Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería en Electrónica y Automática Industrial

Trabajo Fin de Grado

Configuración y ejecución de algoritmos de visión artificial en la tarjeta Nvidia Jetson TK1 DevKit

Autor: Mario Luis Álvarez Pastor

Tutores: Cristina Losada Gutiérrez y David Jiménez Cabello

2017

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería en Electrónica y Automática Industrial

Trabajo Fin de Grado

Configuración y ejecución de algoritmos de visión artificial en la tarjeta Nvidia Jetson TK1 DevKit

Autor: Mario Luis Álvarez Pastor

Directores: Cristina Losada Gutiérrez y David Jiménez Cabello

Tribunal:

Presidente: Elena López Guillén

Vocal 1º: María Soledad Escudero Herranz

Vocal 2º: Cristina Losada Gutiérrez

Calificación: .....

Fecha: .....

Me gustaría dedicar este trabajo a todas aquellas personas que me han apoyado en todo este tiempo, especialmente a mi madre, que ha hecho posible que esté hoy aquí haciendo este trabajo, a mis tutores, que han tenido infinita paciencia conmigo, a mis amigos, que siempre han estado ahí animandome, sobre todo a David Fuentes, que ha estado conmigo incontables horas trabajando a mi lado...

"¡¿Pero quién dijo que la vida fuera justa?!"
David Fuentes

Agradecimientos

¿Por qué esta magnífica tecnología científica, que ahorra trabajo y nos hace la vida mas fácil, nos aporta tan poca felicidad? La repuesta es ésta: simplemente, porque aún no hemos aprendido a usarla con tino.

Albert Einstein (1879-1955) Científico alemán nacionalizado estadounidense

Este trabajo es el fruto de muchas horas de dedicación, tanto por investigación del funcionamiento de la tarjeta y exploración de sus límites, como por la programación y evaluación de los diferentes algoritmos ejecutados en ella. Es por ello que me gustaría escribir estos agradecimientos.

Mención especial merecen mis tutores Cristina Losada, que ha estado ahí siempre que la he necesitado, en todo momento presta a ayudarme, y David Jiménez, el cuál me ha enseñado muchísimo y me ha ayudado a sobrepasar los problemas surgidos durante el desarrollo de este trabajo. También cabe alabar la paciencia que han tenido conmigo, por todo lo que he tardado en desarrollar este proyecto.

También quiero dar las gracias a Jose, Raquel y Marcos, por la compañía que me han hecho mientras desarrollaba el trabajo, y especialmente a David Fuentes, en el cual no sólo he encontrado un compañero de investigación, sino también una maravillosa persona, divertida y amena, que ha hecho más llevadero el desarrollo de este trabajo, y ha hecho posible la realización de nuevos proyectos de futuro entre los dos.

Finalmente, agradecer a mi madre por haberme ayudado a llegar hasta donde estoy, porque sin ella no hubiera sido posible, y a Google, ya que la mayoría de ayudas y ejemplos los encontré gracias a su magia, y aunque seguro que me he dejado alguna referencia hacia los ejemplos, desde aquí les doy las gracias a todos ellos por compartir su saber con el mundo.

Índice general

Índice de figuras

Índice de tablas

Resumen

En el presente Trabajo Fin de Grado (TFG) se aborda la evaluación de la tarjeta de desarrollo NVidia Jetson TK1. Se trata de una tarjeta orientada a la ejecución de algoritmos de visión artificial a través del cálculo en paralelo mediante la Unidad de Procesamiento Gráfico (GPU) de la tarjeta, que dispone de un SOC (System on a Chip) Tegra K1 el cual incluye una GPU NVidia Tegra y un microprocesador ARM Cortex A-15 entre otros periféricos.

La evaluación de la tarjeta se lleva a cabo desde dos perspectivas diferentes. En primer lugar, se realiza un análisis a nivel de *hardware* para encontrar las ventajas y limitaciones para su uso en aplicaciones de visión artificial, en concreto, se evalúa el el uso de las librerías de OpenCV para visión en estéreo, combinadas con un desarrollo de entorno gráfico en OpenGL. Posteriormente, se comparan los tiempos de ejecución de diferentes algoritmos para evaluar los distintos rendimientos de la tarjeta y de su GPU y CPU (Unidad Central de Proceso).

Palabras clave: OpenCV, NVidia Jetson TK1, GPU, CUDA, Visión en estéreo.

Abstract

This final degree thesis (TFG) adresses the evaluation of the NVidia Jetson TK1 development board. It is a board oriented to the execution of computer vision algorithms using paralel computing on the Graphics Processing Unit (GPU) integrated on the board. The Jetson TK1 includes Tegra K1 SOC (System on a Chip) that integrates a NVidia Tegra GPU and an ARM Cortex A-15 microprocessor among other peripherals.

The evaluation of the development board is carried out from two different perspectives. First, a hardware level analisis is made in order to analyze the advantages and limitations for computer vision applications, specially those that use OpenCV libraries for stereo vision, combined with a OpenGL graphical environment. Then, computation cost are evaluated for different algorithms, so a comparaive of the performance can be made between GPU and CPU (Central Processing Unit).

Keywords: OpenCV, NVidia Jetson TK1, GPU, CUDA, Stereo Vision.

Resumen extendido

El objetivo principal de este trabajo fin de grado es la evaluación de la tarjeta de desarrollo NVidia Jetson TK1 DevKit [4], que se muestra en la figura 2. Se trata de una tarjeta de evaluación diseñada para la ejecución de algoritmos en paralelo gracias a la la Unidad de Procesamiento Gráfico (GPU) integrada.

Figura 2: Tarjeta NVidia Jetson TK1

Para la evaluación de la tarjeta se planteó la implementación y evaluación de un algoritmo para la detección de la posición 3D de la mano, con el objetivo de utilizar dicha posición para el control remoto de un robot. El algoritmo para la detección y posicionamiento 3D de la mano se programó tanto para su ejecución secuencial en la Unidad Central de Proceso (CPU) o microprocesador ARM Cortex-A15 disponible en la Jetson TK1, como para su ejecución en paralelo en la GPU utilizando para ello el lenguaje y herramientas CUDA (Arquitectura Unificada de Dispositivos de Cómputo) proporcionada por NVidia. Además, los resultados del procesamiento, así como la información necesaria debían mostrarse a través de las gafas de realidad aumentada ARvision-3D HMD ([5]), que consta de un par de cámaras estéreo (tal como se puede apreciar en la figura 3), así como dos displays que permiten mostrar imágenes a cada ojo de manera separada y poder generar la sensación de realidad aumentada al usuario final.

Figura 3: Gafas de visión ARvision 3D

Para lograr los objetivos, inicialmente, se propuso realizar la captura de información empleando una cámara de profundidad basada en tiempo de vuelo (ToF) [6], que permite obtener información de la distancia de cada punto en la escena a la cámara, facilitando así la obtención de la posición de las manos. Sin embargo, tras las primeras pruebas se determinó que la cámara elegida (PMD Camboard Nano) no era compatible con la tarjeta, debido a que no existían *drivers* compatibles con el *hardware* integrado en la tarjeta porque faltaba soporte para este tipo de *hardware* en el sistema operativo pre-compilado que incluye. Tras realizar una búsqueda exhaustiva de información, se descartó esta posibilidad.

Dada la imposibilidad de utilizar una cámara ToF, a continuación se propuso utilizar el par de cámaras estéreo disponibles en el casco de realidad aumentada, que contaban con conexión *FireWire*. Dado que la placa no contaba con un contector *FireWire*, fue necesario utilizar un adaptador PCI-express a *FireWire*, e instalar una versión no oficial del sistema operativo de la tarjeta (Linux for Tegra R19.3) que incluía los drivers *FireWire*. Sin embargo, al comprobar el funcionamiento de las cámaras se determinó que aunque se detectaban correctamente, no había comunicación con ellas. Tras el análisis de la información disponible sen concluyó que las cámaras disponibles no eran compatibles con el procesador ARM integrado en la Jetson.

Tras las pruebas anteriores, y debido a la falta de compatibilidad del *hardware* de la tarjeta con las cámaras disponibles, se descartó el procesamiento de la información adquirida en tiempo real por la tarjeta. Por este motivo se realizó la grabación de un conjunto de secuencias de imágenes con dos cámaras *FireWire* conectadas a un ordenador, y sincronizadas entre sí. Para realizar la captura de forma síncrona se ha desarrollado un programa con múltiples hilos para la captura y guardado de imágenes.

De forma previa a la captura de imágenes, se realizó la calibración de las cámaras para obtener sus modelos matemáticos reflejados en los parámetros intrínsecos y extrínsecos. Para ello se desarrolló un programa, haciendo uso de las librerías OpenCV, que permitia tanto la captura de imágenes, como la obtanción de los parámetros de calibración.

Tras la calibración de las cámaras, se realizaron múltiples grabaciones con el objetivo de contar con imágenes que incluyeran diferentes condiciones de grabación (cambios en la luminosidad de la imagen, diferentes colores y texturas de fondo, diferentes distancias entre el fondo y las cámaras, así como entre la mano y las cámaras, etc.)

En paralelo a estas pruebas con las cámaras, se comenzó a desarrollar un programa basado en OpenGL cuyo objetivo era manejar en pantalla completa los resultados obtenidos por los algoritmos de visión. El

programa consistía en la creación de una ventana que sería redimensionada a la resolución específica de las pantallas del casco, para que no se distorsionara la imagen, de forma que la imagen creada tuviera un aspecto realista.

Una vez finalizada esta etapa, se procedió a la evaluación de diversos algoritmo para la extracción de información 3D a partir de imágenes estéreo. Se evaluaron tanto algoritmos programados directamente en CUDA (que permitían obtener un mejor rendimiento en la GPU) como otros implementados utilizando las librerías OpenCV. En concreto se evaluó un algoritmo desarrollado integramente en CUDA, y dos alternativas basadas en OpenCV: *Stereo Block Matching* (Stereo BM) que consiste en buscar la correspondencia entre bloques de píxeles en ambas imágenes y así obtener una imagen de distancias en blanco y negro, y *watershed* (Transformación divisoria), que consiste en dividir la imagen en distintas áreas en función de su distancia.

De las pruebas realizadas se concluyó que el algoritmo desarrollado en CUDA presenta un mejor rendimiento al estar optimizado para su ejecución en GPU, sin embargo, no proporciona resultados adecuados para este trabajo, ya que no permite la configuración de los parámetros utilizados en la extracción de información estéreo.

Por otro lado, las soluciones basada en OpenCV permiten una mayor configuración, obteniendo resultados mejores. Estas propuestas han sido evaluadas tanto ejecutándose de forma secuencial en CPU, como realizando algunas tareas en paralelo gracias a la GPU, obteniéndose tiempos de procesamiento menores en este último caso.

Lista de acrónimos

API	Interfaz de Programación de Aplicaciones.
CPU	Central Processor Unit.
CUDA	Compute Unified Device Architecture.
DRAM	Dynamic Random Access Memory.
FPS	Frames por segundo.
GPGPU	General-Purpose Computing on Graphics Processing Units.
GPU	Graphics Processor Unit.
ISP	In-System Programmable.
MIMD	Multiple Instruction stream, Multiple Data stream.
mini-PCIe	mini-Peripheral Component Interconnect express.
RAM	Random Access Memory.
SATA	Serial Advanced Technology Attachment.
SIMD	Single Instruction stream, Multiple Data stream.
SOC	Sysems on a Chip.
Stereo BM	Stereo Block Matching.
TFG	Trabajo de Fin de Grado.

Capítulo 1

Introducción

+¿Podrá llevar una vida normal?
-No, será ingeniero.

Madre de Dilbert y médico suplente, Dilbert

1.1 Presentación

En el presente *Trabajo de Fin de Grado (TFG)* se realiza un estudio del rendimiento de la tarjeta de desarrollo NVidia Jetson TK1 [4,7,8], mostrada en la figura 1.1 a través de la implementación y ejecución de algoritmos de visión artificial. Así podemos destacar las ventajas que posee la programación en paralelo utilizando la Unidad de Procesamiento Gráfico o *Graphics Processor Unit (GPU)* frente a la programación convencional con el procesador o *Central Processor Unit (CPU)*. Así, durante el desarrollo de este trabajo, se han ejecutado diversos algoritmos en la GPU de la tarjeta, para evaluar las limitaciones y propiedades de la misma.

Figura 1.1: Tarjeta NVidia Jetson TK1

El modelo de programación en paralelo basado en GPU puede definirse como el uso de una unidad de procesamiento gráfico en combinación con una CPU para acelerar aplicaciones. Las GPU aceleradoras han pasado a utilizarse en una gran diversidad de sitios de todo el mundo, ya que aceleran las aplicaciones de plataformas diversas, desde automóviles hasta teléfonos móviles y tablets [9], drones [10, 11], o robots [12, 13].

La diferencia entre la CPU y la GPU a la hora de funcionar es la forma en que procesan las tareas. Una CPU está formada por varios núcleos optimizados para el procesamiento en serie, mientras que una GPU consta de millares de núcleos más pequeños y eficientes diseñados para manejar múltiples tareas simultáneamente.

Los programas que incluyen paralelización presentan dificultades añadidas respecto a los secuenciales, debido a que la concurrencia introduce nuevos tipos de errores de *software*, siendo las condiciones de carrera los más comunes. La comunicación y sincronización entre diferentes subtareas son algunos de los mayores obstáculos para obtener un buen rendimiento del programa paralelo.

La tarjeta de desarrollo que se ha utilizado posee unas características suficientes para la ejecución de los algoritmos propuestos, y se eligió por cumplir con las expectativas del trabajo. La GPU posee 192 núcleos con los que se realizan los cálculos en paralelo, y la CPU es un cuadcore Cortex-A15, que junto con los 2 Gigas de *Random Access Memory (RAM)*, y el almacenamiento interno de la tarjeta nos encontramos ante un pequeño ordenador portable con un gran abanico de posibilidades para desarrollar aplicaciones.

Actualmente, la computación en paralelo es una tecnología cada vez más utilizada, ya que día a día los cálculos necesarios para resolver nuevas ecuaciones, planteamientos, y otras investigaciones, son cada vez más elevados; Una CPU convencional sólo podrá realizar unos pocos cálculos simultáneamente, por lo que el tiempo empleado en realizar una elevada cantidad de operaciones sería excesivo. Por ello, se emplean programas y algoritmos especialmente diseñados para ser procesados por GPUs, las cuales pueden efectuar varias operaciones simultáneamente al poder lanzar un elevado número de hilos, reduciendo considerablemente el tiempo empleado en dichos cálculos.

