelementos, v_theta, v_x_transl
      , v_y_transl);
#elif defined(CALSOL_V3)
  ransacOdometriaOriginal_t<<<dimGridR2, dimBlockR2>>>(d_seleccion,
      d_puntuacion, nelementos, d_theta, d_x_transl, d_y_transl,  devStates)
      ;
#elif defined(CALSOL_V2)
  ransacOdometriaOriginal(v_seleccion, v_puntuacion, nelementos, v_theta,
      v_x_transl, v_y_transl);
#endif
#if (defined(CALSOL_V3)||defined(CALSOL_V5))
  CUDA_SAFE_CALL( cudaMemcpy(v_seleccion, d_seleccion, OD_NUM_THREADS*sizeof
      (unsigned int), cudaMemcpyDeviceToHost) );
  CUDA_SAFE_CALL( cudaMemcpy(v_puntuacion, d_puntuacion, OD_NUM_THREADS*
      sizeof(unsigned int), cudaMemcpyDeviceToHost) );
#endif

  memcpy(r_ac_anterior, r_ac, 3*sizeof(float));

  for (int x=0; x<3; x++){
    unsigned int pos=(unsigned int)(x*((OD_NUM_THREADS)/3));

    int limite_inf=(unsigned int)(x*((OD_NUM_THREADS)/3));
    int limite_sup=(unsigned int)((x+1)*((OD_NUM_THREADS)/3));

    for (int i=limite_inf; i<limite_sup; i++){
      if (v_puntuacion[i]>v_puntuacion[pos]){
        pos=i;
      }
    }
```

```
    if (x==0)
      r_ac[0]=v_theta[v_seleccion[pos]];
    else if (x==1)
      r_ac[1]=v_x_transl[v_seleccion[pos]];
    else
      r_ac[2]=v_y_transl[v_seleccion[pos]];
  }

#ifdef FILTRO
  if ((r_ac[1]*r_ac[1]+r_ac[2]*r_ac[2])<UMBRAL_FILTRO){
    r_ac_sel=r_ac;
  }else{
    r_ac_sel=r_ac_anterior;
  }
#else
  r_ac_sel=r_ac;
#endif

  //Liberado de memoria
  theta_ac.push_back(r_ac_sel[0]+theta_ac.back());
  x_ac.push_back(r_ac_sel[1]+x_ac.back());
  y_ac.push_back(r_ac_sel[2]+y_ac.back());

  free(v_theta);
  free(v_x_transl);
  free(v_y_transl);

  CUDA_SAFE_CALL(cudaFree(d_pt1));
  CUDA_SAFE_CALL(cudaFree(d_pt2));
  CUDA_SAFE_CALL(cudaFree(d_theta));
  CUDA_SAFE_CALL(cudaFree(d_y_transl));
  CUDA_SAFE_CALL(cudaFree(d_x_transl));

}
#endif

void ransacOdometria(unsigned int* seleccion, unsigned int* puntuacion, int
    nelementos, float*theta, float*x_transl, float*y_transl)
{

  int j;

  int inliers;

  float rango;
  float sel[3], sel_aux;
  float * vector;

  unsigned int RANDOM[3], RANDOM_aux;

  srand((unsigned int)time(NULL));
```

```
  for (int x=0; x<OD_NUM_THREADS; x++){

    inliers=0;

    for (int i=0;i<3;i++){
      RANDOM[i]=rand()%nelementos;
    }

    if (x<((OD_NUM_THREADS)/3)){
      for (int n=0; n<3; n++){
        sel[n]=theta[RANDOM[n]];
      }
      vector=theta;
      rango=RANGO_THETA;
    }else if (x<(2*((OD_NUM_THREADS)/3))){
      for (int n=0; n<3; n++){
        sel[n]=x_transl[RANDOM[n]];
      }
      vector=x_transl;
      rango=RANGO_X;
    }else{
      for (int n=0; n<3; n++){
        sel[n]=y_transl[RANDOM[n]];
      }
      vector=y_transl;
      rango=RANGO_Y;
    }
    for(int i=1; i<3; i++){
      for(int j=0; j<=2-i;j++){
        if(sel[j]>sel[j+1]){
          sel_aux=sel[j];
          sel[j]=sel[j+1];
          sel[j+1]=sel_aux;
          RANDOM_aux=RANDOM[j];
          RANDOM[j]=RANDOM[j+1];
          RANDOM[j+1]=RANDOM_aux;
        }
      }
    }

    seleccion[x]=RANDOM[1];

    for (j=0; j<nelementos; j++){
      if(abs(sel[1]-vector[j]) < rango){
        inliers++;
      }
    }
    puntuacion[x]=inliers;
  }
}
```

```
void ransacOdometriaOriginal( unsigned int * seleccion,
                unsigned int * puntuacion,
                int num_ptos,
                float * theta,
                float * x_transl,
                float * y_transl)

