Universidad Carlos III de Madrid

Escuela Politécnica Superior

Ingeniería Industrial

Proyecto Fin De Carrera

Desarrollo De Un Algoritmo De Auto-Calibración De Sistemas Estéreo Basado En El Uso Del UV-*Disparity*. Aplicación A La Odometría Visual

Autora: Irene Vázquez García
Tutor: Basam Musleh Lancis

Octubre de 2012

Resumen

El objetivo de este proyecto es el de presentar una solución que calibre de forma autónoma los parámetros extrínsecos de un par de cámaras estéreo cuyo fin es trazar la trayectoria del vehículo mediante un algoritmo de odometría visual.

La calibración de la cámara ha de realizarse en cada instante de captura de la cámara para así poder minimizar los errores debidos a posibles vibraciones del vehículo mientras este está en movimiento. Gracias a ello, los resultados serán más precisos y robustos.

Para ello, se procederá a introducir las fórmulas que relacionan el entorno con el sistema de la cámara, y poder con ello efectuar una calibración de ellas antes de calcular la trayectoria del vehículo.

El algoritmo ha sido diseñado en CUDA (*Compute Unified Device Architecture*) para poder satisfacer con la demanda de bajo tiempo de cómputo necesaria para poder llevar a cabo la empresa.

El código implementado será introducido en el prototipo de vehículo inteligente de la UC3M (*Universidad Carlos III de Madrid*), IVVI (*Intelligent Vehicle Based On Visual Information*) 2.0 para poder formar un sistema de navegación dependiente únicamente de los sensores instalados en él.

Abstract

THE aim of this project is to present a solution that calibrates automatically the extrinsic parameters of stereo vision pairs of cameras, whose objective is to determine the path of the vehicle due to a visual odometry algorithm.

The calibration of the camera must be done in each frame, in order to minimize the errors that occur because of the car's vibrations while it is being driven. Thanks to this improvements, the results will be more accurate and robust.

For this purpose, formulas that relate the environment with the camera coordinate system will be introduced, so that calibration takes place before the algortihm calculates the current position of the vehicle.

The algorithm has been designed in CUDA (*Compute Unified Device Architecture*), in order to satisfy with the low computation time request that is needed in this problem.

The implemented code will be used in the intelligent vehicle owned by the UC3M (*Universidad Carlos III de Madrid*), called IVVI (*Intelligent Vehicle Based On Visual Information*) 2.0, to create a navigation system that only depends on the on-board sensors.

Agradecimientos

Con este proyecto termina una etapa, etapa que pese a algunas decepciones, también ha estado llena de momentos de alegría. No me gustaría terminarla, sin poder expresar mis agradecimientos a aquellas personas que, pese a que no han escrito este proyecto conmigo, les pertenece una parte de él.

Lo primero, a mis padres, por haber entendido que llegar a este punto era importante para mí, por no haber escatimado en ningún aspecto para poder lograrlo, y, ante todo, por la convicción que siempre me han demostrado, de que confiaban en mi para que hoy logre este reto. A mi padre, porque ha celebrado cada batalla como propia, y las frustraciones las ha entendido al rememorar sus experiencias. A mi madre, porque siempre ha tenido palabras amables y tranquilizadoras, cuando todo estaba negro. Al fin y al cabo, ella "es la que peor, de todos, lo ha pasado durante los exámenes"...

A mi familia, porque pese a que han vivido desde la distancia todo el proceso, se han enorgullecido de mí, y se han preocupado por intentar entender cómo funcionaba la universidad y las asignaturas, y han animado de la mejor forma posible. Nunca han dejado de creer tampoco en mis posibilidades, y sé que este proyecto les hace muchísima ilusión.

No puedo dejar de dar las gracias a Carlos, pues nadie entiende mejor que él lo que ha sido el último año. Los buenos momentos, los malos, que también los ha habido, siempre ha estado ahí. Gracias por querer sacar en todo momento lo mejor de mí, por no haber dejado nunca que el pesimismo nos embargase sobremanera. Siempre ha estado dispuesto a explicar las cosas tantas veces fuera necesario, y es el que ha absorbido en primera aproximación cada mal rato pasado, y sin embargo se ha alegrado como si fuera su propio triunfo cada vez que me ha salido algo bien.

Son muchas las personas a las que he conocido, y con las que estos años no hubieran sido lo mismo. Cuando, dentro de un tiempo, rememore estos años, supongo que me acordaré de todas las conversaciones, comidas y paseos que hemos mantenido más que de la teoría impartida. Me gustaría agradecer a Adrián y Yeny, porque la suerte nos fue dispar, pero eso nunca ha sido óbice para mantener una relación duradera. A Javier, Santiago, Sergio, Marta, Mario y otros tantos que no me caben aquí, porque ir a clase ha sido un placer. A los chicos de la especialidad, porque nunca un día fue igual al anterior.

Quiero agradecer a Juan que confiase en mí sin esperar nada a cambio. Fue realmente importante que creyese en mí, y me aportase tanto conocimiento. Desde donde quiera que esté, sé que le hubiera encantado este proyecto.

Finalmente, dar las gracias a mi tutor, por siempre tener tiempo para ayudar a mejorar este proyecto y en todo momento aportar nuevas ideas.

Gracias a todos.

Irene

ÍNDICE GENERAL

1. Introducción **1**

1.1. El Vehículo Inteligente 4

1.1.1. Proyectos Comerciales 5

1.1.2. Proyectos Científicos 8

1.2. La Percepción del Entorno 11

1.2.1. *Cámaras Monoculares* 13

1.2.2. *Cámaras Omnidireccionales* 13

1.2.3. *Sistemas Estéreo* 13

1.2.4. *Autocalibración de la Cámara* 14

1.3. Localización del Vehículo 15

1.3.1. Sistemas De Posicionamiento Global 15

1.3.2. La Odometría Visual 17

1.4. Filosofías de Programación 18

2. Estado del Arte **20**

2.1. Trabajos Existentes Basados en Cámaras Monoculares 20

2.2. Trabajos Existentes Basados en Par Estéreo 21
2.3. Trabajos Existentes Acerca de Calibración 25

3. Fundamentos Teóricos 28

3.1. Conceptos Físicos 29
3.1.1. La Visión Estéreo 29
3.2. Manejo de Imágenes 34
3.2.1. Prepocesamiento de Imágenes 34
Laplaciana de la Gaussiana 35
3.2.2. El Mapa de Disparidad 36
3.2.3. u-v *Disparity* 38
v-*Disparity* 39
u-*Disparity* 40
3.2.4. El Mapa Libre y el Mapa de Obstáculos 40
3.3. Herramientas 41
3.3.1. CUDA 41
Programación en *CUDA* 43
Modelo de *Host* y *Device* 46
Ventajas de Usar *CUDA* 46
3.3.2. OpenCV 48
3.3.3. Matrox MIL 49
3.4. Algoritmos 50
3.4.1. RANSAC 50
3.4.2. SURF 53
El Detector Hessiano 54

Descriptores 56
3.4.3. FLANN 56
3.5. Conocimientos Básicos de Vehículos Necesarios 57

4. Cálculo de la Odometría Visual 59

4.1. Obtención del Mapa de Disparidad y u-v *Disparity* 59
4.2. Obtención y Uso del Perfil de la Calzada 61
4.2.1. Implementación Del Algoritmo 63
4.3. Detección de Puntos Característicos de la Calzada 66
4.3.1. Implementación Del Algoritmo 68
4.4. Cálculo de la Estimación de Movimiento Entre *Frames* Consecutivos 72
4.4.1. Implementación del Algoritmo del Cálculo de Resultados 74
4.4.2. Implementación del Algoritmo de Búsqueda de una Solución Única 75

5. Autocalibración 76

5.1. Calibración del *yaw* 79
5.2. Calibración de la altura, *pitch* y *roll* 80
5.3. Obtención de las Coordenadas del Mundo 85

6. Resultados Experimentales Obtenidos 87

6.1. Resultados Previos al Algoritmo Propuesto 90
6.2. Calibración del *Yaw* 92
6.3. Calibración del *Pitch* 93
6.4. Calibración del *Roll* 94

6.5. Odometría Visual Con Calibración de sus Parámetros. Comprobación de Puntos en la Calzada. 97

6.6. Resultados de la Estimación del *Pitch* 99

6.7. Resultados de la Estimación del *Roll* 101

6.8. Error Cometido 103

7. Conclusiones y Trabajos Futuros 104

8. Costes del Proyecto 107

Apéndices 109

A. Parámetros 110

B. Main 112

C. Kernel 130

D. Host 144

E. SURF 162

F. Funciones de SURF 171

LISTA DE FIGURAS

1.1. Datos Accidentes Año 2000 3

1.2. Datos Accidentes Año 2010 3

1.3. Sistema *HAVEit* 6

1.4. Tecnología *simTD* 7

1.5. Vehículo Inteligente Google 9

1.6. Percepción del Entorno de un Google Car 9

1.7. Vehículo Inteligente *IvvI* 11

1.8. Sistemas de Percepción de un Vehículo Inteligente 12

1.9. Cámara *Bumblebee2* 14

1.10. Sistema *Galileo* 16

2.1. Resultados Arrojados Por El Sistema de Odometría Visual Propuesto por Scaramuzza et al. FOV=Puntos de Vista 21

2.2. Trazado Por Odometría Visual (Verde) Frente A Encoders (Azul) 23

2.3. Curiosity 23

2.4. Captación de Puntos Característicos en la Superficie de Marte 24

2.5. *Pose* de un Satélite (Envisat en la Imagen) 24

2.6. Sistema de autocalibración planteado por Dang et al. C1, C2: Cámaras Estéreo, M: Espejo, C3: Cámara Monocular. 26

2.7. Sistema de autocalibración planteado por Yunde. 26

3.1. Visión Estereoscópica. La misma realidad vista desde cada ojo aporta perspectivas distintas. 29

3.2. Modelo de Lente Fina 30

3.3. Modelo *Pin Hole* 30

3.5. Ángulos *Pitch*, *Yaw* y *Roll* 31

3.4. Modelo de Cámara Estenopeica para un Sistema Estéreo . . . 32

3.6. Imágenes Estéreo. 33

3.7. Geometría Epipolar 34

3.8. Modelo de color RGB 35

3.9. Detección de Bordes 35

3.10. Imágenes Estéreo y sus Correspondientes Mapas de Disparidad. 38

3.11. Detalle de la Realización de un Mapa v-*Disparity* 39

3.12. v-Disparity en una Aplicación de Vehículos 40

3.13. Filosofía de Diseño de una CPU (izq) y una GPU (der) 42

3.14. Estructura de Trabajo de *CUDA* 43

3.15. Hilo, Bloque, *Grid* 45

3.16. Flujo de Información en una Aplicación *CUDA* 46

3.17. Arquitectura CUDA 47

3.18. Logo de las *OpenCV* 48

3.19. Ejemplo de ajuste datos, teniendo todos en consideración. (a) da un resultado correcto, (b) no ha ajustado bien la recta a los datos 51

3.20. Ejemplo de *inliers*-azul-, *outliers*-rojo- y recta ajustada. . . . 51

3.21. Ejemplo Básico del Algoritmo SURF, Aplicado Sobre una Zona Parcial de una Imagen, Aplicada una Rotación. 53

3.22. De izquierda a derecha: Derivadas Parciales, Discretizadas y Cuantificadas en Dirección y y xy, y Aproximaciones Obtenidas tras el uso de Filtros de Caja. 55

3.23. Geometría de Dirección de Ackermann 57

3.24. Desplazamientos de la Masa del Vehículo en Frenadas y Aceleraciones. 58

4.1. Detalle de la Realización del Mapa de Disparidad 60

4.2. Perfil de la Calzada 63

4.3. Extracción del Perfil de la Calzada a partir del v-*Disparity* . . 64

4.4. Diagrama de Flujo del Cálculo del Perfil de la Calzada 65

4.5. Interacción entre Procesos 69

4.6. Funcionamiento del Acceso a la Memoria Compartida 71

4.7. Desplazamientos y Rotaciones de un Vehículo Entre 2 *Frames*. 73

5.1. Diagrama de Flujo de la Odometría Visual Autocalibrada . . . 77

5.2. Parámetros Extrínsecos 79

5.3. Mapa Libre 85

6.1. Trazado usado para la Evaluación de Resultados. Localización 88

6.2. Parámetros Intrínsecos de la Cámara Sobre el Sistema A Bordo Del *IvvI* 2.0 90

6.3. Distintos Resultados de Iteraciones del Algoritmo de Partida. . 90

6.4. Resultados del Algoritmo de Partida. (a) Distintas Ejecuciones Sin Autocalibración. Comparativa. (b) Superposición sobre el Mapa de Satélite de un Resultado de Odometria Sin Autocalibración. 91

6.5. Secuencia de Calibración. 92

6.6. Distintos Resultados de Iteraciones Sin Calibración Alguna. . 92

6.7. Superposición sobre el Mapa de Satélite Sin Calibración Alguna 93

6.8. Distintos Resultados de Iteraciones Con *Yaw* Calibrado. . . . 93

6.9. Resultados del Algoritmo de Partida. (a) Resultados Superpuestos con Calibración del *Yaw*. (b)Superposición sobre el Mapa de Satélite Calibración del *Yaw*. 94

6.11. Distintos Resultados de Iteraciones Con Calibración del *Yaw*, *Pitch*, *Roll*. 95

6.10. Estado de la Secuencia de Calibración 95

6.12. Resultados del Algoritmo de Partida. (a) Resultados Superpuestos con Calibración del *Yaw*, *Pitch*, *Roll*, sin Comprobación de Puntos en Calzada. (b) Superposición sobre el Mapa de Satélite Con Calibración del *Yaw*, *Pitch*, *Roll*, sin Comprobación de Puntos en Calzada. 96

6.13. Estado de la Secuencia de Calibración 97

6.14. Distintos Resultados de Iteraciones Con Calibración del *Yaw*, *Pitch*, *Roll*, Con Comprobación de Puntos en Calzada. 98

6.15. Resultados del Algoritmo de Partida. (a) Resultados Superpuestos con Calibración del *Yaw*, *Pitch*, *Roll*, Con Comprobación de Puntos en Calzada. (b)Superposición sobre el Mapa de Satélite Con Calibración del *Yaw*, *Pitch*, *Roll*, Con Comprobación de Puntos en Calzada. 98

6.16. Valores de la Media Del Valor de *Pitch* en Cada *Frame*, 9 Iteraciones. 99

6.17. Valores de la Media Del Valor de *Pitch* en Cada *Frame*, 9 Iteraciones, Superpuesto con la Gráfica de Mayores Variaciones de las Iteraciones. 100

6.18. *Frames* con Mayores Valores de *Pitch* Calibrado. 101

6.19. Valores de la Media Del Valor de *Roll* en Cada *Frame*, 9 Iteraciones. 102

6.20. Valores de la Media Del Valor de *Roll* en Cada *Frame*, 9 Iteraciones, Superpuesto con la Gráfica de Mayores Variaciones de las Iteraciones. 102

6.21. *Frames* con Mayores Valores de *Roll* Calibrado. 103

7.1. Resultados Antes y Después de la Autocalibración 105

7.2. Comportamiento del *Pitch* en un Badén. 106

LISTA DE TABLAS

3.1. Comparativa CPU-GPU 41

6.1. Especificaciones Técnicas del Ordenador Utilizado en la Toma de Resultados 88

6.2. Especificaciones Técnicas de la GPU Empleada en la Toma de Resultados 89

6.3. Especificaciones Técnicas del Sistema Estéreo Usado Para la Captura de Datos 89

6.4. Error y Desviación Típica Del Proceso. 103

8.1. Estimación de Tiempo Empleado en el Proyecto 107

8.2. Estimación de Costes de Materiales del Proyecto 108

LISTA DE SIGLAS Y ACRÓNIMOS

simTD	*Sichere Intelligente Mobilität-Testfeld Deutschland*. 6
ABS	*Antilock Braking System*. 5
ACC	*Adaptive Cruise Control*. 5
ADAS	*Advanced Driver Assistances Systems*. 5
ALU	*Arithmetic Logic Unit*. 42
BSD	*Berkeley Software Distribution*. 48
CCD	*Charge-Coupled Device*. 13
CMOS	*Complementary Metal-Oxide-Semiconductor*. 13
CPU	*Central Processing Unit*. 10, 19, 22, 41–43, 46, 48, 65, 68, 112, 144
CSIC	*Centro Superior de Investigaciones Científicas*. 10
CUDA	*Compute Unified Device Architecture*. 18, 19, 41, 43–47, 88
DARPA	*Defense Advanced Research Projects Agency*. 8
DGT	*Dirección General de Tráfico*. 2
ESA	*European Space Agency*. 16

FLANN *Fast Library for Approximate Nearest Neighbors*. 56, 59, 66, 68

GPGPU *General-Purpose Computing on Graphics Processing Units*. 41

GPS *Global Positioning System*. 8, 10, 11, 15–17, 105

GPU *Graphics Processing Unit*. 41–43, 46, 59, 60, 68, 74, 112, 130

HAVEit *Highly Automated Vehicles for Intelligent Transport*. 5, 6

HAZCAM *Hazard Avoidance Cameras*. 22

IvvI *Intelligent Vehicle Based On Visual Information*. 10, 88

IV *Intelligent Vehicles*. 1, 4, 11

LSI *Laboratorio Sistemas Inteligentes*. 10, 88

MIL *Matrox Imaging Library*. 49, 89

NASA *National Aeronautics and Space Administration*. 22

OPENCV *Open Source Computer Vision Library*. 48, 49, 56, 66, 88, 108

RANSAC *Random Sample Consensus*. 18, 22, 50, 59, 64, 74, 75, 78, 80, 84, 91, 104

RGB *Red, Green, Blue*. 34

ROI *Region Of Interest*. 68

SICE *Sociedad Ibérica de Construcciones Eléctricas*. 10

SIFT *Scale-invariant feature transform*. 27, 53

SIMD *Single Instruction, Multiple Data*. 42

SSD *Sum of Squared Differences*. 36, 60

SURF *Speeded Up Robust Feature*. 53, 54, 56, 59, 66, 112

UAH	*Universidad Alcalá de Henares*. 10
UC3M	*Universidad Carlos III de Madrid*. 10, 88
UPM	*Universidad Politécnica de Madrid*. 10
URJC	*Universidad Rey Juan Carlos*. 10

1

INTRODUCCIÓN

PARA entender el texto que se expondrá a continuación, primero habrá el lector de entender el contexto actual de la automatización. La revolución industrial cambió el curso de la Historia. Durante el siglo XIX se inventaron nuevas formas de producir, máquinas que eran capaces de hacer el trabajo que en otra época hacían las personas, en busca de poder automatizar al máximo posible los procesos, ahorrando costes, tiempo y favoreciendo que Europa y América principalmente se convirtieran en el motor económico de la época. Ese fue el comienzo de una nueva forma de pensar, en la que se sueña con poder automatizar cada vez más máquinas, haciendo la vida más cómoda. No pasó mucho tiempo desde que se podían ver máquinas de vapor circulando por las vías férreas, o los escasos automóviles de principios del siglo XX, hasta que se empezó a plantear la idea de poder crear un artefacto que hiciera las labores de forma automática, lo que Karel Čapek bautizó con el nombre de "Robot". Ahí empezó a surgir la inquietud por llevarlo de lo que es una mera idea hacia algo tangible, físico. Desde entonces la rama de la robótica ha avanzado de forma irregular con el paso de los años, debido fundamentalmente a la sucesión de acontecimientos ocurridos durante el siglo XX, pero además porque se ha bifurcado en distintas ramas, alejándose de lo que inicialmente era un autómata, para centrarse en objetivos más específicos en campos particulares. Uno de ellos es el de los sistemas de ayuda para conducción basados en los IV (*Intelligent Vehicles*), que son los que centran este trabajo.

Hoy en día, parece imposible encontrar una similitud con los primeros automóviles que rodaron, aparte del fin con el que existen: transportar a las personas y mercancías entre dos enclaves geográficos diferentes. Los coches que se comercializan en la actualidad poseen variedad de características que no tenían los antiguos, y no solo en cuanto a la posibilidad de personalizar el vehículo que se quiere adquirir con propiedades como el número de plazas, el índice de confort de ellas, el color de la carrocería o similares; si no que han avanzado en la calidad de los sistemas de seguridad que ofrecen. Los *airbags*, cinturones de seguridad, las mejoras en las propiedades del chasis en caso de accidentes han avanzado en la idea de conseguir reducir la siniestralidad en transporte por carretera, una de las grandes lacras de la sociedad moderna. Tómense los datos que aporta la DGT (*Dirección General de Tráfico*) para mostrar el alcance de este problema [1]. Se parte del año 1980, año en el se elaborara el primer Plan Nacional de Seguridad Vial. A partir de esa fecha se produjo una disminución en el número de accidentes con víctimas hasta el año 1983 en que se quebró esa tendencia. En 1989 se registra el máximo histórico de accidentes de circulación en carretera. Más recientemente, durante el periodo 1990-1994 se observa un descenso firme y continuado hasta el año 1995 en que nuevamente se rompe esta tendencia descendente. Desde 1998 hasta 2002 las cifras de accidentes con víctimas presentaron ligerísimas variaciones que en ningún caso superaron el 2 %. En el año 2003 se produjo un incremento respecto del año anterior del 6 % en el número de accidentes, del 7 % en el número de víctimas y del 1 % en la cifra de muertos. En el año 2004 se inició una tendencia descendente en cuanto a las cifras de víctimas mortales. Esa tendencia se ha mantenido por quinto año consecutivo en 2008, registrándose un descenso del 19,95 %. En 2006 se implantó un nuevo sistema de recogida de información utilizando las posibilidades que ofrecen las nuevas tecnologías y que claramente ha incrementado el registro de los accidentes más leves, tal y como avala el hecho de que el número de accidentes con víctimas se incrementara ese año un 15,5 %, mientras que el número de accidentes mortales disminuyó un 7,5 %.

Para poder demostrar cómo en la última década han mejorado los resultados de la siniestralidad, se puede ver en las siguientes gráficas la variación entre el año 2000 (Fig.1.1) y el 2010 (Fig.1.2), una década en la que ha disminuido como ya se ha explicado anteriormente el número de víctimas de accidentes, y en la que la implantación de dispositivos de seguridad ha sido muy notable.

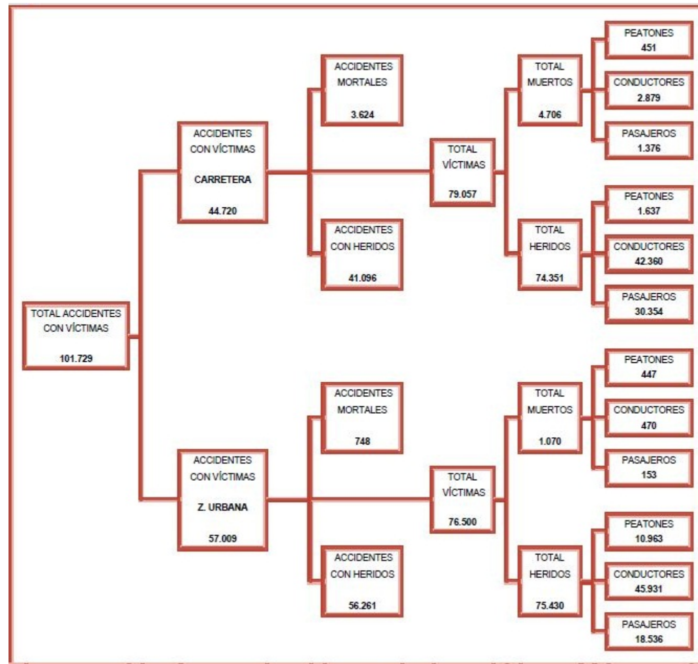


Figura 1.1: Datos Accidentes Año 2000

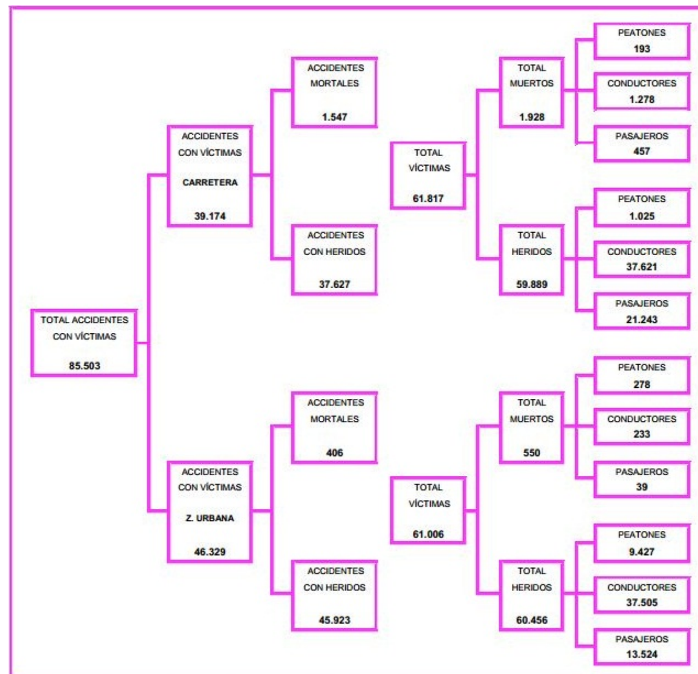


Figura 1.2: Datos Accidentes Año 2010

Una vez que se ha introducido el gran problema que los accidentes de tráfico suponen, y teniendo en cuenta que la enorme mayoría no son por causas mecánicas, sino por causas humanas, resulta necesaria la implantación de los ya mencionados IV.

1.1. El Vehículo Inteligente

Un IV es un vehículo que lleva incorporado sistemas de percepción del entorno, con los que es capaz de detectar obstáculos en la vía, de hacer un ajuste eficaz de las luces, de controlar el estado del vehículo con respecto a la vía... El propósito final es conseguir que estos automóviles sean capaces de desenvolverse naturalmente en una situación normal de tráfico, y basados en las órdenes que ellos mismos deciden mediante esa serie de sensores, cámaras y unidades de procesamiento que llevan incorporadas. Con ello los objetivos que se esperan conseguir son:

1. Aumentar la seguridad vial a través de una conducción más eficiente, gracias a la implantación de sistemas de conducción menos agresivos que los de un conductor humano, en el que una multitud de factores tales como la fatiga o su estado anímico contribuyen a una conducción peligrosa [2].

2. Producir un ahorro en los combustibles.

3. Conseguir que pueden ser capaces de adquirir el control del vehículo en caso de que el conductor haya tenido un percance que haya inhibido sus capacidades para la conducción segura (ataque de epilepsia, somnolencia, etc).

Toda la información captada del entorno, es procesada en tiempo real para poder dar respuesta y efectuar las acciones necesarias para que el vehículo pueda circular normalmente. Resulta muy importante que el tiempo de procesamiento no supere un tiempo crítico, pues en cualquier aplicación de tiempo real, las decisiones tienen caducidad: una decisión acertada, tomada en un tiempo mayor que el tiempo crítico de la aplicación no tiene validez alguna y, en algunos casos, puede resultar incluso contraproducente o dañina. Los vehículos deben analizar su entorno en todo momento, procesar la información

para asegurarse de qué órdenes dar a los distintos actuadores que llevan incorporados, y llevarlas a cabo antes de que sea demasiado tarde. Las unidades de procesamiento que se usan para estos efectos suelen ser sistemas empotrados, pues están diseñados para que sean capaces de utilizar su potencia de computación para una serie de aplicaciones muy particulares, al contrario de lo que pasa en un ordenador personal, en el que se espera que el sistema sea capaz de efectuar una variedad de aplicaciones.

Los vehículos inteligentes llevan una serie de tecnologías de ayuda a la conducción asociadas que se denominan ADAS (*Advanced Driver Assistances Systems*), como son el control de bloqueo de frenos (ABS (*Antilock Braking System*)), control de crucero adaptativo (ACC (*Adaptive Cruise Control*)), sistema de control de la somnolencia del conductor, sistema de aviso a Emergencias automático en caso de urgencia, sistemas de ayuda a la estabilidad, ayuda en conducción nocturna, asistencia para el cambio de carril... Los ADAS son los sistemas precursores de los que van embarcados en los vehículos inteligentes, y son los que están cada vez más implantados en las carreteras. Sin embargo los ADAS, hasta la fecha, solo tienen la posibilidad de ser usados en armonía con el conductor.

Un problema asociado a la implementación de estos sistemas es su alto coste, y más en un mercado, como es el automovilístico, en el que un aumento de costes puede ser un inconveniente difícil de solventar. Otro gran problema para que se puedan ver sobre las vías este tipo de coches es de índole legal, pues existen aún impedimentos para que un coche autónomo pueda circular por la carretera, por ejemplo en caso de siniestro, no se sabría con certeza si la culpa es del fabricante o del conductor [3].

Aún así, se están llevando a cabo estudios sobre estos vehículos tanto por marcas comerciales para proceder a incorporar esta tecnología en sus vehículos como por instituciones científicas, pues es de suponer que el coche del futuro ha de ser inteligente.

1.1.1. Proyectos Comerciales

Especial interés suscita el sistema lanzado en 2008 conocido como HAVEit (*Highly Automated Vehicles for Intelligent Transport*), de la marca *Volkswagen* [4], que incorpora un auto-piloto temporal que maneja funciones como control

Figura 1.3: Sistema *HAVEit*

de salida de carril, sistemas para asegurar la distancia de seguridad con el vehículo precedente o un ajuste de velocidad automático, con opción para el conductor de tener activos estos sistemas o poder prescindir de ellos. HAVEit hace uso de la información que le llega por medio de un radar, una cámara, un sensor láser y sensores de ultrasonidos. La idea es que se use el sistema HAVEit en situaciones de conducción monótonas, o en situaciones de tráfico denso, para asegurar la seguridad.

Por otro lado, *Ford*, entre otras marcas, está trabajando en tecnologías de comunicación coche a coche y coche a infraestructura dentro del proyecto europeo "Movilidad Inteligente y Segura-Campo de Pruebas Alemania", conocido como simTD (*Sichere Intelligente Mobilität-Testfeld Deutschland*) [5], que usa a Alemania (zona de Frankfurt a.M) como campo de pruebas para los próximos cuatro años, contados desde Agosto de 2012. La idea, en palabras del responsable de la marca, es poder probar en situaciones reales estos sistemas para poder incorporarlos en un futuro a la flota comercial de *Ford*. Las tecnologías probadas como parte del proyecto incluyen los siguientes sistemas de seguridad:

- *Luz de freno electrónica*. Su función es avisar al vehículo de detrás de una frenada de emergencia para evitar una colisión por alcance, incluso funcionará si el coche que circula por detrás no tiene en ese momento dentro de su campo de visión al vehículo precedente (por ejemplo en una curva cerrada).
- *Aviso de obstáculos*. Informará al propio vehículo y a todos los demás dentro del radio de acción de la presencia, posición y clase de obstáculo que se encuentra en la calzada.

- *Asistente de señales de tráfico.* El vehículo estará ofreciendo al conductor permanentemente información sobre la señalización con el fin de evitar posibles distracciones, además de permitir que se anticipe a lo que tenga unos kilómetros por delante. Puede ser capaz de avisar de carriles abiertos temporalmente, o cerrados, ya sea por atascos u obras en la carretera.

- *Gestión pública del tráfico.* Aportará un pronóstico exacto con amplia información sobre el tráfico, haciendo posible que no se pierda tiempo en atascos.

- *Acceso Internet Car-In.* Ford pone como ejemplo la posibilidad de reservar y pagar una plaza de aparcamiento mientras se está accediendo a la zona de aparcamiento.