El lenguaje de programación y herramientas *Compute Unified Device Architecture (CUDA)* [14] estudiado para este proyecto es específico para las GPUs de NVidia, optimizado para funcionar a pleno rendimiento con sus GPUs. Está en constante desarrollo, cada vez con más funcionalidades, y adaptándose al *hardware* cada vez más potente que se va desarrollando. El lenguaje de CUDA posee ciertas ventajas con respecto a otros lenguajes de programación de GPUs, como puede ser, por ejemplo, el acceso a nivel de bit de las variables, lecturas de y hacia memoria más rápidas, lecturas dispersas, o la utilización de memoria compartida para todos los hilos ejecutados en la GPU.

1.2 Objetivos

Como ya se ha comentado, el objetivo principal del presente TFG es la evaluación de la tarjeta de desarrollo NVidia Jetson TK1, comprobando su funcionamiento a nivel de *software* y *hardware* como pueda ser el sistema operativo y su capacidad de procesamiento, y comparando las diferencias de rendimiento entre la GPU integrada de la tarjeta y su CPU. Así, los dos aspectos en los que se ha centrado el trabajo son los siguientes:

- **Evaluación de las características de la tarjeta de desarrollo NVidia Jetson TK1.**

  El primer paso del trabajo realizado consiste en el estudio de la arquitectura de la GPU, y el desarrollo y planteamientos que incluye el lenguaje de programación CUDA para la tarjeta. Debido

a que es un lenguaje de programación dirigido especialmente para este tipo de integrados, tiene unas particularidades que deben ser conocidas para poder programar de una manera correcta sobre los recursos que se ofrecen.

También tienen importancia los conectores disponibles en la tarjeta, lo cual limita el tipo de dispositivos que se pueden conectar y ser utilizados para desarrollar aplicaciones con la tarjeta, como por ejemplo, el poder conectar cámaras para realizar programas que puedan grabar imágenes en tiempo real.

Adicionalmente, las librerías precompiladas por NVidia que se incluyen para el desarrollo de programas en la tarjeta, han de ser investigadas para encontrar las posibilidades y limitaciones de la tarjeta. También habrá que evaluar las librerías con el fin de utilizar la tarjeta para procesar algoritmos de visión artificial.

- **Implementación de algoritmos de procesamiento de imágenes, aprovechando la arquitectura de cálculo paralelo en GPU.**

  En la evaluación del rendimiento se estudian dos algoritmos de procesamiento de imágenes que tratan la visión en estéreo (Visión estereoscópica), primero en código escrito directamente en CUDA (GPU), y posteriormente en C++ con el algoritmo *Stereo Block Matching (Stereo BM)*, escrito en CUDA a través de OpenCV.

  En concreto, ambos se implementan en un programa para valorar el rendimiento de la tarjeta, comparando la utilidad que posee la GPU frente a la CPU. Con ello, se podrá generar una comparativa de los distintos rendimientos en cada caso para el mismo algoritmo.

En los siguientes capítulos de esta memoria se describe en detalle el trabajo realizado. En concreto, en el capítulo 2 se presentan los fundamentos teóricos necesarios pra la realización del TFG. A continuación, en el capítulo 3 se describe el trabajo desarrollado para la implementación y evaluación de los algoritmos en la tarjeta de desarrollo. Algunos de los resultados obtenidos se presentan en el capítulo 4. Finalmente, el capítulo 5 recoge las principales conclusiones y posibles líneas de trabajo futuro.

Capítulo 2

Estudio teórico

La capacidad de hablar no te hace inteligente.

Qui-Gon Jinn, Starwars, Episodio I: La amenaza fantasma

En este capítulo se presentan los fundamentos teóricos necesarios para la realización de este TFG. Para comenzar, se describen las principales características de la tarjeta de desarrollo que se evalua (Nvidia Jetson TK1 [4]). Posteriormente, se describe la arquitectura de cálculo paralelo en GPU destacando las particularidades de la programación en este tipo de dispositivos. Finalmente, se presenta el algoritmo implementado para la evaluación experimental de la tarjeta.

2.1 Arquitectura de cálculo en paralelo en GPU

2.1.1 Introducción

CUDA [14] es una arquitectura de cálculo en paralelo desarrollada por NVIDIA que aprovecha la gran potencia de cálculo de la GPU para proporcionar un incremento extraordinario del rendimiento del sistema.

Desde su introducción en el 2006, CUDA se ha desarrollado a través de miles de aplicaciones y trabajos de investigación publicados, y ha sido soportada e instalada en cientos de millones de dispositivos que poseen GPUs compatibles con CUDA.

En un esquema general sobre CUDA, el usuario tiene diversas formas de acceder al manejo de las GPUs de NVidia. Organizándolo por niveles como se muestra en la figura 2.1, se tiene en un nivel más bajo el propio motor de ejecución de programas en paralelo dentro de la GPU. En el siguiente nivel, se encuentra el soporte que proporciona el kernel del sistema operativo con el que se trabaja. Sobre el sistema operativo se encuentran por un lado el soporte de DirectX para poder llamar directamente al kernel de la GPU, y por otro lado el *driver* de CUDA, que puede ser utilizado directamente programando en CUDA o con una nueva capa de abstracción a través de otros lenguajes o *Interfaz de Programación de Aplicaciones (API)* de programación.

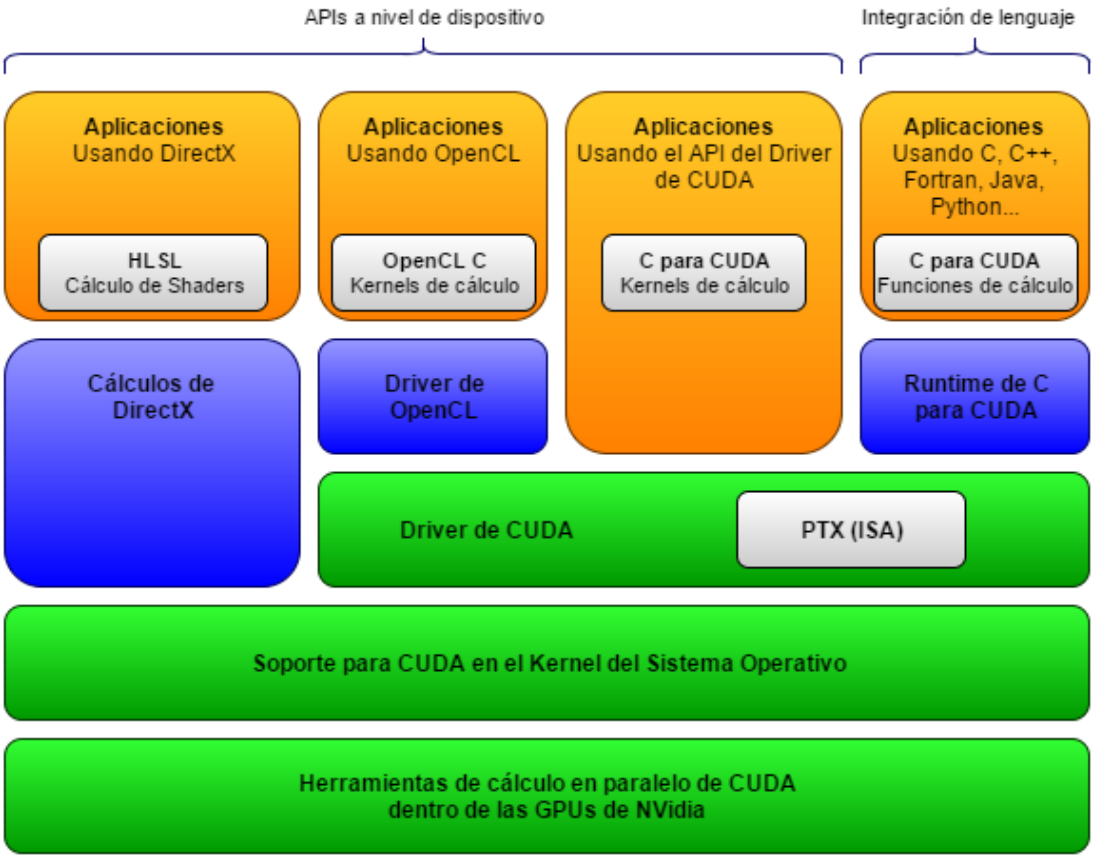


Figura 2.1: Esquema del software para CUDA

2.1.2 Arquitectura general

La arquitectura para el diseño de una GPU está influida por la necesidad de mejorar la capacidad de realizar una gran cantidad de cálculos en coma flotante, especialmente para el procesamiento gráfico. Para optimizar este requisito y obtener un buen rendimiento en este tipo de cálculos, se opta por utilizar un número masivo de flujos de ejecución, o hilos. La estrategia consiste en utilizar estos hilos de tal manera que mientras unos están a la espera para poder escribir en memoria, el resto puede seguir ejecutando tareas pendientes.

En comparación con una CPU, una GPU requiere menos lógica de control para cada flujo de ejecución. Además, cada núcleo de la GPU posee una pequeña memoria caché para permitir a los hilos que se están ejecutando tener un ancho de banda de memoria suficiente para no saturar la *Dynamic Random Access Memory (DRAM)*. Un esquema aproximado de esto se puede observar en la figura 2.2

Algunas características típicas de las arquitecturas GPU son las siguientes:

- Están formadas por un conjunto de multiprocesadores que siguen el modelo *Multiple Instruction stream, Multiple Data stream (MIMD)*, cada uno de los cuales a su vez contiene un grupo de procesadores de tipo *Single Instruction stream, Multiple Data stream (SIMD)*. Se trata de procesadores de tipo vectorial, los cuales permiten ejecutar una misma operación sobre un conjunto o vector de datos simultáneamente.

- La velocidad de ejecución se basa en el aprovechamiento de la localización de los datos, ya sean temporales (Al acceder a un dato, es probable que se vuelva a utilizar el mismo dato en un futuro

cercano) o bien dónde se encuentren (Al acceder a un dato, es muy probable que se utilicen datos adyacentes, y por ello se utilizan memorias caché que guardan varios datos en una línea del tamaño del bus).

- La memoria de una GPU se organiza en diferentes tipos de memoria (local, global, constante y compartida), que tienen diferentes tamaños, tiempos de acceso y modos de acceso, optimizando así la velocidad de procesamiento según cada caso (Como por ejemplo, memorias de sólo lectura o lectura/escritura).
- Las GPUs poseen un ancho de banda de memoria bastante elevado en comparación con una CPU.

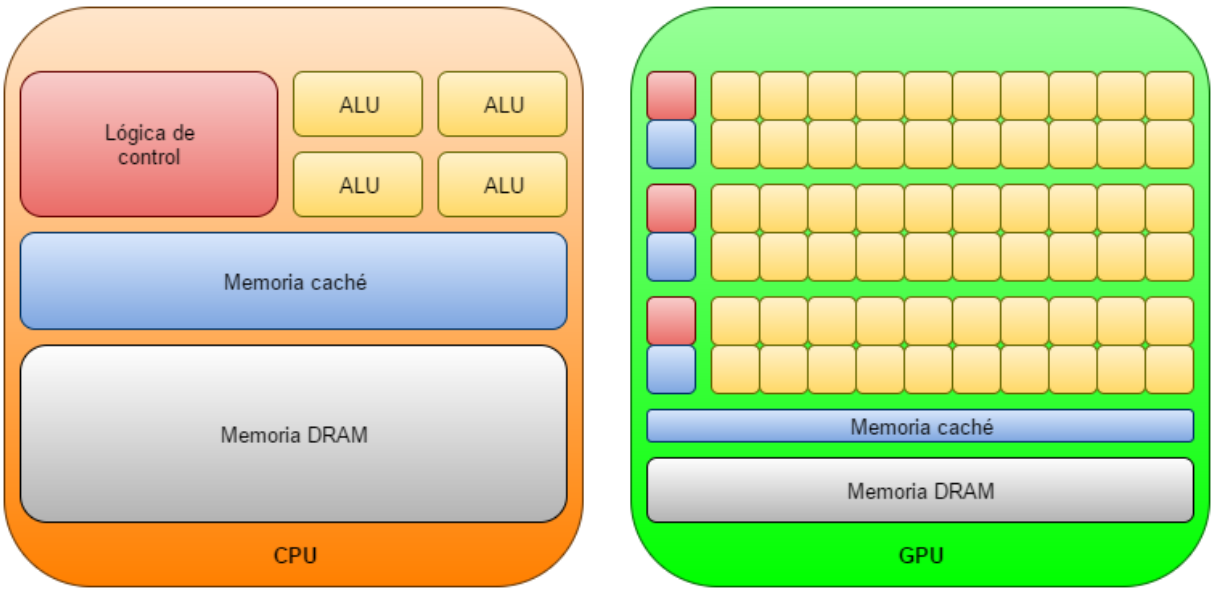


Figura 2.2: Comparativa de la CPU respecto a la GPU en superficie dedicada típicamente a computación, memoria y lógica de control

En contraposición a todas estas ventajas, hay que tener en cuenta que dada la disposición de la CPU y la GPU, esta última no puede acceder a la memoria principal, y la CPU no puede acceder directamente a la memoria de la GPU, por lo que habrá que copiar los datos entre CPU y GPU de manera explícita para ambos sentidos, con sus correspondientes reservas de memoria y transferencia de datos entre ambas.

Además, la capacidad de procesamiento de cada uno de los procesadores de la GPU es inferior a la de una CPU en lo que a frecuencia de operación o de operaciones soportadas se refiere. Siendo así, el rendimiento que se obtiene de la GPU viene dado por el aprovechamiento del gran número de procesadores disponibles para ejecutar un número elevado de hilos que ejecuten un conjunto de operaciones repetidas veces sobre diferentes datos.

2.1.3 Modelo de programación

CUDA es una plataforma que combina *hardware* y *software* que permite a las GPUs de NVidia ejecutar programas escritos en C, C++, Fortran, y una larga lista de lenguajes de programación. Un programa de CUDA invoca unas funciones en paralelo llamadas *kernels* que se ejecutan en muchos y diferentes hilos paralelos.

El programador o el compilador organiza estos hilos en bloques de hilos y a su vez se organiza en una cuadrícula de bloques de hilos, como se muestra en la Figura 2.3.