{

  int j;

  int inliers=0;

  float rango;
  float sel;
  float * vector;

  unsigned int RANDOM;

  srand((unsigned int)time(NULL));

  for (int x=0; x<OD_NUM_THREADS; x++){

    RANDOM = rand()%num_ptos;

    if (x<((OD_NUM_THREADS)/3)){
      sel=theta[RANDOM];
      vector=theta;
      rango=RANGO_THETA;
    }else if (x<(2*((OD_NUM_THREADS)/3))){
      sel=x_transl[RANDOM];
      vector=x_transl;
      rango=RANGO_X;
    }else{
      sel=y_transl[RANDOM];
      vector=y_transl;
      rango=RANGO_Y;
    }

    for (j=0; j<num_ptos; j++){
      if(abs(sel-vector[j]) < rango){
        inliers++;
      }
    }

    puntuacion[x]=inliers;

  }
}
```

```
void calcularPtos(int n, float m, float m_anterior, float horizonte, float
    horizonte_anterior, float pitch, float pitch_anterior, Point2f* v_pt1,
    Point2f* v_pt2, float* v_theta, float *v_y_transl, float*v_x_transl,
    float theta_ac_back)
{
  for (int pos=0; pos<n; pos++){

    Point2f mundo_0, mundo_1,trans;
    Point2f pt_mundo_1;
    Point2f pt_mundo_0;
    float a,b,c,raiz,angulo1,angulo2,angulo;

    mundo_0.y = cos(pitch_anterior)*(m_anterior*F*B/(v_pt1[pos].y+
        ALTURA_SURF-horizonte_anterior));
    mundo_0.x = m_anterior*B*(v_pt1[pos].x-CX)/(v_pt1[pos].y+ALTURA_SURF-
        horizonte_anterior);

    mundo_1.y = cos(pitch)*(m*F*B/(v_pt2[pos].y+ALTURA_SURF-horizonte));
    mundo_1.x = m*B*(v_pt2[pos].x-CX)/(v_pt2[pos].y+ALTURA_SURF-horizonte);

    pt_mundo_0.x = cos(DESVIACION_YAW)*mundo_0.x + sin(DESVIACION_YAW)*
        mundo_0.y;
    pt_mundo_0.y = cos(DESVIACION_YAW)*mundo_0.y - sin(DESVIACION_YAW)*
        mundo_0.x + 1.4f;

    pt_mundo_1.x = cos(DESVIACION_YAW)*mundo_1.x + sin(DESVIACION_YAW)*
        mundo_1.y;
    pt_mundo_1.y = cos(DESVIACION_YAW)*mundo_1.y - sin(DESVIACION_YAW)*
        mundo_1.x + 1.4f;

    a = (pt_mundo_0.x)*(pt_mundo_0.x) + (pt_mundo_0.y)*(pt_mundo_0.y);
    b = 2*pt_mundo_1.x*pt_mundo_0.y;
    c = pt_mundo_1.x*pt_mundo_1.x - pt_mundo_0.x*pt_mundo_0.x;
    raiz = sqrt(b*b-4*a*c);

    angulo1 = asin((-b-raiz)/(2*a));
    angulo2 = asin((-b+raiz)/(2*a));
    angulo = (abs(angulo1)<abs(angulo2)) ? angulo1 : angulo2;

    v_theta[pos]=angulo;
    trans.x = pt_mundo_0.x - pt_mundo_1.x*cos(v_theta[pos])-pt_mundo_1.y*sin
        (v_theta[pos]);
    trans.y = pt_mundo_0.y + pt_mundo_1.x*sin(v_theta[pos])-pt_mundo_1.y*cos
        (v_theta[pos]);
    v_x_transl[pos]=cos(theta_ac_back)*trans.x + sin(theta_ac_back)*trans.y;
    v_y_transl[pos]=cos(theta_ac_back)*trans.y - sin(theta_ac_back)*trans.x;

  }
}
```

A.4 Código de los kernels del programa principal

Las funciones a continuación expuestas son los *kernels* que se ejecutan en la GPU (*device*) en paralelo. Se llaman desde las funciones expuestas en la sección A.3, con sintaxis:

```
funcion<<<GRID, BLOCK>>>(ARGUMENTOS);
```

`GRID` es una estructura `dim3` que contiene la configuración del *grid*; es decir, el número de bloques en cada dimensión. `BLOCK` es análogo para la estructura de los bloques, conteniendo el número de *threads en cada dimensión*.

```
#ifndef _CPP_INTEGRATION_KERNEL_H_
#define _CPP_INTEGRATION_KERNEL_H_
////////////////////
// INCLUSIONES
////////////////////
#include "parametros.h"
#include <curand_kernel.h>

////////////////////
// DEFINICIONES
////////////////////
#define LADO_VENTANA (5) //debe ser numero impar
#define MITAD_LADO (LADO_VENTANA-1)/2
#define SHARED_MEM_SIZE_LAPLACIANA (2*480*sizeof(float))
#define SQ(a) (__mul24(a,a))  // si se pone __mul24(a,a) se realiza SSD,si
    se pone abs(a) entonces SSA
#define mult(a,b) (__mul24(a,b))

typedef struct Point2f
{
    float x;
    float y;
}
Point2f;

int divUp(int a, int b)
{
  if(a%b == 0)
    return a/b;
  else
    return a/b + 1;
}
```