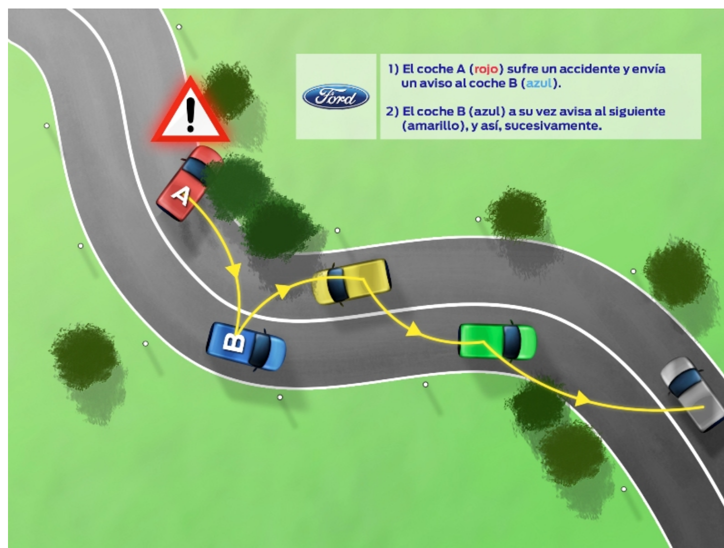


Figura 1.4: Tecnología *simTD*

Se han puesto a disposición del experimento 20 modelos *Ford S-MAX* y una flota de otros 120 vehículos que están especialmente pensados para este proyecto, y que recorrerán miles de kilómetros para recabar información.

1.1.2. Proyectos Científicos

En cuanto a proyectos llevados a cabo por instituciones educativas o científicas, se expondrán algunas de las más representativas tanto a nivel internacional como a nivel nacional.

Desde el DARPA (*Defense Advanced Research Projects Agency*) se organizaron en los años 2004, 2005 y 2007 el "DARPA *Grand Challenge*", carrera de vehículos autónomos que consiste en que las instituciones participantes deben presentar un vehículo capaz de llegar desde la salida hasta la meta sin intervención humana. Las ediciones de 2004 y 2005 tuvieron lugar por el desierto estadounidense, con resultado de que la primera edición ningún vehículo logró terminar (el vehículo que más avanzó lo hizo en 11.84 Km), y en la segunda edición, terminaron varios prototipos, alcanzando el triunfo la Universidad de Stanford (212 Km de trayecto). La edición del año 2007 cambió el desierto por un trazado urbano y se denominó "Urban Challenge", en el que los vehículos debieron coexistir con un entorno real. El próximo reto del DARPA le llevará entre Octubre de 2012 y Agosto de 2014 a alejarse de la investigación con vehículos inteligentes, para centrarse en buscar un humanoide capaz de sustituir al ser humano en situaciones de peligro (ambiente explosivo, etc).

La Universidad de Stanford lleva un proyecto denominado "Shelley" que dirige el *Stanford Dynamic Design Lab* en colaboración con el *Volkswagen Electronics Research Lab* [6]. Se trata de un *Audi TT* que ha alcanzado velocidades de cerca de 200 Km/h sin ayuda de un conductor. Está equipado con sistemas GPS (*Global Positioning System*) y sensores inerciales de bajo coste. Se trata de un prototipo de vehículo de carrera, y como futuras mejoras se espera poder extraer información de dos conductores de carreras profesionales, analizar su comportamiento del cerebro y sus constantes biométricas para mejorar el funcionamiento del vehículo, y que sea capaz de batir a un competidor humano.

Adicionalmente, cabe destacar los esfuerzos de Google con el proyecto Google Car [7], que lleva el responsable del departamento de inteligencia artificial de la universidad de Stanford, y máximo responsable de la victoria en el "DARPA *Grand Challenge*".

Han equipado a 9 vehículos (siete *Toyota Prius*, un *Lexus RX450h* y un *Audi TT*), y han sido capaces de recorrer cerca de 500.000 km de forma

autónoma con ellos. a bordo de los vehículos se encuentra un humano que debe hacerse cargo del coche en caso de que exista peligro, y hasta la fecha, con los automóviles circulando en condiciones normales de velocidad (poseen una base de datos con los datos de velocidad actualizada de cada carretera), solo ha sido necesaria la intervención en dos ocasiones. Para poder detectar el entorno, hacen uso de un lídar de largo alcance que va situado en el techo de los vehículos, videocámaras que detectan desde la perspectiva del interior del habitáculo las marcas viales u obstáculos, radares situados en la parte delantera del coche, y un estimador de posición sobre las ruedas (Fig.1.5).

Figura 1.5: Vehículo Inteligente Google

Los resultados que obtiene el vehículo de Google mediante el conjunto de sensores que lleva incorporado es el que se muestra en la Fig.1.6:

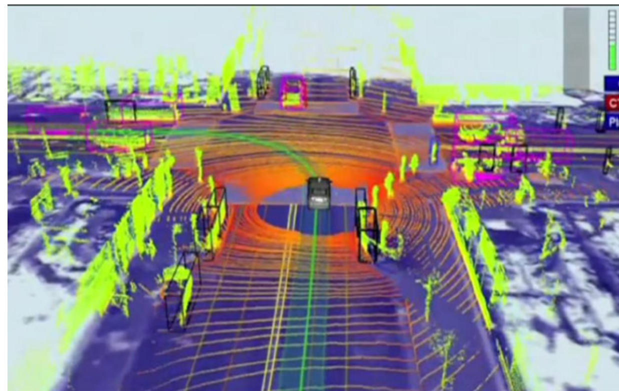


Figura 1.6: Percepción del Entorno de un Google Car

Los vehículos de Google son legales en el estado americano de Nevada desde Marzo de 2012, convirtiendo en un vehículo de este proyecto de investigación en el primer vehículo autónomo registrado en los Estados Unidos.

De gran interés pueden resultar los vehículos experimentales que diversas instituciones españolas están implementando:

- La UPM (*Universidad Politécnica de Madrid*) junto con el CSIC (*Centro Superior de Investigaciones Científicas*) poseen el programa *AUTOPÍA* [8], que funciona mediante dos vehículos uno de los cuales (guiado de forma manual) sirve de guía al otro, que se conduce de forma autónoma. El grupo de investigación ha logrado que su vehículo *Platero* recorriera cerca de 100 km por carreteras convencionales sin necesidad de un conductor humano [9].

- El proyecto *GUIADE*, en el que participa el CSIC ,UAH (*Universidad Alcalá de Henares*), URJC (*Universidad Rey Juan Carlos*) y Albentia, SICE (*Sociedad Ibérica de Construcciones Eléctricas*), tiene como fin la automatización, el posicionamiento y guiado de vehículos de transporte público, con el objetivo de optimizar su eficiencia energética y medioambiental, seguridad y calidad. Para ello, la flota de vehículos del programa disponen de sistemas de percepción multimodal tanto del entorno como del propio vehículo. Mediante la obtención de datos procedentes del *BusCAN* es posible acceder al estado del vehículo, aportando información acerca de éste. Estos datos al unirlos a los obtenidos mediante visión, los cuales aportan información acerca de los vehículos de alrededor, permiten tener información tanto del propio vehículo, como de su alrededor. Además al emplear un GPS permite geo-posicionar y sincronizar en todo momento estas variables [10].

- En la UC3M (*Universidad Carlos III de Madrid*) la investigación se centra en el vehículo IVVI (*Intelligent Vehicle Based On Visual Information*) (Fig.1.7), que es el vehículo en el que se ha probado el código presentado en este proyecto. Se trata del prototipo que el LSI (*Laboratorio Sistemas Inteligentes*) (Departamento de Sistemas y Automática UC3M) tiene bajo estudio. Se trata de un utilitario de la marca *Nissan*, modelo *Note* equipado a tal efecto de cuatro módulos de CPU (*Central Processing Unit*) que procesan la información que reciben gracias a seis cámaras que controlan todo lo que pase tanto dentro como fuera del vehículo:
  - Un par de cámaras estéreo situadas en la luna delantera, que será la que se use en este texto, y dedicada a la detección de obstáculos y de determinación de los límites de la vía.
  - Una cámara a color que sea la que detecte la señalización vertical de la vía.

- Una cámara de infrarrojo lejano para captar obstáculos en caso de que la visibilidad sea reducida.
- Una cámara que está localizada hacia el conductor para evaluar el estado físico y de atención del mismo, que muestra la información obtenida en una pantalla a tal efecto colocada en el salpicadero del coche.

Además, el coche lleva incorporado un GPS para la localización del mismo.

Figura 1.7: Vehículo Inteligente *IvvI*

1.2. La Percepción del Entorno

Uno de los tipos de sensores más utilizados para la percepción del entorno de un IV son las cámaras, que pueden ser monoculares, omnidireccionales, o un par estéreo. Este método de interacción con el exterior del sistema resulta muy interesante porque son dispositivos no invasivos, al contrario de lo que sucede en el caso de los sensores lídar (cálculo de la distancia mediante el retraso entre emitir y recibir un halo de luz) [11], radar (cálculo de la distancia mediante el retraso entre emitir y recibir una señal de radiofrecuencia) [12] o láser, (ver Fig.1.8) que interfieren con su luz o radiofrecuencia en el entorno. Dentro de los sistemas de percepción que se usan actualmente, la extracción de características mediante una cámara es uno de los campos de investigación más interesantes dado que permite poder trabajar con información muy útil de índole visual y de las distancias que hay entre algún objeto y el dispositivo.

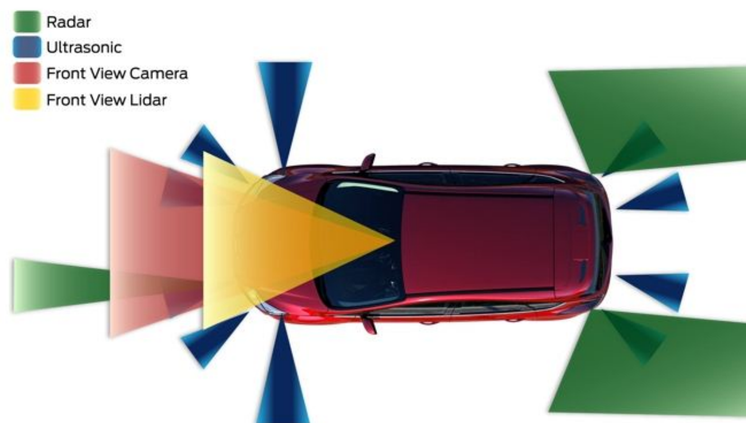


Figura 1.8: Sistemas de Percepción de un Vehículo Inteligente

Las cámaras, por el contrario, dependen fuertemente del entorno, ya que una baja intensidad lumínica puede acarrear que se procesen datos incorrectos, y por tanto se tomen decisiones equivocadas. Las cámaras actuales son capaces de ajustar su ganancia y tiempo de exposición a las condiciones ambientales. Con ello se evita que las imágenes se obtengan sobre expuestas (exceso de luz) o sub expuestas (déficit de luz), y que la ganancia sea la correcta (existe la posibilidad de que se esté amplificando el ruido). Otro de los puntos en contra del uso de las cámaras en un estado de ausencia de luz (por ejemplo, conducción nocturna), una cámara de espectro visible no podría apenas aportar información útil, lo cual hace necesario un cambio de sensor.

La tecnología de cámaras que existe actualmente permite que se puedan tomar numerosas imágenes en poco tiempo, lo cual hace que la información con la que se trabaje no se vea estropeada por el movimiento que tienen los objetos en el mundo. Esto es muy importante sobre todo cuando se presupone que la información más interesante es la de objetos en movimiento.

Otra ventaja de las cámaras es que son sistemas de bajo coste, lo cual en cualquier aplicación siempre es un gran punto a favor, y que no requieren un alto gasto energético.

1.2.1. *Cámaras Monoculares*

Las cámaras monoculares disponen de un sensor CCD (*Charge-Coupled Device*) o de un sensor CMOS (*Complementary Metal-Oxide-Semiconductor*), lo que hace que su implantación sea más barata. Además, no siempre es necesario calibrar la cámara, como se verá que sí lo es cuando se posee un par estéreo. De todas formas, esta es una ventaja relativa, pues es altamente recomendable realizar un proceso de calibración, para asegurar que la toma de datos sea correcta [13].

El principal problema de las cámaras monoculares es que la información recogida es menor que en cualquiera de las opciones que se expondrán a continuación.

1.2.2. *Cámaras Omnidireccionales*

Las cámaras omnidireccionales son cámaras con un campo de visión de 360°. Son usadas cuando se quiere obtener un amplio rango de visión. La ventaja que presentan estas cámaras es que el movimiento en cualquier dirección produce un buen flujo óptico, pero a cambio, al presentar ese gran ángulo de visión, la deformación que experimenta la imagen tomada es muy grande [14].

1.2.3. *Sistemas Estéreo*

En esta configuración se intenta imitar el comportamiento de los sistemas de visión de la mayoría de los animales y de los humanos. Se compone de dos cámaras idénticas separadas una distancia fija, y que están sincronizadas para que tomen a la vez una imagen. Se obtienen dos imágenes por cada disparo que se produzca, en los cuales habrá que aplicarles determinados procesamientos para extraer información de ellas. El primer inconveniente de este sistema es el de que las cámaras deben estar perfectamente calibradas para poder asegurar la correcta rectificación de las imágenes [15].

Las imágenes en las que se basa todo el trabajo posterior son tomadas por un sistema de cámaras estéreo, porque se busca obtener información

Figura 1.9: Cámara *Bumblebee2*

tridimensional del entorno, y, pese al alto coste computacional que requiere, es más robusto. La cámara usada es la que muestra en la Fig.1.9, y es la cámara *Bumblebee2* del fabricante de cámaras *Point Grey* [16].

1.2.4. *Autocalibración de la Cámara*

Las cámaras tienen lo que se denominan parámetros intrínsecos, que son aquellos factores que dependen íntimamente del equipo de cámaras que se use (por ejemplo, distancia focal, concepto que se estudiará en 3.1.1), y que deben obtenerse una única vez mediante un proceso de calibración, y los parámetros extrínsecos que son los que vienen determinados por la posición de la cámara, lo que se denomina en este campo *pose*, y que viene dado por los parámetros conocidos como *altura* h, *pitch* θ y *roll* ρ, que son un sistema de referencia respecto de un sistema de coordenadas global determinado de las medidas tomadas gracias a ese par de cámaras estéreo en particular [17].

La autocalibración de una par de cámaras estéreo hace referencia a la determinación en tiempo real y automática de los parámetros extrínsecos de dicho par estéreo. Dicho de otra forma, hace alusión al proceso de analizar los parámetros a la vez que el sensor está efectuando las operaciones para las que es requerido. En un proceso de autocalibración, no es necesaria un patrón de calibración previo de los parámetros (no importa el estado inicial de la cámara), lo que lleva implícito una reducción de tiempo y costes [18].

Que los resultados que se obtienen de este paso sean correctos es de vital importancia, pues con las cámaras haciendo mal la relación entre lo que se conocerá como *mundo*, y que es el entorno real en el que se encuentra la cámara, y la propia cámara, los datos arrojados por el sistema serán erróneos también. Por ende, es importante no subestimar este paso.

Además, la autocalibración para la introducción de sistemas estéreo en entornos automovilísticos es de gran interés, pues con el paso del tiempo de operación, debido a las vibraciones que el propio vehículo genera, a movimientos de la cámara involuntarios, o incluso por variaciones de temperatura, la cámara sufre desviaciones con respecto a la calibración original, y se incurre en lo mencionado en el párrafo inmediatamente anterior: los resultados obtenidos no arrojan información útil, y todo el proceso en el que ha estado basado ha sido en vano.

1.3. Localización del Vehículo

Los métodos más usados para poder situar el vehículo dentro de un entorno son el sistema GPS y la odometría visual.

1.3.1. Sistemas De Posicionamiento Global

Conociendo la naturaleza y las expectativas de los vehículos inteligentes, se puede entender por qué, pese a que el GPS, el sistema europeo *Galileo* o el sistema ruso *Glonass* se van a introducir en el presente texto como una excelente solución para aplicaciones en las que una leve pérdida de señal no tiene consecuencias relevantes, no se pueden considerar sistemas fiables para aplicaciones más críticas.

Desde los últimos años, los sistemas de posicionamiento global están fuertemente implantados en nuestra sociedad. Se pueden utilizar para ayudar al conductor a realizar una ruta en coche, para practicar deporte, para mejorar la calidad de vida a personas con minusvalías tales como la ceguera, etc. El GPS es el sistema de localización más comúnmente conocido pues su fiabilidad se adecua a dichas aplicaciones.

Mediante una red propia compuesta actualmente de 24 satélites en órbita alrededor de la tierra, el receptor GPS usa el método de triangulación para determinar la posición en la que se encuentra el dispositivo, necesitando un mínimo de 4 satélites para ello. Por supuesto, a medida que el sistema es capaz de captar más satélites, más precisa será la información obtenida. Sincronizando los tiempos locales de los satélites (reloj atómico) y el tiempo

local (reloj interno) del dispositivo, el receptor GPS es capaz de saber la distancia al satélite emisor. Con esa información, mediante lo que se conoce como *efemérides* (toda la información relevante del satélite), es posible indicar la posición sobre la esfera terrestre del dispositivo [19].

El sistema europeo llevado por la ESA (*European Space Agency*) se conoce como *Galileo*. Está compuesto por 30 satélites bajo control civil, que mediante protocolos de frecuencias duales es capaz de aportar datos en tiempo real de posicionamiento hasta un nivel de detalle de errores de menos de un metro, además está diseñado para que pueda funcionar bajo todo tipo de circunstancias y que sea robusto ante aplicaciones críticas, tales como la de guiar vehículos. Esto es posible gracias a que la posición alrededor de la tierra va a ser distinta a la del sistema GPS. Se espera que hasta finales de 2014 no empiecen las pruebas con este sistema [20]. *Galileo* será perfectamente compatible con la tecnología GPS.

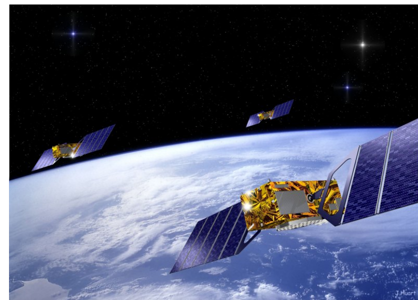


Figura 1.10: Sistema *Galileo*

Las fuentes de error más comunes para que el GPS no pueda situarse correctamente son los siguientes [19]:

- *Disponibilidad selectiva*:el sistema de satélites es controlado por el Departamento de Defensa de los Estados Unidos, y por ello, la precisión que tiene el usuario no siempre es la óptima que se podría obtener, por motivos de seguridad.
- Errores de *efemérides* o *de reloj*:se obtendrán medidas erróneas.
- *Retrasos debidos a la ionosfera* (la capa atmosférica cargada de iones y electrones libres) *y troposfera* (capa de la atmósfera entre 8 y 13 km por encima de la capa terrestre).
- *Interferencias*: entre distintos emisores que entremezclan su señal con la del sensor, provocando retrasos de varios nanosegundos.
- *Ruido de propagación y recepción*: la propagación de la señal de GPS desde el receptor al emisor sufre distorsiones debido a centelleos galácticos

que generan en torno a los 3 nanosegundos de retraso, y el error/retraso del propio receptor puede incurrir en otros 2 o 3 nanosegundos de retraso.

Estos retrasos en los tiempos de sincronización son las que hacen que los sistemas GPS, funcionando correctamente (en el próximo párrafo se presentarán otros errores ajenos a la tecnología), tengan un error de 2.5 metros con un 95 % de fiabilidad.

Sin embargo, si el dispositivo no detecta ningún satélite (o no un mínimo de cuatro), la posición del dispositivo es imposible de determinar. Esto pasa cuando hay problemas para la obtención de la señal de satélite: Un túnel, una zona urbana con edificios altos, una zona con alta posibilidad de interferencias de señales o un punto negro de cobertura GPS hacen imposible la obtención de la posición, y cuando esta información es vital para poder hacer funcionar correctamente otros sistemas, como es el propósito de los *vehículos inteligentes* en estudio, una pérdida de señal es problemática.

La idea que asegura la mayor estabilidad del sistema se basa en un método de apoyo basado en algo "propio", es decir, en un sistema que lleve el vehículo incorporado y que no dependa del entorno. Esta idea es la base para empezar a trabajar en lo que se conoce como la *Odometría Visual* [21], y cuyo fin último es generar un sistema de localización robusto.

1.3.2. La 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. El ser designado con ese nombre se debe a una reminiscencia de los antiguos sensores que iban adjuntos a los encoders de las ruedas de los robots y que se conocían como sensores de odometría. Como el movimiento del sistema de cámaras es solidario con el movimiento del vehículo, a esta técnica también se le conoce como *ego-motion* (movimiento propio).

Para poder realizar una correcta estimación del movimiento mediante esta técnica, es fundamental seguir una serie de pautas que evitarán obtener datos erróneos. Éstas son:

1. La detección de puntos característicos que sean estacionarios, pues si se toman como base puntos en movimiento, los datos salientes serían completamente erróneos, y no se podría hacer uso de ellos.

2. Detectar qué puntos son relevantes, y hacer un cribado de datos no correctos (*outliers*), mediante el algoritmo RANSAC (*Random Sample Consensus*), que será estudiado en 3.4.1.

3. Los circuitos urbanos contienen información desigual: las calzadas se caracterizan por pocos puntos de interés -marcas viales-, o incluso la ausencia de ninguna marca característica. Sin embargo, el resto del entorno se caracteriza por una gran concentración de puntos de interés: árboles, vehículos estacionados... El algoritmo implementado debe ser capaz de extraer los puntos fundamentales.

El mayor problema para la implantación de un sistema de odometría visual es el alto coste computacional que requiere para su correcto uso. Es por ello que en el siguiente epígrafe se presentará una solución para poder reducir ese tiempo lo máximo posible con la tecnología existente actualmente.

1.4. Filosofías de Programación

Para solventar el inconveniente que acaba de ser mencionado, se quiere introducir la arquitectura CUDA (*Compute Unified Device Architecture*) como una solución para reducir el tiempo de cómputo al máximo posible, de tal manera que sea factible implementar una odometría visual que cumpla con los requisitos que se esperan.

Tradicionalmente se han hecho programas en lenguaje C o C++, pero ahora se está introduciendo también la arquitectura CUDA, extensión del lenguaje C++ desarrollada por el fabricante de tarjetas gráficas *nVidia* y que permite la programación multihilo. Dicho en otras palabras: la forma tradicional de programar ha sido de una forma estructurada y en serie. La

tecnología CUDA permite la programación en paralelo: se define la figura del hilo como una línea de programación independiente capaz de acceder a la memoria del programa, y trabajar en paralelo con otros hilos iguales que se encargan cada uno de una región de una imagen, por ejemplo. Para ello es indispensable contar con una tarjeta gráfica de la marca *nVidia*.

Al ser capaz de poder dividir la imagen en bloques, cada uno manejado por un hilo, es posible acelerar el proceso de trabajo sobre la imagen, pues cada hilo escribe los resultados obtenidos en la zona común de memoria de todos los hilos que están implicados en el proceso.

Como ya se indicó anteriormente, el problema de los sistemas de percepción mediante sistemas de visión al nivel al que están hoy en día, es la velocidad de procesamiento que se necesita para obtener resultados correctos. Si se consigue implantar el paralelismo como forma de trabajo, parece razonable creer que los tiempos se reducirán enormemente y que se podrá tender a minimizar esa gran desventaja.

Se verá a lo largo de la sección de resultados obtenidos que al poder usar toda la capacidad de la CPU usada, se ha conseguido reducir los tiempos de ejecución con respecto a la versión anterior de la solución implementada.

2

ESTADO DEL ARTE

Los sistemas de percepción basados en cámaras son una realidad desde hace más de una década, y el fin de este capítulo es poder mostrar algunos de los que más interés han suscitado acerca de los temas tratados en el capítulo anterior.

2.1. Trabajos Existentes Basados en Cámaras Monoculares

En el estudio llevado a cabo por Stein en [13] puede apreciarse un sistema de odometría visual basado en la captación de imágenes mediante una cámara monocular. La cámara se encuentra situada cerca del espejo retrovisor, con el que se puede obtener un alto ángulo de visión. Hace uso de funciones probabilísticas para poder estimar el movimiento del vehículo que ha tenido lugar entre un instante t y el inmediatamente posterior $t + 1$, de tal manera que los resultados expuestos aseguran ser invariantes a destellos luminosos, lluvia y objetos móviles que interfieren en el resultado de la odometría.

En el caso del proyecto de investigación presentado en [22] se utiliza una cámara omnidireccional situada en el techo del vehículo para captar las imágenes, y con dos rastreadores de posición, uno para estimar la desviación

del ángulo del vehículo, y otro para estimar el avance. Los resultados expuestos indican que en un trayecto de 400m son los que pueden apreciarse en la Fig.2.1, dependiendo del ángulo de punto de vista de la cámara omnidireccional.

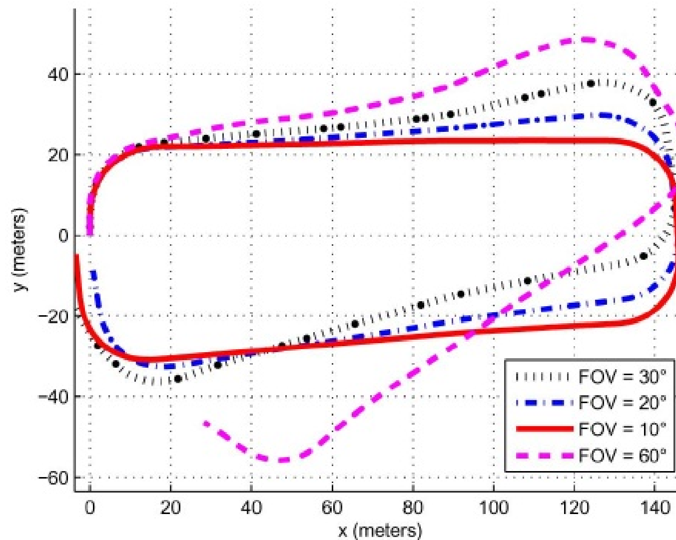


Figura 2.1: Resultados Arrojados Por El Sistema de Odometría Visual Propuesto por Scaramuzza et al. FOV=Puntos de Vista

La tendencia, sin embargo ha sido la de avanzar en el ámbito de la visión estereoscópica, debido a la posibilidad de poder extraer información tridimensional.

2.2. Trabajos Existentes Basados en Par Estéreo

Hasta ahora, gran cantidad de los estudios que se han llevado a cabo se han realizado en entornos *off-road*, pues presentan menos puntos potenciales de ser captados que un entorno urbano (objetos en movimiento, exceso de información, objetos interfiriendo entre sí, etc), y con ello se puede conseguir probar mejor los programas, además de identificar qué información es relevante y cuál no, pues puede darse el caso de que las características que encuentre sean poco relevantes.

En las naves de exploración gemelas *Opportunity* y *Spirit* que la NASA (*National Aeronautics and Space Administration*) mandó a Marte entre los años 2004 y 2005 ya iba acoplado un sistema de visión basado en odometría visual sobre imágenes captadas por un par de cámaras estéreo (sensores HAZCAM (*Hazard Avoidance Cameras*)). Mediante detectores de esquinas de Harris o Forstner se procedía a computar la localización de los puntos característicos entre ambas imágenes. Para valorar el movimiento de la rover se procedió primero a hacer los mínimos cuadrados de la diferencia entre cada punto en la imagen actual y la anterior, y posteriormente se utilizó también en este caso el algoritmo de RANSAC. El hecho de hacer estas dos operaciones es que con los mínimos cuadrados se conseguía la expresión del coste de forma rápida, simple y robusta, y con el RANSAC se aislaban los puntos que no eran de interés, como se hace uso en este proyecto. Dado que los resultados obtenidos fueron razonablemente satisfactorios (con avances de 75 cm en línea recta o en arcos con giros menores de 18° en cada uno de estos avances, se tardaban entre 2-3 minutos de computación sobre la CPU integrada de 20MHz) (Fig.2.2) , se utilizó la odometría visual para determinar la posición diaria de las rover y mejorar sobremanera la efectividad de la misión, al poder dirigir al robot de una forma más segura, esquivando los cráteres. También hay que decir que hubo una serie de fallos en esa determinación de la posición, pero eran debidas a un avance demasiado largo de la nave (el solapamiento de imágenes requerido para que el sistema fuera capaz de identificar los puntos característicos entre una imagen y la anterior hacía imposible la correspondencia entre puntos; por ejemplo, en giros bruscos, de más de 40° sobre el propio eje de la nave se daba este problema), y también hubo falsos positivos (hubo algunos fallos iniciales debido a una serie de parámetros y umbrales que se ajustaban mal al terreno de Marte y que tras ser reprogramados, se solucionaron sin mayor dilación). Spirit terminó la misión con una efectividad del uso de la odometría visual del 97 % (590/609 casos satisfactoriamente procesados), mientras que Opportunity llegó al 95 % (828/875) [23].

El 26 de Noviembre de 2011 fue lanzada la nave *Curiosity*, que llegó a la superficie marciana el 8 de Agosto del siguiente año. Este rover también posee un sistema de odometría visual a bordo, y es una de las claves para que el vehículo pueda moverse como se espera de él por el planeta rojo. El hecho de que la NASA apueste de nuevo por esta solución en un proyecto de tan ambiciosa naturaleza, da una idea de que la odometría visual no es una utopía [24].

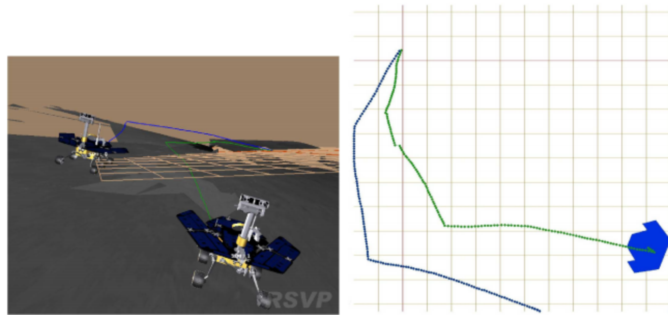


Figura 2.2: Trazado Por Odometría Visual (Verde) Frente A Encoders (Azul)

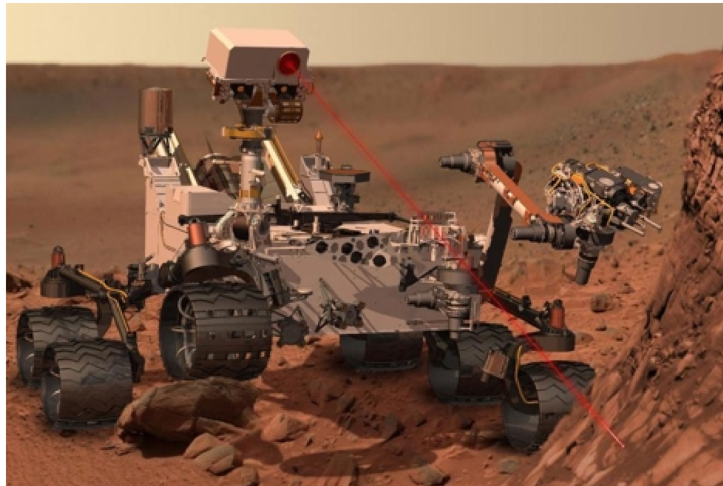


Figura 2.3: Curiosity

Los sistemas de odometría visual que usan las rover se basa en captar puntos característicos entre un instante y el instante anterior, en su entorno y analizar cómo ha variado la posición de dichos puntos (Fig.2.4).

Otros sistemas de determinación de la posición basados en par estéreo interesantes es la de la *pose* de un satélite durante operaciones de aproximación a otros ([25]). Se tratan de operaciones en las que dos satélites están co-existiendo en poco espacio.