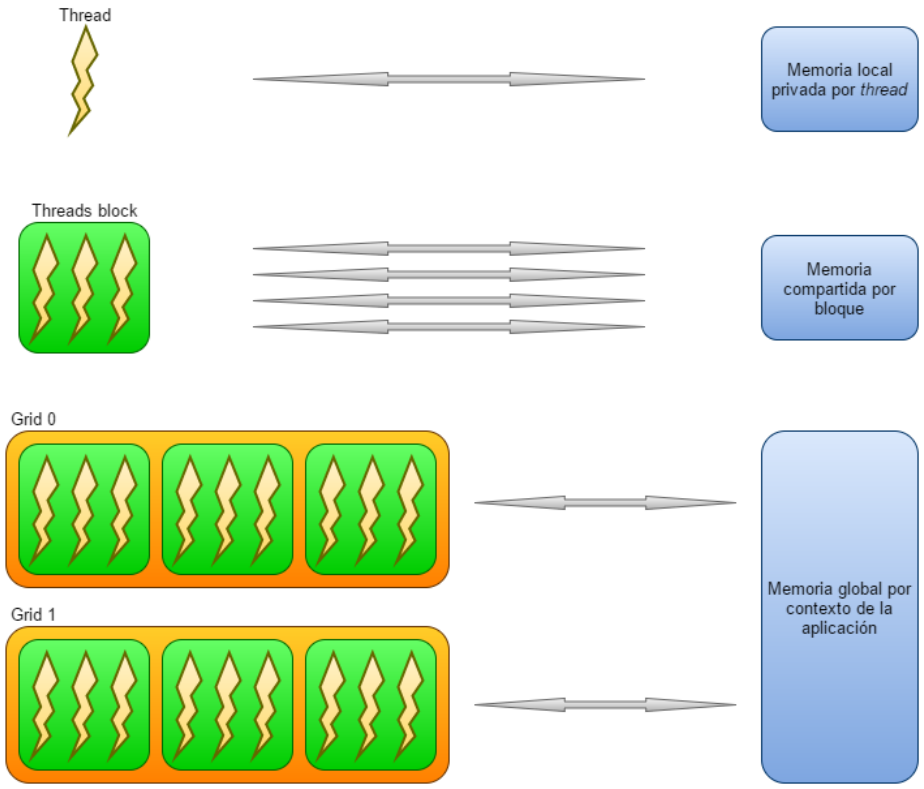


Figura 2.3: Jerarquía de hilos, bloques y cuadrículas en CUDA

Cada hilo (*thread*) de un bloque de hilos ejecuta una instancia del *kernel*. Cada hilo también tiene una identificación de su posición, dentro de su bloque y de su cuadrícula, un contador de programa, registros, memoria privada por hilo, y entradas y resultados de salida.

Un bloque de hilos (*block of threads*) es un conjunto de hilos que se ejecutan concurrentemente y que pueden cooperar entre ellos a través de unas barreras de sincronización y la memoria compartida. Un bloque de hilos tiene una identificación dentro de su rejilla. Si el número de hilos por bloque es demasiado elevado, habrá menos bloques por multiprocesador ejecutándose en paralelo. Existe un máximo de bloques e hilos que un multiprocesador puede manejar físicamente en paralelo. Para un aprovechamiento óptimo de la GPU, se debe tratar de lograr que el número de hilos por bloque sea múltiplo del máximo de hilos que soporta cada multiprocesador, y sea lo suficientemente grande como para que los bloques puedan cubrir todos los recursos disponibles. Es altamente recomendado que sea múltiplo de dos, y que no sea una cantidad pequeña. También se debe intentar disponer de un número de bloques elevado para cubrir los tiempos de latencia, y conseguir así que los multiprocesadores sigan trabajando cuando un bloque queda a la espera por algún motivo. A ser posible, esta cifra debe ser mayor que el doble del número de multiprocesadores del dispositivo.

Una rejilla (*grid*) es un vector de bloques de hilos que ejecutan el mismo *kernel*, leen entradas de la memoria global, escriben los resultados en la memoria global, y sincronizan entre llamadas dependientes del *kernel*.

Para concretar y optimizar más la ejecución de los programas, por debajo de los bloques se encuentran los *warps*, que son una nueva agrupación de hilos que vienen a constituir la unidad básica de ejecución. Actualmente, los *warps* están formados por 32 hilos, lo que implica que una misma instrucción será ejecutada a la vez por grupos de hilos de este tamaño. Cuando esto no sea posible porque distintos hilos dentro del mismo *warp* tengan instrucciones diferentes, cada hilo ejecutará su instrucción en paralelo con

el resto pero en serie con respecto al resto de hilos con instrucciones diferentes, disminuyendo en gran medida el rendimiento y el paralelismo del programa.

En el modelo de programación paralelo de CUDA, cada hilo tiene un espacio de memoria usado para los volcados de registros, las llamadas a funciones, y variables de vectores automáticos de C. Cada bloque de hilos tiene también un espacio de memoria compartida usado para la comunicación entre hilos, compartición de datos, y compartición de resultados para algoritmos en paralelo. Las rejillas de bloques de hilos comparten resultados en el espacio de la memoria global después de la sincronización global por parte del *kernel*.

Desde el punto de vista del programador, el sistema se compone de dos partes: la primera es el procesador principal (*host*), que es la CPU, en nuestro caso, el Cortex-A15 quad-core, que ejecutará el código principal del programa; Y una segunda parte, compuesta por uno o más dispositivos (*devices*), que son las GPUs. En la tarjeta sólo disponemos de una GPU, por lo que el sistema será bastante más simple.

En el esquema de la Figura 2.4 se muestra el flujo de ejecución de un programa que utiliza la GPU para ejecutar parte de su código:

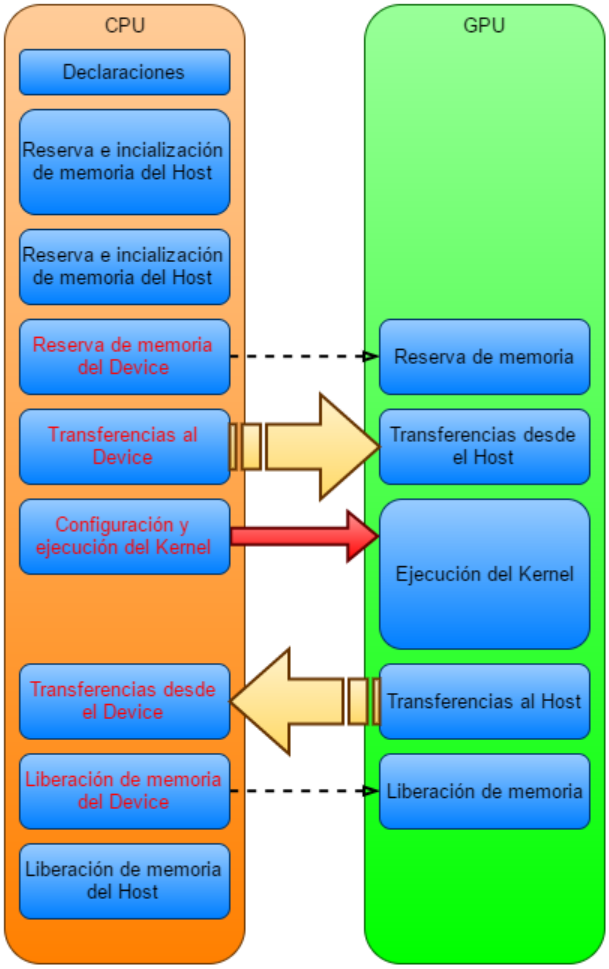


Figura 2.4: Esquema del flujo de ejecución de un programa en CUDA

Debido a que los procesadores de la GPU son mucho menos potentes que los de una CPU, se deben utilizar ambas partes para obtener el máximo rendimiento a un programa, que se verá afectado directamente por la optimización del código en ambos bloques.

Un programa en CUDA consta de varias fases que pueden ser ejecutadas o bien en el *host*, o bien en

los *devices*. Las fases en las que haya muy poco o ningún paralelismo a nivel de datos se implementan para ser ejecutadas en el procesador principal, y las fases con un nivel de paralelismo a nivel de datos elevado se implementan en el código que se ejecuta en los posibles dispositivos disponibles.

Como la GPU es un dispositivo independiente de la CPU, y por tanto de la memoria principal del sistema, necesitará transferencias explícitas de datos con los que poder trabajar en las funciones paralelas, lo que conlleva un retardo de tiempo que afectará en gran medida al rendimiento del programa, aunque sigue siendo menor que el tiempo que se tardaría en ejecutar esa parte de código en una CPU de manera secuencial.

En este proceso, el *host* es el encargado de reservar y liberar memoria en cada dispositivo, además de ser el encargado de hacer las transferencias de memoria entre ambos dispositivos, y determinar cuándo el dispositivo comienza a ejecutar cada uno de los *kernel* que se hayan programado.

2.1.4 Modelo de memoria

De forma general, las GPUs ofrecen seis tipos de memoria diferentes para almacenar los datos, con lo que existirán distintos niveles o ámbitos de visibilidad en la memoria. Estos tipos son los registros (*registers*), memoria local (*local memory*), memoria compartida (*shared memory*), memoria constante (*constant memory*), memoria de teturas (*texture memory*) y memoria global (*global memory*). Un esquema general de la organización de estas memorias se puede observar en la figura 2.5.

Por norma general, toda la memoria de la GPU debe ser tratada manualmente, en el sentido de que las reservas de memoria, liberación de la misma, o transferencia de datos entre GPU y CPU deben ser programadas explícitamente por el usuario.

2.1.4.1 Memoria global

La memoria global de una GPU es la más abundante, la que más latencia tiene, y la que menor ancho de banda maneja. El equivalente en una CPU sería como la memoria RAM, ya que se encuentra fuera del chip que procesa los datos y su principal función es la comunicación de datos entre el *host* y el *device*. Esta transferencia de datos debe ser programada y controlada por el usuario, y el tiempo de vida de esos datos será la duración de todo el programa, en la que cualquier hilo de cualquier bloque tendrá visible y podrá acceder a los mismos.

Los hilos acceden en grupos de *half-warps*, con lo que el mayor rendimiento se obtendrá cuando los hilos accedan de una forma que se unan todos como si fueran un único bloque. Esto ocurrirá cuando se cumplan estas características:

- El tamaño de palabra accedido por hilo sea de 4, 8 o 16 bytes.
- Para que todos los elementos accedan al mismo segmento de memoria, el primer elemento de cada *half-warp* debe estar alineado a un múltiplo de 16 veces su tamaño.
- Cada hilo debe acceder secuencialmente a los datos, esto quiere decir que cada hilo debe acceder a su palabra correspondiente. Pueden existir hilos que no accedan a ningún dato siempre que esto no altere el orden en el que accedan los hilos siguientes.

Si algo de esto no se cumple, entonces el programa no estará optimizado y funcionará haciendo accesos secuenciales de 32 bytes por cada hilo del *half-warp*, lo que conlleva la pérdida del acceso paralelo a la memoria y el mayor retardo posible en los accesos a esta memoria.

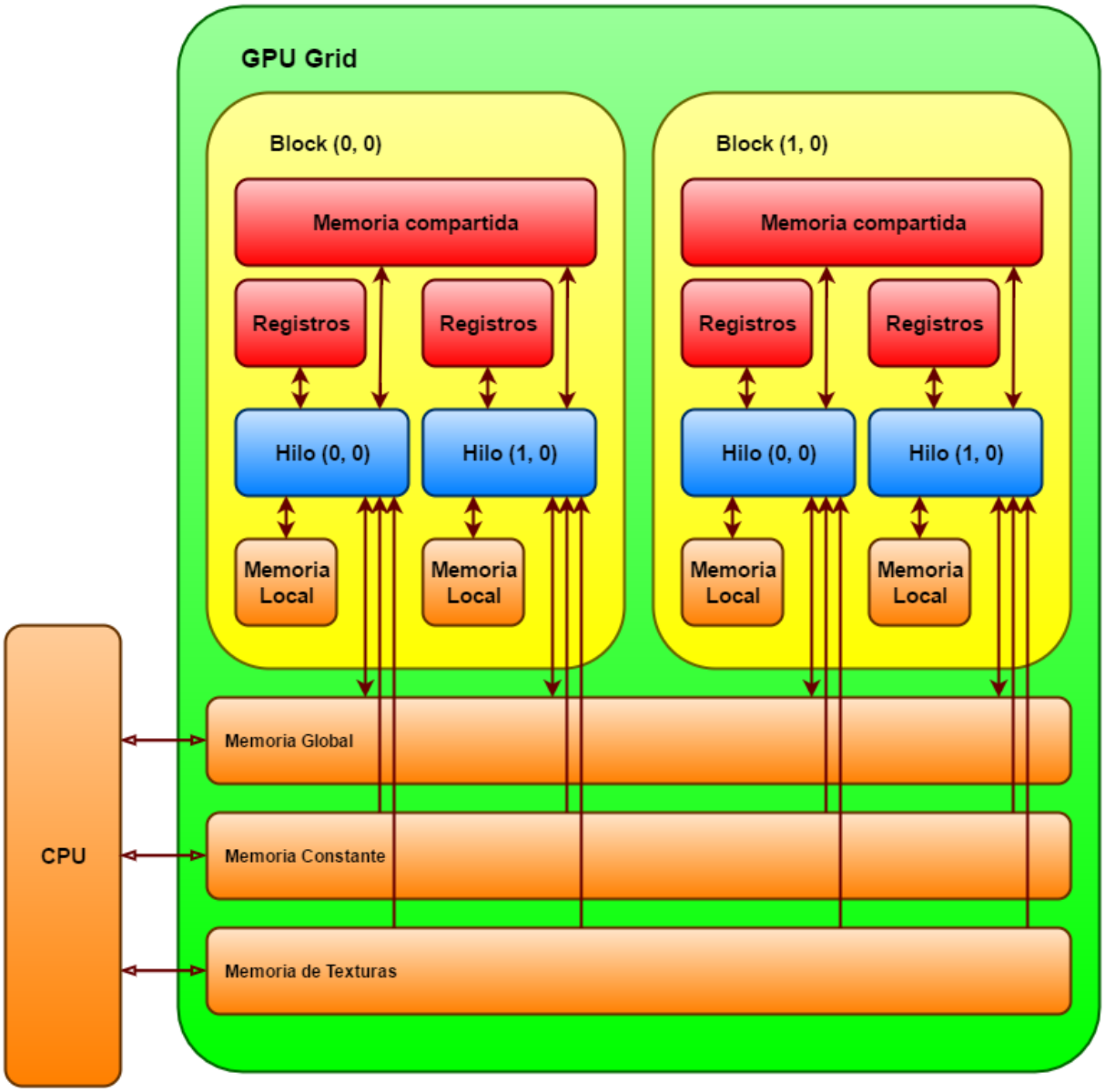


Figura 2.5: Esquema del modelo de memoria para cada tipo de memoria utilizado en CUDA

2.1.4.2 Memoria local

La memoria local es otra posibilidad de uso de la memoria global, ya que no es otro tipo de memoria y se pueden aplicar las mismas características de la memoria global a ésta, con algunas excepciones. La memoria local limita la visibilidad y el tiempo de vida de los datos a nivel de hilo y de *kernel*, haciendo que se libere la memoria ocupada por los datos una vez se han terminado de ejecutar el *kernel*. Normalmente, esta memoria se utiliza como lugar de descarga de los registros o para almacenar estructuras de datos de varios elementos locales a cada hilo. El compilador y el *runtime* de CUDA son los encargados de decidir cuándo y cómo se almacenan los datos para su utilización, de manera que se obtienen los resultados óptimos de acceso.