```
////////////////////
// TEXTURAS, MEMORIAS
////////////////////

texture<unsigned char, 2, cudaReadModeElementType> tex_izq,tex_der;

 __shared__ float f_1[480];
 __shared__ float f_2[480];


__global__ void
Laplaciana_gausiana_doble( unsigned char* g_odata_izq, unsigned char*
    g_odata_der,int width, int height)
{
    float pix_0;
  float pix_1;
  float pix_2;
    float f0;


  // calculate texture coordinates
    unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;

  //read from texture

    pix_0 = tex2D(tex_izq, x,y);
    pix_1 = tex2D(tex_izq, x-1,y)+tex2D(tex_izq,x+1,y);
    pix_2 = tex2D(tex_izq, x-2,y)+tex2D(tex_izq,x+2,y);



    f0     = 76*pix_0+10*pix_1-6*pix_2;
    f_1[y] = 10*pix_0-12*pix_1-5*pix_2;
    f_2[y] = -6*pix_0-5*pix_1-1*pix_2;

    __syncthreads();

    g_odata_izq[y*width + x] = (unsigned char)(127+(f0 + f_1[(y-1)] + f_1[(y
        +1)] + f_2[(y+2)] + f_2[(y-2)])/116);

    __syncthreads();

    pix_0 = tex2D(tex_der, x,y);
    pix_1 = tex2D(tex_der, x-1,y)+tex2D(tex_der,x+1,y);
    pix_2 = tex2D(tex_der, x-2,y)+tex2D(tex_der,x+2,y);

    f0     = 76*pix_0+10*pix_1-6*pix_2;
    f_1[y] = 10*pix_0-12*pix_1-5*pix_2;
    f_2[y] = -6*pix_0-5*pix_1-1*pix_2;
```

```
    __syncthreads();

    g_odata_der[y*width + x] = (unsigned char)(127+(f0 + f_1[(y-1)] + f_1[(y
        +1)] + f_2[(y+2)] + f_2[(y-2)])/116);

}

__device__ void

inicializar(unsigned char* g_odata,int x,int y, int width)
{
  g_odata[y*width + x] = 127; //Inicializamos el coste con el valor con
                //el valor de diferencia maxima posible
}

////////////////////
// KERNEL PARA EL CÁLCULO DE LA DISPARIDAD IZQUIERDA
////////////////////
__global__ void disparidadizda(unsigned char* izquierda,
               unsigned char* derecha,
               int *disparityMinSSD_izq,
               int *disparityMinSSD_der,
               int width,
               int height)
{

  extern __shared__ int col_ssd[];

  int d;
  int diff;
  int ssd;
  int x_tex;
  int y_tex;
  int x_der;
  int row;
  int i;

    #define X (__mul24(blockIdx.x,BLOCK_W) + threadIdx.x)
  #define Y (__mul24(blockIdx.y,ROWSperTHREAD))

  int extra_read_val = 0;
  if(threadIdx.x < (2*RADIUS_H)) extra_read_val = BLOCK_W+threadIdx.x;

  if(X<width )
  {
    for(i = 0;i<ROWSperTHREAD && Y+i < height;i++)
    {
      izquierda[__mul24((Y+i),width)+X] = VALOR_INICIAL;  // initialize that
          indicates no match
      derecha[__mul24((Y+i),width)+X] = VALOR_INICIAL;
```

```
      disparityMinSSD_izq[__mul24((Y+i),width)+X] = MIN_SSD;
      disparityMinSSD_der[__mul24((Y+i),width)+X] = MIN_SSD;
    }
  }
  __syncthreads();

  if( X < (width+2*RADIUS_H) && Y < height )
  {
  x_tex = X - RADIUS_H;
  for(d = STEREO_MIND; d <= STEREO_MAXD; d++)
  {

    col_ssd[threadIdx.x] = 0;
    if(extra_read_val>0)  col_ssd[extra_read_val] = 0;

    y_tex = Y - RADIUS_V;

    for(i = 0; i <= 2*RADIUS_V; i++)
    {
      diff = tex2D(tex_izq,x_tex,y_tex) - tex2D(tex_der,x_tex-d,y_tex);
      col_ssd[threadIdx.x] += SQ(diff);

      if(extra_read_val > 0)
      {
        diff = tex2D(tex_izq,x_tex+BLOCK_W,y_tex) - tex2D(tex_der,x_tex+
            BLOCK_W-d,y_tex);
        col_ssd[extra_read_val] += SQ(diff);
      }
      y_tex += 1;
    }
    __syncthreads();

    if(X < width && Y < height)
    {
      ssd = 0;
      for(i = 0;i<=(2*RADIUS_H);i++)
      {
        ssd += col_ssd[i+threadIdx.x];
      }

      if( ssd  < disparityMinSSD_izq[__mul24(Y,width) + X])
      {
        izquierda[__mul24(Y,width) + X]= (unsigned char)(d);
        disparityMinSSD_izq[Y*width + X] = ssd;
      }
      x_der = X -d;
      if((x_der >=0) && ssd  < disparityMinSSD_der[__mul24(Y,width) + x_der
          ])
      {
        derecha[__mul24(Y,width) + x_der]= (unsigned char)(d);
        disparityMinSSD_der[Y*width + x_der] = ssd;
```