Los costes asociados de enviar a un humano al espacio cada vez que ocurre algún percance en un satélite comienza a ser demasiado elevado para poder asumirlos, y en muchas ocasiones solo renta para operaciones en las que no existe otra opción. Es por ello que la visión estéreo se ha convertido en un gran aliado para estos menesteres, siendo posible llevar a cabo una

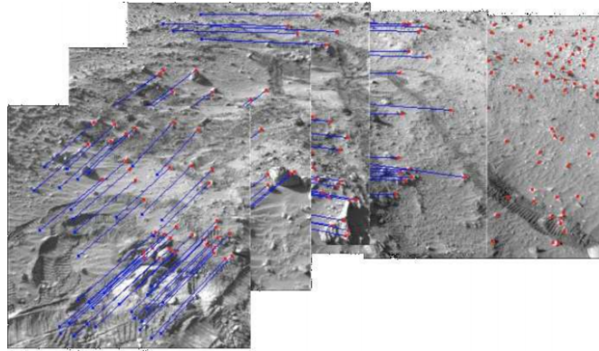


Figura 2.4: Captación de Puntos Característicos en la Superficie de Marte

reorientación de la *pose* del sistema (Fig.2.5) mediante el procesamiento de información en 3 dimensiones.

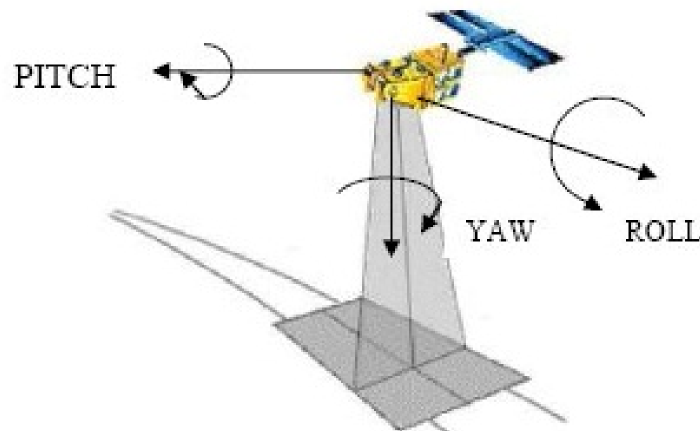


Figura 2.5: *Pose* de un Satélite (Envisat en la Imagen)

Uno de los problemas de la operación de satélites es la necesidad de robustez en sus sistemas, dado que una situación común es que no pueda existir comunicación con centro de control terrestre (interferencias, retrasos en la señal...), y gracias a los avances en la visión estéreo y sus resultados, se han podido empezar a implementar estos sistemas.

La calibración de la posición del satélite se basa en visión estéreo y también monocular, y captan la trayectoria y la velocidad del satélite, y capta su entorno (tiene un alcance de aproximadamente 100m), buscando un satélite

a su alrededor. Mediante una estimación tridimensional del satélite, hace una primera aproximación del la *pose* del satélite. A partir de esta primera aproximación, hace un seguimiento del movimiento del satélite hasta obtener los resultados buscados.

La ventaja que presenta este peculiar entorno es que por no estar en un ambiente con atmósfera y entornos ricos en características hace que procesar las imágenes sea un proceso poco costoso. Sin embargo, los cambios de iluminación si que son de consideración, y pueden llevar a pérdidas temporales de efectividad.

La visión en el ámbito espacial se resume en:

- Detección de puntos característicos generales en el satélite.
- Detección de puntos característicos claramente identificables y modelización 3D.
- Encontrar las características 3D y las correspondencias entre el mundo y la cámara.

Por lo tanto, cuando los dos satélites están en proximidad, es posible tener la información de velocidad y trayectoria de uno frente al otro, además de la información técnica en si misma, mediante un sistema barato, no invasivo y de poca exigencia energética.

2.3. Trabajos Existentes Acerca de Calibración

Especial interés suscita el trabajo presentado en [26], el cual ha sido el punto de partida para el estudio del algoritmo presentado, pero otros sistemas de autocalibración, como la mencionada por [18] es digna de mención.

En la solución planteada por [18], el sistema del que se parte consta de tres cámaras: un par estéreo y una cámara monocular, y situadas de tal forma que las tres cámaras pueden moverse de forma independiente entre

ellas (las cámaras estéreo pueden rotar libremente por su eje *yaw* y la cámara monocular puede rotar horizontal y verticalmente (Fig.2.6). Las bases para la calibración se basa en minimizar el error de ajuste entre las coordenadas del mundo y las de la cámara en todos los *frames* de la secuencia, y en minimizar la restricción epipolar (se estudiará en la sección 3.1.1). Posteriormente, se aplica un filtro de Kalman.

Figura 2.6: Sistema de autocalibración planteado por Dang et al. C1, C2: Cámaras Estéreo, M: Espejo, C3: Cámara Monocular.

Existen otras soluciones, como la realizada por [27], y cuyo esquema puede apreciarse en 2.7, cuyo funcionamiento para conseguir la calibración de un sistema de cámaras estéreo mediante el uso de espejos.

Figura 2.7: Sistema de autocalibración planteado por Yunde.

Se trata de un sistema embebido de calibración que es capaz de conseguir por sí mismo la calibración del sistema, sin necesidad de ningún elemento externo (ni patrón ni semejantes). El sistema de cámaras se introduce dentro de un conjunto de espejos reflectores, y en frente de un espejo que permite

solamente que el 50 % de luz sea reflejada, mientras el otro 50 % pasa a través. Por lo tanto, el sistema capta a la vez lo que pasa por el espejo y lo que se refleja a lo largo de los demás del sistema (para la mejor estimación de la profundidad) y de un espejo poliédrico, cuyo fin es generar múltiples imágenes virtuales de la unidad de calibración para aumentar la precisión con ello. Mediante este método se es capaz de calibrar la cámara para que funcione con el entorno.

Otro estudio interesante es el que muestra [28], en el que la calibración de una única cámara se lleva a cabo gracias a la posibilidad de que la misma tenga un grado de libertad para trasladarse, y gracias a la captura de tres imágenes consecutivas captadas en distintas posiciones de la cámara. Basándose en que las horóptera[1] de tres imágenes distintas tomadas bajo una traslación intersectan en un punto real y dos complejos conjugados, es capaz de calibrar la cámara.

Otros métodos, como el presentado por [29], se basa en el detector SIFT (*Scale-invariant feature transform*), que capta diferencias entre imágenes que pueden estar rotadas o trasladadas entre sí. Tras la calibración previa de dos de los parámetros intrínsecos, se hace uso de dicho detector para estimar la matriz esencial de la cámara. Aplicando descomposición en valores singulares se pueden obtener a partir de la matriz esencial las matrices de traslación y rotación. Con ello son capaces de estimar la autocalibración de la cámara.

[1] Una horóptera es una curva espacial dada por el conjunto de puntos que proyectan en puntos con idénticas coordenadas en dos cámaras con iguales parámetros intrínsecos.

3

FUNDAMENTOS TEÓRICOS

En este capítulo se procederá a introducir los fundamentos teóricos necesarios en el desarrollo del algoritmo implementado, el cual será descrito más profundamente en el capítulo siguiente.

Los temas tratados son los que sientan la base para poder entender en todas sus dimensiones el tema tratado, por lo que el capítulo puede entenderse como estructurado, siendo la primera parte la que trata acerca de los conceptos físicos que permiten la realización del estudio, para luego adentrarse en los procesos que se aplican a las imágenes captadas por esos procesos físicos, otra sección en la que introducir los entornos de programación, y finalmente, un epígrafe dedicado a los algoritmos que resultan necesarios para los fines buscados. Se adjunta además una breve sección con conceptos de los vehículos a motor que se recomienda poseer para el correcto entendimiento del proceso.

3.1. Conceptos Físicos

3.1.1. La Visión Estéreo

Después de que en el capítulo anterior se explicaran las ventajas de trabajar con un par estéreo, se pretende en este epígrafe fundamentar cómo funciona esta topología.

La visión binocular, como punto de partida para la visión estereoscópica, genera una ilusión de profundidad, como puede apreciarse en la Fig.3.1, pues al poseer dos imágenes (una captada por cada ojo) del mismo objeto desde dos perspectivas diferentes, el cerebro es capaz de calcular la diferencia entre ambas imágenes e inferir la distancia al objeto, darle sensación de volumen. Si el objeto está lejos del individuo, las imágenes se diferenciarán menos entre sí (en cuanto a la perspectiva) que si el objeto está cercano.

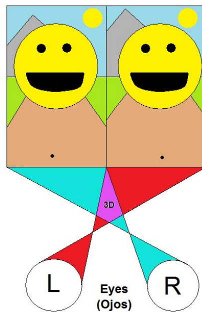


Figura 3.1: Visión Estereoscópica. La misma realidad vista desde cada ojo aporta perspectivas distintas.

La visión por computador basada en visión estéreo pretende imitar este comportamiento humano, y hacer uso de él en aplicaciones industriales, o aplicaciones de robótica, por ejemplo.

Un par de cámaras estéreo ha de estar constituido por dos cámaras idénticas (igual *distancia focal*), y que tengan sus *ejes ópticos* paralelos.

Siguiendo el modelo de lente fina (se designa así a los sistemas en los que se considera despreciable el tamaño de la lente), los rayos inciden de manera perpendicular sobre la lente del sistema (ver Fig.3.2), y al pasar por la lente, éstos se concentran en un único punto, que está situado a la *distancia focal* de

la lente, y que se simboliza comúnmente con la letra f minúscula. Los rayos que atraviesan la lente de forma perpendicular a la misma y por su punto medio son los únicos que no sufren distorsión alguna al pasar por la lente, y se les define con el nombre de *eje óptico*.

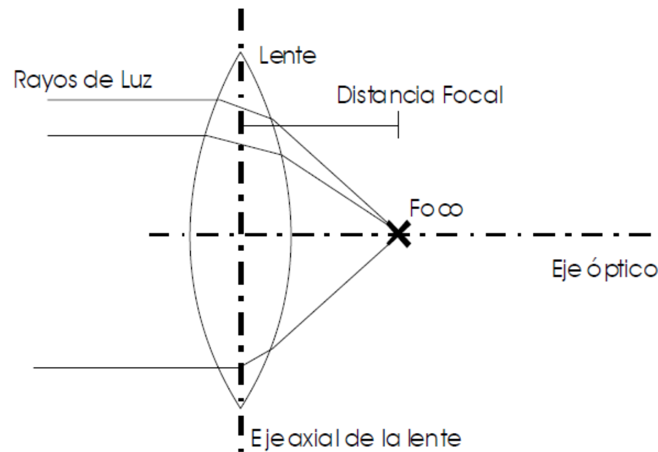


Figura 3.2: Modelo de Lente Fina

El modelo de cámara estenopeica (también conocida como modelo *pin-hole*) es de gran interés para poder entender cómo se podrá estimar la posición de un punto en el mundo $P(X, Y, Z)$ sobre el sistema de la cámara (u, v).

El modelo *pin-hole* es una cámara que no posee lente, de tal manera que la cámara se reduce a una caja a prueba de luz con un pequeño orificio (diámetro de 1/100 de la distancia al objeto como mínimo) por donde entra la luz. La luz reflejada por la escena que ha de ser captada pasa a través de dicho orificio y proyecta una imagen invertida en el extremo opuesto de la caja, donde se encuentra la película o el sensor de la cámara (Fig.3.3).

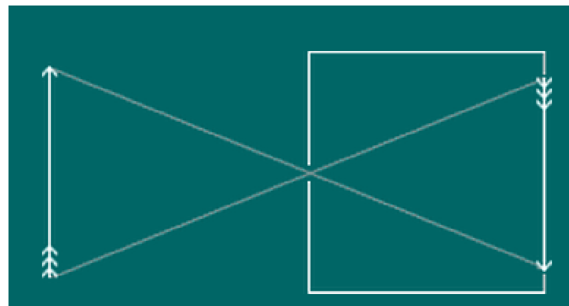


Figura 3.3: Modelo *Pin Hole*

Las ecuaciones que relacionan el mundo con el sistema óptico son las mostradas en las ecuaciones 3.1 y 3.2:

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

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

El problema que presenta esta solución es la falta de información de la tercera dimensión: la coordenada Z que representa la profundidad del objeto, no se puede determinar. Esta es la razón fundamental por la que resulta tan interesante la visión estéreo, como se verá en las ecuaciones 3.3, 3.4, 3.5, donde ya se supone un sistema estéreo basado en dos cámaras *pin hole* (Fig.3.4) con coordenadas $p_1 = (u_1, v_1)$ y $p_2 = (u_2, v_2)$

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

$$Y = \frac{b \cdot v_1}{d} \tag{3.4}$$

$$Z = \frac{f \cdot b}{u_1 - u2} = \frac{f \cdot b}{d} \tag{3.5}$$

donde d expresa la diferencia entre un punto captado por la cámara derecha y la izquierda (se introducirá en el texto como *disparidad*) y b es el valor de distancia a las que se encuentran las dos cámaras (*baseline*).

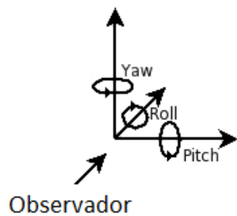


Figura 3.5: Ángulos *Pitch*, *Yaw* y *Roll*

Las cámaras usadas en aplicaciones de visión estéreo son idealmente idénticas (aunque en la práctica no es posible de ser llevado a cabo) y desplazadas una de la otra una distancia que dependerá del posterior uso que se quiera hacer de ellas (típicamente esta distancia será entre 45 y 75 mm), al que se conoce con el nombre de *baseline*. Además, hay que tener en cuenta otros parámetros denominados *roll* (ρ), *pitch* (θ) y *yaw* (ϕ), que se muestran en la Fig.3.5. Para entender su significado, puede entenderse que ϕ hace referencia a girar la cabeza de izquierda a derecha ($\longleftrightarrow$) y θ corresponde a mover la cabeza de arriba a abajo ($\updownarrow$). Estos son

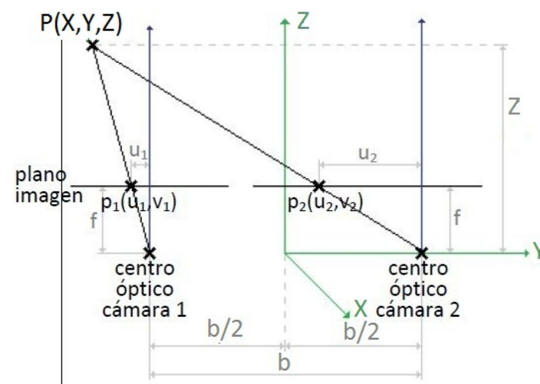


Figura 3.4: Modelo de Cámara Estenopeica para un Sistema Estéreo

parámetros denominados extrínsecos, y que no dependen del sistema óptico en sí, si no de la posición particular del sistema.

El disparo, y la toma de imágenes ha de ser sincronizada, de tal manera que lo que se espera obtener es un par de imágenes tomadas a la vez, pero con distinta perspectiva. Es con ello que se consigue un efecto tridimensional, pues al igual que pasa con el ojo humano, dependiendo de la distancia entre un punto del mundo en la imagen izquierda (análogo con la derecha) y la imagen derecha -lo que se designa como *disparidad*-, se puede valorar la distancia del objeto a la cámara.

La sensación de volumen, depende de la disparidad, que a su vez depende de la distancia entre los sensores de tal manera que:

- A mayor distancia entre sensores, mayor disparidad posible y mejor captación de la profundidad en objetos lejanos.
- A menor distancia entre sensores, menor disparidad y mejor captación de la profundidad en objetos cercanos.

Una imagen estéreo será la fusión de la imagen vista desde el sensor izquierdo con el derecho o viceversa. En la Fig.3.6 pueden apreciarse dos imágenes estéreo, y el resultado de superponer una a la otra.

No ha de dejar de explicarse de ningún modo el concepto de la denominada *geometría epipolar* (Fig.3.7): Se define el plano epipolar como aquel que

(a) Imagen Estéreo Izquierda

(b) Imagen Estéreo Derecha

(c) Superposición de Imágenes 3.6a y 3.6b

Figura 3.6: Imágenes Estéreo.

contiene los dos centros ópticos, C_L y C_R, y el punto X del mundo. Análogamente, se definen las líneas epipolares como la intersección del plano imagen con el plano epipolar. Los puntos de corte de la recta que une los centros ópticos de ambas cámaras con los planos de proyección son los epipolos. Si se determinan la aplicación entre puntos de la imagen izquierda (o derecha) y las rectas epipolares de la imagen derecha (o izquierda), se puede restringir la búsqueda para el emparejamiento de la proyeccion del punto de una imagen a lo largo de la línea epipolar correspondiente. El resultado práctico es la reducción del problema de dos dimensiones a una sola.

La restricción epipolar hace referencia a que los puntos correspondientes deben estar sobre líneas epipolares conjugadas.

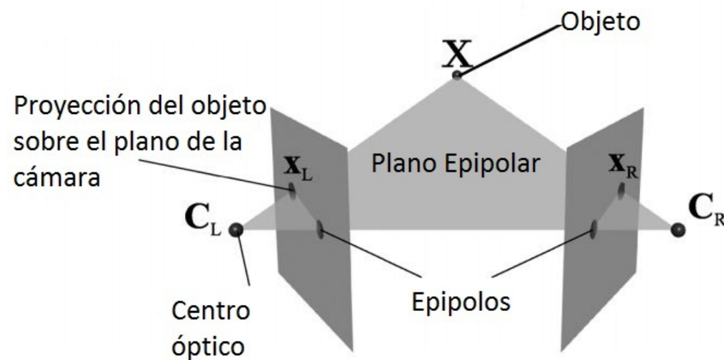


Figura 3.7: Geometría Epipolar

3.2. Manejo de Imágenes

3.2.1. Prepocesamiento de Imágenes

Las imágenes en formato RGB (*Red, Green, Blue*) son las más usadas para las imágenes que no tienen aplicación en ámbitos industriales, pues capturan el entorno tal cual se percibe por el ojo humano. Sin embargo, a la hora de aplicaciones industriales, prevalece la facilidad de cómputo frente a la captura de todas las características del medio. Esto es debido a que en la mayoría de las aplicaciones industriales no interesa tener la información que aporta el color, si no tan solo la información de los objetos (tamaño, posición en el mundo, etc). Como puede apreciarse en la Fig.3.8 puede entenderse por qué: al trabajar con el modelo de color RGB, se trabaja con tres canales, uno para cada color, y hay que almacenar el triple de información que en el caso de solo trabajar con un canal.

Se elige que las imágenes se presenten en distintos niveles de gris, con 255 posibles valores distintos. Aunque el ser humano no es capaz de distinguir tan enorme cantidad de tonalidades entre el blanco y el negro, para una máquina aporta una inmensa información con la que se puede trabajar, y todo ello con escaso almacenamiento en memoria. Esta es la razón por la que en todo el algoritmo se usan imágenes en blanco y negro.

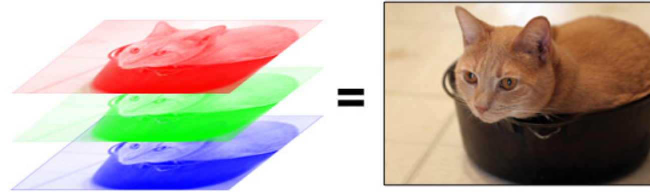


Figura 3.8: Modelo de color RGB

Laplaciana de la Gaussiana

La laplaciana de la gaussiana (LoG) de una imagen corresponde a aplicar la acción conjunta de un filtro gaussiano y el operador laplaciana, y su resultado es el que puede apreciarse en la Fig.3.9. Se basa en encontrar saltos bruscos de niveles de gris entre píxeles vecinos, y en la imagen resultante del filtro aplicado a tal fin, se aprecian solo los bordes, el resto de información de los objetos ha sido eliminado por no ser importante para el fin buscando. El fin último de este paso es el de poder solventar los errores que las mínimas diferencias entre las cámaras tengan en cuanto a su ganancia.

Figura 3.9: Detección de Bordes

La laplaciana es la segunda derivada de una función y representa la derivada de esta respecto a todas las direcciones:

$$\nabla^2 f(x, y) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \tag{3.6}$$

Para convolucionar una imagen con una gaussiana se hace uso de la expresión:

$$G(x, y) = e^{-\frac{(x+y)^2}{2\sigma^2}} \tag{3.7}$$

Por lo tanto, denominando $I(x, y)$ a la imagen sobre con la que se trabaja, aplicar el filtro resulta en:

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

Desarrollando 3.8 sabiendo 3.6 y 3.7 y llamando $r^2 = x^2 + y^2$

$$\nabla^2 G(x, y) = \frac{r^2 - 2\sigma^2}{\sigma^4} \cdot e^{-\frac{r^2}{2\sigma^2}} \tag{3.9}$$

Por último, denotar que cuanto más estrecha sea la gaussiana, más bordes se detectarán [30].

3.2.2. El Mapa de Disparidad

Una vez que se han captado ambas imágenes, se procede a construir el que se conoce como *mapa denso de disparidad*. Se conoce así a la representación en una única imagen de los valores de la disparidad de cada píxel de la imagen de las dos imágenes originales, y en el cual, a través de los distintos niveles de gris que se obtienen en la imagen, se es capaz de poder ver la distancia del objeto a la cámara. Los objetos más lejanos presentan un nivel de gris más intenso que el que tienen los objetos cercanos.

El mapa de disparidad consiste en encontrar la proyección del mismo punto sobre las dos imágenes, y detectar a qué distancia, medida en píxeles, está una del otro, lo que se designará con *disparidad d*. Para ello, se seguirá el algoritmo 1 [31].

El resultado obtenido puede observarse en la Fig.3.10.

Para poder efectuar el paso 6, es necesario realizar cuatro pasos:

1. *Calcular la función de coste*: La función de coste analiza la diferencia entre el mismo píxel en las dos imágenes estéreo. En esta aplicación se hace uso de la SSD (*Sum of Squared Differences*) para calcularlo.

Algoritmo 1 Algoritmo para Mapa Denso de Disparidad

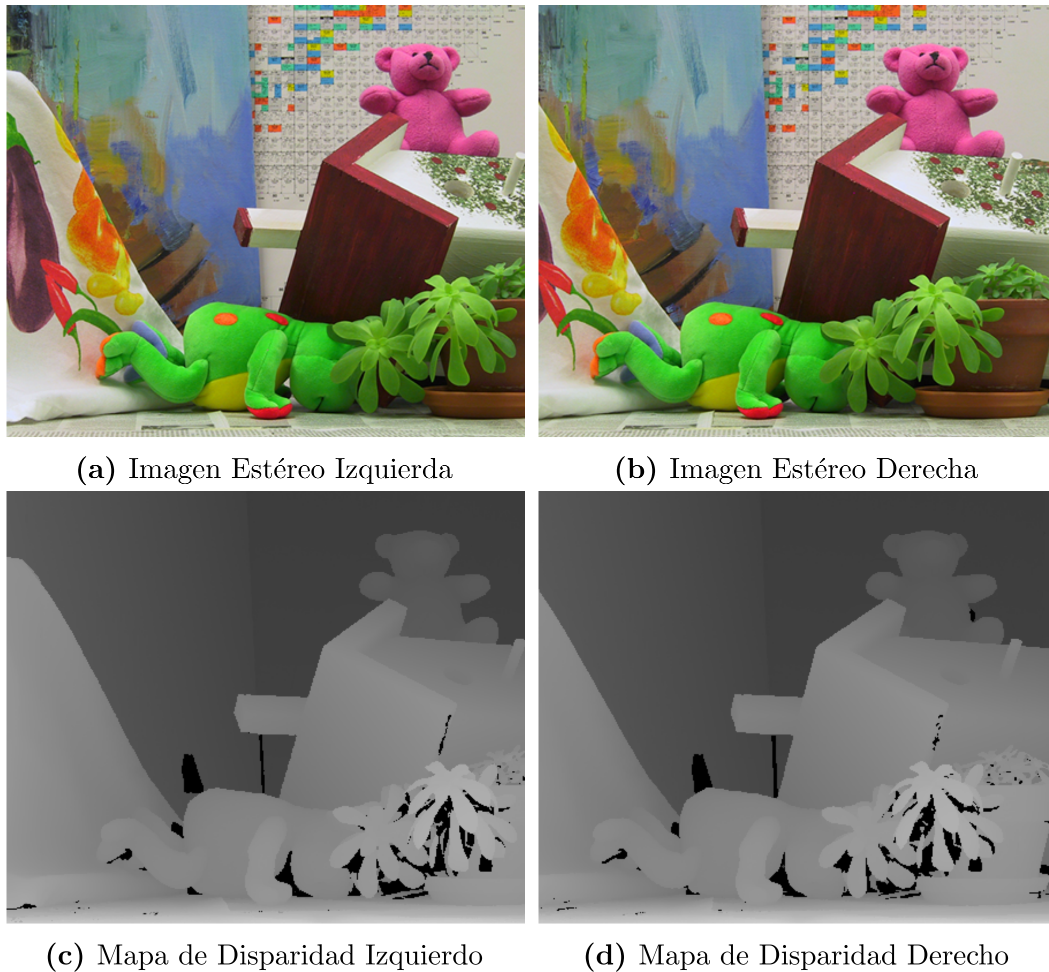
1: **para** un sistema estéreo con todos los parámetros intrínsecos conocidos **hacer:**
2:   **para** un par de imágenes estéreo **hacer:**
3:     **si** las imágenes no cumplen con la geometria epipolar, proceder a una rectificación para asegurar que las coordenadas verticales de un mismo punto en cada imagen (izquierda y derecha) sean iguales. **entonces:**
4:       Se aplica un filtro para eliminar la información irrelevante: Solo han de permanecer los bordes de la imagen. Laplaciana de la Gaussiana (*Filtro Marr-Hildreth*).
5:       Recorrer la imagen derecha buscando la proyección del mismo punto en la imagen izquierda.
6:       Dependiendo de la distancia d entre dichas proyecciones, se obtendrá el nivel de gris que abarca de 0 a $d_{máx}$ que el píxel adquirirá en el mapa de disparidad.
7:     **fin si**
8:   **fin para**
9: **fin para**

2. *Agregación del coste*: Se comparan los valores de intensidad de los píxeles que forman la región de soporte, agregándose los costes sumados o promediados.

3. *Cálculo de la disparidad*: Existen varias formas distintas de ejecutarlo (métodos globales, optimización global, programación dinámica o algoritmos cooperativos).

4. *Postprocesamiento*: Son etapas que aseguran que los resultados obtenidos son los correctos. Un caso típico es el comparar los mapas de disparidad tomando la imagen derecha como base y el correspondiente tomando la imagen izquierda.

El mapa de disparidad es una imagen de dimensiones igual a la imagen, con valores en cada punto que van de 0 hasta $d_{máx}$ (disparidad máxima), que es un dato que viene dado por los parámetros del sistema, tales como la distancia focal f, o la *baseline*.



(a) Imagen Estéreo Izquierda (b) Imagen Estéreo Derecha

(c) Mapa de Disparidad Izquierdo (d) Mapa de Disparidad Derecho

Figura 3.10: Imágenes Estéreo y sus Correspondientes Mapas de Disparidad.

3.2.3. u-v *Disparity*

El mapa de disparidad es de gran utilidad para determinar la distancia de la cámara a los objetos, pero se ha de introducir también la utilidad de las imágenes u y v [32].

Con el u-v *Disparity*, se consigue hacer una estimación espacial de la localización de los objetos.

v-*Disparity*

El v-*Disparity* es una imagen que se obtiene tomando como base el mapa de disparidad. Tiene como dimensiones la altura de la propia imagen, y como ancho, el de la disparidad máxima. Se recorre la imagen por filas, calculando el histograma para cada fila, de tal manera que la imagen resultante representa en cada una de sus filas el histograma de la correspondiente fila en el mapa de disparidad. Para mayor aclaración, se presenta el esquema de la Fig.3.11, en el que la primera fila de la imagen se ha hecho como ejemplo con un código de colores.

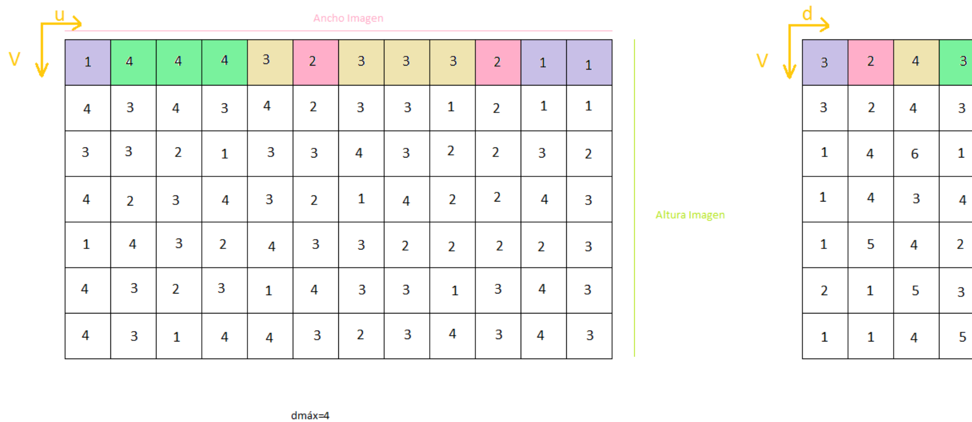


Figura 3.11: Detalle de la Realización de un Mapa v-*Disparity*

En la Fig.3.12 puede verse cómo es un v-*Disparity* típico en aplicaciones de vehículos inteligentes.

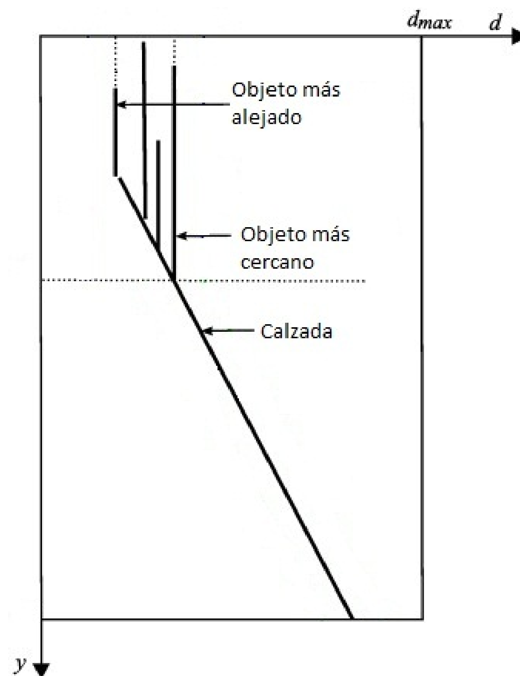


Figura 3.12: v-Disparity en una Aplicación de Vehículos

u-*Disparity*

La realización del u-*Disparity* es análoga a la del v-*Disparity*, con la salvedad de que se realiza para cada columna del mapa de disparidad en lugar de para cada fila.

En el caso de que uno de los píxeles del v o del u *Disparity* sean mayores de 255, que es el valor más alto en una imagen de 8 bits, el valor asignado es simplemente de 255.

3.2.4. El Mapa Libre y el Mapa de Obstáculos

El *mapa libre* es una variación del mapa denso de disparidad, con la diferencia de que solo aparecen en él los objetos de mayor tamaño. Esto es debido a que en el u-*Disparity*, se buscan las posiciones en las que el valor de dicho mapa es mayor que un predeterminado valor, y solo se queda con estos. El resultado obtenido es un mapa denso de disparidad en el que solo están

representados las zonas libres de obstáculos, al contrario de lo que pasa en el mapa de obstáculos, en el que solo se pueden distinguir los obstáculos que hay en el entorno.