2.1.4.3 Registros

Los registros son la parte de memoria más rápida de la GPU, junto con la memoria compartida. Se encuentran dentro del chip de procesamiento y los datos que se encuentren en ella su visibilidad y

tiempo de vida son semejantes a los de la memoria local. Estos registros no tienen tanta capacidad de almacenamiento y son mucho más limitados, de forma que cuando no queden registros disponibles se utiliza la memoria local como lugar donde volcar el desbordamiento producido.

2.1.4.4 Memoria compartida

La memoria compartida es la memoria más importante del dispositivo. Como los registros, también se encuentra dentro del chip, pero a diferencia de las anteriores, su visibilidad es a nivel de bloque. Esto implica que todos los hilos del mismo bloque pueden compartir datos utilizando esta memoria. Igual que los registros, es un tipo de memoria limitado, y es el programador el encargado de mover los datos entre esta memoria y el resto de recursos. El tiempo de vida de esta memoria es a nivel de *kernel*, y el retardo de acceso podría compararse a la de los registros siempre que los accesos estén bien optimizados. Igual que en la memoria global, los accesos a memoria se llevan a cabo en grupos de *half-warps*, por lo que se dispone de 16 bancos con un tamaño de palabra de 4 bytes. Cuando varios hilos pertenecientes al mismo *half-warp* intentan acceder al mismo banco, aunque intenten alcanzar posiciones diferentes, se realizarán los accesos en serie, por lo que habrá retrasos en los accesos a esta memoria.

2.1.4.5 Memoria constante

La memoria constante es un tipo de memoria dentro del dispositivo que trata de ofrecer un nivel de caché por encima de la memoria global. Se trata de una memoria de sólo lectura por parte de la GPU, así que los datos introducidos en ella deberán ser transferidos y asociados por el *host* antes de la ejecución del *kernel* que acceda a estos datos. Su visibilidad y tiempo de vida coinciden con los de la memoria global, con la diferencia de que la memoria constante no guarda coherencia con ella.

Los accesos a la memoria constante pueden ser tan rápidos como una lectura de un registro. Para que esto ocurra, todos los hilos de un *half-warp* deben leer la misma posición de memoria. En cualquier otro caso, los accesos a memoria serán en serie, ralentizando nuevamente la ejecución del programa linealmente con el número de posiciones de memoria diferentes solicitadas por el mismo *half-warp*.

2.1.4.6 Memoria de texturas

La memoria de texturas se utiliza nativamente para leer texturas de imágenes en aplicaciones gráficas. Esta memoria pretende servir de una manera similar a la memoria constante, pero con unas restricciones de acceso distintas. Esta memoria se organiza en dos niveles y está diseñada para optimizar accesos de datos colocados en dos dimensiones. De esta manera, se consigue que los hilos de un mismo *half-warp* que lean los datos de esta memoria obtengan mejores tiempos de acceso.

2.1.5 Estructura del programa

Como se puede observar en la figura 2.4, la estructura de un programa basado en CUDA consta de una serie de pasos que se repiten cada vez que quiera ejecutarse parte del código en la GPU:

- Reserva de memoria en el dispositivo.
- Transferencia de datos entre el *host* y el espacio de memoria reservado en el *device*.
- Invocación del *kernel* en el dispositivo.

- Transferencia de los datos entre el *device* y el *host*, y liberación la memoria del *device* si ya no es necesaria.

El entorno de programación de CUDA posee unas funciones que simplifican estas tareas de transferencia de memoria, siendo necesario solamente unos pocos parámetros como los datos a copiar en el dispositivo, o qué datos recuperar del mismo.

Hay que destacar que la invocación de un *kernel* es asíncrona, lo que implica que una vez realizada la llamada al *kernel* la CPU seguirá ejecutando código. Para esperar a que la GPU termine de hacer los cálculos antes de seguir ejecutando el código, habrá que hacer una sincronización entre hilos que actúe de barrera hasta que todos los hilos del *kernel* hayan terminado su ejecución, y se pueda extraer de la memoria del dispositivos los resultados obtenidos.

2.1.6 Ventajas y dificultades

Como se ha ido describiendo a lo largo de los apartados del presente trabajo, CUDA posee determinadas ventajas sobre otros tipos de programas en GPU utilizando aplicaciones gráficas. Entre ellas, cabe destacar:

- Lectura de memoria dispersa: se puede acceder a cualquier posición de memoria.
- Memoria compartida: CUDA posee una memoria especial de una capacidad de 16KB (48KB en las GPU de la serie Fermi) que se comparte entre hilos. La principal característica de esta memoria es la velocidad, por lo que se puede utilizar de caché.
- Lecturas más rápidas de y hacia la GPU, debido a su estructura y *hardware* y *software* especialmente diseñados para ello.
- Soporta enteros y operaciones a nivel de bit.

A pesar de estas buenas características, también existen carencias que se tienen que tener en cuenta a la hora de programar en CUDA:

- La carencia más importante es que no se puede utilizar la recursividad, variables estáticas dentro de funciones, punteros a funciones, o funciones con número de parámetros variable. Todo debe ser fijado antes de la compilación.
- CUDA no soporta el renderizado de texturas.
- Los números desnormalizados o NaNs no están soportados en precisión simple.
- Las latencias y los anchos de banda de los buses entre la CPU y la GPU pueden generar un cuello de botella a la hora de acelerar la velocidad de ejecución de la aplicación.
- Los hilos de ejecución, por eficiencia, deben lanzarse en grupos múltiplos de 32.

2.2 Visión estéreo para la obtención de información de profundidad

2.2.1 Introducción

La visión en estéreo es una técnica capaz de recopilar información visual tridimensional para crear una ilusión de profundidad mediante un conjunto de pares de imágenes, tomadas con dos cámaras ligeramente desplazadas entre sí. Esta técnica trata de relacionar los píxeles correspondientes en las imágenes de derecha e izquierda, creando así información de profundidad a partir de las dos imágenes.

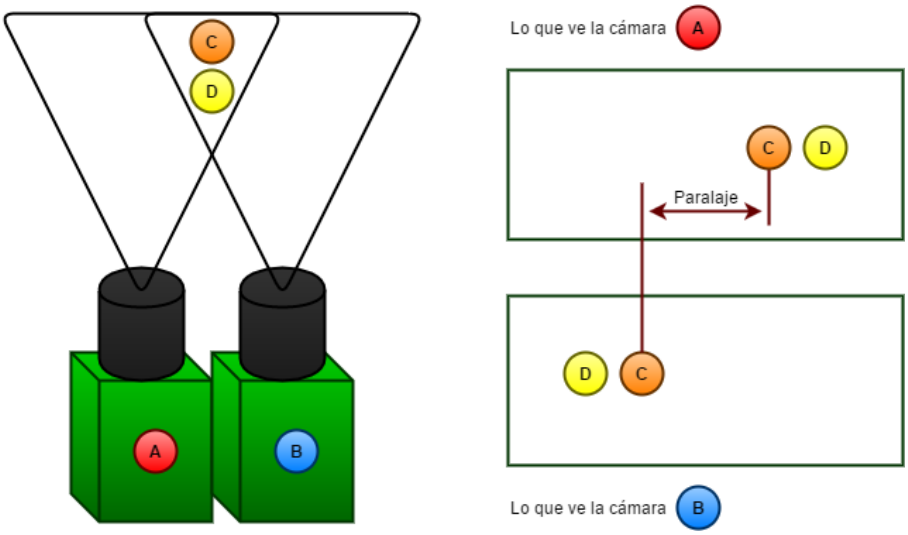


Figura 2.6: Colocación de dos cámaras para realizar visión en estéreo

Dos puntos diferentes capturados en imágenes capturadas con diferente punto de vista, pueden colocarse en correspondencia, teniendo así los dos puntos en posiciones diferentes dentro de la misma línea de vista entre las dos imágenes correspondidas, con lo que se puede determinar un punto 3D en el espacio (Figura 2.6).

2.2.2 Base teórica

Los principales problemas a resolver de la visión en estéreo son dos:

- **El problema de la correspondencia**

  Consiste en encontrar los puntos correspondientes, ya que cada proyección de esos puntos vendrá dado por un punto 3D.

  La visión en estéreo se reduce a un simple problema de triangulación (Figura 2.7), pero hay que tener en cuenta que es crucial encontrar estos pares de puntos que nos dicen cómo se encuentra el punto observado con respecto a las cámaras. Muchas veces, la correspondencia ambigua entre dos puntos puede dar lugar a la interpretación incorrecta del entorno, obteniendo diferentes posibles escenas, o incluso no hallar correlación alguna y no obtener ningún dato de ciertas zonas.

  Para la reconstrucción de la escena a partir de imágenes estéreo, es necesario encontrar la disparidad entre los puntos de ambas imágenes, entendiendo disparidad como la diferencia entre la posición de un determinado punto en cada una de las imágenes del par. Para ello, se utilizan unas subimágenes pertenecientes a una de ellas, y se busca la relación con la segunda imagen por medio de un proceso

de correlación. Es importante que el punto que se quiera encontrar tenga la misma intensidad en cada imagen, lo que quiere decir que las superficies sean mates, y las cámaras estén bien calibradas, ya que si no por culpa de los brillos y diferencias de intensidad puede no encontrarse disparidad alguna.

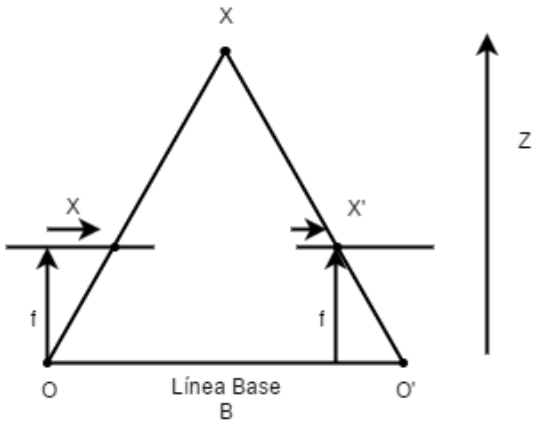


Figura 2.7: Problema de triangulación para la visión en estéreo

El proceso sería el siguiente:

- En una de las dos imágenes, izquierda por ejemplo, se considera una subventana alrededor de un punto (o píxel).
- Se busca esta región en la imagen contraria (derecha), que está asociada a un píxel en la imagen anterior (izquierda).
- Para cada píxel en la primera imagen (izquierda), se busca el desplazamiento del mismo en la segunda imagen (derecha), buscando una "norma", una correlación entre los píxeles de ambas imágenes.

  Si además de esto se quiere acelerar el cálculo, es necesario fijar un límite para las comparaciones, además del tamaño de la ventana a comparar. Esto conlleva que una ventana demasiado pequeña provocará falta de información, pero una ventana excesivamente grande causará que el tiempo que se tarda en realizar los cálculos se incremente notablemente.

  La mayoría de estos problemas sólo pueden resolverse por medio de la observación y la valoración de los resultados obtenidos.

Ahora bien, estos pasos plantean una serie de complejidades, como por ejemplo, cómo determinar la ventana inicial, teniendo en cuenta que puede elegirse una ventana que no esté presente en la segunda imagen, o la distancia desde la que se empieza a buscar.

- **El problema de la reconstrucción**

  Una vez se haya obtenido cada punto correspondiente en las imágenes, podemos representar el mapa de disparidad, que incluso puede ser convertido en una representación 3D del entorno.

  La reconstrucción de la imagen no es más que recuperar la profundidad de la escena observada, que se interpreta a partir de la disparidad de las imágenes, la cual es la diferencia en la posición entre los puntos correspondientes de las dos imágenes.

  El problema que se encuentra en la reconstrucción, es el posible error de que los puntos no estén bien correspondidos en cada imagen, lo que levará a un inevitable error en el mapa de disparidad.

Todo esto se basa en la geometría epipolar [15]. Consiste en hallar las relaciones geométricas que existen entre los puntos en 3D y sus proyecciones en 2D en las imágenes que determinan los parámetros de este método.

Para ello, el principio de esta visión binocular es la triangulación: Dada una única imagen, la localización tridimensional de cualquier punto de un objeto visible está determinada por una línea recta que pasa entre ese punto y el centro de la proyección en la imagen. El hallar la intersección de dos de estas rectas generadas en dos imágenes independientes es lo que se conoce como triangulación.

Esta forma de determinar un punto de la escena a través de la triangulación depende de emparejar la localización de un punto de una imagen en la localización de su mismo punto en la otra imagen. Este proceso de emparejamiento se conoce como correspondencia, y requiere buscar en toda la imagen, pero la geometría epipolar impone unas restricciones que permiten buscar únicamente en una sola línea.

El epipolo es el punto de la intersección de la línea que une los dos centros ópticos, que es la línea base, con el plano de la imagen. Así el epipolo en la imagen de una cámara es el centro óptico de la otra.

El plano epipolar es el plano formado por tres puntos, que serían cada centro óptico de cada imagen, y el punto en 3D en concreto que se esté tratando.

La línea epipolar es la intersección entre el plano epipolar y el plano de la imagen. Todas las líneas epipolares intersecan con el epipolo.

De esta manera, un punto en una imagen genera una línea epipolar que corresponderá con otra línea epipolar en la otra imagen donde debe estar su punto correspondiente, limitando así la búsqueda en una región a una línea.

En un entorno de cámaras calibradas, esta representación algebraica se conoce como matriz esencial. En uno que no esté calibrado, se conoce como matriz fundamental. Ahora bien, con dos puntos de vista, los dos sistemas de coordenadas están relacionados con unas matrices de rotación R y traslación T.

Dado un punto de una imagen **x** se puede hallar su correspondiente punto **x'** con una ecuación que relaciona ambas imágenes.

Las matrices esencial y fundamental tienen las siguientes propiedades:

- La matriz fundamental contiene tanto los parámetros intrínsecos como los extrínsecos, mientras que la esencial sólo contiene los extrínsecos.
- La matriz esencial es una matriz 3×3 con solo 5 grados de libertad. Para estimarla usando puntos correspondientes en las imágenes, los parámetros intrínsecos de ambas cámaras deben ser conocidos.
- La matriz fundamental se encarga de transformar puntos en sus correspondientes líneas epipolares.
- La matriz fundamental se encarga de transformar los epipolos al origen de de su correspondiente plano de imagen.
- La matriz fundamental tiene 7 grados de libertad. Tiene 9 elementos en ella, pero sólo su relación es significativa, lo que le deja 8 grados de libertad. Además, la limitación de que su determinante sea 0 le deja sólo 7 grados de libertad.

2.2.3 Visión estéreo en paralelo utilizando CUDA

Para resolver un problema de visión en estéreo, existen diferentes algoritmos, como podemos observar en la Figura 2.8 (extraída de [2]) que resuelven los problemas mencionados anteriormente. La diferencia entre

unos y otros, son las operaciones y comprobaciones que se realizan, y por tanto la elección del algoritmo que se va a utilizar afectará notablemente al tiempo que se tardan en ejecutar todos los cálculos. El hecho de pasar estos cálculos a un código que pueda ejecutarse en paralelo, no hará más que acelerar el proceso, pero los resultados obtenidos serán los mismos que si el programa se hubiera ejecutado secuencialmente.