```
      }
    }
    __syncthreads();

    y_tex = Y - RADIUS_V;
    for(row = 1;row < ROWSperTHREAD && (row+Y < (height+RADIUS_V)); row++)
    {

      diff = tex2D(tex_izq,x_tex,y_tex) - tex2D(tex_der,x_tex-d,y_tex);
      col_ssd[threadIdx.x] -= SQ(diff);

        diff = tex2D(tex_izq,x_tex,y_tex + 2*RADIUS_V+1) - tex2D(tex_der,
            x_tex-d,y_tex + 2*RADIUS_V+1);
      col_ssd[threadIdx.x] += SQ(diff);

      if(extra_read_val > 0)
      {
        diff =  tex2D(tex_izq,x_tex+BLOCK_W,y_tex) - tex2D(tex_der,x_tex-d+
            BLOCK_W,y_tex);
        col_ssd[extra_read_val] -= SQ(diff);

        diff = tex2D(tex_izq,x_tex+BLOCK_W,y_tex + 2*RADIUS_V+1) - tex2D(
            tex_der,x_tex-d+BLOCK_W,y_tex + 2*RADIUS_V+1);
        col_ssd[extra_read_val] += SQ(diff);
      }
      y_tex += 1;
        __syncthreads();

      if(X<width && (Y+row) < height)
      {
        ssd = 0;

        for(i = 0;i<=(2*RADIUS_H);i++)
        {
          ssd += col_ssd[i+threadIdx.x];
        }
        if(ssd < disparityMinSSD_izq[__mul24(Y+row,width) + X])
        {
          izquierda[__mul24(Y+row,width) + X] = (unsigned char)(d);
          disparityMinSSD_izq[__mul24(Y+row,width) + X] = ssd;
        }
        if((x_der >=0) && ssd < disparityMinSSD_der[__mul24(Y+row,width) +
            x_der])
        {
          derecha[__mul24(Y+row,width) + x_der] = (unsigned char)(d);
          disparityMinSSD_der[__mul24(Y+row,width) + x_der] = ssd;
        }
      }
      __syncthreads();
    }
  }
```

```
  }
}
__global__ void cross_checking(unsigned char* izquierda,int width)
{

    unsigned int x_tex = blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int y_tex = blockIdx.y*blockDim.y + threadIdx.y;

  int diff = tex2D(tex_izq,x_tex,y_tex) - tex2D(tex_der,x_tex,y_tex);
  if(abs(diff) > 1)
  {
    izquierda[__mul24(y_tex,width) + x_tex] = 0;
  }
  else
  {
    izquierda[__mul24(y_tex,width) + x_tex] = tex2D(tex_izq,x_tex,y_tex);
  }
}

__global__ void u_disparity(unsigned char* u,int width,int height)
{

    unsigned int x_tex = blockIdx.x*blockDim.x + threadIdx.x;

  int histograma[(int)STEREO_MAXD];

  for(int i=0;i<STEREO_MAXD;i++)
  {
    histograma[i]= 0;
  }
  for(int i=0;i<height;i++)
  {
    histograma[tex2D(tex_izq,x_tex,i)] = histograma[tex2D(tex_izq,x_tex,i)
        ]+1;
  }
  histograma[0] = 0;
  for(int i=0;i<STEREO_MAXD;i++)
  {
#ifdef U_DISP_UMBRALIZADO
    if(histograma[i] > 25)
    {
      histograma[i] = 255;
    }
    else
    {
      histograma[i] = 0;
    }
#endif
    u[__mul24(i,width) + x_tex] = histograma[i];
  }
}
```

```
__global__ void v_disparity(unsigned char* v,int width,int height)
{

    unsigned int y_tex = blockIdx.y*blockDim.y + threadIdx.y;
  int histograma[(int)STEREO_MAXD];


  for(int i=0;i<STEREO_MAXD;i++)
  {
    histograma[i] = 0;
  }
  for(int i=0; i<width; i++)
  {
    if(histograma[tex2D(tex_izq,i,y_tex)] < 255)
    {
      histograma[tex2D(tex_izq,i,y_tex)] = histograma[tex2D(tex_izq,i,y_tex)
          ]+1;
    }
  }
  histograma[0] = 0;
  for(int i=0;i<STEREO_MAXD;i++)
  {
    v[__mul24(y_tex,STEREO_MAXD) + i] = (unsigned char)histograma[i];
  }

}

__global__ void mapa_obstaculos(unsigned char* mapa_obstaculos,unsigned char
    * u_disparity,int d,int width,int height)
{

    unsigned int x_tex = blockIdx.x*blockDim.x + threadIdx.x;

  if(u_disparity[__mul24(d,width) + x_tex] == 255)
  {
    for(int i=0;i<height;i++)
    {
      if (tex2D(tex_izq,x_tex,i) == d)
      {
        mapa_obstaculos[__mul24(i,width) + x_tex] = (unsigned char)(tex2D(
            tex_izq,x_tex,i));
      }
    }
  }
}
__global__ void inicializacion(unsigned char* imagen,int width,int height)
{

    unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;
```

```
  imagen[__mul24(y,width) + x] = 0;