3.3. Herramientas

3.3.1. CUDA

Las GPU (*Graphics Processing Unit*) son procesadores especiales que tradicionalmente fueron usados para liberar de carga de computación a las CPU en tareas de índole gráfico, y con ello lograr que se acelerasen los procesos en cuestión (ver tabla 3.1). En la actualidad, las GPU son más bien unidades de procesamiento multinúcleo altamente paralelizables que permiten el manejo de operaciones de propósito general en unidades de procesamiento de gráficos, a lo que se conoce como GPGPU (*General-Purpose Computing on Graphics Processing Units*). Las GPGPU son una realidad desde el año 2002, pero no fue hasta que *nVidia* lanzase CUDA en 2007, cuando empezaron a suscitar un interés creciente. CUDA es una arquitectura de programación basada en paralelización desarrollada por la marca comercial de tarjetas gráficas *nVidia* para el procesamiento de aplicaciones de propósito general. Es la forma de programar que han creado para poder usar las GPU, basado en lenguajes de programación C/C++, pero con algunas instrucciones propias. El éxito de esta arquitectura radica en el acierto de *nVidia* de ofrecer una forma fácil de poder programar sobre tarjetas gráficas, al contrario de lo que pasaba hasta su introducción en el mercado. Es una arquitectura útil, con gran poder de programabilidad y resultados robustos. Los competidores directos de este entorno son principalmente *AMD* con *ATI Stream*, y los estándares *OpenCL* y *DirectX11 DirectCompute* [33].

CPU	**GPU**
Tecnología Madura	Alto Ritmo de Desarrollo
Muchas Tareas de Carácter General	Alta especialización
Gestión de Operaciones Secuencial	Posibilidad de Paralelizar Operaciones
Frecuencia de Reloj Alta (3.8-4GHz)	Frecuencia de Reloj Baja (600-800 MHz)

Tabla 3.1: Comparativa CPU-GPU

El modelo básico de las GPU radica en el procesamiento de flujo (*stream processing*), basada en en la técnica de computación denominada SIMD (*Single Instruction, Multiple Data*) (Sistema de búsqueda de la paralelización mediante instrucciones que aplican una misma operación sobre un conjunto de datos de tamaño variable).

La aplicación de las técnicas de paralelización, sin embargo no deben ser usadas de forma universal, pues estas técnicas presentan ventajas con respecto a la CPU en operaciones de alta intensidad aritmética, paralelismo en los datos, y datos localizados. No es recomendable el uso de estas prácticas en el caso de que los datos presenten dependencia entre si, o se necesite acceso a memoria no estructurado. En aplicaciones gráficas, donde vértices, fragmentos e incluso cada píxel puede ser procesado de forma independiente, con regiones de memoria claramente delimitados, es donde se puede aprovechar todo el potencial de las GPU.

La diferencia entre una GPU y una CPU es tan notable que incluso el propio hardware es completamente diferente (Fig.3.13). Para ilustrar con un ejemplo el por qué de semejante varianza, baste decir que sin existir dependencia de datos, las cachés pueden reducirse de tamaño, y esos transistores pueden ser usados para una ALU (*Arithmetic Logic Unit*). Así, una CPU usa la mayoría de sus transistores para controlar las cachés, y la GPU lo hace para tener más ALU [1]

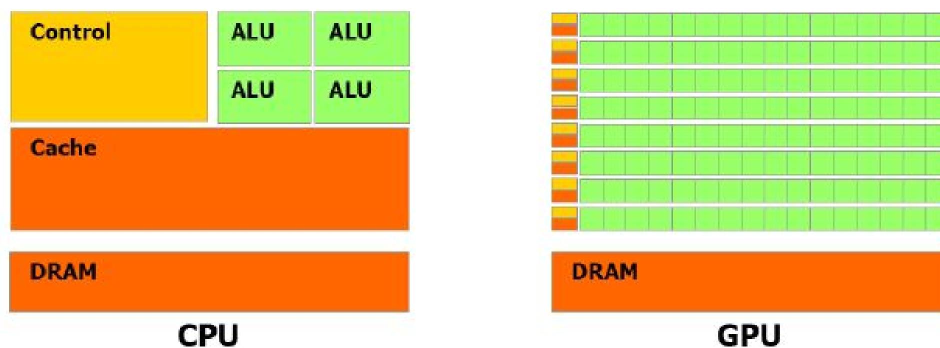


Figura 3.13: Filosofía de Diseño de una CPU (izq) y una GPU (der)

[1]. La caché es una memoria diminuta y rápida, que almacena copias de datos ubicados en la memoria principal que se utilizan con más frecuencia, y la ALU es un circuito digital que calcula operaciones aritméticas (como suma, resta, multiplicación, etc.) y operaciones lógicas (si, y, o, no), entre dos números.

La estructura de funcionamiento de CUDA se puede ver en la Fig.3.14, en la que cada parte responde a:

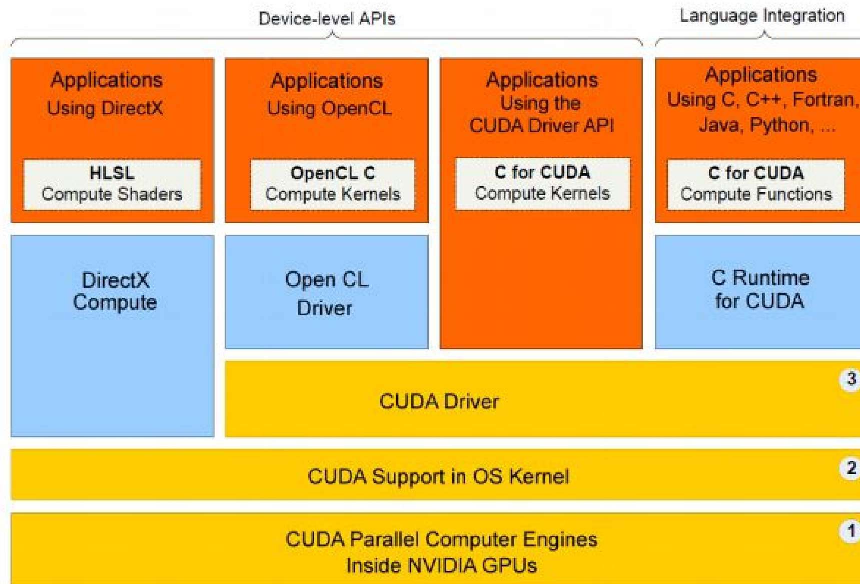


Figura 3.14: Estructura de Trabajo de *CUDA*

1. Tarjeta gráfica soportada por CUDA.
2. Soporte del sistema operativo de la arquitectura CUDA.
3. Driver de CUDA

La ventaja de trabajar con CUDA es la posibilidad de usar tanto la CPU como la GPU al mismo tiempo de forma heterogénea, de forma que la primera efectúa las operaciones altamente secuenciales y la segunda soporta las operaciones más factibles de ser paralelizadas [34].

Programación en *CUDA*

La filosofía de programación con CUDA se basa preferentemente en base a dos pasos:

1. Dividir el problema planteado en conjuntos independientes ⇒ *Grid*
2. Dividir dichos conjuntos en subtareas que pueden ser llevadas a cabo de forma cooperativa ⇒ Bloques de hilos.

Se ha de escribir un código en lenguaje C que invoque *kernels* (parte del código que es ejecutado por cada hilo) paralelos, que son típicamente ejecutados por cientos de hilos, y que son agrupados en bloques de hilos, cada uno de ellos aplicando las mismas operaciones que los demás hilos de su bloque, pero sobre una región de los datos distinta. Acceden a una zona de memoria específica de cada bloque para guardar los datos que hayan calculado, y son capaces de ser sincronizados para terminar el bloque cuando el último hilo haya terminado su operación. Para acceder a cada hilo, ha de saberse que cada uno tiene una identidad específica dada por `threadIdx`.

Los bloques están agrupados en un *Grid*, pero los bloques dentro de un mismo *Grid* son igualmente independientes. De forma análoga a los hilos, cada bloque tiene su propia identificación mediante `blockIdx` dentro del *Grid*. Los *Grid* pueden ser lanzados de forma dependiente o independiente: así, *Grid* independientes son lanzados a su vez en paralelo, y los dependientes en forma secuencial. La forma de lanzarse depende de la potencia del hardware usado. La estructura de organización de hilos, bloques y *Grid* puede verse en Fig.3.15.

Cada hilo usa una memoria local privada para registros, y operaciones de pila. Cada bloque usa una memoria compartida. La situación más común en el manejo de CUDA, es que el *kernel* inicialice variables en la memoria compartida, los hilos del bloque ejecuten sus operaciones usando esas variables, y los resultados sean copiados a una memoria global, visible para todo el proceso, y que permite la comunicación entre bloques de *Grids* independientes (Fig.3.16). En una arquitectura típica de CUDA, la memoria compartida está implementada en un on-chip [2] *RAM* de baja frecuencia de latencia, y situada cerca del procesador, mientras que la memoria global se encuentra alojada en una *DRAM*. La memoria de cada hilo pervive mientras existe el hilo.

[2] Sistemas, que al contrario que un microcontrolador, integran todos o gran parte de los módulos componentes de un ordenador o cualquier otro sistema informático o electrónico en un único circuito integrado o chip.

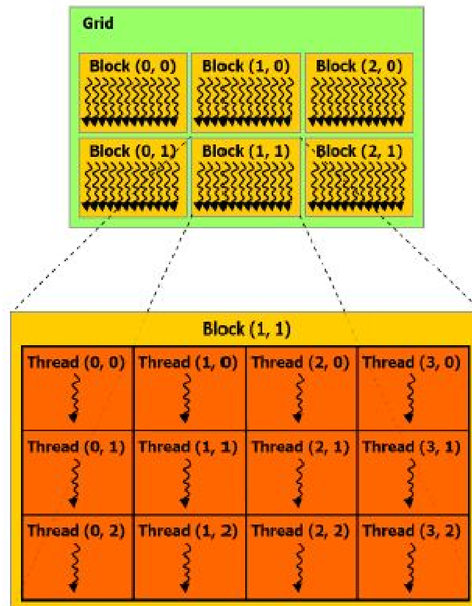


Figura 3.15: Hilo, Bloque, *Grid*

A la hora de programar en el entorno de CUDA, es importante tener en cuenta ciertas restricciones que harán que el rendimiento del proceso mejore, a saber [35], [36]:

1. No es posible crear hilos o bloques dentro de otros ya existentes.
2. Los bloques dentro de un mismo *Grid* no pueden comunicarse entre si, pues son independientes, pese a que si que comparten memoria global. Esto aumenta la escalabilidad, pues los bloques puede darse el caso de que estén distribuidos en distintos espacios de memoria.
3. No es recomendable efectuar funciones de recursividad, pues viola los fundamentos de corriente de flujo que gobiernan esta aproximación.

La mejor aproximación para guardar datos es mediante lo que se conoce como "textura", pues son datos que son alojados en caché, y que permiten un manejo de datos más rápido y eficiente.

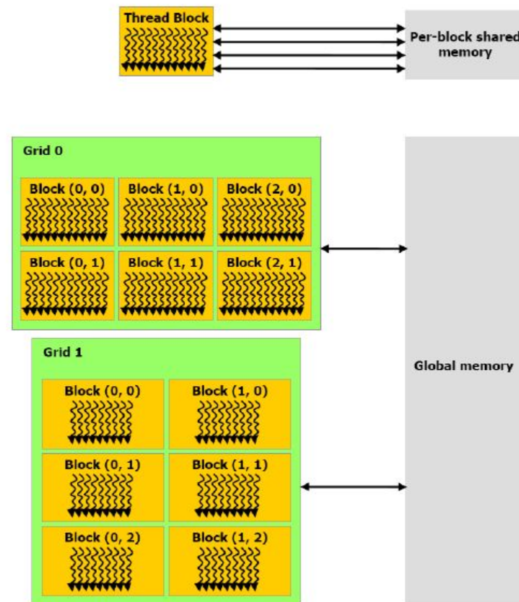


Figura 3.16: Flujo de Información en una Aplicación *CUDA*

Modelo de *Host* y *Device*

Según se muestra en la Fig.3.17, el *Host* hace referencia a las operaciones llevadas a cabo en la CPU, mientras que el *Device* es el que hace referencia a las operaciones llevadas a cabo en la GPU.

Los hilos se pueden ejecutar sobre un *Device* físicamente separado y que opera como coprocesador de ayuda al *Host*, que está ejecutando el programa en C. Un ejemplo es cuando los *Kernels* se están ejecutando en la GPU y el resto del programa se lleva a cabo en la CPU. De nuevo hay que mencionar la diferencia entre memorias: existe una memoria del *Device* distinta de la del *Host*.

Ventajas de Usar *CUDA*

La elección de CUDA como forma de trabajo para este proyecto puede darse en función de siete criterios:

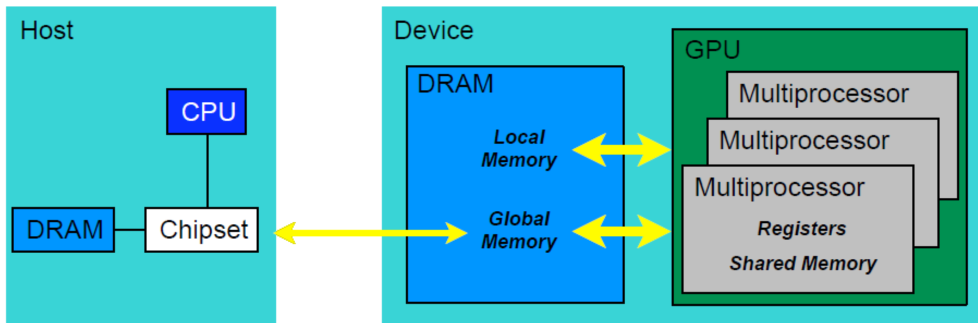


Figura 3.17: Arquitectura CUDA

- *Difusión*: La primera tarjeta gráfica en soportar CUDA fue la *G80* (serie *GeForce 8800*) en Noviembre de 2006, que poseía 16 multiprocesadores, cada uno de ellos con 16 KB de memoria compartida, y 8 procesadores para mejorar la corriente de flujo. Desde ese momento *nVidia* ha vendido más de 100 millones de tarjetas gráficas con soporte para CUDA.

- *Inversión*: El *software* para poder usar CUDA es gratuito, y el *hardware* no implica coste privativo, debido en parte al punto anterior: los costes de fabricación de tarjetas son competitivos por el alto volumen de ventas que presenta. Además, al ser un campo de interés creciente, la competencia obliga a estar en continua guerra de precios.

- *Mercado*: Las alternativas a CUDA son *ATI Stream*, *OpenCL*, y *Direct-Compute*, de las cuales, las dos últimas presentan la ventaja de ser independientes del *hardware* usado. La ventaja de CUDA en este punto, es que soporta la programación en ellas también.

- *Innovación*: *nVidia* no es nueva en el mercado, y mucho antes de la introducción de CUDA ya distribuía tarjetas gráficas. El bagaje del *know-how* que presenta la marca es fundamental para optar por este entorno.

- *Herramientas de desarrollo*: *nVidia* está constantemente desarrollando nuevas herramientas para incrementar el rendimiento de la programación en CUDA. El siguiente paso de la compañía es lograr que Microsoft Visual Studio integre la programación en CUDA en futuras versiones del *software*.

- *Modelo de programación*: Esta aproximación reduce la dificultad de programación en paralelo con respecto a las antiguas. Es una forma de

programar fácilmente escalable, y no requiere conocimientos de lenguajes nuevos de programación. Permite además, la opción de paralelizar solo ciertas partes del código.

- *Aplicaciones y rendimientos*: Para aplicaciones científicas, los aumentos de velocidad son considerables, con respecto a efectuar las operaciones en CPU. Los manejos de memoria son rápidos y robustos.

3.3.2. OpenCV

En 1999 *Intel* creó la librería de visión artificial OPENCV (*Open Source Computer Vision Library*) (Fig.3.18). Su uso es libre bajo la licencia BSD (*Berkeley Software Distribution*), y es *cross-platform*, lo que significa que es capaz de funcionar en una gran variedad de computadores, sin depender de sistema operativo. Su funcionalidad más relevante es la posibilidad de poder procesar imágenes en tiempo real. Se puede hacer uso de ella a través de los lenguajes de programación C#, Ch, Python, Ruby y Java [37].

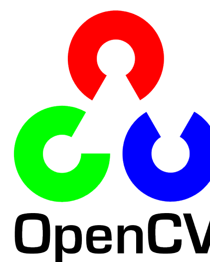


Figura 3.18: Logo de las *OpenCV*

Las áreas de aplicación de OPENCV son comúnmente:

- Manejo de Imágenes en 2D y 3D.
- Estimación de la Odometría Visual.
- Sistemas de Reconocimiento Facial.
- Reconocimiento de Gestos.
- Interacción Humano-Robot.
- Robots Móviles.
- Identificación de Objetos.
- Segmentación y Reconocimiento.
- Visión Estereoscópica.

- Extracción de Características en Movimiento.
- Seguimiento del Movimiento.

Para llevarlas a cabo, OPENCV hace uso de librerías que incluyen:

- Boosting.
- Árboles de Decisión.
- Algoritmo esperanza-maximización
- $\mathcal{K}$-nn.
- Clasificador Bayesiano.
- Redes Neuronales.
- Bosques de Árboles de Decisión.
- Máquinas de vectores de soporte.

3.3.3. Matrox MIL

En el código expuesto se usan tanto las librerías OPENCV como las que se introducirán en esta sección: las *Matrox* MIL (*Matrox Imaging Library*).

Matrox es un fabricante de chips gráficos y componentes para PC canadiense, que tiene tres líneas diferenciadas de operación: *Matrox Graphics*, *Matrox Video* y *Matrox Imaging*. *Matrox Graphics* es la marca orientada al usuario final. *Matrox Video* está orientada a la edición de vídeo digital, por último *Matrox Imaging* vende capturadoras de vídeo de alta gama y cámaras inteligentes orientadadas a visión artificial.

MIL es un conjunto de *software* que permite la implementación de aplicaciones de visión estéreo, análisis de imágenes y también aporta herramientas para el entorno de la visión por computador para aplicaciones médicas.

3.4. Algoritmos

3.4.1. RANSAC

En cualquier proceso experimental, los datos obtenidos tienden no solo a ser diferentes entre si (el fin de cualquier experimento práctico es analizar cuán bien se ajustan los datos a una previa hipótesis formulada), sino que en ocasiones, existen datos atípicos que no son debidos al fenómeno en sí, si no que se deben a errores en el aparataje de medida, a errores humanos, a fallos de precisión, etc. Estos errores pueden terminar por ser decisivos a la hora de plantear conclusiones al experimento, suponiendo que existan desviaciones frente a los valores teóricos de carácter notable. En ocasiones, estos datos pueden ser eliminados, y posteriormente, efectuar un ajuste a un modelo (recta, por ejemplo) por el método de los mínimos cuadrados, o cualquier otro procedimiento similar. Estos modelos escogen todos los datos y estiman los parámetros de aquella recta, parábola, etc, que mejor reúne a todos los puntos.

Sin embargo, en determinadas áreas es inabarcable, bien por falta de medios, bien por falta de tiempo, el estimar qué puntos son adecuados y cuáles no. Por ejemplo, en la visión por computador, en el caso a estudio en el presente texto, no se podría detectar qué puntos son válidos por el enorme gasto de tiempo que ello supondría, y que rompería por completo la especificación necesaria de computación en tiempo real. Los resultados del modelo resultante serían erróneas, y todo ello, sin la certeza de si el fallo en el ajuste ha sido debido a una mala aproximación a la hora de tomar los datos, o si por el contrario, se de el caso de que los datos sean correctos, pero los errores ajenos tengan demasiado peso. Un ejemplo de datos bien ajustados y otro de lo contrario se aprecia en la Fig.3.19.

Se presenta por tanto la necesidad de elegir un nuevo algoritmo, introducido en 1981 [38] que presenta como novedad que no usa todos los datos para hacer una estimación de los parámetros de la recta o curva que mejor agrupa a la mayoría de datos. Así, ha de definirse la figura del *inlier*, que es aquel punto que entra dentro de un rango de distancia predefinido (tolerancia ϵ) tomado desde el punto de la recta o curva ajustada hasta el propio punto. Este método es el conocido como RANSAC, que lo que hace es elegir dos puntos aleatorios entre los datos existentes, calcular la recta que pasa por ambos, y

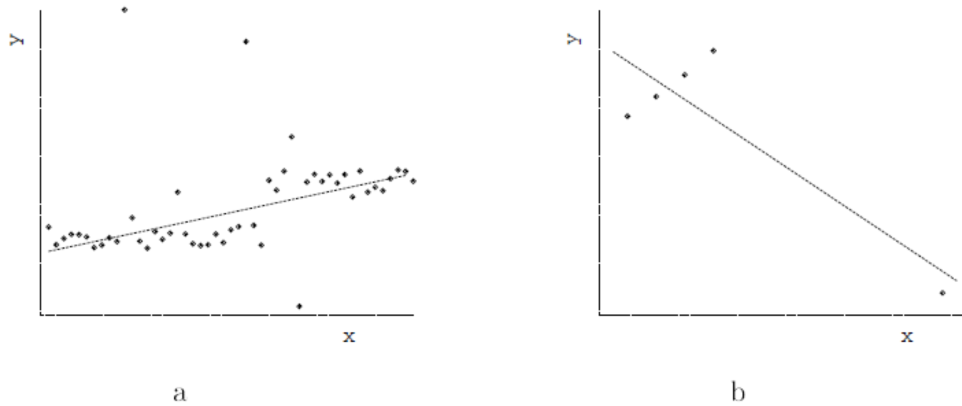


Figura 3.19: Ejemplo de ajuste datos, teniendo todos en consideración. (a) da un resultado correcto, (b) no ha ajustado bien la recta a los datos

obtener el número de *inliers*. El resultado puede verse en la Fig.3.20, donde en naranja se simboliza el límite para considerar un punto *inlier* (el punto se sitúa dentro de las dos líneas naranjas), los puntos de color rojo son *outliers*, y los azules están ajustados a una recta [39].

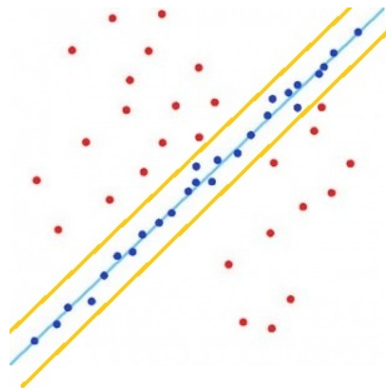


Figura 3.20: Ejemplo de *inliers*-azul-, *outliers*-rojo- y recta ajustada.

El proceso suele constar de varias iteraciones, en cada una de la cual se va almacenando el valor de los parámetros estimados, así mismo, del número de *inliers* obtenidos, siendo elegida la solución que más puntos ha tenido en consideración al ajustar la recta. El objetivo es, por tanto, no considerar todos los datos de la muestra, para no obtener una recta que tenga en cuenta valores erróneos, pero tampoco desechar un número alto de puntos por considerarlos "contaminados".

En la actualidad, este método es muy robusto y se utiliza en multitud de aplicaciones de visión por computador, pues las operaciones a llevar a cabo no son complejas.

El número de iteraciones k que ha de llevarse a cabo para poder asegurar que los parámetros mínimos necesarios n para poder dar una expresión del ajuste de datos obtenidos son correctos con una probabilidad z, ha sido estimado de acuerdo a la expresión 3.10.

$$k = \frac{\log(1-z)}{\log(1-w^n)} \tag{3.10}$$

w representa la probabilidad de que un valor esté en distancia por debajo del valor de tolerancia para poder asegurar que un valor es *inlier* (dicho en otras palabras, la probabilidad de que un punto determinado sea *inlier*), pero este dato no siempre es conocido, por lo que en ocasiones resulta más interesante estimar k y de ahí obtener el valor que debe tener w.

El caso de la tolerancia ϵ, que divide los datos en *inliers* y *outliers*, es similar al de k, y no es posible hacer una estimación de él a priori de él en muchos casos.

La búsqueda de hipótesis se basa en minimizar la función de costes dada por el modelo $\mathcal{M}$ (ecuación3.11 y 3.13) [40]:

$$\mathcal{M} = \{d \in \mathbb{R}^d : f_{\mathcal{M}}(d;\theta)\} \tag{3.11}$$

donde $D = \{d_1...d_N\}$ son los datos de partida, compuestos por N elementos, θ es un vector que contiene a los parámetros estimados, y $f_{\mathcal{M}}$ es una función suave cuyo nivel cero contiene a todos los puntos que se ajustan al modelo $\mathcal{M}$ en un momento dado aplicando el modelo dado por θ.

La función de costes:

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

d representa cada uno de los conjuntos mínimos de datos extraído de D y ρ es la *función de pérdida*:

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

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

3.4.2. SURF

SURF (*Speeded Up Robust Feature*) es un descriptor invariante a rotación y escala que es capaz de encontrar puntos iguales en dos imágenes distintas, bien pueden ser éstas dos imágenes estéreo consecutivas, o la misma imagen a la que se le ha aplicado una transformación, tal como girar, o cambiar su tamaño, o incluso aunque el objeto esté parcialmente oculto.

Figura 3.21: Ejemplo Básico del Algoritmo SURF, Aplicado Sobre una Zona Parcial de una Imagen, Aplicada una Rotación.

Las ventajas de usar SURF frente a otras soluciones que aportan las mismas soluciones (principalmente SIFT), es su mejora en cuanto a tiempos de cálculo.

El algoritmo que sigue SURF es el mostrado en el algoritmo 2.

Algoritmo 2 SURF

1: Buscar puntos de interés. Tales puntos de interés son bordes, esquinas, cambios de tonalidad, etc
2: Se representa el entorno de cada punto de interés en un vector de características. Comúnmente se les denomina “descriptores” del punto de interés. Ha de ser robusto, invariante al ruido, a deformaciones geométricas o fotométricas, y asegurar la repetitividad.
3: Localizar los descriptores en las demás imágenes que se han introducido al sistema, basándose en la similitud entre vectores.

Lo más importante en un algoritmo como el SURF es su repetitividad, que hace referencia a la posibilidad de encontrar el punto característico en distintas condiciones e imágenes, pues un pobre resultado en la búsqueda de puntos característicos bajo distintas condiciones hace al algoritmo perder la robustez. [41]

Para buscar los puntos de interés existen multitud de soluciones, de las cuales solo se presentará el detector Hessiano, pues son sistemas que han sido implementados para poder ejecutar códigos como SURF, y que son efectivos a la par que invariantes a escala y rotación, con buena repetitividad. Y todo ello con bajo tiempo de procesamiento y precisión.

En aquellos píxeles en los que al hacer esta operación, se obtengan los valores máximos, se supone que son un punto de interés.

El Detector Hessiano

Dado un punto $X = (x, y)$ de una imagen I, se define la matriz Hessiana $\mathcal{H}(X, \sigma) =$ del punto X, bajo escala σ como

$$\mathcal{H}(X,\sigma) = \begin{pmatrix} L_{xx}(X,\sigma) & L_{xy}(X,\sigma) \\ L_{yx}(X,\sigma) & L_{yy}(X,\sigma) \end{pmatrix} \tag{3.14}$$

Donde $L_{xx}(X, \sigma)$ hace referencia a la convolución de la derivada de segundo orden de la Gaussiana $\frac{\delta^2}{\delta x^2}g(\sigma)$ con la imagen I en X. El planteamiento es análogo con $L_{xy}(X, \sigma)$ y $L_{yy}(X, \sigma)$.

Cabe destacar que el filtro es simétrico y por tanto, $L_{xy}(X, \sigma) = L_{yx}(X, \sigma)$.

El uso de Gaussianas es óptimo teóricamente, pero en la práctica necesitan ser discretizadas y cuantificadas, lo cual lleva a que se pierda repetitividad. La discretización y cuantificación, presentan la ventaja de la velocidad de cálculo, y la pérdida de dicha repetitividad es aceptable.

Dado que ningún filtro es ideal, el próximo paso es usar simplemente filtros-caja, que son aproximaciones a la derivada de segundo orden de la Gaussiana (3.22).

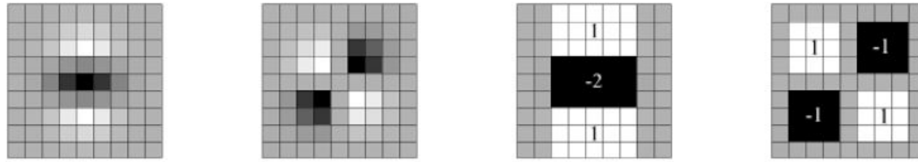


Figura 3.22: De izquierda a derecha: Derivadas Parciales, Discretizadas y Cuantificadas en Dirección y y xy, y Aproximaciones Obtenidas tras el uso de Filtros de Caja.

Así pues, las aproximaciones de $L_{xx}(X, \sigma), L_{xy}(X, \sigma), L_{yy}(X, \sigma)$ se designarán con D_{xx}, D_{xy}, D_{yy}. Por tanto, el valor del determinante de la Hessiana queda:

$$det(\mathcal{H}_{aprox}) = D_{xx} \cdot D_{yy} - w \cdot {D_{xy}}^2 \tag{3.15}$$

Donde w es el peso relativo de la respuesta del filtro y se usa para regular la expresión del determinante.

El determinante aproximado de la matriz Hessiana se utiliza para la localización de puntos y para determinar la escala. Para conservar el equilibrio entre la solución sin aproximación y la aproximada, por ejemplo si se usa un filtro 9x9 (el más pequeño) para una Gaussiana con $\sigma = 1,2$, se obtiene que $w = 0,9$.

$$w = \frac{|L_{xy}(1,2)|_F |D_{xx}(9)|_F}{|L_{yy}(1,2)|_F |D_{xy}(9)|_F} = 0,912... \simeq 0,9 \qquad (3.16)$$

donde $|x|_F$ simboliza la norma de Frobenius, la cual es invariante al tamaño del filtro usado.

El resultado de este paso es haber encontrado los puntos de interés.

Descriptores

Para localizar los puntos de interés en la imagen original y en las escaladas, se procede a la eliminación de los puntos que no sean máximos en la región vecina $3 \times 3 \times 3$ (ancho, altura, escala). El máximo determinante de la matriz Hessiana se interpola en la escala y el espacio de la imagen.

A continuación se busca la orientación del descriptor basada en la información de una zona circular alrededor del punto de interés y se procede a construir una región cuadrada alineada con dicha orientación. Así se obtiene ya el descriptor SURF.

3.4.3. FLANN

El último paso para encontrar el mismo punto en dos imágenes distintas, es el algoritmo implementado en la librería FLANN (*Fast Library for Approximate Nearest Neighbors*). Hace uso de los descriptores encontrados en el paso anterior por SURF, y su función es encontrar el emparejamiento de los descriptores entre ambas imágenes.

FLANN reduce en un orden de magnitud el tiempo de computación con respecto a los demás códigos semejantes [42].

FLANN se encuentra en una biblioteca de la OPENCV, por lo que su uso será inmediato.

3.5. Conocimientos Básicos de Vehículos Necesarios

Aunque no se pretende un estudio exhaustivo del funcionamiento mecánico de los automóviles, si se expondrán aquellos conceptos necesarios para el fin de este proyecto.

Los vehículos comerciales siguen el denominado modelo de geometría de Ackermann , el cual será el que se siga en este texto.

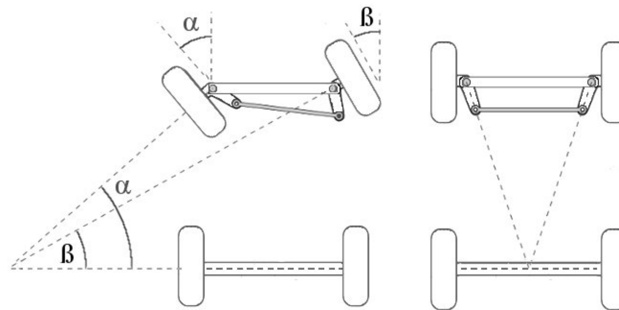


Figura 3.23: Geometría de Dirección de Ackermann

La geometría de Ackermann hace referencia al problema que presenta la dirección de un vehículo al tomar una curva, y que se soluciona haciendo que las dos ruedas directrices no giren el mismo radio con respecto a un punto fijo, tal como se puede distinguir en Fig.3.23, sino que la rueda situada en el lado interior de la curva gira un ángulo ligeramente mayor que la otra, siendo posible que los ejes perpendiculares a cada una de las ruedas interesecten en un único punto[43].