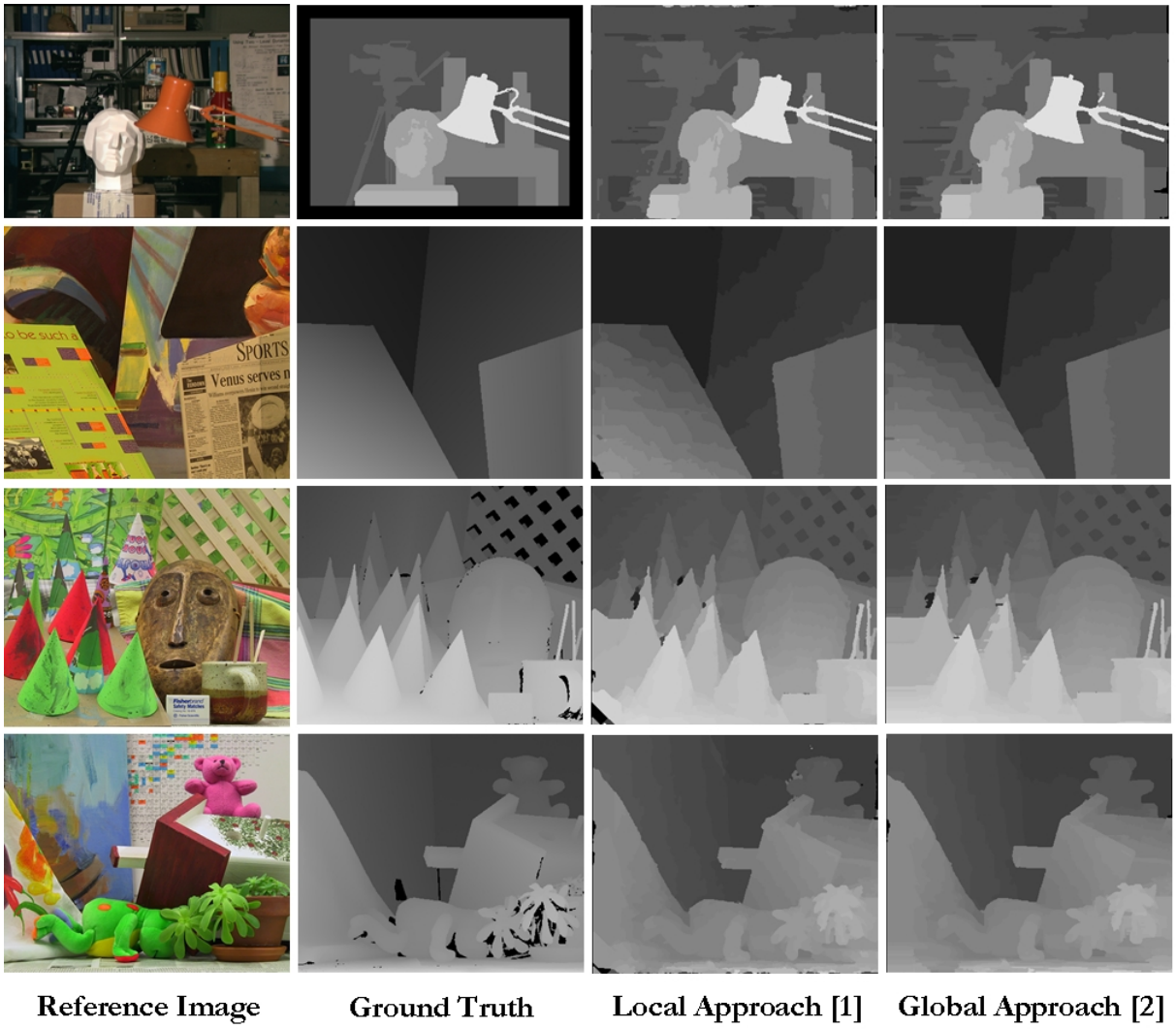


Figura 2.8: Distintos tipos de algoritmos de visión en estéreo, cada uno con diferentes características de imágenes por segundo y precisión alcanzada en ellas. *Local approach* [1] y *Global approach* [2]

En la Figura 2.8 se pueden apreciar algunos ejemplos de resultados de dos algoritmos diferentes. El *ground truth* es el resultado ideal de la imagen de disparidad, en este caso, los valores de distancia obtenidos desde las imágenes en estéreo con la distancia que debería de haber en la realidad. El *local approach* [1] consiste en buscar la correspondencia entre un píxel y un pequeño entorno a él, con unos píxeles y entornos alineados al píxel de referencia en la otra imagen, para así hallar el píxel correspondiente y por tanto poder calcular la imagen con la que observaremos las distancias entre los objetos. Por último, el *global approach* [2] es igual que el local, con la diferencia de que intenta buscar el píxel que mejor que corresponda en la otra imagen sin necesidad de estar alineado, aunque en un entorno cercano al del píxel original.

En el caso de este trabajo, se utilizan las librerías de OpenCV, que poseen una clase específica para realizar el algoritmo de *Block Matching* (*Local Approach* en la Figura 2.8) y ejecutarlo en la GPU de la tarjeta. Este algoritmo es más limitado que el de su versión en CPU, ya que no admite la modificación de

algunos parámetros específicos, porque no está el código escrito para ellos, pero también es el algoritmo más rápido, ya que con él se obtienen más imágenes por segundo que con los otros dos, a costa de perder precisión en las disparidades obtenidas.

2.3 Tarjeta de desarrollo NVidia Jetson TK1

2.3.1 Introducción

En este capítulo del trabajo se exponen las principales características de la tarjeta sobre la que se va a trabajar, así como las ventajas y limitaciones que posee respecto a otras alternativas.

Se justifican además los motivos por los que finalmente se ha utilizado esta tarjeta en lugar de otra.

2.3.2 Características del *hardware* y *software* de la tarjeta de desarrollo NVidia Jetson TK1

La tarjeta Jetson TK1 es una plataforma embebida de NVidia con Linux que posee un *Sysems on a Chip (SOC)* que integra todos los módulos que componen una computadora en un único circuito integrado o chip, lo que implica mayor capacidad de memoria, velocidad de comunicación entre los módulos, y menor coste de producción. Esta tarjeta posee un Tegra K1 (CPU+GPU+*In-System Programmable (ISP)*). La Jetson TK1 viene con el sistema operativo Linux4Tegra pre-instalado (Ubuntu 14.04 con drivers pre-configurados).

Además de la CPU quad-core 2.3GHz ARM Cortex-A15 y de la revolucionaria GPU Tegra K1, la placa Jetson TK1 incluye características similares a una Raspberry Pi [16], convirtiéndolo en un mini-ordenador portable con unas características sorprendentes para su tamaño y consumo de energía, y también algunas funcionalidades orientadas a ordenador como *Serial Advanced Technology Attachment (SATA)*, que es es una interfaz de transferencia de datos entre la placa base y algunos dispositivos de almacenamiento, como la unidad de disco duro, de disco óptico, de estado sólido u otros dispositivos de altas prestaciones. También posee *mini-Peripheral Component Interconnect express (mini-PCIe)*, que es un bus de comunicación serie de alta velocidad para conectar periféricos directamente a la placa base, y un ventilador para permitir un funcionamiento continuo incluso con grandes cargas de trabajo.

La figura 2.9 presenta las principales partes en que se divide la NVidia Jetson TK1:

Las siguientes señales están disponibles a través del puerto de expansión con el conector de 125 pines de 2mm de separación:

- **Puertos de cámara:** 2 puertos fast CSI-2 MIPI (uno 4-lane y otro 1-lane)
- **Puertos de LCD:** Panel de LVDS y eDP Display
- **Puertos de pantalla táctil:** Touch SPI 1 x 4-lane + 1 x 1-lane CSI-2
- **UART**
- **HSIC**
- **I2C:** 3 puertos
- **GPIO:** 7 pines GPIO a 1,8V. Los pines de cámara CSI también pueden utilizarse para GPIOs extra si no se usan ambas cámaras

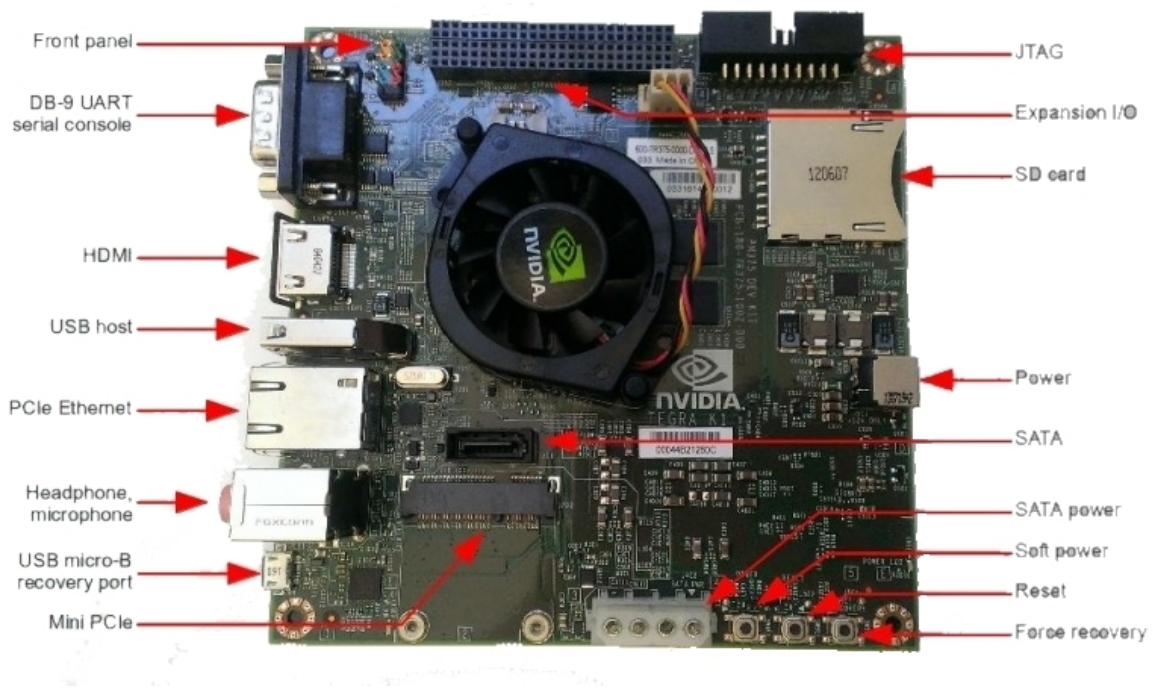


Figura 2.9: Partes de la NVidia Jetson TK1 [3]

Conector de panel frontal:

- **Verde:** LED de encendido
- **Naranja:** LED del HDD
- **Rojo:** Botón de encendido
- **Morado / azul:** Botón de reset

APIs soportadas con aceleración hardware:

- **CUDA 6.0** (SM3.2, como la versión de escritorio SM3.5)
- **OpenGL 4.4**
- **OpenGL ES 3.1**
- **OpenMAX IL multimedia:** códec que incluye H.264, VC-1 y VP8 a través de Gstreamer
- **NPP:** NVidia Performance Primitives (NPP) optimizadas para CUDA
- **OpenCV4Tegra:** NEON + GLSL + optimizaciones quad-core CPU
- **VisionWorks**

2.3.3 Ventajas y limitaciones

Una de las principales ventajas de la tarjeta de desarrollo NVidia Tegra TK1 es que está diseñada para usos móviles y por ello contiene numerosos sistemas de reducción de consumo para controlar cuándo distintas partes del *hardware* deben ir más rápido o más despacio, o incluso apagarse, basado en el uso

de ejecución. Esto es positivo para la mayoría de los casos, y la configuración por defecto dará un alto rendimiento para muchos proyectos intensos y un bajo consumo para muchas tareas ligeras.

Pero lo realmente interesante de esta característica es que se puede forzar un menor o mayor rendimiento a algunas partes del *hardware* como para ejecutar las aplicaciones a máximo rendimiento, o hacer cumplir con un menor consumo de energía. El encendido y apagado automático de los 4 núcleos de las CPUs principales y el quinto núcleo de compañía se hace principalmente en el *kernel* L4T usando *cpuquiet*, un mecanismo para dinámicamente encender o apagar núcleos de la CPU basado en su carga de trabajo o en las directrices que le ordenemos.

Sobre la GPU de la tarjeta, el SOC de la NVidia Tegra K1 proporciona un excelente rendimiento de *General-Purpose Computing on Graphics Processing Units (GPGPU)* por vatio, incluso mejor que la mayoría de CPUs o GPUs ya sean para móvil, ordenador personal, o superordenador.

Sin embargo , su diseño favorece las operaciones de coma flotante de 16 bits que pueden limitar su utilidad para las tareas de GPGPU que por lo general necesitan 32 bits, y a veces la precisión de coma flotante de 64 bits.

La tarjeta de desarrollo NVidia Jetson TK1 posee unas características de *hardware* deseables para la ejecución de los algoritmos propuestos en la GPU, y se eligió por cumplir con las expectativas del trabajo. En concreto, la GPU posee 192 núcleos con los que se realizan los cálculos en paralelo ejecutando el algoritmo de visión en estéreo, y que suponemos suficientes para mediar con las imágenes y poder dar una velocidad asequible al programa; la CPU es un cuadcore Cortex-A15, pudiendo así tener bastante libertad a la hora de utilizar programas multihilo para realizar diversas tareas simultáneamente, conjuntamente con la CPU. Además, cuenta con 2 Gigabytes de RAM, y el almacenamiento interno de la tarjeta permite guardar gran cantidad de imágenes.

Capítulo 3

Desarrollo

La cosa más útil que puedes hacer es cometer un error.
No puedes aprender nada siendo perfecto.

Adam Osborne

En este capítulo se aborda el desarrollo del trabajo, explicando la obtención de las imágenes y el manejo de su programa, al igual que el proceso de calibración llevado a cabo. Se comienza explicando la estructura y funcionalidades del programa principal, para posteriormente definir las diferentes aplicaciones del mismo.

3.1 Algoritmo desarrollado

Como ya se ha comentado, para la evaluación de la tarjeta de desarrollo NVidia Jetson TK1 DevKit, se ha implementado el algoritmo Stereo BM de OpenCV, que consiste en la comparación de bloques de píxeles entre dos imágenes, tomadas por dos cámaras apuntando en la misma dirección y próximas entre sí, para encontrar una correspondencia entre estos bloques, y con ella obtener la distancia aproximada a la que se encuentran los objetos que se pueden ver en ambas imágenes.

En la figura 3.1 se muestra un esquema general del proyecto desarrollado, especificando qué elementos se han implementado en CPU y cuáles en la GPU.