}
__global__ void mapa_libre(unsigned char* mapa_libre,int width,int height)
{

    unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;

  if(tex2D(tex_der,x,y) == 0)
  {
    mapa_libre[__mul24(y,width) + x] = (unsigned char)(tex2D(tex_izq,x,y));
    //mapa_obstaculos[__mul24(i,width) + x_tex] = tex2D(tex_izq,x_tex,i);
  }
}
__global__ void filtradas(unsigned char* obstaculos_umbralizado,int width,
    int height)
{

    unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;

  unsigned char permitadas[(int)STEREO_MAXD] =
      {0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};

  if(tex2D(tex_der,x,y) > UMBRAL_AREA_ESTUDIO && (abs(tex2D(tex_der,x-1,y)-
      tex2D(tex_der,x+1,y)) <= permitadas[tex2D(tex_der,x,y)-1]) && (abs(
      tex2D(tex_der,x,y-1)-tex2D(tex_der,x,y+1)) <= permitadas[tex2D(tex_der
      ,x,y)-1]))
  {
      obstaculos_umbralizado[__mul24(y,width) + x] = 255;
  }
}

////////////////////
// Kernel para la implementación de RANSAC en rectas
////////////////////
__global__ void ransac_t(unsigned int * puntos_blancos, int num_ptos_blancos
    , int * puntuacion, float * puntos_seleccionados, curandState *
    globalState, float rango)
{

  int x = blockIdx.x*blockDim.x + threadIdx.x;


  // Generación números aleatorios
  curandState localState=globalState[x];
  unsigned int RANDOM1 = num_ptos_blancos*curand_uniform(&localState);
  unsigned int RANDOM2;
  do{
```

```
  RANDOM2 = num_ptos_blancos*curand_uniform(&localState);
  }while (RANDOM1==RANDOM2);
  globalState[x]=localState;

  float m=0,b=0;
  float x1,y1,x2,y2;

  int num_inliers=0;
  unsigned int n;

  x1=*(puntos_blancos+RANDOM1);
  y1=*(puntos_blancos+RANDOM1+MAX_PTOS_BLANCOS);
  x2=*(puntos_blancos+RANDOM2);
  y2=*(puntos_blancos+RANDOM2+MAX_PTOS_BLANCOS);

  if (x1!=x2){
    m=(y1-y2)/(x1-x2);
    b=y1-m*x1;

    for (n=0;n<num_ptos_blancos;n++){
      if ((abs(*(puntos_blancos+n+MAX_PTOS_BLANCOS)-m*(*(puntos_blancos+n))-
          b))<rango){
        num_inliers++;
      }
    }
  }else{
    num_inliers=-1;
  }

  *(puntuacion+x)=num_inliers;
  *(puntos_seleccionados+x)=m;
  *(puntos_seleccionados+x+RP_NUM_THREADS)=b;
}

////////////////////
// Kernel para generar la semilla de generación de números aleatorios
////////////////////
__global__ void setup_kernel(curandState * state, unsigned long seed=1234)
{
  int id = blockIdx.x*blockDim.x + threadIdx.x;
  curand_init(seed, id, 0, &state[id]);
}

////////////////////
// Kernel para la implementación de RANSAC en el cálculo de la solución
    final CON MEDIANA
////////////////////
__global__ void ransac0dometria_t(unsigned int * seleccion,
                  unsigned int * puntuacion,
                  int num_ptos,
                  float * theta,
```

```
                    float * x_transl,
                    float * y_transl,
                    curandState *globalState)

{
  int x = blockIdx.x*blockDim.x + threadIdx.x;

  int j;

  int inliers=0;

  float rango;
  float sel[3], sel_aux;
  float * vector;

  unsigned int RANDOM[3], RANDOM_aux;

  curandState localState=globalState[x];
  for (int n=0; n<3; n++){
    RANDOM[n] = num_ptos*curand_uniform(&localState);
  }
  globalState[x]=localState;

  if (x<((OD_NUM_THREADS)/3)){
    for (int n=0; n<3; n++){
      sel[n]=theta[RANDOM[n]];
    }
    vector=theta;
    rango=RANGO_THETA;
  }else if (x<(2*((OD_NUM_THREADS)/3))){
    for (int n=0; n<3; n++){
      sel[n]=x_transl[RANDOM[n]];
    }
    vector=x_transl;
    rango=RANGO_X;
  }else{
    for (int n=0; n<3; n++){
      sel[n]=y_transl[RANDOM[n]];
    }
    vector=y_transl;
    rango=RANGO_Y;
  }

  for(int i=1; i<3; i++){
    for(int j=0; j<=2-i;j++){
      if(sel[j]>sel[j+1]){
        sel_aux=sel[j];
        sel[j]=sel[j+1];
        sel[j+1]=sel_aux;
        RANDOM_aux=RANDOM[j];
        RANDOM[j]=RANDOM[j+1];
```