Interesa saber el comportamiento de la rueda del vehículo con el trazado: La resistencia a la rodadura se presenta cuando un cuerpo rueda sobre una superficie, deformándose uno de ellos o ambos.

El concepto de coeficiente de rodadura es similar al del coeficiente de rozamiento, con la diferencia de que este último hace alusión a dos superficies que deslizan o resbalan una sobre otra, mientras que en el coeficiente de rodadura no existe tal resbalamiento entre la rueda y la superficie sobre la que rueda, disminuyendo por regla general la resistencia al movimiento.

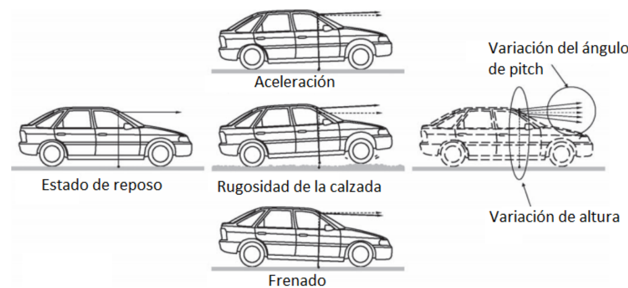


Figura 3.24: Desplazamientos de la Masa del Vehículo en Frenadas y Aceleraciones.

Por un lado, a escala microscópica una rueda no presenta un alzado exactamente circular, y la superficie sobre la que rueda no constituye tampoco un perfil plano, puesto que en ambos casos existen irregularidades. No obstante, este no es el principal factor que influye en el coeficiente, sino la histéresis. La rueda, en función del material con el que esté construida y su propio peso, además del de la carga que soporta, sufre una deformación que al rotar provoca repetidos ciclos de deformación y recuperación, estos ciclos propician la disipación de energía por calor. Además, esta deformación supone que no apoye una línea únicamente sobre la calzada, sino una superficie. Todo ello genera vibraciones en el vehículo durante la marcha.

Al trazar el vehículo una curva aparece una fuerza centrífuga que tiende a desplazarlo hacia el exterior de la curva. Para contrarrestar esta fuerza, aparece una fuerza de rozamiento entre pavimento y neumático proporcional a un coeficiente de rozamiento estático, de forma que la fuerza de rozamiento es perpendicular a la trayectoria del vehículo.

Se genera un desplazamiento de la masa del vehículo hacia la rueda exterior a la curva, por lo que todo se traduce en que existe una diferencia de altura al tomar un vehículo una curva, concepto que será tratado en el proyecto para poder entender varios resultados obtenidos. Lo mismo pasa al frenar o acelerar el vehículo (Fig.3.24).

Estos cambios que experimenta el coche cuando se está desplazando son imprescindibles para entender las variaciones a las que está sujeta la cámara y que hace necesaria su calibración.

4

CÁLCULO DE LA ODOMETRÍA VISUAL

Este capítulo tiene como fin explicar las aproximaciones llevadas a cabo a la hora de realizar la programación del algoritmo que se encarga de calcular la odometría visual; mientras que el próximo capítulo se discernirá acerca de la autocalibración.

El algoritmo que hace la odometría visual está basada en encontrar puntos característicos en la calzada y emparejarlos entre dos *frames* consecutivos, para con ello poder estimar el movimiento que realiza el vehículo. Para ello se hará uso a lo largo de todo el proceso de la programación multihilo en GPU, y de los algoritmos ya explicados en el capítulo 3 RANSAC, SURF y FLANN [44] [45].

4.1. Obtención del Mapa de Disparidad y u-v ***Disparity***

En la sección 3.1.1 ya se explicó el concepto de los mapas de disparidad, su utilidad y sus usos. En el algoritmo que se presenta, este es el primer paso a realizar, y del cual puede encontrarse mayor información en [46] y [47].

El mapa de disparidad está realizado sobre la GPU, y responde a los siguientes pasos:

1. *Filtrar las imágenes*: Las imágenes tienen ruido que ha de ser eliminado, además de la información que es irrelevante, como ya se hizo alusión en 3.1.1, por lo que es necesario aplicar el filtro de la Laplaciana de la Gaussiana.

2. *Cálculo del mapa de disparidad*: El mapa de disparidad se consigue mediante la programación multihilo, en base a lanzar un *Kernel* que coge una ventana de píxeles en la imagen izquierda, la sitúa sobre la imagen derecha y busca el mejor valor de disparidad (ver Fig.4.1). Las diferencias se calculan mediante SSD, almacenándose en cada momento el valor mínimo de SSD obtenido, y comparando el valor actual con dicho valor mínimo. En caso de que sea mayor, no se tiene en cuenta ese valor, y si fuera menor que el valor almacenado, ese sería el nuevo patrón. El objetivo de este paso es encontrar el valor que hace mínimo la SSD el punto de la mejor disparidad.

   Cada hilo se ejecuta sobre una columna, de manera que el cálculo del mapa de disparidad está siendo calculado de forma paralela. La obtención del mapa de disparidad es una de las labores más costosas en cuanto a requerimiento de tiempo.

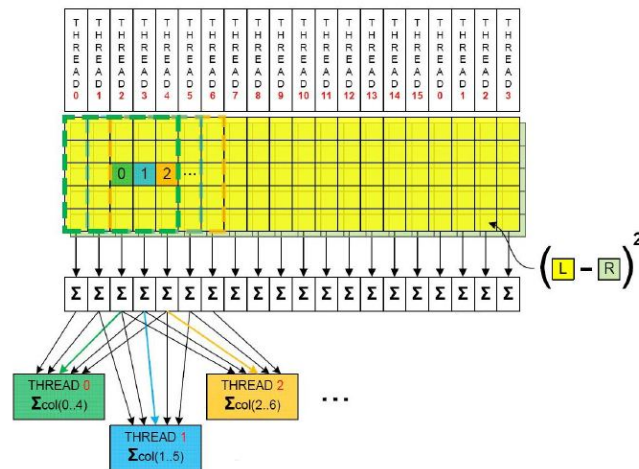


Figura 4.1: Detalle de la Realización del Mapa de Disparidad

Es importante hacer notar que este método calcula tanto el mapa de disparidad derecho como el izquierdo de forma simultánea.

Una vez obtenido el mapa de disparidad, es factible conseguir los mapas u-v *Disparity*, para los cuales, como ya se explicó en la sección 3.2.3. Al igual que el mapa de disparidad, se lanza un hilo por cada fila o cada columna (como proceda para la imagen deseada-u-*Disparity* por cada fila, v-*Disparity* por cada columna), y que son los que llevan a cabo el histograma, que a última instancia son los mapas buscados.

4.2. Obtención y Uso del Perfil de la Calzada

A partir del v-*Disparity* se puede obtener el conocido como "perfil de la calzada" (*road profile*), que es una línea recta oblicua que ya ha sido mostrada en la Fig.3.12. Esto es debido a que al ser el v-*Disparity* la representación del histograma sobre los valores de disparidad de cada fila del mapa de disparidad, la carretera al irse alejando desde la perspectiva del sistema óptico, hace, que por ende, el mapa de disparidad sufra variaciones también. Esta variación será de forma lineal, y es la que da como resultado la ya mencionada recta en estudio [48]. Dicha aproximación, no obstante, no es válida en caso de que la calzada no sea plana [49].

Pese a que el v-*Disparity* tiene como ventaja el poder ser capaz de detectar tanto obstáculos como la calzada, a lo largo de todo el algoritmo solo interesa la información de la calzada sobre la que circula el vehículo, por ser de sumo interés el saber el movimiento que sigue el vehículo sobre ella. La solución más óptima es la de obviar el mapa denso de disparidad y trabajar en su lugar con el mapa libre directamente (3.2.4).

El perfil de la calzada puede idealmente representarse según la ecuación 4.1

$$v = m \cdot d + h \tag{4.1}$$

Donde v es la coordenada vertical de la imagen de los puntos de la calzada, d el valor de la disparidad en ese punto, m representa la pendiente del sistema y su ordenada en el origen h, es el valor teórico del horizonte del sistema estéreo.

Sin tener en cuenta la autocalibración de los parámetros extrínsecos, la profundidad del punto P de coordenadas en el mundo $(X, Y, Z, 1)$, denominada Z se puede calcular mediante la fórmula 4.2, a partir de la disparidad y las propiedades del sistema óptico directamente, y sabiendo que la proyección de dicho punto P en el plano de la imagen viene dado por (u_L, v_L) para la imagen izquierda y (u_R, v_R) para la derecha.

$$Z = \frac{f \cdot b}{u_L - u_R} = \frac{f \cdot b}{d} \tag{4.2}$$

Siendo d la disparidad, f la distancia focal y b la *baseline*.

Para la estimación del primero de los parámetros extrínsecos de la cámara, el *pitch* θ (el ángulo que forma el sistema con respecto a la calzada), es necesario conocer la coordenada vertical del centro óptico ($v_{\Delta 0}$), de tal manera, que se obtiene según la ecuación 4.3

$$\theta = \arctan \frac{v - v_{\Delta 0}}{f} \tag{4.3}$$

Mediante las ecuaciones 4.1 y 4.2, es posible representar la relación de profundidad entre los puntos pertenecientes a la calzada (Z) y su coordenada vertical en la imagen, corregidas además con la información del *pitch*, para que la información sea dada respecto al sistema de coordenadas del vehículo y no desde el de la cámara.

$$Z = \frac{m \cdot f \cdot b}{v - h} \cdot \cos\theta \tag{4.4}$$

La coordenada X de un punto de la calzada puede obtenerse a partir de sus coordenadas en el sistema de la imagen, denotado por (u, v), mediante 4.5

$$X = \frac{Z \cdot (u - C_u)}{f} = \frac{m \cdot b \cdot (u - u_{\Delta 0})}{v - h} \tag{4.5}$$

Donde $u_{\Delta 0}$ hace referencia a la coordenada horizontal del centro óptico.

Para obtener una mejor estimación de las coordenadas $[XZ]^T$, (se verá que $Y = 0$ en adelante), se consideran la base $[X'Z']^T$, y son obtenidas por tener en cuenta la desviación del *yaw* ϕ, de acorde a 4.6.

$$\begin{bmatrix} X' \\ Z' \end{bmatrix} = \begin{bmatrix} \cos\phi & -\operatorname{sen}\phi \\ \operatorname{sen}\phi & \cos\phi \end{bmatrix} \begin{bmatrix} X \\ Z \end{bmatrix} \tag{4.6}$$

En el capítulo siguiente estas fórmulas serán optimizadas gracias a la incorporación de la calibración de los parámetros extrínsecos, y en el capítulo de *Resultados* podrá apreciarse los pingües beneficios que dicha calibración aporta a la robustez del sistema en su conjunto.

Los pasos que a continuación se explican son invariantes a la fórmula en uso, y es por ello que se describirán con detalle.

4.2.1. Implementación Del Algoritmo

Para poder obtener el perfil de la calzada, es imprescindible trabajar con el v-*Disparity* del mapa libre de disparidad, como ya ha sido justificado, lo cual da como resultado una imagen como la representada en la Fig.4.2

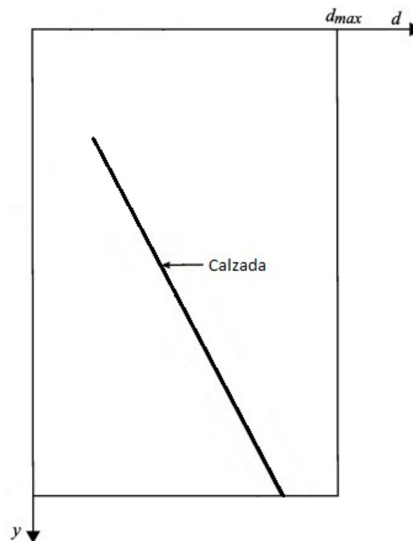


Figura 4.2: Perfil de la Calzada

El v-*Disparity* no necesita ningún tratamiento previo tal como una umbralización o ecualización, pues por su propia definición ya representa 255 valores distintos de grises diferentes. En esta imagen, pese a que existen valores muy dispares, los puntos de la calzada siguen una cierta tendencia que los del entorno no siguen, lo que se traduce en que la extracción del perfil de la calzada se resume en buscar y almacenar aquellos puntos que superen un cierto umbral, recorriendo la imagen por filas y columnas.

120	56	28	38	141	164	76
163	205	218	89	229	54	237
53	52	225	240	38	74	58
92	81	203	111	65	250	244
123	17	87	89	46	34	99
99	78	136	227	156	131	133
199	62	84	192	231	207	84
249	130	127	167	139	3	110
1	20	63	208	109	241	167
150	202	19	91	97	11	135
211	117	14	228	32	165	106
42	145	32	207	234	183	89
202	227	4	4	194	94	135
0	116	34	5	7	229	117
213	235	15	237	143	44	192
107	231	239	70	86	59	35
92	247	87	210	183	80	4
240	206	126	98	207	97	127
144	188	227	137	225	121	86
156	189	134	189	251	135	86

0	0	0	0	255	255	0
255	255	255	0	255	0	255
0	0	255	255	0	0	0
0	0	255	0	0	255	255
0	0	0	0	0	0	0
0	0	255	255	255	255	255
255	0	0	255	255	255	0
255	255	0	255	255	0	0
0	0	0	255	0	255	255
255	255	0	0	0	0	255
255	0	0	255	0	255	0
0	255	0	255	255	255	0
255	255	0	0	255	0	255
0	0	0	0	0	255	0
255	255	0	255	255	0	255
0	255	255	0	0	0	0
0	255	0	255	255	0	0
255	255	0	0	255	0	0
255	255	255	255	255	0	0
255	255	255	255	255	255	0

Figura 4.3: Extracción del Perfil de la Calzada a partir del v-*Disparity*

Los puntos que superan dicho valor umbral son almacenados en una matriz especialmente diseñada para ello . Sin embargo, los puntos que se obtienen no forman una recta estructurada, si no una nube de puntos que precisa el uso del procedimiento RANSAC, el cual ajustará dicha nube de puntos a la recta que mejor reúna a los puntos. A dicha recta resultante, es a la que se la denomina el perfil de la calzada. En la Fig.4.3 se representa un ejemplo de cómo se efectúa este paso, estando remarcados los valores que superan el valor umbral (128 en el ejemplo), que se sitúan a nivel alto (255), y que representan la nube de puntos ya mencionada.

En este paso, como ya se ha indicado a la hora de explicar la filosofía RANSAC, es necesario generar dos puntos aleatorios, y encontrar un modelo de

recta que ajuste esos dos puntos, y calcular el número de *inliers*. Este proceso es realizado un número repetitivo de veces, y tras la finalización de todas las iteraciones se escoge aquel modelo de recta que mejor haya encuadrado los puntos. Este es el primer uso de la filosofía multihilo en el algoritmo, de tal manera que se lanzan el número de hilos adecuado para que se ejecuten en paralelo, y almacenando en una memoria compartida por todos los hilos los modelos obtenidos y el número de *inliers*, tal y como se muestra en la Fig.4.4, para que se pueda elegir en el menor tiempo posible la mejor estimación del perfil de la calzada.

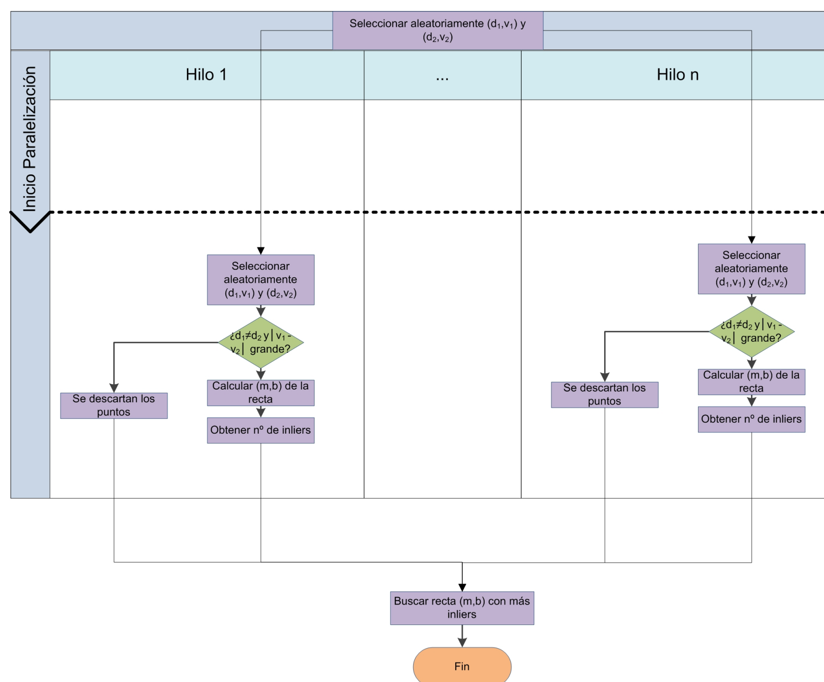


Figura 4.4: Diagrama de Flujo del Cálculo del Perfil de la Calzada

Se ha generado un *Kernel* que es el que todos los hilos siguen para la ejecución de estas operaciones, de forma paralela. La búsqueda de los puntos a nivel alto (se denomina así a los puntos que tienen nivel de gris por encima del valor umbral) se efectúa en la CPU, al igual que la búsqueda de la mejor recta posible después de que hayan accedido a la memoria compartida todos los hilos. La matriz de los valores se encuentra en la memoria global del dispositivo.

Un caso posible es que al recorrer la matriz, los puntos que se obtienen son nulos, o escasos, en ese caso, y tal como puede verse en Fig.4.4 y 3, no se procede a calcular los parámetros. La razón es que la recta que sale como

solución no tiene validez alguna para el fin buscado, pues al haberse obtenido con pocos puntos, los cambios de pendiente son extraordinariamente notables.

Otro posible caso es aquel en el que los dos puntos de la matriz resultante de puntos por encima del umbral sean el mismo, en cuyo supuesto, la pendiente queda una indeterminación del tipo $\frac{0}{0}$, y esa es la razón por la cual también se hace una comprobación de este supuesto antes de empezar las operaciones.

La última comprobación que hay que llevar a cabo, es aquel supuesto en el que las coordenadas y_1, y_2 son semejantes, lo cual incurre en que la diferencia de ambas es cercana a 0, y las recta resultante sería horizontal, caso que evidenciaría la presencia de errores en el v-*Disparity*.

Si el paso 14 diera lugar, se le asigna un valor negativo (-1) al número de *inliers*, para así asegurar que bajo ningún concepto se usarán esos datos.

Por último, indicar que los valores usados en el algoritmo 3 han sido obtenidos de forma empírica, y son fácilmente modificables, al estar definidos como macro del preprocesador en un archivo cabecera.

4.3. Detección de Puntos Característicos de la Calzada

Para la detección de puntos característicos se hace uso del detector y descriptor SURF, haciendo uso de la implementación que viene en la biblioteca correspondiente en OPENCV.

Para emparejar los puntos se hace uso del algoritmo FLANN.

Debido a que los únicos puntos que importan en este caso son los de la calzada, se tiene en consideración solo el tercio inferior de la imagen para la extracción de los puntos característicos. Esto es así dado que de forma experimental se puede asegurar que la calzada se encuentra en dicha región de la imagen. Las ventajas a cambio es una aceleración del tiempo de cálculo de la detección de puntos característicos.

Algoritmo 3 Obtención de los Parámetros del Perfil de la Calzada

Parámetro(s): Número de Hilos Lanzados (i), Valor Umbral Para Asegurar el Proceso ($Umbral$), Rango de *Inliers* ($Rango$)

1: **para** un número i de hilos ejecutándose en paralelo **hacer:**
2: Generar dos números aleatorios distintos para cada hilo (haciendo uso de la librería cuRAND.
3: Acceder a las coordenadas (x_1, y_1) y (x_2, y_2) correspondientes a las posiciones en la matriz de valores a nivel alto de dichos números aleatorios.
4: **si** los puntos no son el mismo ($x_1 \neq x_2$) y $|y_1 - y_2| > Umbral$ **entonces:**
5: Asignar la pendiente de forma que para cada hilo m_i corresponde a $\frac{y_1 - y_2}{x_1 - x_2}$
6: Asignar la ordenada en el origen que para cada hilo b_i corresponde a $y_1 - m \cdot x_1$
7: **para** todos los puntos a nivel alto en el v-*Disparity*, $(x, y)_i$ **hacer:**
8: **si** están a una distancia por debajo del rango ($|y_i - (m \cdot x_i + b)| < Rango$) **entonces:**
9: Incrementar en 1 el número de *inliers* correspondiente al par (m_i, b_i).
10: **fin si**
11: Cada hilo guarda los parámetros m_i, b_i y número de *inliers*.
12: **fin para**
13: **si no**
14: Abortar el proceso para ese hilo en particular, indicando que no existe recta y, por ende, no existen inliers.
15: **fin si**
16: **fin para**
17: Buscar posición n correspondiente a la recta con mayor número de *inliers*, de entre todos los registros obtenidos por los hilos.
18: **devolver** (m_n, b_n)

4.3.1. Implementación Del Algoritmo

Debido a que este proceso es el más costoso en cuanto a tiempo, la detección de puntos se lleva a cabo en paralelo con todo el proceso, en otro proceso independientemente. Para evitar problemas de comunicación por la memoria compartida entre los dos procesos (principal y detección de puntos), se han implementado varios semáforos.

Solo es necesaria la imagen izquierda para este paso, pues es posible calcular la localización en el mundo $[X, Z]^T$ solo con las coordenadas de la imagen izquierda, que viene dada en coordenadas (u, v).

El primer paso es seleccionar la ROI (*Region Of Interest*), que será el tercio inferior de la imagen.

Para poder hacer el posterior proceso de *matching* de puntos característicos, es necesario almacenar en cada momento no solo los puntos del *frame* actual, si no también los del *frame* anterior, para lo cual habrá de hacerse un llamamiento a la instrucción `CvCloneSeq`, seguida de `cvClearMemStorage`.

Para la detección y extracción de los puntos característicos se hace uso de la función `cvExtractSURF`.

Esa información ya es suficiente para poder efectuar la llamada al proceso FLANN, responsable de encontrar las correspondencias entre los puntos característicos de los dos *frames* consecutivos.

La visión global del funcionamiento entre los dos procesos (el principal y el detector de puntos) puede observarse en la Fig.4.5, donde se puede apreciar que a partir solo de la imagen estéreo, se pueden bifurcar los dos procesos, que de tal manera consiguen reducir el tiempo de cómputo.

Una de las ventajas de esta forma de proceder radica en que las operaciones designadas dentro de *Programa Principal* hacen un uso intensivo de la GPU, mientras que para el proceso de detección de puntos, la CPU es la más indicada para llevarla a cabo. Resulta de esta manera que se optimizan los recursos al hacer uso de ambas unidades de procesamiento en el mismo tiempo.

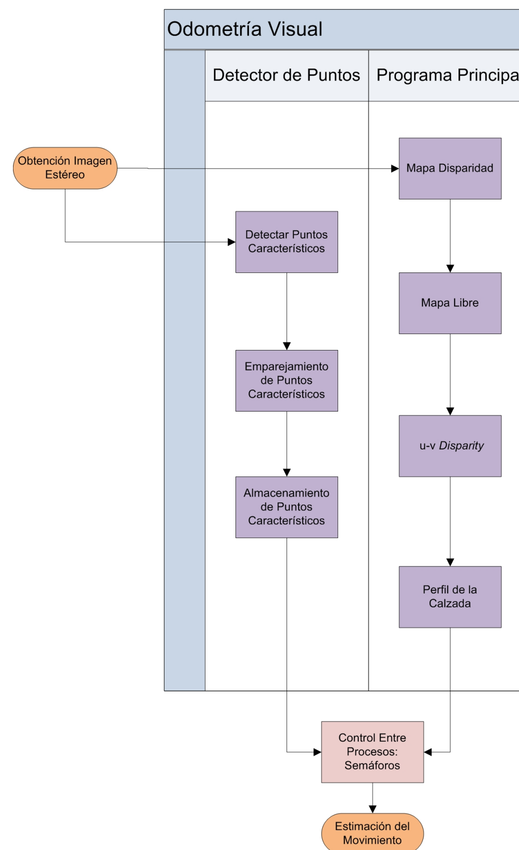


Figura 4.5: Interacción entre Procesos

Resulta interesante mencionar el momento de unión de ambos en aras de proseguir con el correcto funcionamiento del algoritmo, y que ya ha sido mencionado, se hace mediante el uso de semáforos, al estar los datos en memorias compartidas del archivo de paginación de *Windows*.

Un semáforo, en el contexto de la programación, hace uso a un sistema de control de acceso a una zona de memoria, mediante el cambio del valor de dicho semáforo por cada una de las partes que desean acceder a dichos datos. Tienen valor 0 o 1, y este valor se cambia al acceder un sistema al semáforo. Normalmente el valor unitario indica que se puede acceder a él, siendo necesario que en cuanto un proceso acceda a él, modifique ese valor a 0, hasta que termine las operaciones correspondientes, cuando vuelve a incrementarlo a 1 para permitir el acceso a otro proceso (o a él mismo en un instante de tiempo posterior). Los semáforos por tanto permiten que la lectura y escritura en la memoria a la que vigilan no sufra accesos que conlleven una

indeterminación en los datos (accesos de escritura y/o lectura en la misma zona de memoria en el mismo instante t).

Un esquema del funcionamiento de estos semáforos puede apreciarse en la Fig.4.6.

Como se representa en 4.6, existen cuatro semáforos, a saber:

- 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

De nuevo, de acuerdo a la Fig.4.6, los valores iniciales de los semáforos son:

- Semáforo 1: 1 $\longrightarrow$ Dispuesto a cargar una imagen en cualquier momento.
- Semáforo 2: 0 $\longrightarrow$ Bloquea el acceso del detector de puntos a la imagen hasta que no esté cargada.
- Semáforo 3: 1 $\longrightarrow$ Dispuesto a dejar escribir los vectores.
- Semáforo 4: 0 $\longrightarrow$ Bloquea el acceso a los vectores hasta que no estén escritos correctamente.

Existen tres memorias compartidas, una para la imagen estéreo, y otros dos para los dos vectores que almacenarán los puntos característicos en el *frame* inmediatamente anterior y el actual.

El tamaño asignado a cada una de las memorias es de 2048 pares de puntos, valor que es sobradamente superior al de puntos que han podido llegar a ejecutarse durante las pruebas realizadas con el algoritmo.

Se ha implementado adicionalmente, la condición de que los puntos característicos que se emparejen pertenezcan realmente a la calzada, para lo

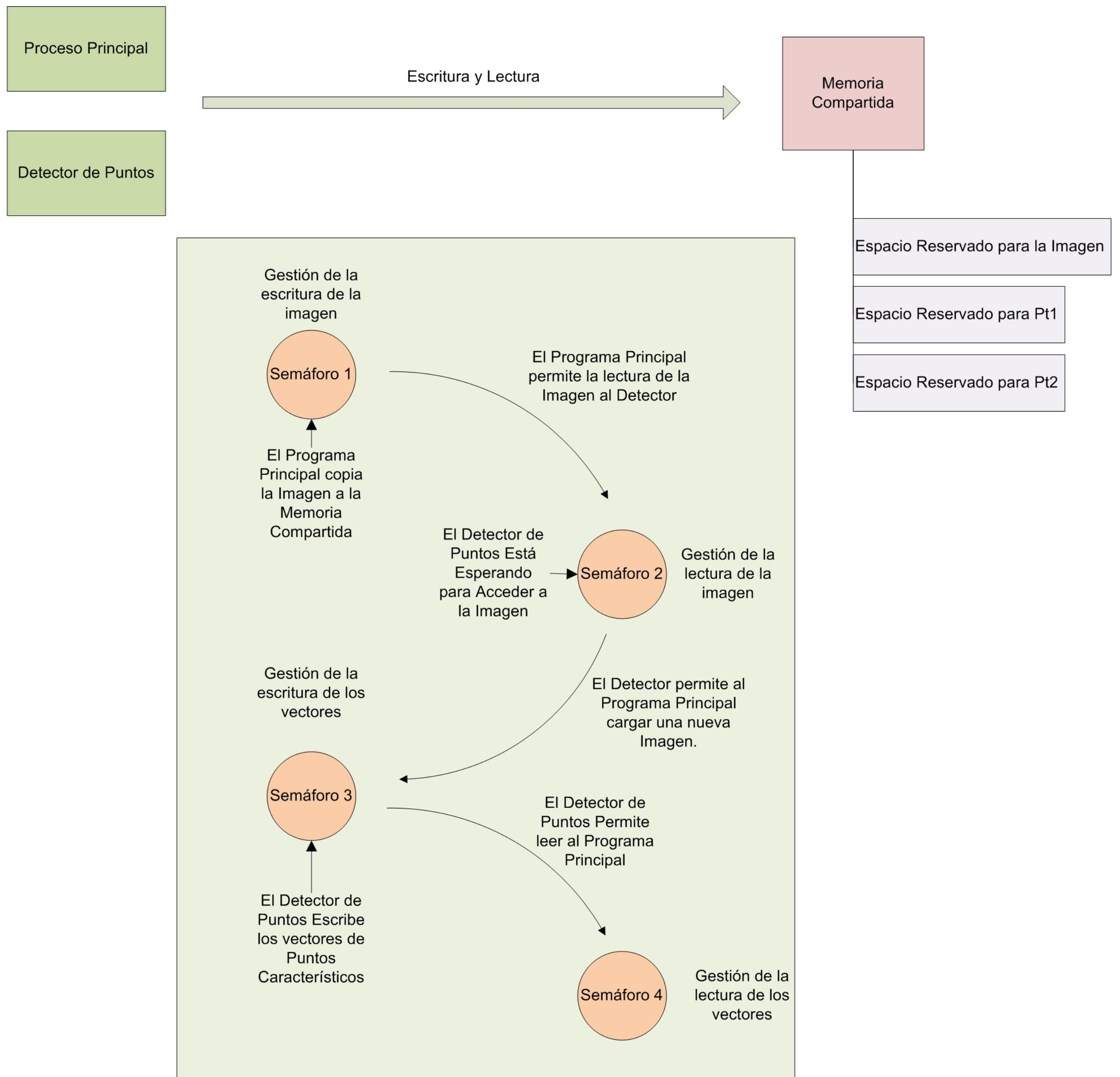


Figura 4.6: Funcionamiento del Acceso a la Memoria Compartida

cual se comprueba que están dentro del mapa libre. Es una aportación de gran relevancia, pues se evita considerar puntos que no aportan información de utilidad, y se consigue que el algoritmo sea más robusto. Esta es una mejora con respecto al algoritmo original que ha sido introducida en este proyecto, y cuya utilidad se analizará en el apartado de resultados.

La idea detrás de esta implementación radica en que en algunas ocasiones, los puntos detectados están en la acera (entornos urbanos), o simplemente fuera de la calzada, y las fórmulas que se manejan son solo válidas para la calzada.

4.4. Cálculo de la Estimación de Movimiento Entre *Frames* Consecutivos

Para este apartado se han supuesto algunas simplificaciones:

1. El movimiento del vehículo entre dos imágenes consecutivas puede usarse como dos etapas de velocidad constante, con una rotación en torno al centro del eje trasero, de ángulo α, y con una traslación hacia delante después de la rotación, descrita de acuerda a su proyección en los dos ejes cartesianos del plano de la calzada, $(\Delta X, \Delta Z)$. Fig.4.7
2. No hay deslizamiento en ninguna dirección.

El fin de este apartado es buscar los puntos en el mundo que corresponden a las coordenadas de los puntos (u, v) que han salido como característicos entre dos *frames* consecutivo, gracias a las fórmulas 4.4, 4.5 y 4.6, y para una mejor estimación, las que se introducirán en el capítulo siguiente.