El algoritmo Stereo BM se ejecuta en la GPU de la tarjeta, el cual se llama desde una función en un bucle en la CPU, con el que se van eligiendo los siguientes pares de imágenes para poder hacer la visión en estéreo en tiempo real, y se muestran los resultados por pantalla. Dado que no se pudo realizar esto último en tiempo real por medio de la captura de imágenes, ya que las cámaras no eran compatibles con la tarjeta, se simula de manera que una vez capturadas todo un conjunto de pares de imágenes (pero no un vídeo), el programa carga en memoria pares de imágenes y ejecuta el algoritmo para mostrarlo por pantalla antes de cargar el siguiente par de imágenes.

Todo el programa funciona en un bucle de OpenGL, utilizado para desarrollar una interfaz gráfica para la aplicación, pudiendo crear un ejecutable del programa en pantalla completa, y en un futuro el poder hacer añadido el uso de texturas para la creación de un programa de realidad virtual sobre las imágenes mostradas como resultado de los algoritmos. Esto hubiera sido especialmente interesante de haber sido compatible la tarjeta con el casco de realidad virtual que disponíamos en el laboratorio, el

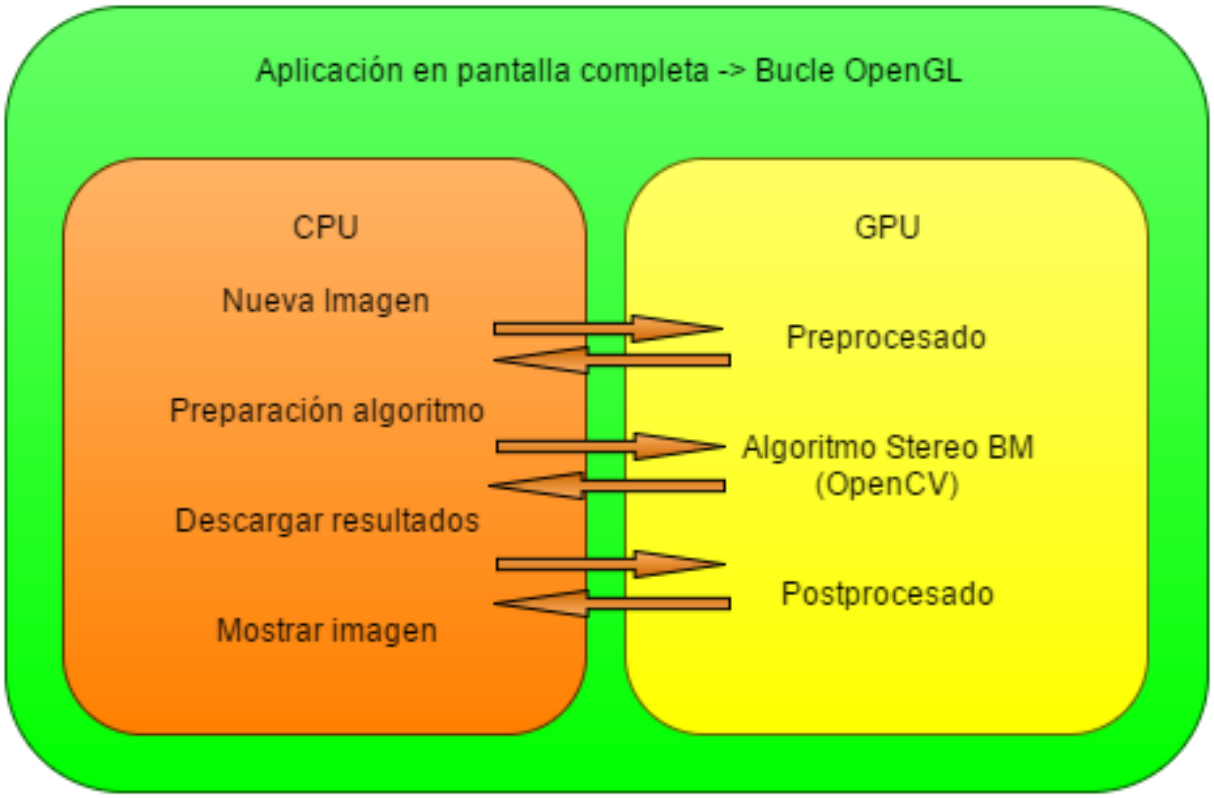


Figura 3.1: Esquema general del proyecto

cual posee dos cámaras frontales junto con dos pantallas a modo de visor con el que se podría haber creado una aplicación de realidad virtual.

Todo el trabajo se ha desarrollado en la tarjeta NVidia Jetson TK1 previamente descrita, debido a las características disponibles en un tamaño reducido que permitían pensar en desarrollar un sistema portable.

El programa que se ha escrito para la implementación de este algoritmo se describe a lo largo de este capítulo, desglosando el contenido de cada uno de sus archivos con sus funcionalidades para formar el conjunto de la aplicación.

3.2 Captura de imágenes estéreo

Para llevar a cabo el proceso de obtención de profundidad a partir de imágenes estéreo, el primer paso ha sido la instalación y calibración de las cámaras para la captura de dichas imágenes. Para ello, se han utilizado dos cámaras industriales con conexión *FireWire* cuyas características se detallan en el apartado 3.2.1. Una vez instaladas en la posición correcta se procedió a la calibración de las mismas tal como se describe en el apartado 3.2.2.

3.2.1 Características de las cámaras

Las cámaras que se han utilizado (figura 3.2, [17]) son unas cámaras industriales con conexión por *FireWire* de las que cabe destacar las siguientes especificaciones:

- **Sensibilidad:** 0.3 lux

- **Rango dinámico:** 8 bits
- **Formato de vídeo a máximas imágenes por segundo:** 640x480 (0.3 MP) Y800 a 60 fps (*frames* por segundo)
- **Interfaz:** FireWire 400
- **Tipo de sensor:** CCD
- **Alimentación:** Entre 8 V y 30 V DC
- **Tipo de obturador:** Obturador global
- **Tiempo del obturador:** Entre 0.1 ms y 30 s
- **Ganancia:** Entre 0 dB y 36 dB

Figura 3.2: Cámaras utilizadas

3.2.2 Calibración de las cámaras

El primer paso, antes de implementar y llevar a cabo la calibración por *software*, es el ajuste de los diferentes parámetros físicos de las cámaras, con las lentes que vienen incorporadas que permiten variar el zoom, enfoque y apertura. Para ello, con la ayuda de un ordenador, se monitorizaron las dos cámaras, y se fueron variando los discos giratorios que poseen las lentes, hasta obtener imágenes nítidas, y lo más similares posibles entre si, teniendo en cuenta que las dos tengan los mismos niveles de zoom, enfoque y aertura. La configuración final seleccionada fue: sin zoom, con enfoque infinito y apertura media, colocando los discos en unas posiciones similares, aunque no coincidieran exactamente entre ellas.

A continuación, fue necesario preparar un tablero de damero en blanco y negro, para tomar imágenes por medio de un programa de captura de imágenes, en las que aparecerá el damero en diferentes posiciones e inclinaciones, como se muestra en los ejemplos de la Figura 3.3. Se utiliza este tipo de imágenes para calibrar las cámaras porque cualquier distorsión y detalle erróneo que pueda aparecer en las imágenes se aprecia fácilmente viendo cómo las líneas y los recuadros del tablero no siguen la realidad, como por ejemplo curvándose.

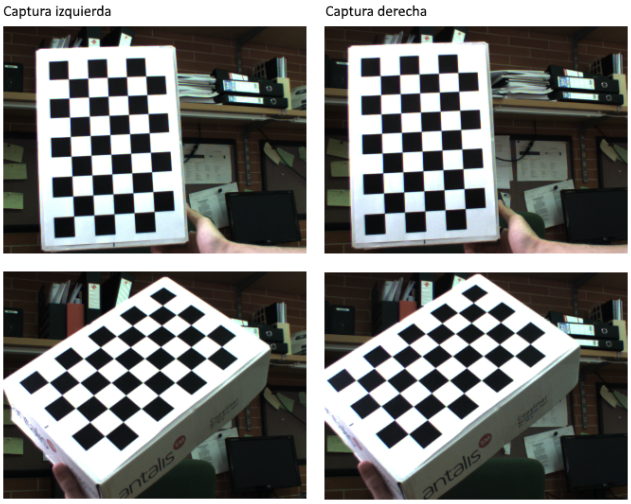


Figura 3.3: Ejemplos de capturas tomadas por las cámaras del damero para la calibración de las mismas

Para poder calibrar las cámaras, se escribió un nuevo programa para la extracción de las características esenciales de las imágenes con el objetivo de conocer cuáles son las diferentes distorsiones de las mismas, y finalmente, las matrices que corrigen y manejan las diferencias entre las cámaras. El código fuente correspondiente a este programa se encuentra en el apéndice A.3.

Los progresos del programa para calibrar las cámaras, como por ejemplo encontrar las esquinas de cada cuadro del damero, pueden observarse también, como se muestra en la imagen de la Figura 3.4.

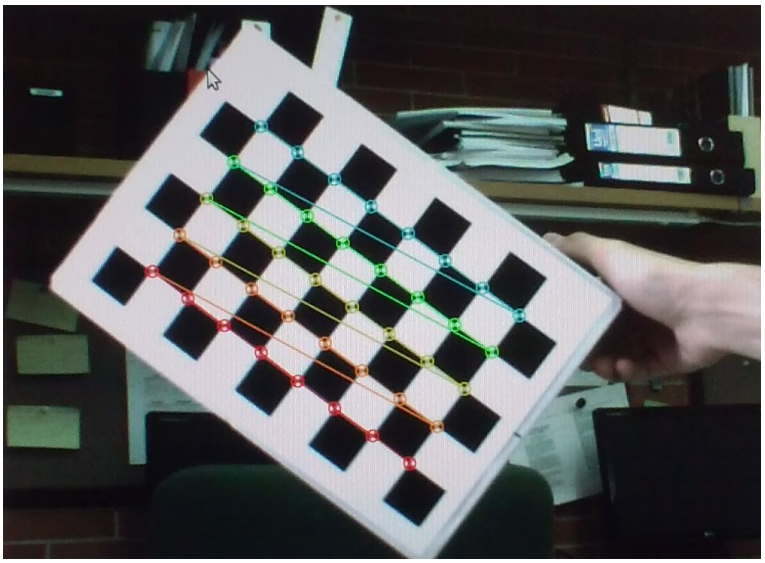


Figura 3.4: Esquinas del damero encontradas marcadas por círculos y unidas por orden

Tras la calibración, los parámetros obtenidos son las matrices de parámetros intrínsecos de cada cámara, y las de parámetros extrínsecos que relacionan ambas. Estas matrices se emplean en el programa final para corregir la distorsión de cada cámara, y determinar la relación geométrica entre ellas. Los resultados arrojados por el programa de calibración desarrollado se presentan a continuación:

```
Right camera calibration
```

```
21 right images found
There are no more right images

Right camera matrix: INTRINSIC

[1123.252225105901, 0, 339.4456657680444;
0, 1123.971343979718, 226.5911398268449;
0, 0, 1]

Right camera matrix: EXTRINSIC

[-0.1242607410556098, 3.115517586917172, 0.0002284733690985072,
-0.003829718740541797, 14.83251027386601, 0.1406434860041577,
0.3991843519067836, 37.86442045489201]

Right camera rms error:
0.260447

Left camera calibration

21 left images found
There are no more left images

Left camera matrix: INTRINSIC

[1115.488680426108, 0, 316.4635984037795;
0, 1114.75805126877, 223.8783082757508;
0, 0, 1]

Left camera matrix: EXTRINSIC

[-0.1234396176236101, 0.8704171312319118, -0.001391213663301998,
9.345524996964323e-05, -125.8131946723568, 0.1374422112325338,
-0.7906605634785033, -114.0148492230641]

Left camera rms error:
0.253945

Calibrating stereo
Cameras ROTATION matrix:

[0.9990613255725089, -0.002165551523724455, -0.04326405126544833;
0.002384738940593295, 0.9999845797601961, 0.005015302783402157;
0.04325252322681838, -0.005113768514713795, 0.9990510840823363]

Cameras TRANSLATION:

[49.21567666544566; 0.745348709965439; 5.11424373573004]

Cameras ESSENTIAL matrix:
```

```
[0.02004207620157852, -5.117976413630723, 0.7189919558675388;
2.980740927823582, 0.240602419447482, -49.39023802962585;
-0.627282529560094, 49.21653183894393, 0.2790783249658107]

Cameras FUNDAMENTAL:

[8.263171111843611e-08, -2.110107301055242e-05, 0.007856545183112936;
1.228939464746923e-05, 9.919926931873091e-07, -0.2334083570063444;
-0.005592251835848664, 0.2351518943449702, 1]

Stereo rms error:
0.31233

Exit program
```

3.2.3 Proceso de captura de imágenes estéreo

Una vez obtenidas las matrices de calibración por medio del programa anterior, se procedió a realizar otro programa que se encargue de asegurar que las imágenes captadas por el estéreo se capturaran exactamente al mismo tiempo. Para ello, fue necesario utilizar la programación por hilos y semáforos, consiguiendo así que la captura de ambas imágenes se produjera simultáneamente y de forma síncrona mediante dos hilos diferentes.

Cabe destacar además, que en este punto era importante asegurar que ambas cámaras tuvieran exactamente los mismos valores de propiedades a la hora de realizar las capturas, como pueden ser el brillo, ganancia, tiempo de apertura, etc. De no ser así, a la hora de realizar el estéreo se encontrarían errores que pueden llegar a resultar bastante graves, llegando al punto de no encontrar correspondencia entre ambas imágenes, como por ejemplo en el caso de que una cámara, y por tanto imagen, tuviera más brillo que la otra: los píxeles de la imagen que deberían ser iguales, al ser más oscuros en una que en otra, no encontrarán su correspondiente pareja por color en la otra imagen.

Con los parámetros ajustados a los mismos valores para ambas cámaras, el primer paso es inicializar los hilos y la comunicación con las cámaras. Una vez hecho esto, y en caso de que no se produzca ningún error, el siguiente paso es iniciar el bucle de captura de imágenes, en el cual mediante el uso de semáforos se espera hasta que ambos estén listos para capturar, entonces se lanzan los dos hilos de captura de imágenes para cada cámara, se espera a que terminen, y se realiza la acción correspondiente, ya sea guardar las imágenes en la memoria del ordenador, o seguir capturando, todo ello en función de la tecla que se esté pulsando en ese momento.

El código fuente de este programa se puede encontrar en el apéndice A.4.