```
        RANDOM[j+1]=RANDOM_aux;
      }
    }
  }

  seleccion[x]=RANDOM[1];

  for (j=0; j<num_ptos; j++){
    if(abs(sel[1]-vector[j]) < rango){
      inliers++;
    }
  }

  puntuacion[x]=inliers;

}

////////////////////
// Kernel para la resolución de las ecuaciones
////////////////////
__global__ void calcularPtos_t(int n, float m, float m_anterior, float
    horizonte, float horizonte_anterior, float pitch, float pitch_anterior,
    Point2f* pt1, Point2f* pt2, float* theta, float *y_transl, float*
    x_transl, float theta_ac_back)
{

  int pos = mult(blockIdx.x,blockDim.x) + threadIdx.x +1;

  if (pos<=n){

  Point2f mundo_0, mundo_1,trans;
  Point2f pt_mundo_1;
  Point2f pt_mundo_0;
  float a,b,c,raiz,angulo1,angulo2,angulo;

  mundo_0.y = cos(pitch_anterior)*(m_anterior*F*B/(pt1[pos].y+ALTURA_SURF-
      horizonte_anterior));
  mundo_0.x = m_anterior*B*(pt1[pos].x-CX)/(pt1[pos].y+ALTURA_SURF-
      horizonte_anterior);

  mundo_1.y = cos(pitch)*(m*F*B/(pt2[pos].y+ALTURA_SURF-horizonte));
  mundo_1.x = m*B*(pt2[pos].x-CX)/(pt2[pos].y+ALTURA_SURF-horizonte);

  pt_mundo_0.x = cos(DESVIACION_YAW)*mundo_0.x + sin(DESVIACION_YAW)*mundo_0
      .y;
  pt_mundo_0.y = cos(DESVIACION_YAW)*mundo_0.y - sin(DESVIACION_YAW)*mundo_0
      .x + 1.4;

  pt_mundo_1.x = cos(DESVIACION_YAW)*mundo_1.x + sin(DESVIACION_YAW)*mundo_1
      .y;
```

```
  pt_mundo_1.y = cos(DESVIACION_YAW)*mundo_1.y - sin(DESVIACION_YAW)*mundo_1
      .x + 1.4;

  a = (pt_mundo_0.x)*(pt_mundo_0.x) + (pt_mundo_0.y)*(pt_mundo_0.y);
  b = 2*pt_mundo_1.x*pt_mundo_0.y;
  c = pt_mundo_1.x*pt_mundo_1.x - pt_mundo_0.x*pt_mundo_0.x;
  raiz = sqrt(b*b-4*a*c);

  angulo1 = asin((-b-raiz)/(2*a));
  angulo2 = asin((-b+raiz)/(2*a));
  angulo = (abs(angulo1)<abs(angulo2)) ? angulo1 : angulo2;

  theta[pos]=angulo;
  trans.x = pt_mundo_0.x - pt_mundo_1.x*cos(theta[pos])-pt_mundo_1.y*sin(
      theta[pos]);
  trans.y = pt_mundo_0.y + pt_mundo_1.x*sin(theta[pos])-pt_mundo_1.y*cos(
      theta[pos]);
  x_transl[pos]=cos(theta_ac_back)*trans.x + sin(theta_ac_back)*trans.y;
  y_transl[pos]=cos(theta_ac_back)*trans.y - sin(theta_ac_back)*trans.x;

  }
}
#endif

////////////////////
// Kernel para la implementación de RANSAC en el cálculo de la solución
    final SIN MEDIANA
////////////////////
__global__ void ransacOdometriaOriginal_t(unsigned int * seleccion,
                  unsigned int * puntuacion,
                  int num_ptos,
                  float * theta,
                  float * x_transl,
                  float * y_transl,
                  curandState *globalState)

{
  int x = blockIdx.x*blockDim.x + threadIdx.x;

  int j;

  int inliers=0;

  float rango;
  float sel;
  float * vector;


  curandState localState=globalState[x];
  unsigned int RANDOM = num_ptos*curand_uniform(&localState);
  seleccion[x]=RANDOM;
```

```
  globalState[x]=localState;


  if (x<((OD_NUM_THREADS)/3)){
    sel=theta[RANDOM];
    vector=theta;
    rango=RANGO_THETA;
  }else if (x<(2*((OD_NUM_THREADS)/3))){
    sel=x_transl[RANDOM];
    vector=x_transl;
    rango=RANGO_X;
  }else{
    sel=y_transl[RANDOM];
    vector=y_transl;
    rango=RANGO_Y;
  }


  for (j=0; j<num_ptos; j++){
    if(abs(sel-vector[j]) < rango){
      inliers++;
    }
  }
  puntuacion[x]=inliers;
}

#endif // #ifndef _CPP_INTEGRATION_KERNEL_H_
```

A.5 Código del programa auxiliar

El código fuente que se expone a continuación constituye el programa auxiliar destinado a ejecutarse en paralelo con el programa principal, y que se encarga de detectar los puntos característicos mientras este se encuentra aún procesando el mapa de disparidad y el perfil de la calzada.