La desviación de ángulo α puede expresarse según la ecuación 4.7

$$\alpha = \arctan \frac{\Delta X}{\Delta Z} \tag{4.7}$$

y los desplazamientos $\Delta Z, \Delta X$ vienen dados según la ecuación

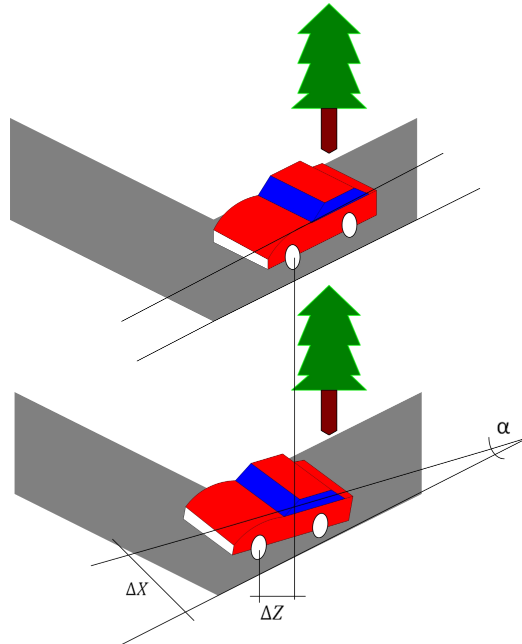


Figura 4.7: Desplazamientos y Rotaciones de un Vehículo Entre 2 *Frames*.

$$\begin{bmatrix} X^t \\ Z^t \end{bmatrix} = \begin{bmatrix} \cos\alpha & \operatorname{sen}\alpha \\ -\operatorname{sen}\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} X^{t+1} \\ Z^{t+1} \end{bmatrix} + \begin{bmatrix} \Delta X \\ \Delta Z \end{bmatrix} \tag{4.8}$$

Que depende del ángulo α, y donde el superíndice $t+1$ hace alusión al *frame* actual y t al anterior.

Si se despejan los incrementos de posición $\Delta X, \Delta Z$ (ecuaciones 4.9 y 4.10), queda resuelto, pero depende del ángulo α:

$$\Delta X = X^t - X^{t+1}\cos\alpha - Z^{t+1}\operatorname{sen}\alpha \tag{4.9}$$

$$\Delta Z = Z^t - Z^{t+1}\cos\alpha + X^{t+1}\operatorname{sen}\alpha \tag{4.10}$$

Debido a que el parámetro α es incógnita, se procede a dividir 4.9 entre 4.10:

$$\tan\alpha = \frac{\operatorname{sen}\alpha}{\cos\alpha} = \frac{\Delta X}{\Delta Z} = \frac{X^t - X^{t+1}\cos\alpha - Z^{t+1}\operatorname{sen}\alpha}{Z^t - Z^{t+1}\cos\alpha + X^{t+1}\operatorname{sen}\alpha} \tag{4.11}$$

Puede obtenerse α a partir de la ecuación de segundo grado 4.12:

$$((X^t)^2 + (Z^t)^2)\operatorname{sen}^2\theta + (2 \cdot X^{t+1} \cdot Z^t)\operatorname{sen}\theta + ((X^{t+1})^2 - (X^t)^2) = 0 \quad (4.12)$$

Que, al ser la única incógnita que impedía la obtención directa de los desplazamientos $\{\Delta Z, \Delta X\}$, hace que los parámetros que dirigen el movimiento, es decir, la terna $\{\alpha, \Delta Z, \Delta X\}$, sean todos ya conocidos. Dado que esto ha sido logrado gracias al conocimiento de la posición de cada punto en un *frame* $(Z^t + 1, X^t + 1)$ y en el anterior, (Z^{t+1}, X^{t+1}), se obtendrán $\{\alpha, \Delta Z, \Delta X\}_k$ soluciones, siendo k el número de pares de puntos que han sido detectados como característicos.

El siguiente paso es estimar el movimiento real del vehículo, teniendo en cuenta todas las ternas $\{\alpha, \Delta Z, \Delta X\}$ obtenidas. Para ello se hace uso de la mediana de los valores, y de nuevo el método RANSAC.

4.4.1. Implementación del Algoritmo del Cálculo de Resultados

Para llevar a cabo la implementación de este paso, se vuelve a hacer uso de la paralelización que aporta la GPU, de acuerdo al algoritmo 4.

Algoritmo 4 Algoritmo para el cálculo de los parámetros del movimiento

Parámetro(s): Desviación del *yaw* 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:    **para** para cada par de puntos encontrados, lanzar un hilo **hacer:**
3:       Calcular coordenadas en el mundo (Z^t, X^t) y (Z^{t+1}, X^{t+1}) (4.4, 4.5).
4:       Corregir coordenadas en el mundo con la desviación del ϕ (4.6).
5:       Obtener α (4.12) y almacenarla.
6:       Obtener ΔZ y ΔX (4.9 y 4.10).
7:       Obtener X_T, Y_T (4.14, 4.15) y almacenarlas.
8:    **fin para**
9: **fin para**

Expresar los resultados del movimiento del vehículo en término de desplazamientos $(\Delta Z, \Delta X)$ no es óptimo, por lo que se expresarán en función de (Y_T, X_T), que son coordenadas referidas al punto inicial del movimiento, y que guardan la relación con respecto a los desplazamientos según las ecuaciones 4.14 y 4.15.

$$\begin{bmatrix} X_T \\ Y_T \end{bmatrix} = \begin{bmatrix} \cos\theta & \operatorname{sen}\theta \\ -\operatorname{sen}\theta & \cos\theta \end{bmatrix} \begin{bmatrix} \Delta X \\ \Delta Z \end{bmatrix} \tag{4.13}$$

$$X_T = \Delta X \cos\theta + \Delta Z \operatorname{sen}\theta \tag{4.14}$$

$$Y_T = \Delta Z \cos\theta - \Delta X \operatorname{sen}\theta \tag{4.15}$$

4.4.2. Implementación del Algoritmo de Búsqueda de una Solución Única

Para poder efectuar la estimación del movimiento del vehículo se procede a encontrar cuál de entre todas las ternas (α, Y_T, X_T) correspondientes a cada uno de los pares de puntos de la calzada, es la que mejor representa el avance del sistema completo. Para ello, se efectúa la mediana y RANSAC.

El primer punto importante es el de remarcar que se han considerado a los parámetros por separado, no como un conjunto asociado a un punto, para poder obtener mejores resultados.

5

AUTOCALIBRACIÓN

La autocalibración de la cámara ya ha sido introducida en el presente escrito [1.2.4] , pero es en este apartado en el que se tratará ampliamente sobre este aspecto, que constituye el objeto de este trabajo.

La autocalibración pretende actualizar continuamente la ya mencionada *pose* del sistema estéreo para reducir los errores que ocurren en cuanto a precisión de la solución obtenida en aquellos casos en los que no se tiene esta actualización de los parámetros, debido a los cambios de la *pose* de la cámara entre dos *frames* consecutivos.

Esta falta de precisión es debida a que la calibración se lleva a cabo al comienzo del proceso, pero por las causas ya mencionadas que hacen que los parámetros sufran cambios durante su funcionamiento, induciendo de forma directa errores en todos los pasos posteriores.

Dentro del programa descrito que efectúa la odometria visual, este apartado tiene cabida una vez calculado el perfil de la calzada, tal como se muestra en la Fig.5.1:

Para determinar los parámetros extrínsecos de la cámara, existen tres aproximaciones distintas:

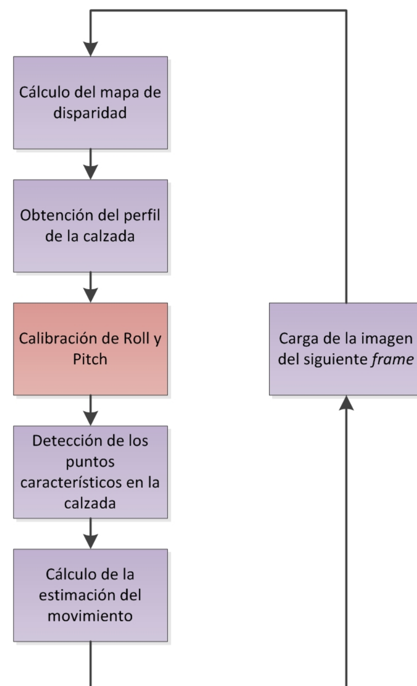


Figura 5.1: Diagrama de Flujo de la Odometría Visual Autocalibrada

1. Usar un patrón de calibración, que puede estar situado en el suelo, o en el capó del vehículo [50].

2. Usar las marcas viales. No es recomendable, pues esto hace que la dependencia de las marcas viales sea necesaria, y no están siempre disponibles [51] [50].

3. Hacer una estimación geométrica del suelo en frente del vehículo [26] [52]. Opción más recomendable pues no se hace necesario la depedencia de factores externos al sistema.

   Esta aproximación a la autocalibración de la *pose* del sistema consiste de tres pasos. En el primer paso se intenta ver el movimiento del vehículo sobre la carretera, y con ello se es capaz de calcular el parámetro *yaw*. En los últimos pasos, se toma el suelo (plano) como referencia, y es en los que se puede determinar el *pitch* y *roll* [52].

   Para el primer paso, es necesario tomar referencias del entorno cuando el vehículo se está moviendo con respecto a la carretera. Deben ser referencias que estén fijas y que sean detectables para el sistema. Se suelen usar líneas de la propia carretera, en caso de que se esté llevando a cabo la calibración en un entorno con carreteras señalizadas como

sistema de referencia, pese a que puede ocurrir el caso de que estas señales estén poco visibles bien por un pobre mantenimiento de la vía, que las haga indetectables, o bien porque haya obstáculos en la vía que no permitan a la cámara captarlos correctamente. En realidad, se indica que se debe tomar las líneas de la señalización de carreteras como marca característica, pero vale cualquier característica que sea particular, reconocible e inconfundible para el fin que se quiere obtener [17].

A la hora de llevar a la práctica este último paso, es importante no llevar al vehículo demasiado rápido. Lo ideal es asegurarse que la velocidad de la toma de imágenes asegure que se puedan tomar como mínimo cuatro imágenes por cada metro que avance el vehículo.

Para determinar los otros dos parámetros extrínsecos del sistema estéreo, se toman como referencia tres puntos del suela cuya correspondencia en el plano del suelo es conocida (gracias a las ecuaciones 3.3, 3.4, 3.5). Conociendo sobre la cámara la distribución de esos puntos, es posible encontrar la ecuación del plano que representa a esos tres puntos.

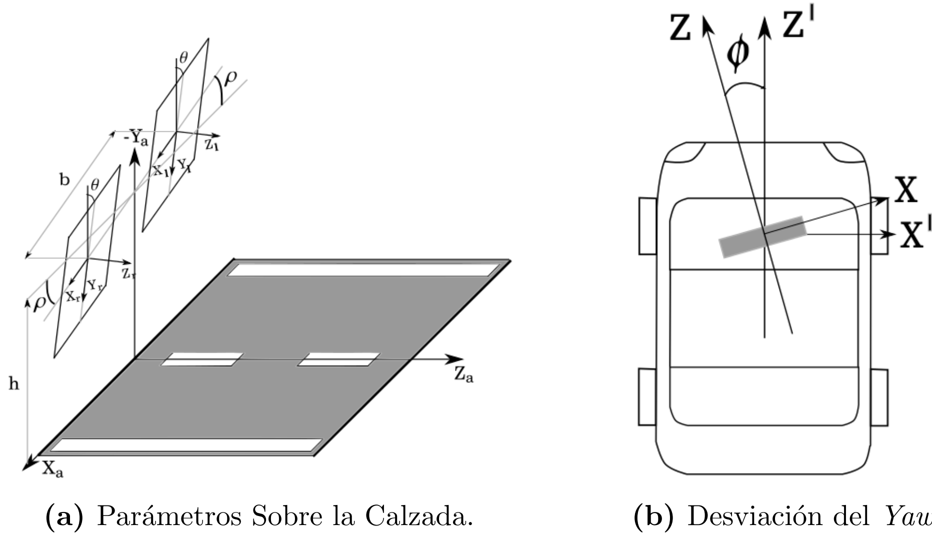
Sin embargo, dado que el suelo, tomado como referencia, no se puede considerar tampoco como ideal, es conveniente no usar solo tres puntos, sino más, para asegurarse de que la autocalibración es correcta. La razón por la que es necesaria por lo menos una terna de puntos es porque es el mínimo conjunto de puntos con los que se determina inmediatamente un plano.

Entre cada imagen consecutiva se detectan los puntos característicos de las referencias que se han tomado inicialmente, para lo cual se hace uso de detectores de esquinas de Harris. Reforzando la idea expuesta en el párrafo anterior, cuanto mayor sea la distancia que haya pasado entre dos imágenes consecutivas, mayor es el esfuerzo computacional requerido para hacer el encuentro de los puntos característicos entre las imágenes consecutivas. Un algoritmo de RANSAC vuelve a ser necesario para este paso.

Con estos procedimientos es posible calcular los ángulos entre el sistema de coordenadas global (el suelo) y el sistema de la cámara que se esté usando, y con ello poder obtener los parámetros necesarios para llevar a cabo el resto de pasos para poder implementar la solución de odometría visual propuesta.

En este texto se presentará una solución para la autocalibración basada en la tercera opción.

En las Fig.5.2a y Fig.5.2b se muestran la posición de los parámetros extrínsecos objeto de la calibración sobre la calzada, y sobre el vehículo, la desviación del *yaw*.



(a) Parámetros Sobre la Calzada. **(b)** Desviación del *Yaw*

Figura 5.2: Parámetros Extrínsecos

5.1. Calibración del *yaw*

Cuando el vehículo está efectuando un movimiento rectilíneo, se determina el punto de fuga entre dos *frames* consecutivos. En el caso de que no exista una desviación del *yaw*, se obtendrá que la coordenada horizontal del punto de fuga (u_{vp}) es igual que la coordenada horizontal del centro óptico del sistema de cámaras (u_0).

En caso de que no se cumpla, la desviación del *yaw* puede ser calculada de acuerdo a la ecuación 5.1.

$$\phi = \arctan \frac{u_{vp} - u_0}{f} \tag{5.1}$$

Donde se denomina con f a la distancia focal.

Se busca el punto de fuga en la línea de horizonte entre dos *frames* consecutivos. A las coordenadas de los puntos característicos n-ésimos en la imagen izquierda se les designa con (u_{n1}, v_{n1}). y en el *frame* siguiente, con (u_{n2}, v_{n2}).

Se puede construir una línea recta r_n 5.2 para cada punto característico encontrado, sabiendo que el punto de fuga corresponde a la intersección de cada línea recta con el resto de líneas que se obtienen de buscar puntos característicos entre dos *frames* consecutivos.

$$v = \frac{v_{n2} - v_{n1}}{u_{n2} - u_{n1}} \cdot u + v_{n2}(u_{n2} + u_{n1}) \tag{5.2}$$

Debido a que este proceso de calibración se lleva a cabo en entornos no controlados, es frecuente que existan errores en los valores obtenidos, haciendo por tanto que no todos los puntos que idealmente deberían juntarse en un único punto, lo hagan en distintos puntos. Es por tanto necesario repetir el proceso para varios *frames* distintos para posteriormente aplicar RANSAC a los valores obtenidos y con ello sí ser capaz de obtener un valor del *yaw* aceptable.

5.2. Calibración de la altura, *pitch* y *roll*

Para poder obtener los valores descritos el primer paso es relacionar el mundo exterior con el del sistema estéreo.

Para ello, supóngase un punto $P = (X, Y, Z, 1)$ perteneciente al mundo y para el cual, su proyección sobre los planos de las cámaras es $(u_i \cdot S, v \cdot S, S, 1)$, donde $i = r$ simboliza la cámara derecha y $i = l$ es para la cámara izquierda. Así, se pueden relacionar de la siguiente manera 5.3.

$$\begin{bmatrix} u_i \cdot S \\ v \cdot S \\ S \\ 1 \end{bmatrix} = M_{proj}(f, u_0, v_0) \cdot M_{Translx}(-\varepsilon_i \cdot \frac{b}{2}) \cdot M_{Rotx}(\theta) \cdot M_{Rotz}(\rho) \cdot M_{Transly}(-h) \cdot \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{5.3}$$

Donde las matrices son las que se pueden ver en 5.4, 5.5, 5.6 y 5.7.

ε tiene valor 1 cuando se trata de la cámara derecha y -1 cuando es sobre la cámara izquierda.

$$M_{proj}(f,u_0,v_0) = \begin{bmatrix} f & 0 & u_0 & 0 \\ 0 & f & v_0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot M_{Translx}(-\varepsilon_i \cdot \frac{b}{2}) = \begin{bmatrix} 1 & 0 & 0 & -\varepsilon_i \cdot \frac{b}{2} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.4}$$

$$M_{Rotx}(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\operatorname{sen}(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.5}$$

$$M_{Rotz}(\rho) = \begin{bmatrix} \cos(\rho) & -\operatorname{sen}(\rho) & 0 & 0 \\ \operatorname{sen}(\rho) & \cos(\rho) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.6}$$

$$M_{Transly}(-h) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -h \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.7}$$

Así pues, la ecuación 5.3 puede simplificarse como 5.8

$$\begin{bmatrix} u_i \cdot S \\ v \cdot S \\ S \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & u_0 & 0 \\ 0 & f & v_0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\rho & -\operatorname{sen}\rho & 0 & h\operatorname{sen}\rho - \frac{\varepsilon_i b}{2} \\ \cos\theta\operatorname{sen}\rho & \cos\theta\cos\rho & -\operatorname{sen}\theta & -h\cos\theta\cos\rho \\ \operatorname{sen}\theta\operatorname{sen}\rho & \cos\rho\operatorname{sen}\theta & \cos\theta & -h\cos\theta\operatorname{sen}\rho \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{5.8}$$

La ya mencionada disparidad (en 3.1.1), se introducirá ahora matemáticamente denominándola Δ, y que se puede calcular según la ecuación 5.9. Representa la profundidad de cada punto del mundo.

$$\Delta = \frac{u_l \cdot S - u_r \cdot S}{S} = \frac{f \cdot b}{Z \cos\theta + (Y - h) \cdot \cos\rho \operatorname{sen}\theta + X \operatorname{sen}\rho \operatorname{sen}\theta} \tag{5.9}$$

Donde b simboliza la distancia denominada *baseline*, ya comentada en este texto.

De la ecuación 5.8 la coordenada que más interesa es la Y, dado que para saber si el punto está en el suelo, dicha coordenada debe ser 0, y resulta imprescindible expresar Y como función de los demás parámetros del sistema $\{f, b, u_0, v_0, h, \rho, \theta\}$, y de la disparidad Δ. Es por ello, que se procederá a despejar Y de las ecuaciones anteriores.

De 5.9

$$\Delta = \frac{u_l \cdot S - u_r \cdot S}{S} \Longrightarrow S = \frac{f \cdot b}{\Delta} \tag{5.10}$$

De 5.3, despejando el vector $[X\,Y\,Z\,T\,1]^T$, y suponiendo $\varepsilon_l = -1$

$$\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = M^{-1}_{Transly}(-h) \cdot M^{-1}_{Rotz}(\rho) \cdot M^{-1}_{Rotx}(\theta) \cdot M^{-1}_{Translx}\frac{b}{2} \cdot M^{-1}_{proj}(f, u_0, v_0) \begin{bmatrix} u \cdot \dfrac{f \cdot b}{\Delta} \\ v \cdot \dfrac{f \cdot b}{\Delta} \\ \dfrac{f \cdot b}{\Delta} \\ 1 \end{bmatrix} \tag{5.11}$$

$$\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\rho & \cos\theta \operatorname{sen}\rho & \operatorname{sen}\theta \operatorname{sen}\rho & -\frac{b\cos\rho}{2} \\ -\operatorname{sen}\rho & \cos\theta\cos\rho & \cos\rho \operatorname{sen}\theta & h + \frac{b \operatorname{sen}\rho}{2} \\ 0 & -\operatorname{sen}\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \frac{1}{f} & 0 & -\frac{u_0}{f} & 0 \\ 0 & \frac{1}{f} & -\frac{v_0}{f} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} u \cdot \frac{f \cdot b}{\Delta} \\ v \cdot \frac{f \cdot b}{\Delta} \\ \frac{f \cdot b}{\Delta} \\ 1 \end{bmatrix} \quad (5.12)$$

Haciendo las operaciones matriciales para despejar el valor de Y, esto resulta

$$Y = \frac{(v - v_0) \cdot b \cdot \cos\rho \cdot \cos\theta}{\Delta} - \frac{(u - u_0) \cdot b \cdot \operatorname{sen}\rho}{\Delta} + \frac{f \cdot b \cdot \cos\rho \cdot \operatorname{sen}\theta}{\Delta} + h + \frac{b \cdot \operatorname{sen}\rho}{2} = 0 \quad (5.13)$$

Se iguala a 0 la ecuación 5.13, pues lo que interesa es que $Y = 0$ como ya ha sido mencionado anteriormente.

El siguiente paso es obtener una ecuación que relaciona las coordenadas de la imagen entre sí

$$(v - v_0) = \frac{\tan\rho}{\cos\theta} \cdot (u - u_0) - \frac{h}{b \cdot \cos\rho \cdot \cos\theta} \cdot \Delta - \frac{\tan\rho}{2 \cdot \cos\theta} \cdot -f \cdot \tan\theta \quad (5.14)$$

La ecuación 5.14 relaciona (u, v) mediante una recta de la forma $v = c \cdot u + d$, donde

$$c = \frac{\tan\rho}{\cos\theta} d = -\frac{h}{b \cdot \cos\rho \cdot \cos\theta} \cdot \Delta - \frac{\tan\rho}{2 \cdot \cos\theta} \cdot -f \cdot \tan\theta \quad (5.15)$$

El valor del *roll* suele ser muy pequeño en entornos automovilísticos, haciendo que normalmente $\cos\rho \approx 1$ y $\operatorname{sen}\rho \approx 0$, lo que permite poder expresar 5.14 de la manera expuesta en 5.16, en la que se ha sustituido los términos por las aproximaciones:

$$v = -\frac{h}{b \cdot \cos\theta} \cdot \Delta + v_0 - f \cdot \tan\theta \tag{5.16}$$

Mediante la ecuación 5.16 se ha logrado relacionar de forma lineal la posición v de la imagen con el valor de la disparidad Δ, que es a su vez, la ecuación que rige al v-disparity. A esta línea se la denomina *road profile*, y se la suele representar como $v = C_r \cdot \Delta + v\Delta 0$, donde C_r es el valor de la pendiente $(-\frac{h}{b \cdot \cos\theta})$ y $v_{\Delta 0}$ es el valor de v para el cual la disparidad $\Delta = 0$.

Esta es toda la información necesaria para poder obtener la altura h (5.17) y el *pitch*, θ (5.18):

$$h = -C_r \cdot b \cdot \cos\theta \tag{5.17}$$

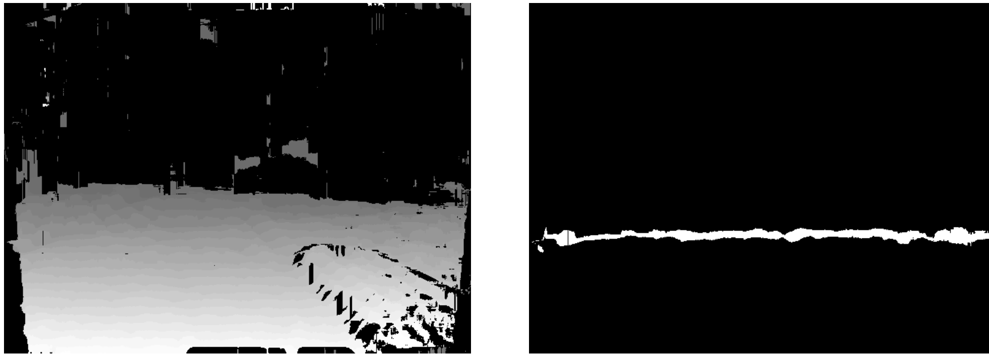
$$\theta = \arctan\frac{v_0 - v_{\Delta 0}}{f} \tag{5.18}$$

El *roll* ρ se obtiene a partir del mapa libre (donde solo se tiene en cuenta la calzada), para lo cual, se fija un valor de Δ, y solo se tiene en cuenta la parte del mapa que representa a la calzada que se sitúa frente al vehículo y que cumple con ese valor de disparidad. Puntos con la misma disparidad se encuentran a la misma distancia de la cámara. Se obtiene una nube de puntos que simbolizan la calzada (Fig.5.3), y que para el presente trabajo ha sido elegido un valor de $\Delta = 15$.

Este valor de disparidad ha sido elegido empíricamente por ser el valor que mejor soluciones aportaba, por estar a una distancia no muy lejana del vehículo, y que permitiese obtener los suficientes puntos como para poder obtener buenos resultados.

Tras aplicar de nuevo RANSAC a esa nube de puntos, se puede obtener una línea recta que sigue la ecuación $v = C_u + d_\Delta$, la cual es necesaria para poder obtener un valor del parámetro, como puede apreciarse en la fórmula 5.19

$$C = \frac{\tan\rho}{\cos\theta} \Longrightarrow \rho = \arctan\left(C \cdot \cos\theta\right) \tag{5.19}$$



(a) Mapa Libre Obtenido Para un Fotograma

(b) Mapa Libre Binarizado con Disparidad 15.

(c) Valores a Disparidad 15 del Mapa Libre

Figura 5.3: Mapa Libre

5.3. Obtención de las Coordenadas del Mundo

Una vez que se han obtenido todos los parámetros extrínsecos e intrínsecos $\{f, b, u_0, v_0, h, \rho, \theta\}$, ya es posible recuperar la ecuación 5.11, y despejar X y Z.

Despejando X de 5.12:

$$X = \frac{b \cdot \cos\rho}{\Delta} \cdot (u - u_0) + \frac{b \cdot \cos\theta \cdot \operatorname{sen}\rho}{\Delta} \cdot (v - v_0) + \frac{f \cdot b \cdot \operatorname{sen}\theta \cdot \operatorname{sen}\rho}{\Delta} \quad (5.20)$$

$$Z = \frac{b \cdot \operatorname{sen}\theta}{\Delta} \cdot (v - v_0) + \frac{f \cdot b \cdot \cos\theta}{\Delta} \tag{5.21}$$

Para tener en cuenta la desviación del *yaw*, es imprescindible pasar a una nueva base de coordenadas $[X'Z']^T$ siguiendo la ecuación 5.22.

$$\begin{bmatrix} X' \\ Z' \end{bmatrix} = \begin{bmatrix} \cos\phi & -\operatorname{sen}\phi \\ \operatorname{sen}\phi & \cos\phi \end{bmatrix} \begin{bmatrix} X \\ Z \end{bmatrix} \tag{5.22}$$

6

RESULTADOS EXPERIMENTALES OBTENIDOS

TODOS los cambios que se han llevado a cabo, desde la versión original del algoritmo de la cual se partió se exponen a continuación, de forma que pueda verse de forma clara y concisa a través de gráficos y tablas.

La versión original, previa a ninguna modificación objeto en este proyecto era una solución para la odometría visual basada en la captación de puntos característicos de la calzada. Dicha solución es estable y robusta, pero presenta la desventaja de que, pese a que obtiene buenos resultados, estos varían considerablemente entre dos iteraciones independientes. La aproximación dada es aceptable, pues es capaz de dar resultados de odometría, pero es en los detalles de precisión donde presenta grandes deficiencias. La motivación de este trabajo es mejorar la estimación de la posición del vehículo sobre la calzada *frame* a *frame*, haciendo que los resultados finales sean lo más precisos y ajustados a la realidad posible.

Para ello, como ya se ha indicado en varios epígrafes del texto se procede a autocalibrar los parámetros extrínsecos del sistema estéreo para mejorar la determinación de su *pose*. Además, se estudiará la utilidad de la mejora

aportada a la estimación de los puntos característicos, en forma de que solo se consideren aquellos puntos que realmente pertenezcan a la calzada.

Para probar los resultados se ha usado una secuencia tomada por el vehículo IVVI [3] y con la cámara *Bumblebee*. Consta de 982 *frames*, y es un recorrido de aproximadamente 500 m, situado en las cercanías del campus de Leganés de la *Universidad Carlos III de Madrid*, tal como puede apreciarse en la Fig.6.1, donde se muestra en tono verde el trazado.

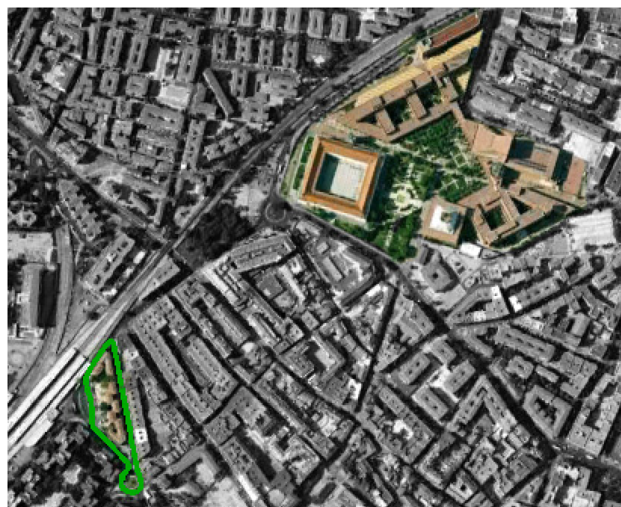


Figura 6.1: Trazado usado para la Evaluación de Resultados. Localización

El equipo usado para la ejecución del algoritmo es el que posee el *Laboratorio Sistemas Inteligentes* de la *Universidad Carlos III de Madrid*, y cuyas características se muestran en la tabla 6.1.

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

Para poder trabajar con la arquitectura CUDA, este equipo está provisto de una tarjeta gráfica *nVidia*, modelo *FX 380 LP*, que presenta los parámetros técnicos mostrados en la Tabla 6.2.

La versión de las librerías de OPENCV son la 2.4.2, las cuales están disponibles desde Julio de 2012. Para el manejo de imágenes distinto del

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

cálculo de los puntos característicos de la calzada, se emplean las bibliotecas MIL [53]. El entorno de desarrollo ha sido *Visual C++ 2008* [54].

La cámara estéreo es, como ya ha sido mencionada, la cámara *Bumblebee2*, que puede verse en la Fig.1.9, y cuyas características técnicas son las enumeradas en la Tabla 6.3 [16].

Frames por Segundo	48 a resolución 640×480 20 a resolución 1024×768
Baseline	12 cm
Modo de Captura	Blanco y negro con posterior rectificación
Distancia Focal	6 mm

Tabla 6.3: Especificaciones Técnicas del Sistema Estéreo Usado Para la Captura de Datos

En la Fig.6.2 puede verse el sistema que va implementado en el vehículo con la propia cámara que está instalada. Se muestran sobre la figura los parámetros que procederán a ser calibrados.