El objetivo incial de la aplicación desarrollado era poder distinguir la localización de una mano pasando por enfrente de las cámaras, para poder realizar acciones según su posición y proximidad. Siendo así, con este programa se generaron distintos conjuntos de imágenes para ambas cámaras en diferentes condiciones de iluminación, distintos fondos, diferentes distancias entre las cámaras, la mano y el fondo, etc. Todo ello, con el objetivo de evaluar distintos resultados y rendimientos a la hora de obtener la posición del objeto más cercano, que en este caso se trata de una mano moviéndose delante de la cámara, con la que se pretendía controlar la velocidad y posición de un robot móvil.

Para poder realizar esto, se necesita hallar las imágenes de disparidad, que consiste en una serie de cálculos realizados etre ambas imágenes que dan como resultado la imagen de disparidad. Esta imagen

contiene información de la distancia que hay entre los objetos que aparecen y las cámaras, normalmente en escala de grises, donde el blanco significa más cercano y el negro implica más lejano.

3.3 Extracción de la información de profundidad a partir de visión estéreo

Una vez generada una base de datos con suficientes imágenes, el siguiente paso es la implementación de un programa para la tarjeta Jetson TK1 que se encargue de extraer la información de distancia en cada píxel, de cada par de imágenes que hemos capturado previamente. Para ello, se han empleado las librerías de OpenGL además de OpenCV, que permiten utilizar el entorno gráfico que proporciona la tarjeta, dando así una mayor versatilidad a la hora de elegir cómo hacer funcionar el programa y obtener una mejor respuesta de los controles que el usuario pueda ejercer sobre el mismo.

Para el correcto funcionamiento del programa, una vez se han establecido las funciones básicas de programación y funcionamiento del entorno de OpenGL sobre la tarjeta, con las que se podrá visualizar una ventana con una determinada resolución, o a pantalla completa, es la creación de un *Makefile* que incluya los comandos de compilación que permitan incluir todas las librerías necesarias para la creación del ejecutable. Después de numerosas pruebas y resolución de conflictos, el archivo final empleado es el siguiente:

```
C_FLAGS = -c #-DLOAD_SRCS #-DSTORE_SRCS #-DQUANTITATIVE_RESULTS #-DWITH_REGUL
#-DWITH_ARROWS
#CU_FLAGS = --ptxas-options=-v
#-target-cpu-arch ARM
C_SOURCES = ./src/main.cpp ./src/system.cpp ./src/graphics.cpp
./src/graphicApplication.cpp ./src/stereo.cpp
#CU_SOURCES = ./src/stereo_cuda.cu

#CUDA_PATH = /usr/local/cuda-6.0

INCLUDE = `pkg-config --cflags opencv` -I./include -I/usr/include/GL
-I/usr/local/include/opencv2
#-I/usr/local/cuda-6.0/include -I/usr/local/cuda-6.0/samples/common/inc
`pkg-config --cflags opencv`

LDLIBS = `pkg-config --libs opencv` -L. -lstdc++ -lpthread -lGL -lGLU -lglut -lpng
# -lpthread -lcudart -lcublas -lcufft -L/usr/local/cuda-6.0/lib
-L/usr/local/cuda-6.0/samples/common/lib
# -lcuda `pkg-config --libs opencv`
LDFLAGS = `pkg-config --libs opencv` -OPENCV_BUILD_3RDPARTY_LIBS
#-Wl -E `pkg-config --libs opencv`

C_OBJS = $(patsubst %.cpp, %.o, $(C_SOURCES))
#CU_OBJS = $(patsubst %.cu, %.o, $(CU_SOURCES))

CC = g++
#NVCC = /usr/local/cuda-6.0/bin/nvcc
#-march=native -mtune=native -finline-functions -m32
# -funroll-loops -msse2 -ftree-vectorize -fno-rtti -fno-exceptions
```

```
-fomit-frame-pointer -O3 -ffast-math -m32

EXE = Stereo_project

all: $(EXE)
  #$(CU_OBJS): %.o : %.cu
  # $(NVCC) $(CU_FLAGS) -c $(INCLUDE) -o $@ $<
  $(C_OBJS): %.o : %.cpp
  $(CC) $(C_FLAGS) $(INCLUDE) -o $@ $<

$(EXE): $(C_OBJS)
  $(CC) -o $(EXE) $(C_OBJS) $(INCLUDE) $(LDFLAGS) $(LDLIBS)
  #$(NVCC) $(CU_OBJS)
  #rm -f *.o *~ core.* ./src/*.o ./src/*~ #./src/*.cu_o

clean:
  rm -f *.o *~ core.* ./src/*.o ./src/*~
  $(EXEC) clear
  rm $(EXE)
```

El código está compuesto por archivos que son puramente funcionales en lo respectivo al entorno gráfico de OpenGL, como son el main, system, graphics, y graphicApplication (Con sus respectivos .h); y por los archivos que gestionan de una manera separada y ejecutando en la GPU de la tarjeta código, se encuentran los archivos de stereo.cpp, junto con el fichero de cabecera common.h

El código se puede encontrar en los anexos según siguen: main.cpp anexo A.5, system.cpp anexo A.6, system.h anexo A.7, graphics.cpp anexo A.8, graphics.h anexo A.9, graphicApplication.cpp anexo A.10, y graphicApplication.h anexo A.11. Los archivos de interés de este trabajo se encuentran en los anexos: stereo.cpp anexo A.12, stereo.h anexo A.13 y common.h en el anexo A.14.

A continuación, se explica brevemente el funcionamiento de cada uno de los archivos desarrollados:

- **main.cpp**

  En este programa principal, se inicializan las variables relacionadas con la ventana que se usará durante el programa, se procede a la inicialización del motor gráfico de openGL, también conocido como *glut*, y se entrará en el bucle principal del programa de openGL en el que se manejarán las actualizaciones de pantalla y los estímulos del usuario hacia el programa, como pudiera ser una tecla presionada.

- **system.cpp**

  Contiene funciones relacionadas con el sistema, que pueden resultar útiles o cruciales a la hora de calcular determinados parámetros del programa, como por ejemplo los *Frames por segundo (FPS)* que es capaz de generar. En este archivo se encuentra la declaración de la función que nos dará los milisegundos que han transcurrido desde que comenzó el programa, y otras funciones para gestionar posibles errores que pudiera dar el programa.

- **system.h**

  Este encabezado tendrá las declaraciones de las funciones en su archivo de código, system.cpp.

- **graphics.cpp**

El archivo de graphics contiene todas las funciones que gestionan lo que ocurra en la aplicación desarrollada, desde el punto de vista de OpenGL. Aquí están las definiciones de las funciones que registran la función que se encargará de hacer cada cosa, desde la que inicializa la ventana que se creará para la aplicación, hasta las funciones que permiten manejar la respuesta frente a las diferentes teclas que permiten realizar diversas operaciones, o qué hacer cuando no tenga que actualizar la pantalla o por el contrario cuando sí tenga que hacerlo.

- **graphics.h**

  Este encabezado tendrá únicamente las declaraciones de las funciones de su correspondiente archivo de código, graphics.cpp.

- **graphicApplication.cpp**

  El archivo graphicsApplication se encarga de gestionar todas las órdenes que se reciben del teclado, así como gestionar lo que se muestra por pantalla, y llamar a las funciones que se encargan de hacer los cálculos correspondientes con las imágenes.

  Este archivo contiene los constructores para la clase Aplication que será el núcleo del programa. Se inicializan aquí los mensajes que tiene que mostrar la aplicación tanto al inicio de la ejecución como al finalizar el programa, y se declaran las funciones para el manejo de la ventana, desde su creación hasta su redimensionado si ocurriera durante la ejecución del programa.

  Las funciones que manejan la respuesta frente a haber presionado una tecla, consisten mayormente en enviar un mensaje por la consola de comandos que responde con un parámetro de la visión en estéreo que se está utilizando en ese momento, y algunas de ellas lo modifican, incrementando o decreciendo su valor. Hay que tener en cuenta que estos comandos son sensibles a las mayúsculas, ya que la letra mayúscula incrementará el valor, mientras que la minúscula lo hará más pequeño.

  Cabe destacar dos funciones que se encargan de ejecutar código en cada vuelta del bucle de OpenGL, que son las funciones OnIdle y OnRender. OnIdle se ejecutará cuando no ocurra nada más en el programa, pero al menos se ejecutará una vez en cada bucle. Esta función se encarga en nuestro caso de calcular la cantidad de frames que se están generando, para poder tener una idea de la capacidad de procesamiento que tiene nuestro programa. La otra función, OnRender, será la encargada de hacer que se vea todo correctamente en nuestro programa. Prepara los parámetros de la ventana y configura las imágenes para poder mostrarlas correctamente por la pantalla y que no salgan invertidas. Ahora bien, a la hora de elegir la imagen que debe mostrar y los algoritmos que debe aplicar, se debe haber cambiado previamente los define del archivo common.h, ya que es allí desde donde se configuran las partes del código que se compilarán y por tanto ejecutarán y cuáles no. Esto se explicará más adelante en el capítulo de resultados.

  El programa además, admite la posibilidad de guardar las imágenes que se han procesado, para poder observarlas luego una a una, y no sólo por pantalla, aunque esto ralentiza considerablemente el ritmo de ejecución del programa.

  El resto de funciones que contiene este fichero de código, están relacionadas con el posicionamiento de la ventana, con la obtención de los milisegundos transcurridos desde la última vez que se calcularon (Para poder sacar valores como los frames por segundo que se están generando en el programa), y la creación y liberación de la ventana de ejecución del programa.

- **graphicApplication.h**

  Este encabezado tendrá la clase “Aplication”que define la ejecución del programa, con la declaración de los bucles de OpenGL y todas sus funciones asignadas.

- **stereo.cpp**

  El archivo stereo.cpp contiene todos los datos de las cámaras obtenidos previamente con la calibración de las mismas, almacenados en unas matrices para que sea posible luego operar correctamente con las imágenes, corrigiendo las distorsiones halladas con la calibración. En este fichero se manejan las funciones de la clase Stereo, la cual se encarga de hacer todas las operaciones con las imágenes. En la creación de esta clase, se establecen todos los parámetros necesarios para comenzar haciendo unos cálculos básicos, además de configurar qué imágenes se utilizarán para los cálculos, o las matrices para contener las imágenes correctas, por ejemplo si son en color o en blanco y negro para la imagen de disparidades.

  La función más importante de esta clase será stereo_vision, puesto que es la encargada en mandar las imágenes a la GPU, ordenar los cálculos que deben ejecutarse en esta parte de la tarjeta, y recoger nuevamente los resultados a la CPU para poder procesarla posteriormente en el programa y mostrarla por pantalla o guardar la imagen en el disco.

  El resto de funciones serán las encargadas de procesar las imágenes con distintos algoritmos. Por ejemplo, la función process_image se encargaría de hacer que los valores de los píxeles de la imagen se tornen completamente negros o blancos, dependiendo de dos parámetros que acotarán qué valores deben ser blancos y cuáles negros, pudiendo así obtener unas imágenes en las que se ha resaltado más lo que nos interesa destacar o conseguir separar del resto de la imagen, como es el caso de resaltar la mano pasando por delante de las cámaras y eliminar el fondo de la imagen. Hay que destacar que esta función se intentó ejecutar en GPU, pero los resultados de la función predefinida para ello que tiene OpenGL no eran tan satisfactorios como los que se consiguieron ejecutando un for que recorría cada píxel de la imagen para hacer este procesamiento, lo que ralentizaba considerablemente la ejecución del programa y por tanto reducía notablemente los fps que se podían lograr en la ejecución del programa, como veremos más adelante en el capítulo de resultados.

  La función watershed_algorithm, traducido como algoritmo de marca de agua, pretende separar los objetos que se hallan en la imagen del fondo de la misma. El algoritmo intenta encontrar los contornos de los objetos que encuentra, y luego rellenarlos en una única mancha sólida, definiendo perfectamente cada objeto que se halla en nuestra imagen. Añadiendo algún cálculo adicional a este algoritmo, además se pueden separar los distintos objetos hallados por lejanía o proximidad, obteniendo una imagen en escala de grises con los objetos más cercanos en un color más claro, y los más lejanos en uno más oscuro. En la práctica resultó que el objeto más cercano, que era la mano, se separaba en diferentes objetos, siendo imposible definir la mano como un único objeto sólido, y dando confusión con otros que se encontraban más alejados. Aun así, fue el algoritmo que mejores resultados dio.

  Las demás funciones que controla esta clase ya son para calcular la siguiente imagen que debe mostrar o guardar, la calibración de los parámetros de los cálculos para las imágenes del programa, mostrar los parámetros que tiene actualmente configurados, o para el cálculo de tiempo que tarda en hacer los trabajos de la GPU.

- **stereo.h**

  Este encabezado tendrá la clase “Stereo”que contiene las funciones que ejecutan el código de la GPU, y que aplican los algoritmos utilizados para hallar las imágenes de disparidad.

- **common.h**

  El fichero de common.h es el más importante de todos, no por sus funciones, sino porque desde él se configura todo el funcionamiento del programa. En él se incluyen todos los encabezados de librerías

externas que se utilizan en el programa, desde los de OpenGL o OpenCV hasta los de CUDA y los propios del sistema.

La parte de las definiciones es la que contiene las elecciones de los segmentos de código que se ejecutarán o no. Así pues, basta con comentar o descomentar las líneas en concreto que queramos para que el programa funcione a nuestro gusto. Estas definiciones se explicarán una a una en la sección del manual del programa, y se pueden encontrar defines desde los que muestran la información de FPS tanto de la CPU como de la GPU, hasta los que eligen el algoritmo que se aplicará en la ejecución del código, con los valores por defecto que mejor funcionan para cada conjunto de datos.

El resto de declaraciones que contiene este fichero son las estructuras básicas para el funcionamiento del programa, como la de la ventana de visualización, la estructura que conforma un punto en dos dimensiones, o la que aúna los posibles valores de un color separado en sus cuatro variables de rojo, verde, azul y transparencia.

Una vez finalizado el programa, el siguiente paso fue el ajuste de los diferentes parámetros implicados en el cálculo de las imágenes de disparidad. Esto se llevó a cabo de forma experimental, analizando cada uno de los conjuntos de datos hasta obtener el conjunto de parámetros que proporcionaban mejores resultados. Puesto que existe un número elevado de opciones para configurar la disparidad de las imágenes, como pueden ser el tamaño de bloque o la cantidad de disparidades calculadas para encontrar los puntos correspondientes, el conjunto de parámetros se guardó para que se cargara de manera automática cada vez que se utilizara ese *dataset* en concreto.

Estos resultados, por el orden en el que aparecen, nos indican:

- Datasheet utilizado
- Las imágenes de disparidad se guardarán o no.
- Las imágenes de disparidad se mostrarán o no.
- El número de disparidades nos indica la cantidad de píxeles que el algoritmo de visión en estéreo busca en su imagen correspondiente para calcular las distancias de los objetos que aparecen. Siempre es un múltiplo de 8, no superior a 256.
- El tamaño del bloque de las disparidades indica el número de píxeles de una forma circular en los que el algoritmo buscará su píxel correspondiente en la otra imagen. Si este parámetro es muy pequeño, el algoritmo no encontrará fácilmente el píxel en la otra imagen, pero si lo encuentra será una información más precisa. En caso de ser muy grande, tendrá mayor facilidad para encontrar su homónimo en la otra imagen, pero introducirá más ruido a la imagen de disparidad final.