```
//Define si se usa la´"búsqueda aproximada del vecino más próximo"
#define USE_FLANN

//INCLUDES
//Inclusión de las bibliotecas del sistema
#include <iostream>
#include <windows.h>
using namespace std;

//Inclusión de las bibliotecas de visión
```

```
//OpenCV
#include <opencv2/nonfree/nonfree.hpp>
#include <cv.h>
#include <cxcore.h>
#include <highgui.h>
using namespace cv;

//Inclusión de los parametros de control
#include "parametros.h"

//Inclusión del funciones complementarias para el SURF
#include "funcionesSURF.cpp"

//ESTRUCTURAS
struct punto{
  int x;
  int y;
};

//////////////////
// PROGRAMA
//////////////////
int main(int argc, char** argv)
{
  initModule_nonfree();

  //Declaración de variables
  int const size = WIDTH_IMAGEN*HEIGHT_IMAGEN*sizeof(unsigned char);
  bool hayPuntos = FALSE;
  vector<int> ptpairs;

  //-Memoria compartida
  HANDLE hMapFile=NULL, hMapFile2=NULL, hMapFile3=NULL;
  unsigned char* pBuf;
  Point2f* pBuf2;
  Point2f* pBuf3;
  TCHAR szName[]=TEXT("Global\\ShMemImage");
  TCHAR szName2[]=TEXT("Global\\ShMemV1");
  TCHAR szName3[]=TEXT("Global\\ShMemV2");

  //-Semáforos
  HANDLE ghSemaphoreWIm=NULL, ghSemaphoreRIm=NULL, ghSemaphoreWVec=NULL,
      ghSemaphoreRVec=NULL;
  TCHAR semNameWIm[]=TEXT("Global\\SemWriteImage");
  TCHAR semNameRIm[]=TEXT("Global\\SemReadImage");
  TCHAR semNameWVec[]=TEXT("Global\\SemWriteVectors");
  TCHAR semNameRVec[]=TEXT("Global\\SemReadVectors");
  DWORD dwWaitResult=WAIT_OBJECT_0;

  //Reserva de memoria
  unsigned char* h_izq= (unsigned char*) malloc(size);
```

```
unsigned char* h_der= (unsigned char*) malloc(size);

Point2f* pt1=(Point2f*)malloc((MAX_PTOS_SURF+1)*sizeof(Point2f));
Point2f* pt2=(Point2f*)malloc((MAX_PTOS_SURF+1)*sizeof(Point2f));

//-OpenCV
IplImage *izq_visible_1 = cvCreateImageHeader(cvSize(WIDTH_IMAGEN,
    HEIGHT_IMAGEN),8,1);
char* visible_aux =  (char*)calloc(((WIDTH_IMAGEN+2)*HEIGHT_IMAGEN),sizeof
    (char));

CvSeq *anteriorKeypoints = 0, *anteriorDescriptors = 0;
CvSeq *Keypoints = 0, *Descriptors = 0;
CvMemStorage* storage = cvCreateMemStorage(0);
CvMemStorage* storage_anterior = cvCreateMemStorage(0);

const CvSURFParams params = cvSURFParams(100, 0);

cout << "Buscando programa principal..." << endl;

while((hMapFile==NULL)||(hMapFile2==NULL)||(hMapFile3==NULL)||(
    ghSemaphoreWIm==NULL)||(ghSemaphoreRIm==NULL)||(ghSemaphoreWVec==NULL)
    ||(ghSemaphoreRVec==NULL)){

  //MEMORIA COMPARTIDA
  //-Para el paso de la imagen
  hMapFile = OpenFileMapping(FILE_MAP_ALL_ACCESS,   // read/write access
                FALSE,                 // do not inherit the name
                szName);               // name of mapping object

  if (hMapFile !=NULL){
    pBuf = (unsigned char*) MapViewOfFile(  hMapFile,       // handle to
        map object
                        FILE_MAP_ALL_ACCESS,  // read/write permission
                        0,
                        0,
                        BUF_SIZE);
  }

  //-Para el paso del primer vector de puntos
  hMapFile2 = OpenFileMapping(FILE_MAP_ALL_ACCESS,   // read/write access
                FALSE,                 // do not inherit the name
                szName2);              // name of mapping object
  if (hMapFile2 != NULL){

    pBuf2 = (Point2f*)MapViewOfFile(hMapFile2,        // handle to map
        object
                    FILE_MAP_ALL_ACCESS,  // read/write permission
                    0,
                    0,
                    BUF_SIZE2);
```

```
    }

    //-Para el paso del segundo vector de puntos
    hMapFile3 = OpenFileMapping(FILE_MAP_ALL_ACCESS,   // read/write access
                  FALSE,                 // do not inherit the name
                  szName3);              // name of mapping object

    if (hMapFile3 != NULL){

      pBuf3 = (Point2f*)MapViewOfFile(hMapFile3,        // handle to map
          object
                      FILE_MAP_ALL_ACCESS,  // read/write permission
                      0,
                      0,
                      BUF_SIZE2);