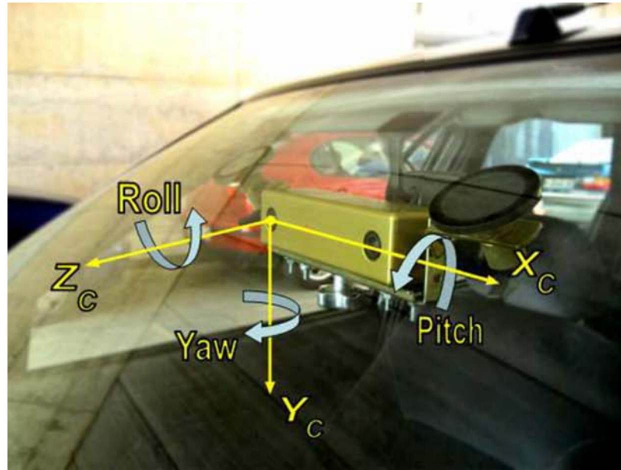
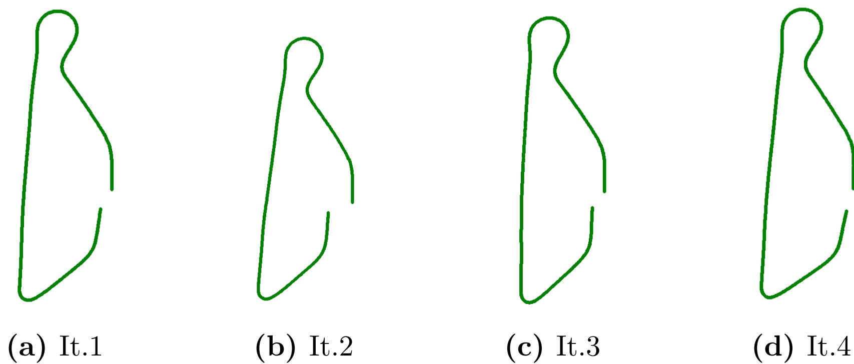


Figura 6.2: Parámetros Intrínsecos de la Cámara Sobre el Sistema A Bordo Del *IvvI* 2.0

6.1. Resultados Previos al Algoritmo Propuesto

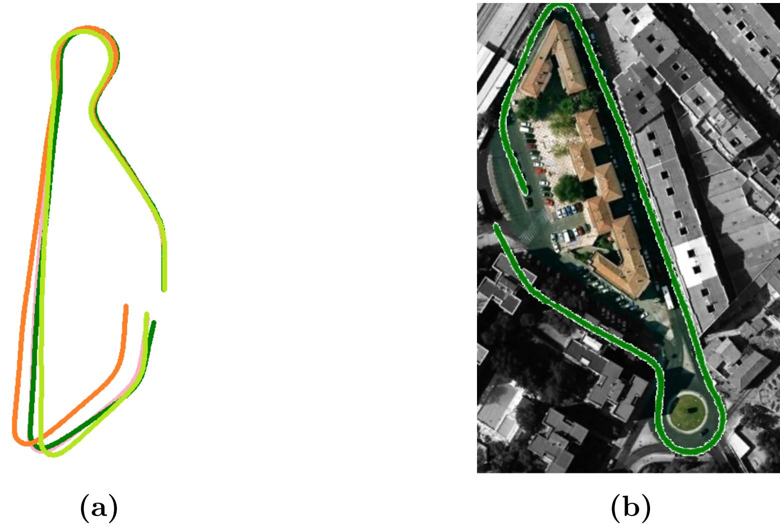
Primero han de resumirse los resultados previos obtenidos, y que pueden verse en Fig.6.3:



(a) It.1 **(b)** It.2 **(c)** It.3 **(d)** It.4

Figura 6.3: Distintos Resultados de Iteraciones del Algoritmo de Partida.

Debido al alto grado de error integral el la estimación de la odometría visual, el error entre la posición real y la estimada se va incrementando, como puede verse en Fig.6.4a:



(a) (b)

Figura 6.4: Resultados del Algoritmo de Partida. (a) Distintas Ejecuciones Sin Autocalibración. Comparativa. (b) Superposición sobre el Mapa de Satélite de un Resultado de Odometria Sin Autocalibración.

A partir de estos resultados se puede entender la necesidad de la autocalibración, pues en los primeros metros, hasta la glorieta, las desviaciones que presentan las trayectorias son debidas a la arbietrariedad de los valores tomados para poder efectuar la odometría (los distintos procesos que incluyen RANSAC predominantemente), pero es después de la glorieta donde las trayectorias siguen destinos diferentes. El resultado final resulta en una falta de precisión del punto final, como puede apreciarse en la Fig.6.4b

El procedimiento de calibración de los parámetros es: primero estimar el *yaw*, después el *pitch*, y por último el *roll*. El orden a seguir en este capítulo será el mostrado en Fig.6.5, en el que primero se expondrán la aportación de la calibración de cada uno de estos parámetros, añadiéndolos uno a uno, y sin tener en cuenta la mejora acerca de la comprobación de que realmente los puntos pertenecen a la calzada. Posteriormente, y a la vista de los resultados obtenidos, se hará un estudio de cómo afecta la comprobación ya mencionada. Por último, se analizará la evolución de los parámetros *pitch* y *roll* a lo largo de la secuencia de fotogramas.

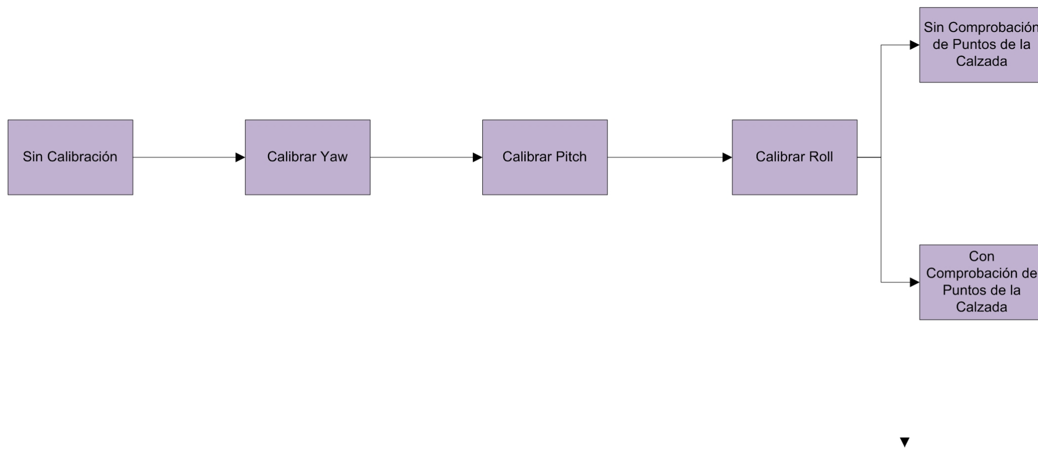
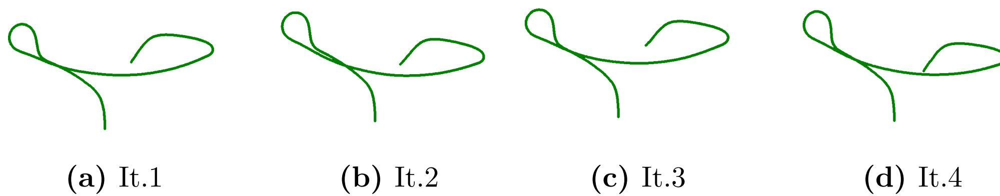


Figura 6.5: Secuencia de Calibración.

6.2. Calibración del *Yaw*

El parámetro que ha de calibrarse primero es el del *yaw*, pues tras la adquisición de datos experimentales, se ha podido constatar que la cámara es muy sensible a cambios en él. Para poder demostrar la necesidad de la calibración, baste ver los resultados obtenidos cuando no se lleva a cabo dicha calibración (Fig.6.6):



(a) It.1 **(b)** It.2 **(c)** It.3 **(d)** It.4

Figura 6.6: Distintos Resultados de Iteraciones Sin Calibración Alguna.

Se demuestra por tanto la importancia del ajuste de este parámetro.

Cabe destacar que para este ensayo, tanto el *pitch* como el *roll* no han sido calibrados tampoco.

La superposición del resultado que se muestra en Fig.6.6d con el mapa de satélite se muestra en la Fig.6.7.

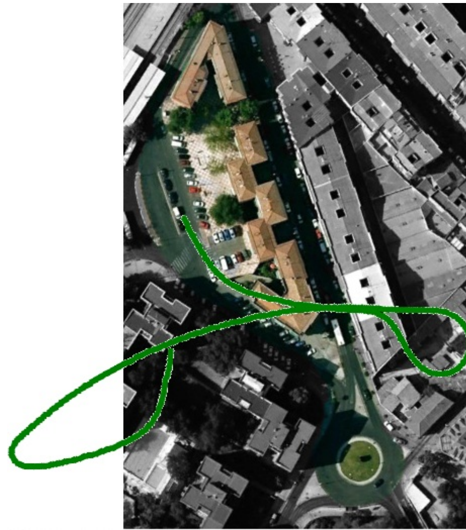
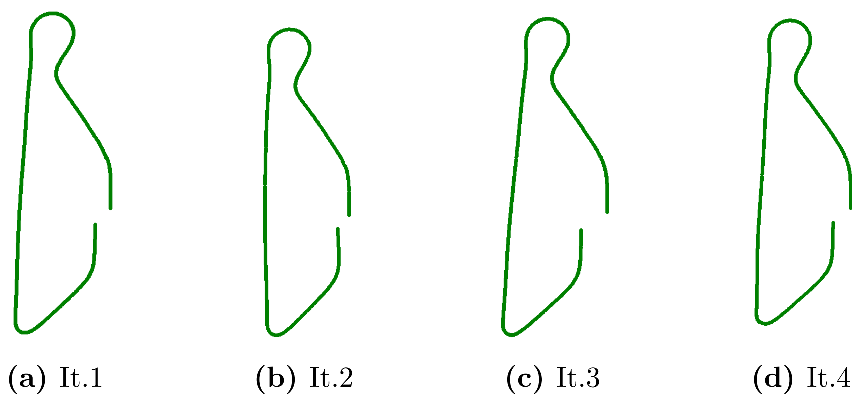


Figura 6.7: Superposición sobre el Mapa de Satélite Sin Calibración Alguna

6.3. Calibración del *Pitch*

El siguiente parámetro que ha de ser calibrado es el *pitch*, para lo cual primero se analiza las variaciones entre aquellas iteraciones en las que no se tuvo en cuenta más que el *yaw* calibrado, y aquellas en las que si y que pueden verse en la Fig.6.8:



(a) It.1 **(b)** It.2 **(c)** It.3 **(d)** It.4

Figura 6.8: Distintos Resultados de Iteraciones Con *Yaw* Calibrado.

El resultado de la superposición de las distintas soluciones de odometría solo teniendo en cuenta el *yaw* calibrado son las que se exponen a continuación en la Fig.6.9a:

(a) **(b)**

Figura 6.9: Resultados del Algoritmo de Partida. (a) Resultados Superpuestos con Calibración del *Yaw*. (b)Superposición sobre el Mapa de Satélite Calibración del *Yaw*.

Estos resultados sobrepuestos sobre la imagen de satélite corresponde a la mostrada en Fig.6.4b.

Los resultados una vez calibrado el *pitch* son los que ya se habían expuesto en el apartado 6.1.

6.4. Calibración del *Roll*

Una vez se han calibrado los dos primeros parámetros, es el momento de calibrar el último de ellos, y con ello se alcanzaría el estado resaltado de la Fig.6.10, que representa los tres parámetros calibrados, y sin tener en cuenta la comprobación de que los puntos característicos escogidos sean puntos de la calzada.

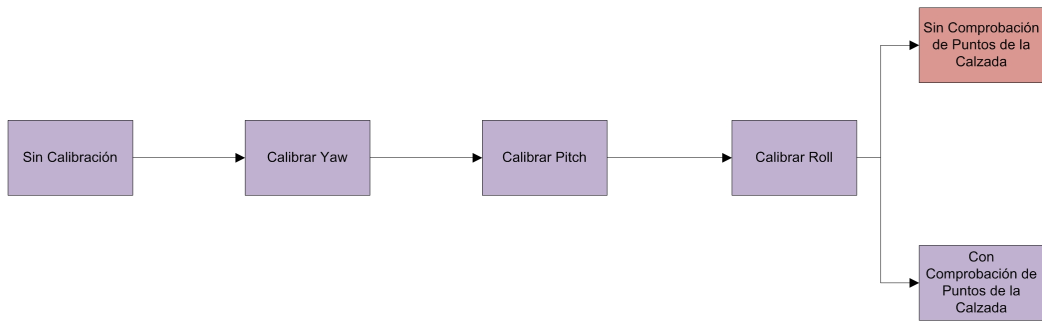
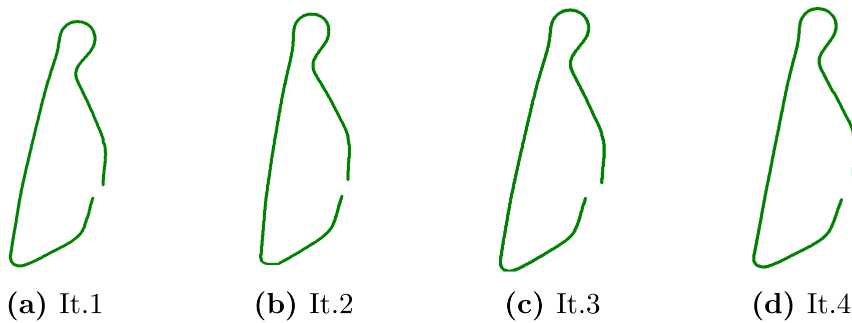


Figura 6.10: Estado de la Secuencia de Calibración



(a) It.1 **(b)** It.2 **(c)** It.3 **(d)** It.4

Figura 6.11: Distintos Resultados de Iteraciones Con Calibración del *Yaw*, *Pitch*, *Roll*.

Se han tomado de nuevo cuatro iteraciones del proceso tras la calibración, que se exponen en la Fig.6.11.

Resultando más claros los resultados que se encuentran en la Fig6.12.

(a) **(b)**

Figura 6.12: Resultados del Algoritmo de Partida. (a) Resultados Superpuestos con Calibración del *Yaw*, *Pitch*, *Roll*, sin Comprobación de Puntos en Calzada. (b) Superposición sobre el Mapa de Satélite Con Calibración del *Yaw*, *Pitch*, *Roll*, sin Comprobación de Puntos en Calzada.

De Fig.6.11, Fig.6.12a y Fig.6.9b se pueden extraer las siguientes conclusiones:

- El sistema se aproxima al recorrido real realizado por el vehículo.
- El sistema presenta repetitibilidad y robustez, pues las cuatro iteraciones realizadas arrojan datos similares, y con menos variabilidad que las anteriores versiones.
- No exige requerimientos de tiempo mayores que las versiones anteriores, dado que la diferencia radica en el cálculo de los parámetros que conectan el mundo y el sistema de la cámara.

6.5. Odometría Visual Con Calibración de sus Parámetros. Comprobación de Puntos en la Calzada.

A la vista de que los mejores resultados obtenidos por el sistema de odometría visual son los que tienen todos los parámetros ajustados, se parte de esta solución para poder introducir la opción de solo evaluar los puntos que efectivamente están en la calzada, y que no sean pertenecientes al entorno (Fig.6.13).

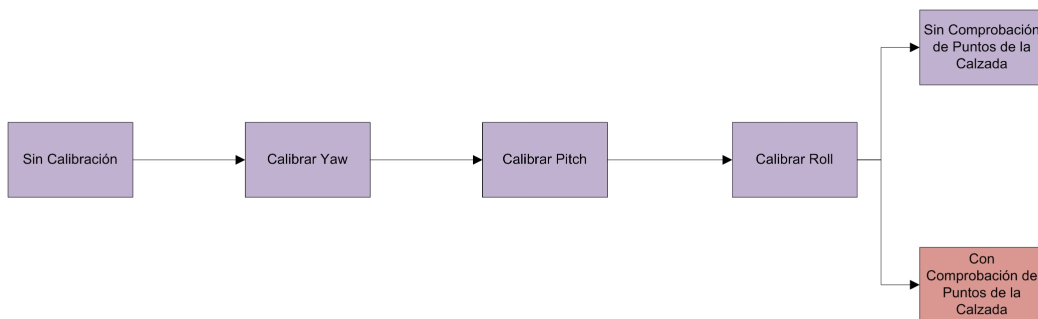


Figura 6.13: Estado de la Secuencia de Calibración

Los resultados obtenidos al añadir al algoritmo la detección de puntos de la calzada ha permitido obtener mejores resultados que todas las anteriores versiones, como muestra, se pueden ver las iteraciones captadas en Fig.6.14

(a) It.1 **(b)** It.2 **(c)** It.3 **(d)** It.4

Figura 6.14: Distintos Resultados de Iteraciones Con Calibración del *Yaw*, *Pitch*, *Roll*, Con Comprobación de Puntos en Calzada.

Al superponer los cuatro resultados unos sobre otros se obtiene un resultado que demuestra que el método empleado de calibración, unido con la mejora implementada que hace que solo se capten puntos de la calzada, genera unos resultados muy robustos, repetitivos y precisos, como se verá en las Fig.6.15a y Fig.6.15b.

(a) **(b)**

Figura 6.15: Resultados del Algoritmo de Partida. (a) Resultados Superpuestos con Calibración del *Yaw*, *Pitch*, *Roll*, Con Comprobación de Puntos en Calzada. (b)Superposición sobre el Mapa de Satélite Con Calibración del *Yaw*, *Pitch*, *Roll*, Con Comprobación de Puntos en Calzada.

6.6. Resultados de la Estimación del *Pitch*

Se han tomado datos del *pitch* estimado en cada uno de los distintos *frames* para estimar la variabilidad y la consiguiente utilidad de la calibración.

Para ello se han tomado nueve iteraciones distintas de la versión, y se ha calculado la media de todas ellas, para poder concluir que las variaciones del parámetro es pequeña a lo largo de todos los *frames*, con valores elevados (picos) localizados, como se puede ver en las Fig.6.16 y Fig.6.17. Se ha considerado para la Fig.6.17 que el gráfico denominado "Máximos Valores" es la que se compone por los valores máximos de las nueve iteraciones de cada fotograma, y de forma idéntica ocurre con la gráfica "Mínimos Valores".

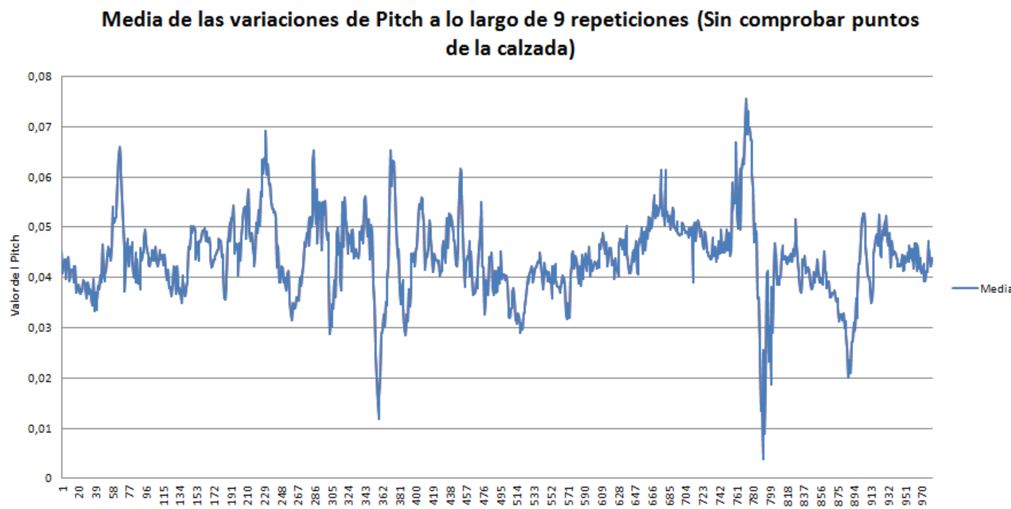


Figura 6.16: Valores de la Media Del Valor de *Pitch* en Cada *Frame*, 9 Iteraciones.

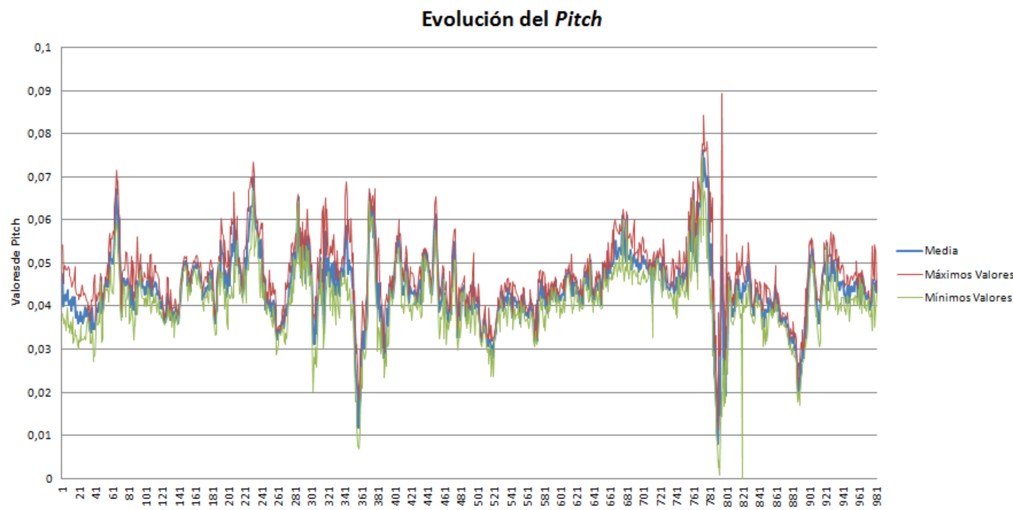


Figura 6.17: Valores de la Media Del Valor de *Pitch* en Cada *Frame*, 9 Iteraciones, Superpuesto con la Gráfica de Mayores Variaciones de las Iteraciones.

De los resultados mostrados en Fig.6.16 y Fig.6.17 puede sacarse como conclusiones lo siguiente:

- Los valores de *pitch* se deben a vibraciones del vehículo, y son de reducido valor dado que la carretera por la que se ha tomado los datos no presenta grandes variaciones.
- Todas las iteraciones arrojan valores muy similares (la diferencia entre valores máximos y mínimos con respecto a la media es mínima) , lo que refuerza lo ya comentado acerca de la robustez del sistema y la repetitividad.
- Los valores de *pitch* apenas sufren variaciones, y son cercanos a 0, obteniéndose los valores mayores de este parámetro en las curvas cerradas, tal como puede apreciarse en la Fig.6.18. Esto es causado por lo que ya se avanzó en la sección 3.5, y que es debido a la especial distribución de masas que sufre un vehículo al tomar una curva, y que produce cambios en la altura que la cámara detecta y que por *software* se corrige.

  Dependiendo del sentido de la toma de la curva, los valores se corrigen obteniéndose valores positivos, o al contrario, negativos.

Figura 6.18: *Frames* con Mayores Valores de *Pitch* Calibrado.

6.7. Resultados de la Estimación del *Roll*

En el caso de la estimación del *roll*, según los datos arrojados por las Fig.6.19 y Fig.6.20, se puede indicar que la estimación que se hizo según la cual $\operatorname{sen}\rho = 0$ y $\cos\rho = 1$ era acertada.

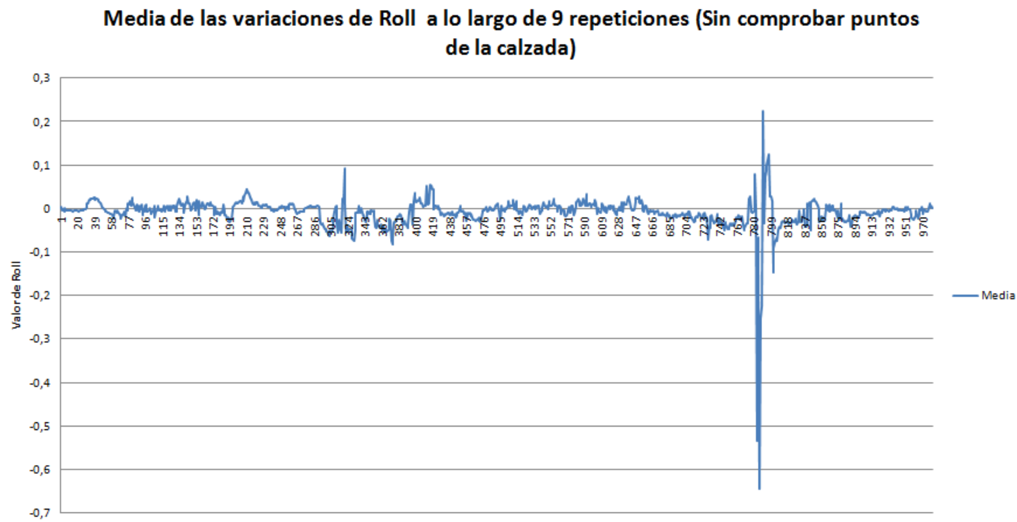


Figura 6.19: Valores de la Media Del Valor de *Roll* en Cada *Frame*, 9 Iteraciones.

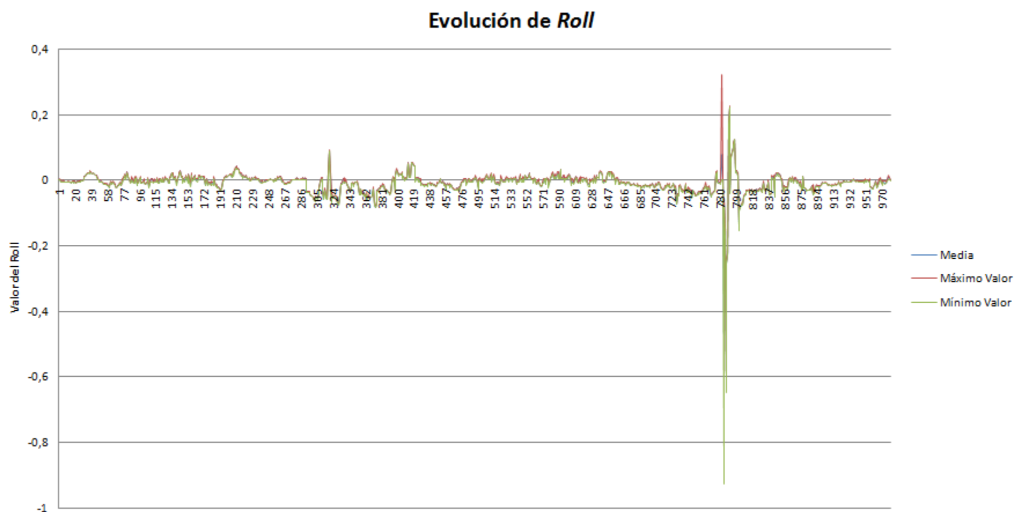


Figura 6.20: Valores de la Media Del Valor de *Roll* en Cada *Frame*, 9 Iteraciones, Superpuesto con la Gráfica de Mayores Variaciones de las Iteraciones.

Existen ciertos datos atípicos en los *frames* 782, 784 y 791, que corresponden a los puntos señalados en la Fig.6.21, momentos en los que el vehículo efectúa curvas muy cerradas.

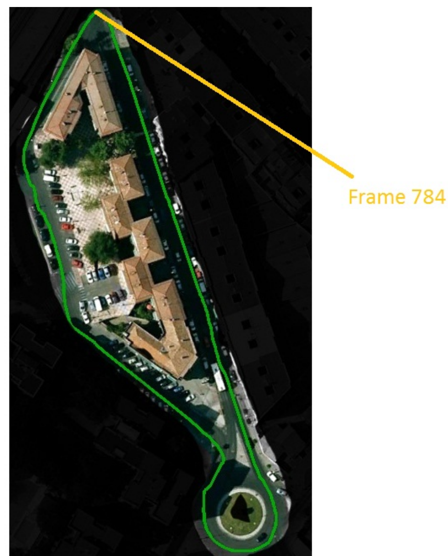


Figura 6.21: *Frames* con Mayores Valores de *Roll* Calibrado.

6.8. Error Cometido

Tras la realización de las iteraciones, se procedió a calcular el error cometido, mediante las coordenadas del punto final del trazado (idealmente, debería ser $(0, 0)$ la posición final), y mediante la distancia euclídea, se ha podido estimar no solo el error, sino también la desviación típica, y que pueden verse en la Tabla 6.4.

Error Cometido (m)	Error Cometido (%)	Desviación Típica (m)
13,75	3,13	0,672

Tabla 6.4: Error y Desviación Típica Del Proceso.

7

CONCLUSIONES Y TRABAJOS FUTUROS

Tras realizar las pruebas pertinentes con el algoritmo planteado en el trabajo, se puede constatar que los resultados han logrado una mejora considerable frente a la versión inicial de la solución de odometría visual existente, y que son las enumeradas a continuación:

1. La *repetitividad* en las soluciones obtenidas es alta, de manera que para distintas repeticiones del código, las soluciones que aporta son similares. A lo largo del algoritmo se hace uso en numerosas ocasiones de operaciones basadas en números aleatorios (cada paso que implicaba hacer uso de RANSAC), por lo que la repetitividad adquiere mayor valor.

2. La *precisión* que alcanza en las soluciones que aporta sobre el trazado de prueba es como ya se ha indicado de $13,75$ metros de error sobre el trazado que consta de 500 metros, lo cual da un error de precisión de aproximadamente el $3,13\,\%$.

3. El algoritmo introducido es fácilmente escalable: no precisa de un estado de calibración previo ni de ningún tipo de proceso de aprendizaje del entorno sobre el que debe funcionar. Es rápidamente implementable en cualquier situación.

4. Es un sistema rápido, no precisa de tiempos de *set-up* como otros sistemas (el GPS necesita de un tiempo hasta que encuentra señal).

Una comparativa acerca de cómo ha variado el trazado de la odometría antes de la autocalibración y después se muestra en Fig.7.1.

Figura 7.1: Resultados Antes y Después de la Autocalibración

Debido al peligro que entraña el ámbito de los vehículos inteligentes en el caso de que ocurra el más mínimo error, resulta importante poseer un sistema de respaldo, basado en el GPS diferencial preferiblemente, trabajando en cooperación para asegurar en todo momento que los resultados sean coherentes entre ambos sistemas, y por ende, se pueda concluir que son fiables.

Para futuras implementaciones, se puede considerar la opción de calibrar *frame* a *frame* la desviación de *yaw*. En el presente trabajo esta desviación ha sido considerada constante, pero esta asunción deja de ser válida en el

momento en el que se tiene en cuenta la vibración del vehículo, que producen fluctuaciones en los valores del parámetro.

Otra posible mejora que no ha sido introducida en este proyecto es el de conseguir que el algoritmo pueda funcionar sobre cualquier tipo de vía, sin depender de la hipótesis de que el suelo es plano, en caminos campestres o en vías con badenes. Por ejemplo, en el caso de estos últimos, la variación de *pitch* sufriría variaciones bruscas, como puede verse en la Fig.7.2.

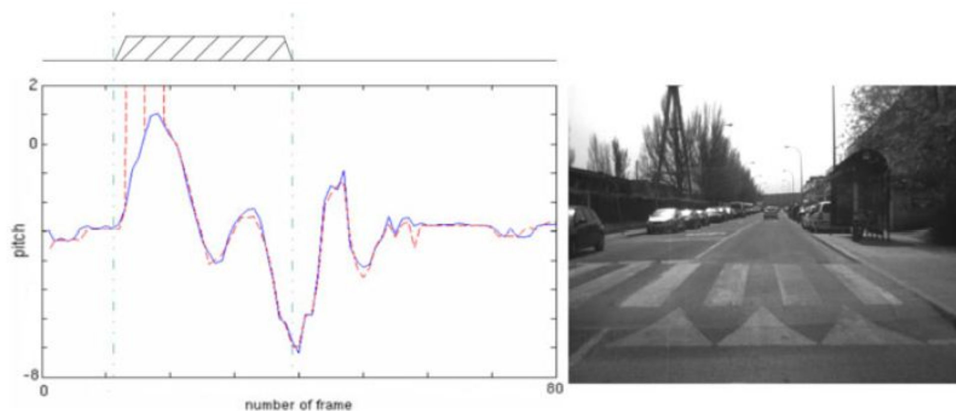


Figura 7.2: Comportamiento del *Pitch* en un Badén.

8

COSTES DEL PROYECTO

En el presente capítulo se hace una estimación de los costes del proyecto, de forma pormenorizada, de forma que se estima el tiempo necesario en la ejecución, los costes asociados a la equipación necesaria y los salarios del personal que lo lleva a cabo.

El primero de los puntos a tratar es la estimación del tiempo necesario, que se adjunta en la Tabla8.1:

Etapa	Tiempo
Estudio y comprensión del problema a resolver	30 h
Documentación Y Estado del Arte	50 h
Redacción de la memoria (parte teórica)	30 h
Implementación del código	250 h
Pruebas del algoritmo	30 h
Redacción de la memoria (parte práctica)	25 h
TOTAL	415 h

Tabla 8.1: Estimación de Tiempo Empleado en el Proyecto

Suponiendo un salario base bruto de 35€/h del ingeniero, esto resulta en:

$$415h \cdot 35€/h = 14525€ \quad (8.1)$$

Los materiales necesarios para poder llevar a la práctica el proyecto ha constado de los siguientes materiales, con los costes asociados en la Tabla 8.2.