    }

    //SEMÁFOROS
    //-Para controlar la escritura de la imagen
    ghSemaphoreWIm = OpenSemaphore(SYNCHRONIZE|SEMAPHORE_MODIFY_STATE, FALSE
        , semNameWIm);

    //-Para controlar la lectura de la imagen
    ghSemaphoreRIm = OpenSemaphore(SYNCHRONIZE|SEMAPHORE_MODIFY_STATE, FALSE
        , semNameRIm);

    //-Para contolar la escritura de los vectores de puntos
    ghSemaphoreWVec = OpenSemaphore(SYNCHRONIZE|SEMAPHORE_MODIFY_STATE,
        FALSE, semNameWVec);

    //-Para contolar la lectura de los vectores de puntos
    ghSemaphoreRVec = OpenSemaphore(SYNCHRONIZE|SEMAPHORE_MODIFY_STATE,
        FALSE, semNameRVec);
  }

  cout << "Programa principal encontrado" << endl;

  if ((pBuf != NULL)&&(pBuf2 != NULL)&&(pBuf3 != NULL)){

  for(;;){

    //Recepción de la imagen
    dwWaitResult=WaitForSingleObject(ghSemaphoreRIm, 10000);
        //Si en 10 s no se recibe imagen, se cierra el programa.

    if (dwWaitResult==WAIT_TIMEOUT){
      cout << "No se ha recibido imagen" << endl;
      break;
    }
```

```
memcpy(h_izq, pBuf, size);

if(!ReleaseSemaphore(ghSemaphoreWIm,1,NULL)) printf("ReleaseSemaphore
    error: %d\n", GetLastError());

//SURF
ptpairs.clear();

for(int y=0; y<HEIGHT_IMAGEN; y++)
{
  for(int x=0; x<WIDTH_IMAGEN; x++)
  {
    visible_aux[izq_visible_1->widthStep*y+x*izq_visible_1->nChannels]=
        h_izq[y*(int)WIDTH_IMAGEN+x];
  }
}
izq_visible_1->imageData = visible_aux;

cvEqualizeHist(izq_visible_1, izq_visible_1);

cvSetImageROI(izq_visible_1,cvRect(0,ALTURA_SURF,WIDTH_IMAGEN,
    HEIGHT_IMAGEN));

cvClearMemStorage(storage_anterior);

if(hayPuntos)
{
  anteriorKeypoints = cvCloneSeq((const CvSeq*)Keypoints,
      storage_anterior);
  anteriorDescriptors = cvCloneSeq((const CvSeq*)Descriptors,
      storage_anterior);
}

cvClearMemStorage(storage);

cvExtractSURF(izq_visible_1, 0, &Keypoints, &Descriptors, storage,
    params);

if(hayPuntos)
{
#ifndef USE_FLANN
  findPairs( anteriorKeypoints, anteriorDescriptors, Keypoints,
      Descriptors, ptpairs);
#else
  flannFindPairs( anteriorKeypoints, anteriorDescriptors, Keypoints,
      Descriptors, ptpairs);
#endif
}

hayPuntos=((Keypoints->total)!=0);
```

```
    int n = (int)(ptpairs.size()/2);

    pt1[0].x=(float)n;
    pt2[0].x=(float)n;

    for(int pos = 0; pos < n; pos++ )
    {
      pt1[pos+1] = ((CvSURFPoint*)cvGetSeqElem(anteriorKeypoints,ptpairs[pos
          *2]))->pt;
      pt2[pos+1] = ((CvSURFPoint*)cvGetSeqElem(Keypoints,ptpairs[pos*2+1]))
          ->pt;
    }

    //Envío de los vectores de puntos
    WaitForSingleObject(ghSemaphoreWVec, INFINITE);

    CopyMemory((PVOID)pBuf2, pt1, ((int)(pt1[0].x)+1)*sizeof(Point2f));
    CopyMemory((PVOID)pBuf3, pt2, ((int)(pt2[0].x)+1)*sizeof(Point2f));

    if(!ReleaseSemaphore(ghSemaphoreRVec,1,NULL)) printf("ReleaseSemaphore
        error: %d\n", GetLastError());

  }

  }


  free(visible_aux);
  free(h_izq);
  free(h_der);
  free(pt1);
  free(pt2);
  cvClearSeq(Keypoints);
  cvClearSeq(anteriorKeypoints);
  cvClearSeq(Descriptors);
  cvClearSeq(anteriorDescriptors);
  cvClearMemStorage(storage);
  cvClearMemStorage(storage_anterior);
  cvReleaseMemStorage(&storage);
  cvReleaseMemStorage(&storage_anterior);
  cvReleaseImage(&izq_visible_1);

  UnmapViewOfFile(pBuf);
  UnmapViewOfFile(pBuf2);
  UnmapViewOfFile(pBuf3);

  CloseHandle(hMapFile);
  CloseHandle(hMapFile2);
  CloseHandle(hMapFile3);

  cout << "\nENTER para cerrar" << endl;
```

```
  getchar();
}
```