Material	Coste
Ordenador de sobremesa	1.200 €
Interfaz salpicadero (Pantalla Xenarc 8 pulgadas)	400 €
Cámara estéreo (Modelo *Bumblebee* del fabricante *Pointgrey*)	3.000 €
Paquete de librerías MIL	500 €
Paquete de librerías OPENCV	Gratuito
Otro material (cables, soportes, etc.)	100 €
TOTAL	4.900 €

Tabla 8.2: Estimación de Costes de Materiales del Proyecto

De acuerdo con 8.1 y la Tabla 8.2, el precio total del proyecto asciende a:

$$14525€ + 4900€ = 19425€ \quad (8.2)$$

APÉNDICES

A

PARÁMETROS

El primero de los apéndices mostrados corresponde al archivo del algoritmo que almacena los parámetros que se han definido como cabecera de programa y que si son modificados, lo hacen a lo largo de todos los archivos.

```
#define OFF_LINE
#define USE_FLANN
#define PERFIL_V3
//#define MOSTRAR_PERFIL
#define SURF_V2
//#define COMPROBAR_PTOS_MAPA_LIBRE
#define MOSTRAR_SURF
#define FILTRO
//¡OJO! NO se puede usar CALPTOS_V2 sin alguna CALSOL
   distinta de V1
#define CALPTOS_V2
#define CALSOL_V5
#define CALIBRACION
//#define MOSTRAR_ROLL
//////////////////////////
#define WIDTH_IMAGEN 640
#define HEIGHT_IMAGEN 480
#define NUM_IMAGENES 982
//////////////////////////
#define ROWSperTHREAD 40  // Número de filas que procesará un
    hilo
#define BLOCK_W 128      // Ancho del hilo para calcular la
   disparidad
```

```
#define ANCHO_BLOQUE 64   // Valor del ancho del bloque para
    el cálculo de la laplaciana de la gausiana
#define RADIUS_H 8      // Kernel Radius 5V & 5H = 11x11
    kernel
#define RADIUS_V 8
#define MIN_SSD 5737500   //((RADIUS_H*2+1)*(RADIUS_V*2+1)
    *255)   // Valor mínimo de la SSD en la ventana (por
    ejemplo el mayor posible en ventana de 7x7)
#define STEREO_MIND 0.0f  // Mínimo valor de disparidad a
    comprobar
#define STEREO_MAXD 30.0f // Máximo valor de disparidad a
    comprobar
#define SHARED_MEM_SIZE ((BLOCK_W + 2*RADIUS_H)*sizeof(int) )
       // Memoria compartida usada
#define VALOR_INICIAL 0   //Valor inicial para el array de
    salida, será el que se usará si disp no ha sido menor que
    MIN_SSD
#define U_DISP_UMBRALIZADO
#define UMBRAL_AREA_ESTUDIO 1
/////Perfil de la calzada////
#define MAX_PTOS_BLANCOS 14400
#define RP_NUM_THREADS 8192
#define Y_THRESHOLD 24
#define RP_RANGO 1
#define RP_UMBRAL 128
/////Odometría////
#define B 0.119915f
#define F 811.9104f
#define CX 322.0614f
#define CY 247.6637f
#define DESVIACION_YAW 0.0547f
#define ALTURA_SURF (HEIGHT_IMAGEN-(int)(HEIGHT_IMAGEN/3))
#define RESOLUCION_MAPA 3
#define ALTO_MAPA 800
#define ANCHO_MAPA 1000
#define UMBRAL_FILTRO 2
//////RANSAC Odometría///////
#define OD_NUM_THREADS 2048
#define RANGO_THETA 0.01f
#define RANGO_X 0.01f
#define RANGO_Y 0.01f
//////Memoria compartida
#define MAX_PTOS_SURF 2048
#define BUF_SIZE WIDTH_IMAGEN*HEIGHT_IMAGEN*sizeof(unsigned
    char)
#define BUF_SIZE2 (MAX_PTOS_SURF+1)*sizeof(Point2f)
//////Autocalibración
#define ROLL_RANGO 5
```

B

MAIN

Se muestra el archivo desde el cual se hacen las llamadas a las funciones de la GPU, CPU y el proceso que ejecuta SURF. Este algoritmo trabaja 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;
  }
};

//Temporizador basado en las OpenCV
class TimerCV{
public:
  string nombre;
  int64 t;

  TimerCV(string nombre){
    this->nombre=nombre;
    t=getTickCount();
  }
  void Parar(){
    t=getTickCount()-t;
    cout << nombre << " time: " << t*1000/getTickFrequency()
        << " ms" << endl;
  }

};

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
////////////////////////////////////////
//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);

#ifdef CALIBRACION
extern "C" void odometria(PerfilCalzada perfil, PerfilCalzada
    perfil_anterior, Point2f* pt1, Point2f* pt2, vector<float
   > &theta_ac, vector<float> &x_ac, vector<float> &y_ac,
   float aroll , float arollant,FILE *f);
#else
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);
#endif

//////////////////////////////////////
// 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
#ifdef MOSTRAR_ROLL
  MIL_ID MilPerfil;
  MdispAlloc(MilSystem,M_DEFAULT ,"M_DEFAULT",M_DEFAULT ,&
     MilPerfil);
  MdispSelect(MilPerfil,MilDisparitylibre);
  Overlay *dibujoPerfil = new Overlay(MilPerfil);
#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));

#ifdef COMPROBAR_PTOS_MAPA_LIBRE
  Point2f* PtosC1=(Point2f*)malloc((MAX_PTOS_SURF+1)*sizeof(
     Point2f));
  Point2f* PtosC2=(Point2f*)malloc((MAX_PTOS_SURF+1)*sizeof(
     Point2f));
#endif

  //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

TimerMedia  t1("Carga de imágenes"),
      t2("Semáforo 1"),
      t3("Copia de imagen"),
      t4("Mapa de disparidad"),
      t5("Perfil de la calzada"),
      t6("Procesamiento de imágenes"),
      t7("Semáforo 2"),
      t8("SURF/Copia de vectores"),
      t9("Dibujo SURF"),
      t10("Odometría"),
      t11("Dibujo mapa"),
      t0("Total");

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

  //cout << i << endl;
  //getchar();

  t0.Iniciar();
  t1.Iniciar();

//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);

  t1.Parar();

  //Envío de la imagen izquierda al programa auxiliar
#ifdef SURF_V2
  t2.Iniciar();
  WaitForSingleObject(ghSemaphoreWIm, INFINITE);
  t2.Parar();

  t3.Iniciar();
  CopyMemory((PVOID)pBuf, h_izq, size);
  if(!ReleaseSemaphore(ghSemaphoreRIm,1,NULL)) printf("
     ReleaseSemaphore error: %d\n", GetLastError());
  t3.Parar();
#endif

  t4.Iniciar();

  //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

  t4.Parar();
  t5.Iniciar();

  //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

  t5.Parar();
  t6.Iniciar();

  //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);

  t6.Parar();

#ifdef CALIBRACION
  MimBinarize(MilDisparitylibre, MilDisparitylibre,M_IN_RANGE
     ,10,10);
  MbufGet(MilDisparitylibre,h_LG_izq);
  perfil_rollant=perfil_roll;
  ransac(h_LG_izq, perfil_roll, ROLL_RANGO, 128, WIDTH_IMAGEN
     ,HEIGHT_IMAGEN);

#ifdef MOSTRAR_ROLL
  dibujoPerfil->Iniciar();
  dibujoPerfil->Cambiar_Color(M_RGB888(255,0,0));
  //punto p0 = {0,(int)*(recta_res+1)};
```

```
  //punto p1 = {(int)(STEREO_MAXD-1),(int)((*(recta_res))*(
     STEREO_MAXD-1)+(*(recta_res+1)))};
  punto p0 = {0, (int)perfil_roll.horizonte};
  punto p1 = {(int)(WIDTH_IMAGEN-1), (int)(perfil_roll.m*(
     WIDTH_IMAGEN-1)+perfil_roll.horizonte)};
  dibujoPerfil->Linea(p0,p1);

  dibujoPerfil->Cambiar_Color(M_RGB888(0,255,0));

  punto p2 = {0, (int)perfil_roll.horizonte+ROLL_RANGO};
  punto p3 = {(int)(WIDTH_IMAGEN-1), (int)(perfil_roll.m*(
     WIDTH_IMAGEN-1)+perfil_roll.horizonte+ROLL_RANGO)};
  dibujoPerfil->Linea(p2,p3);


  punto p4 = {0, (int)perfil_roll.horizonte-ROLL_RANGO};
  punto p5 = {(int)(WIDTH_IMAGEN-1), (int)(perfil_roll.m*(
     WIDTH_IMAGEN-1)+perfil_roll.horizonte-ROLL_RANGO)};
  dibujoPerfil->Linea(p4,p5);
  dibujoPerfil->Finalizar();
#endif MOSTRAR_ROLL

  float aroll,arollant;
  arollant=aroll;

  if (perfil_roll.m<0.5)
    aroll=atan(perfil_roll.m*cos(perfil.pitch));
  else
  {aroll=0; printf("Roll no considerado\n");}

#endif

  //Puntos característicos



#ifdef SURF_V2
  t7.Iniciar();
  //Recibir de programa auxiliar
  WaitForSingleObject(ghSemaphoreRVec, INFINITE);
  t7.Parar();

  t8.Iniciar();

#ifdef COMPROBAR_PTOS_MAPA_LIBRE
  memcpy(PtosC1, pBuf2, ((int)(pBuf2[0].x)+1)*sizeof(Point2f)
     );
  memcpy(PtosC2, pBuf3, ((int)(pBuf3[0].x)+1)*sizeof(Point2f)
     );
```

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


  if(!ReleaseSemaphore(ghSemaphoreWVec,1,NULL)) printf("
     ReleaseSemaphore error: %d\n", GetLastError());
#else
  t8.Iniciar();
  surf (h_der, h_izq, i, v_pt1, v_pt2);
#endif

  t8.Parar();

#ifdef COMPROBAR_PTOS_MAPA_LIBRE
  v_pt1[0].x=0;
  v_pt2[0].x=0;

  for(int p=1;p<=(int)PtosC1[0].x;p++)
  {
    if (h_der[(int)PtosC2[p].x+WIDTH_IMAGEN*((int)PtosC2[p].y
       +ALTURA_SURF)]>0){
      v_pt1[(int)v_pt1[0].x+1]=PtosC1[p];
      v_pt2[(int)v_pt2[0].x+1]=PtosC2[p];
      v_pt1[0].x++;
      v_pt2[0].x++;
    }
  }
#endif

  t9.Iniciar();

#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

  t9.Parar();
  t10.Iniciar();

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

#ifdef CALIBRACION
    odometria(perfil, perfil_anterior, v_pt1, v_pt2, theta_ac
        , x_ac, y_ac, aroll ,arollant ,file);
#elif (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
    t10.Parar();
    t11.Iniciar();

    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();

  t11.Parar();

#ifdef VIDEO
  MbufExportSequence("Video.avi", M_AVI_DIB, &MilColor, 1, 8,
      M_APPEND);
#endif
  t0.Parar();

}

t1.Escribir(f);
t2.Escribir(f);
t3.Escribir(f);
t4.Escribir(f);
t5.Escribir(f);
t6.Escribir(f);
t7.Escribir(f);
t8.Escribir(f);
```

```
t9.Escribir(f);
t10.Escribir(f);
t11.Escribir(f);
t0.Escribir(f);


#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);
}
```

C

KERNEL

Se presenta el código ejecutado en la GPU.

```
#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 número 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[]; // column squared
     difference functions

  int d;     // disparity value
  int diff;  // difference temporary value
  int ssd;   // total SSD for a kernel
  int x_tex; // texture coordinates for image lookup
  int y_tex;
  int x_der;
  int row;   // the current row in the rolling window
  int i;       // for index variable
```

```
// use define's to save registers
  #define X (__mul24(blockIdx.x,BLOCK_W) + threadIdx.x)
#define Y (__mul24(blockIdx.y,ROWSperTHREAD))

// for threads reading the extra border pixels, this is the
    offset
// into shared memory to store the values

int extra_read_val = 0;
if(threadIdx.x < (2*RADIUS_H)) extra_read_val = BLOCK_W+
   threadIdx.x;

// initialize the memory used for the disparity and the
   disparity difference
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;

  // do the first row
  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();

    // now accumultate the total
    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();

    // now do the remaining rows
    y_tex = Y - RADIUS_V; // this is the row we will remove
    for(row = 1;row < ROWSperTHREAD && (row+Y < (height+
        RADIUS_V)); row++)
    {
      // subtract the value of the first row from column sums
      diff = tex2D(tex_izq,x_tex,y_tex) - tex2D(tex_der,x_tex
          -d,y_tex);
      col_ssd[threadIdx.x] -= SQ(diff);

      // add in the value from the next row down
        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();  // wait for everything to complete
    } // for row loop
  } // for d loop
  } // if 'int the image' loop
}
__global__ void cross_checking(unsigned char* izquierda,int
    width)
{
  // calculate texture coordinates
    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)
{
  // calculate texture coordinates
    unsigned int x_tex = blockIdx.x*blockDim.x + threadIdx.x;
    //unsigned int y_tex = blockIdx.y*blockDim.y + threadIdx.
       y;
  //unsigned char histograma[(int)STEREO_MAXD];
  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)
{
  // calculate texture coordinates
    //unsigned int x_tex = blockIdx.x*blockDim.x + threadIdx.
       x;
    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)
{
  // calculate texture coordinates
    unsigned int x_tex = blockIdx.x*blockDim.x + threadIdx.x;
    //unsigned int y_tex = blockIdx.y*blockDim.y + threadIdx.
       y;
  //unsigned char histograma[(int)STEREO_MAXD];
  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)
{
  // calculate texture coordinates
    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)
{
  // calculate texture coordinates
    unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;
    //unsigned int y_tex = blockIdx.y*blockDim.y + threadIdx.
       y;
  //unsigned char histograma[(int)STEREO_MAXD];
  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)
{
  // calculate texture coordinates
    unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;

  unsigned char permitadas[(int)STEREO_MAXD] =
     {0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};


  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);

#ifdef UMBRAL_Y_RANSAC
  unsigned int const y_thres=Y_THRESHOLD;
  if ((x1!=x2)&&(abs(y1-y2)>y_thres){
#else
  if (x1!=x2){
#endif
    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 ransacOdometria_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
/////////////////////////////////////
#ifdef CALIBRACION
__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, float roll, float rollant)
{
  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,delta,
     delta_anterior;

  delta_anterior=(pt1[pos].y+ALTURA_SURF-horizonte_anterior)/
     m_anterior;
  delta=(pt2[pos].y+ALTURA_SURF-horizonte)/m;

  mundo_0.x = ((cos(rollant)*B)/delta_anterior)*(pt1[pos].x-
     CX)+((B*cos(pitch_anterior)*sin(rollant))/delta_anterior
     )*(pt1[pos].y+ALTURA_SURF-CY)+((F*B*sin(rollant)*sin(
     pitch_anterior))/delta)-(B*cos(rollant)/2);
  mundo_0.y = ((B*sin(pitch_anterior))/delta_anterior)*(pt1[
     pos].y+ALTURA_SURF-CY)+((F*B*cos(pitch_anterior))/
     delta_anterior);

  mundo_1.x = ((cos(roll)*B)/delta)*(pt2[pos].x-CX)+((B*cos(
     pitch)*sin(roll))/delta)*(pt2[pos].y+ALTURA_SURF-CY)+(F*
     B*sin(roll)*sin(pitch))/delta-(B*cos(roll))/2;
  mundo_1.y = ((B*sin(pitch))/delta)*(pt2[pos].y+ALTURA_SURF-
     CY)+((F*B*cos(pitch))/delta);

#ifdef CALIBRACION_FORMULAS_SIN_CALIBRACIÓN
  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);
#endif

  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
```

D

HOST

Corresponde al código ejecutado en la CPU.

```
////////////////////////////////
// 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 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////////////////////
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_LG_izq,d_izq, 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");}
}

#ifdef CALIBRACION
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,
              float aroll,
              float arollant,
              FILE*f)
{

  //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;

CUDA_SAFE_CALL(cudaMalloc((void**) &d_pt1, (nelementos+1)*
    sizeof(Point2f)));
CUDA_SAFE_CALL(cudaMalloc((void**) &d_pt2, (nelementos+1)*
    sizeof(Point2f)));

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));

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,
      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 soluciones
  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 , aroll, arollant);

  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) );

  //RANSAC
  ransacOdometria_t<<<dimGridR2, dimBlockR2>>>(d_seleccion,
     d_puntuacion, nelementos, d_theta, d_x_transl,
     d_y_transl,  devStates);

  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) );

  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

  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());

  //Liberado de memoria
  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));
}
#else
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)
{
```

```
  static TimerCUDA  t1("Inicializ."),
            t2("CalcularPtos"),
            t3("RANSAC"),
            t4("CalcularMayor"),
            t5("Liberar Mem.");
  t1.Iniciar();

  //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;


#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);

  t1.Parar();
  t2.Iniciar();

  //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

  t2.Parar();
  t3.Iniciar();

  //RANSAC
#ifdef CALSOL_V5
  ransacOdometria_t<<<dimGridR2, dimBlockR2>>>(d_seleccion,
     d_puntuacion, nelementos, d_theta, d_x_transl,
     d_y_transl,  devStates);
#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

  t3.Parar();
  t4.Iniciar();

  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

  t4.Parar();
  t5.Iniciar();

  //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));

  t5.Parar();

  if(i==(NUM_IMAGENES-1)){

    t1.Escribir(f);
    t2.Escribir(f);
    t3.Escribir(f);
    t4.Escribir(f);
    t5.Escribir(f);

  }

}
#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 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;

  }
}
```

E

SURF

Este anexo corresponde al proceso que se ejecuta de forma paralela y que lleva a cabo la detección de puntos característicos.

```
//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>
#include <fstream> //para el manejo de ficheros
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;
};

//Temporizadores
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;
    //fprintf(f, "%s \t %f ms\n", nombre, tiempo);
    f<<nombre<<" - "<<tiempo<<" ms"<<endl;
  }
};

class TimerCV{
public:
```

```
  string nombre;
  int64 t;

  TimerCV(string nombre){
    this->nombre=nombre;
    t=getTickCount();
  }
  void Parar(){
    t=getTickCount()-t;
    cout << nombre << " time: " << t*1000/getTickFrequency()
        << " 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;
  }
};

//////////////////////////////////
// PROGRAMA
//////////////////////////////////
int main(int argc, char** argv)
{
  ofstream f("TIEMPOS.txt", ios::app);

  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, INFINITE
     );

  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);

    //free(pt1);
    //free(pt2);

    //pt1=(Point2f*)malloc((n+1)*sizeof(Point2f));
    //pt2=(Point2f*)malloc((n+1)*sizeof(Point2f));

    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();
}
```

F

FUNCIONES DE SURF

Funciones que son necesarias para la ejecución del proceso descrito en Apéndice E.

```
void
flannFindPairs( const CvSeq*, const CvSeq* objectDescriptors,
             const CvSeq*, const CvSeq* imageDescriptors,
                 vector<int>& ptpairs )
{
  int length = (int)(objectDescriptors->elem_size/sizeof(
      float));

    cv::Mat m_object(objectDescriptors->total, length, CV_32F
        );
  cv::Mat m_image(imageDescriptors->total, length, CV_32F);


  // copy descriptors
    CvSeqReader obj_reader;
  float* obj_ptr = m_object.ptr<float>(0);
    cvStartReadSeq( objectDescriptors, &obj_reader );

    for(int i = 0; i < objectDescriptors->total; i++ )
    {
        const float* descriptor = (const float*)obj_reader.
            ptr;
        CV_NEXT_SEQ_ELEM( obj_reader.seq->elem_size,
            obj_reader );
```

```
        memcpy(obj_ptr, descriptor, length*sizeof(float));
        obj_ptr += length;
    }

    CvSeqReader img_reader;
  float* img_ptr = m_image.ptr<float>(0);
    cvStartReadSeq( imageDescriptors, &img_reader );

    for(int i = 0; i < imageDescriptors->total; i++ )
    {
        const float* descriptor = (const float*)img_reader.
           ptr;
        CV_NEXT_SEQ_ELEM( img_reader.seq->elem_size,
           img_reader );
        memcpy(img_ptr, descriptor, length*sizeof(float));
        img_ptr += length;
    }


    // find nearest neighbors using FLANN
    cv::Mat m_indices(objectDescriptors->total, 2, CV_32S);
    cv::Mat m_dists(objectDescriptors->total, 2, CV_32F);
    cv::flann::Index flann_index(m_image, cv::flann::
       KDTreeIndexParams(4));  // using 4 randomized kdtrees

    flann_index.knnSearch(m_object, m_indices, m_dists, 2, cv
       ::flann::SearchParams(64) ); // maximum number of
       leafs checked

    int* indices_ptr = m_indices.ptr<int>(0);
    float* dists_ptr = m_dists.ptr<float>(0);
    for (int i=0;i<m_indices.rows;++i) {
      if (dists_ptr[2*i]<0.6*dists_ptr[2*i+1]) {
        ptpairs.push_back(i);
        ptpairs.push_back(indices_ptr[2*i]);
      }
    }
}

double
compareSURFDescriptors( const float* d1, const float* d2,
   double best, int length )
{
    double total_cost = 0;
    assert( length % 4 == 0 );
    for( int i = 0; i < length; i += 4 )
    {
        double t0 = d1[i] - d2[i];
        double t1 = d1[i+1] - d2[i+1];
```

```
        double t2 = d1[i+2] - d2[i+2];
        double t3 = d1[i+3] - d2[i+3];
        total_cost += t0*t0 + t1*t1 + t2*t2 + t3*t3;
        if( total_cost > best )
            break;
    }
    return total_cost;
}

int
naiveNearestNeighbor( const float* vec, int laplacian,
                      const CvSeq* model_keypoints,
                      const CvSeq* model_descriptors )
{
    int length = (int)(model_descriptors->elem_size/sizeof(
        float));
    int i, neighbor = -1;
    double d, dist1 = 1e6, dist2 = 1e6;
    CvSeqReader reader, kreader;
    cvStartReadSeq( model_keypoints, &kreader, 0 );
    cvStartReadSeq( model_descriptors, &reader, 0 );

    for( i = 0; i < model_descriptors->total; i++ )
    {
        const CvSURFPoint* kp = (const CvSURFPoint*)kreader.
            ptr;
        const float* mvec = (const float*)reader.ptr;
      CV_NEXT_SEQ_ELEM( kreader.seq->elem_size, kreader );
        CV_NEXT_SEQ_ELEM( reader.seq->elem_size, reader );
        if( laplacian != kp->laplacian )
            continue;
        d = compareSURFDescriptors( vec, mvec, dist2, length
            );
        if( d < dist1 )
        {
            dist2 = dist1;
            dist1 = d;
            neighbor = i;
        }
        else if ( d < dist2 )
            dist2 = d;
    }
    if ( dist1 < 0.6*dist2 )
        return neighbor;
    return -1;
}
```

```
void
findPairs( const CvSeq* objectKeypoints, const CvSeq*
    objectDescriptors,
            const CvSeq* imageKeypoints, const CvSeq*
                imageDescriptors, vector<int>& ptpairs )
{
    int i;
    CvSeqReader reader, kreader;
    cvStartReadSeq( objectKeypoints, &kreader );
    cvStartReadSeq( objectDescriptors, &reader );
    ptpairs.clear();

    for( i = 0; i < objectDescriptors->total; i++ )
    {
        const CvSURFPoint* kp = (const CvSURFPoint*)kreader.
            ptr;
        const float* descriptor = (const float*)reader.ptr;
        CV_NEXT_SEQ_ELEM( kreader.seq->elem_size, kreader );
        CV_NEXT_SEQ_ELEM( reader.seq->elem_size, reader );
        int nearest_neighbor = naiveNearestNeighbor(
            descriptor, kp->laplacian, imageKeypoints,
            imageDescriptors );
        if( nearest_neighbor >= 0 )
        {
            ptpairs.push_back(i);
            ptpairs.push_back(nearest_neighbor);
        }
    }
}
```

BIBLIOGRAFÍA

[1] DGT, "Anuario accidentes tráfico 2000." http://www.dgt.es/was6/portal/contenidos/es/seguridad_vial/estadistica/publicaciones/anuario_estadistico/anuario_estadistico003.pdf, 2000. [Online; disponible el 1 de Octubre de 2012].

[2] N. Gregersen, B. Brehmer, and B. Morén, "Road safety improvement in large companies. an experimental comparison of different measures," *Accident Analysis & Prevention*, vol. 28, no. 3, pp. 297–306, 1996.

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

[4] Volkswagen, "Proyecto HAVEit." http://www.haveit-eu.org, 2012. [Online; disponible el 1 de Octubre de 2012].

[5] SIM, "Sichere Intelligente Mobilität Testfeld Deutschland." http://www.simtd.de/index.dhtml/385065a50c58a450308k/-/deDE/-/CS/-/, 2012. [Online; disponible el 1 de Octubre de 2012].

[6] S. D. D. Lab, "Stanford dynamic design lab webpage." http://ddl.stanford.edu/node/45, 2012. [Online; disponible el 1 de Octubre de 2012].

[7] G. Blog, "The self-driving car logs more miles on new wheels." http://googleblog.blogspot.com.es/2012/08/the-self-driving-car-logs-more-miles-on.html, 2012. [Online; disponible el 1 de Octubre de 2012].

[8] UPM-CSIC, “Proyecto AUTOPIA.” http://www.car.upm-csic.es/autopia/, 2012. [Online; disponible el 1 de Octubre de 2012].

[9] IVSymposium2012, “Congreso vehículos inteligentes 2012.” http://www.robesafe.es/iv2012/, 2012. [Online; disponible el 1 de Octubre de 2012].

[10] P. Guiade, “Proyecto guiade.” http://www.proyectoguiade.com/, 2010. [Online; disponible el 1 de Octubre de 2012].

[11] USGS, “Lidar information coordination and knowledge.” http://lidar.cr.usgs.gov/, 2012. [Online; disponible el 1 de Octubre de 2012].

[12] M. Skolnik, “Radar handbook,” 1970.

[13] G. Stein, O. Mano, and A. Shashua, “A robust method for computing vehicle ego-motion,” in *IEEE Intelligent Vehicles Symposium (IV2000*.

[14] D. Scaramuzza, F. Fraundorfer, and R. Siegwart, “Real-time monocular visual odometry for on-road vehicles with 1-point ransac,” in *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pp. 4293–4299, Ieee, 2009.

[15] 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.

[16] P. Grey, “Bumblebee2.” http://www.ptgrey.com/products/bumblebee2/bumblebee2_stereo_camera.asp, 2012. [Online; disponible el 1 de Octubre de 2012].

[17] T. Marita, F. Oniga, S. Nedevschi, T. Graf, and R. Schmidt, “Camera calibration method for far range stereovision sensors used in vehicles,” in *Intelligent Vehicles Symposium, 2006 IEEE*, pp. 356–363, IEEE, 2006.

[18] T. Dang, C. Hoffmann, and C. Stiller, “Self-calibration for active automotive stereo vision,” in *Intelligent Vehicles Symposium, 2006 IEEE*, pp. 364–369, IEEE, 2006.

[19] P. Dana, “Global positioning system (gps) time dissemination for real-time applications,” *Real-Time Systems*, vol. 12, no. 1, pp. 9–40, 1997.

[20] ESA, “Galileo position system.” http://www.esa.int/esaNA/GGG0H750NDC_galileo_0.html, 2012. [Online; disponible el 1 de Octubre de 2012].

[21] 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.

[22] 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.

[23] Y. Cheng, M. Maimone, and L. Matthies, "Visual odometry on the mars exploration rovers-a tool to ensure accurate driving and science imaging," *Robotics & Automation Magazine, IEEE*, vol. 13, no. 2, pp. 54–62, 2006.

[24] NASA, "Curiosity Visual Odometry." http://www.nasa.gov/mission_pages/msl/news/msl20120829f.html?fb_action_ids=4453210059078&fb_action_types=og.likes&fb_source=aggregation&fb_aggregation_id=246965925417366, 2012. [Online; disponible el 1 de Octubre de 2012].

[25] P. Jasiobedski, M. Greenspan, and G. Roth, "Pose determination and tracking for autonomous satellite capture," 2001.

[26] R. Labayrade and D. Aubert, "A single framework for vehicle roll, pitch, yaw estimation and obstacles detection by stereovision," in *Intelligent Vehicles Symposium, 2003. Proceedings. IEEE*, pp. 31–36, IEEE, 2003.

[27] J. Yunde and Q. Xiameng, "An embedded calibration stereovision system," in *Intelligent Vehicles Symposium (IV), 2012 IEEE*, pp. 1072–1077, IEEE, 2012.

[28] F. Espuny, J. Aranda, and J. Burgos Gil, "Camera self-calibration with parallel screw axis motion by intersecting imaged horopters," *Image Analysis*, pp. 1–12, 2011.

[29] R. Liu, H. Zhang, M. Liu, X. Xia, and T. Hu, "Stereo cameras self-calibration based on sift," in *Proceedings of the 2009 International Conference on Measuring Technology and Mechatronics Automation-Volume 01*, pp. 352–355, IEEE Computer Society, 2009.

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

[31] 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.

[32] 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.

[33] D. Kirk and W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach.* Applications of GPU Computing Series, Elsevier Science, 2010.

[34] nVidia, "Cuda." `http://www.nvidia.es/object/cuda_home_new_es.html`, 2012. [Online; disponible el 1 de Octubre de 2012].

[35] C. Zeller, "Nvidia tutorial cuda," *Disponibile in rete: http://people. maths. ox. ac. uk/˜ gilesm/hpc/NVIDIA/NVIDIA_CUDA_Tutorial_No_NDA_Apr08. pdf*, 2008.

[36] C. Zeller, "Cuda performance," *ACM SIGGRAPH GPGPU Course Notes*, 2007.

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

[38] 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.

[39] T. Svoboda, "Random sample consensus,"

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

[41] H. Bay, T. Tuytelaars, and L. Van Gool, "Surf: Speeded up robust features," *Computer Vision–ECCV 2006*, pp. 404–417, 2006.

[42] M. Muja and D. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration," in *International Conference on Computer Vision Theory and Applications (VISSAPP'09)*, pp. 331–340, 2009.

[43] P. Rodríguez, D. Mántaras, and C. Vera, *Ingeniería del automóvil: Sistemas y comportamiento dinámico.* Editorial Paraninfo, 2004.

[44] 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.

[45] C. Guindel Gómez, "Algoritmo de odometrí visual estéreo para sistemas de ayuda a la conducción. Implementación en GPU mediante CUDA," 2012.

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

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

[48] 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.

[49] 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.

[50] S. Hold, C. Nunn, A. Kummert, and S. Muller-Schneiders, "Efficient and robust extrinsic camera calibration procedure for lane departure warning," in *Intelligent Vehicles Symposium, 2009 IEEE*, pp. 382–387, IEEE, 2009.

[51] S. Li and Y. Hai, "Easy calibration of a blind-spot-free fisheye camera system using a scene of a parking space," *Intelligent Transportation Systems, IEEE Transactions on*, vol. 12, no. 1, pp. 232–242, 2011.

[52] Q. Wang, Q. Zhang, and F. Rovira-Mas, "Auto-calibration method to determine camera pose for stereovision-based off-road vehicle navigation," *Environment control in biology*, vol. 48, no. 2, pp. 59–72, 2010.

[53] Matrox, "Matrox imaging library." `http://www.matrox.com/imaging/en/products/software/mil/`, 2012. [Online; disponible el 1 de Octubre de 2012].

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