Estos dos últimos valores, se precargan por defecto, pero posteriormente se ajustan para cada dataset, como se puede observar más abajo en cada configuración, que varía ligeramente de los 128 y 11 respectivamente.

Una vez cargada la calibración de las cámaras se carga la configuración que se encontró más adecuada en pruebas anteriores, cuyos parámetros son:

- Prefiltercap es el valor de truncado para los píxeles de la imagen prefiltrada.
- Prefilter size es el tamaño del prefiltro de la imagen.

- Block size es el parámetro que hemos explicado antes en la anterior enumeración.
- Min disparity es el valor mínimo que se asigna a la disparidad para poder hallar la imagen de disparidad.
- Number of disparities es el parámetro previamente explicado, que será diferente para cada dataset.
- Texture threshold es el umbral o valor límite que se aplica a un objeto para diferenciarlo del resto en la imagen de disparidad, dependiendo de su textura.
- Uniqueness ratio es el parámetro que define el margen en porcentaje que existe entre el mejor valor calculado de la función de costes y el segundo mejor valor considerado para ser un candidato posible a ser un pixel homónimo.
- Speckle window size es el tamaño máximo de las regiones de disparidad dudosas para poder considerarlas como ruido e invalidarlas.
- Speckle range es la variación máxima de disparidad entre cada componente conectado.
- Disp 12 max diff es la máxima diferencia permitida de disparidad en píxeles que puede haber entre las imágenes 1 y 2 (Izquierda y derecha).
- Min process value es el valor utilizado cuando la función de procesar la imagen está activo, y se utiliza para marcar el margen por debajo que debe usarse para considerar lo que hay que resaltar y lo que no en la imagen.
- Max process value es el valor complementario al anterior, utilizado para marcar el margen por encima que debe usarse para procesar la imagen.

En la tabla **??** se observan las configuraciones utilizadas para cada conjunto de datos.

Tabla 3.1: Configuración del conjunto de datos 0

Dataset	0	1	2	3
Prefiltercap	32	32	32	32
Prefilter size	0	0	0	0
Block size	19	19	9	19
Min disparity	0	0	0	0
Number of disparities	128	128	144	128
Texture threshold	5	5	0	8
Uniqueness ratio	0	0	0	0
Speckle window size	0	0	0	0
Speckle range	0	0	0	0
Disp 12 max diff	0	0	0	0
Min process value	120	200	180	160
Max process value	255	255	255	255

Los resultados obtenidos utilizando los diferentes algoritmos en los datasets grabados se presentan el el capítulo 4.

Capítulo 4

Resultados

Un ingeniero es alguien que soluciona un problema que no sabías que tenías, de una manera que no entiendes.

Anónimo

4.1 Introducción

Aunque en los apartados anteriores se han mostrado algunos resultados parciales, en este apartado se muestran los resultados del proceso completo, incluyendo los ya mostrados anteriormente.

Todos los resultados presentados se han obtenido realizando el procesamiento de las imágenes en la tarjeta de evaluación Jetson TK1 analizada en este TFG. Como ya se ha comentado, la captura de las imágenes se realiza en un PC debido a los problemas de compatibilidad con las cámaras encontrados en la tarjeta.

Para comenzar, se muestran algunas imágenes de ejemplo pertenecientes a los datasets grabados. A continuación se presentan los resultados de disparidad de las imágenes estéreo (diferencia de posición de cada punto de una imagen con el punto correspondiente en la otra imagen). Se comienza presentado los resultados obtenidos aplicando la técnica Stereo BM descrita en el capítulo 2, para posteriormente presentar los resultados obtenidos al aplicar dos alternativas diferentes para la mejora de los resultados. Por cada uno de los pasos, se muestra el coste computacional de los algoritmos comparando el número de frames por segundo que pueden procesarse en la tarjeta de desarrollo Jetson TK1 en función de los pasos algoritmos introducidos.

4.2 Imágenes estéreo

Como se ha descrito en el capítulo 3, para la consecución de los objetivos se han grabado cuatro datasets en diferentes condiciones. Todos ellos consisten en conjuntos de imágenes adquiridos por un par de cámaras fijas en una estructura y previamente calibradas. En todas las imágenes aparece una mano en primer plano frente a diferentes tipos de fondo.

Algunas imágenes de los conjuntos tomados se pueden observar en la Figura 4.1. El primer paso para poder realizar todas las pruebas de disparidad fue conseguir una base contundente de imágenes con distintos fondos. Se puede observar claramente la diferencia entre ambas imágenes por la posición de la mano en cada una de ellas.

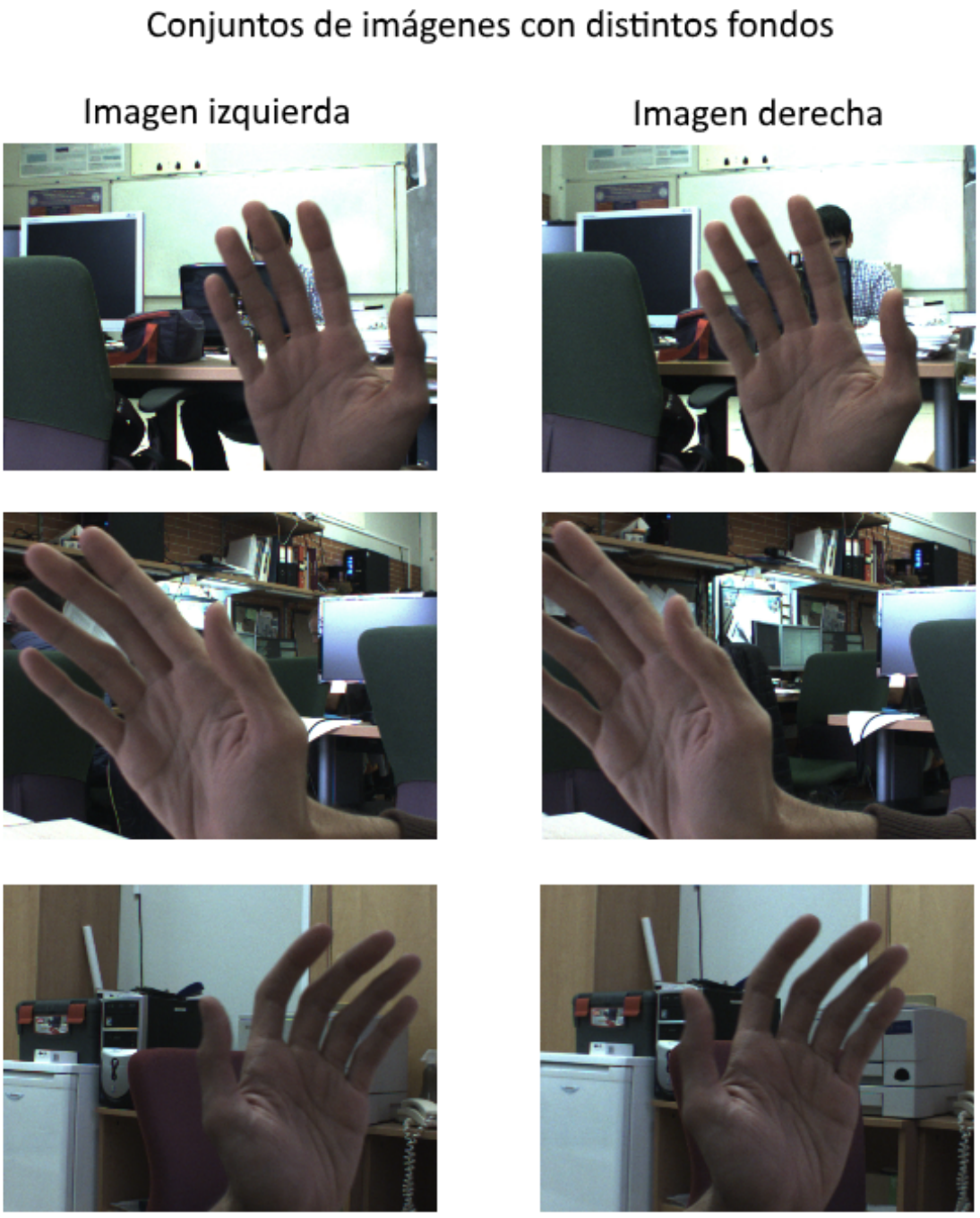
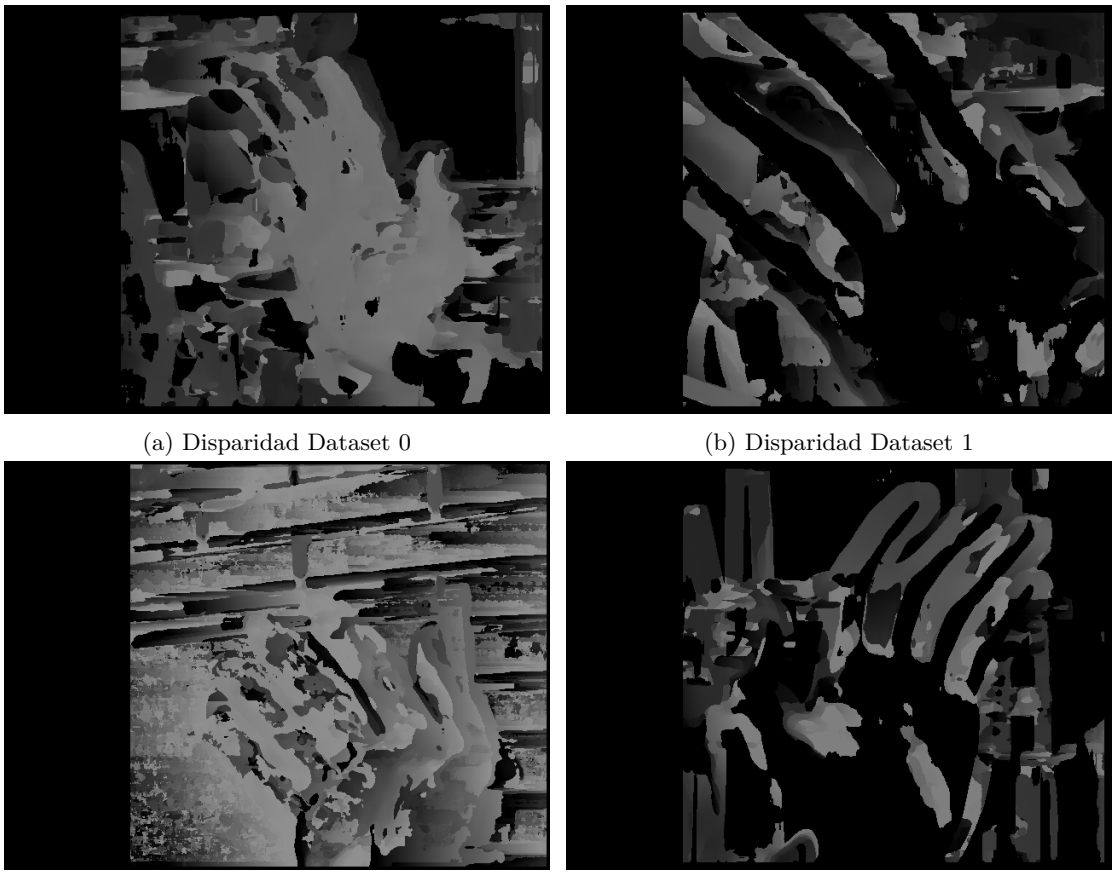


Figura 4.1: Algunos ejemplos de las imágenes capturadas para la realización del estéreo



(a) Disparidad Dataset 0

(b) Disparidad Dataset 1

(c) Disparidad Dataset 2

(d) Disparidad Dataset 3

Figura 4.2: Disparidades para cada conjunto de datos

4.3 Resultados de disparidad

Para comenzar, se muestran las imágenes con su configuración más básica, (algoritmo Stereo BM) ejecutado en la GPU de la tarjeta.

Según podemos ver en la figura 4.2, observamos con alguna dificultad que aparece una mano aproximadamente en el centro de cada imagen, en las de la izquierda detectada de una manera más correcta, como si estuviera delante al estar resaltada en un color más claro, y en las de la derecha de una manera errónea aunque aparentemente más clara, ya que el objetivo de la disparidad no es detectar la mano con un fondo negro, ya que eso se asigna a los objetos más alejados.

Respecto a la velocidad de procesamiento, se analiza el número de frames por segundo (fps) procesados con y sin guardado de datos a disco, para cada conjunto de datos, son los que se muestran en la tabla 4.1. Adicionalmente, se pueden hallar los FPS que alcanza la GPU, y se puede ver las velocidades que tarda en responder la carga y descarga de datos de la CPU a la GPU.

Observamos que los FPS conseguidos guardando todas las imágenes calculadas con la disparidad, se rebajan a unos 3 FPS de media, mientras que si no guardamos la imagen en disco y simplemente la procesamos y mostramos, los FPS aumentan hasta unos 5 FPS aproximadamente, lo que hace que el programa sea claramente más fluido ya que prácticamente se duplica la cantidad de imágenes por segundo que el programa es capaz de procesar. Los FPS obtenidos se pueden observar en la tabla 4.1.

Sobre la información de la GPU, podemos observar que en cualquiera de los casos, ya que se calcula independientemente de si se guarda o no la imagen, la GPU es capaz de procesar más imágenes de

Tabla 4.1: Comparativa FPS obtenidos

FPS	FPS Con guardado	FPS Sin guardado
Máximo	4.61	5.94
Mínimo	2.49	2.44
Media	3.17	5.19

las que la CPU puede guardar o mostrar, alcanzando unos 6 FPS, aunque como se puede observar es muchísimo más inestable y variable esta cantidad, ya que se producen variaciones de 2 hasta 7 FPS. Los FPS obtenidos en la GPU se pueden observar en la tabla 4.2.

En relación a la velocidad de respuesta que tiene la GPU y el tiempo que tarda en traspasar información entre CPU y GPU, podemos observar en la cuarta imagen que el tiempo de descarga de la información de la GPU es aproximadamente un tercio del tiempo que tarda en cargar las imágenes a la GPU, ya que al hacer la carga tiene que traspasar dos imágenes y reservar memoria para la tercera de resultados que luego se obtiene. Los tiempos obtenidos en la GPU de carga y descarga de datos se pueden observar en la tabla 4.2. A la vista de estos datos, cabe destacar la importancia de una correcta planificación de las transferencias de información entre la memoria de la CPU y la de la GPU para evitar posibles cuellos de botella que incrementen los costes computacionales.

Tabla 4.2: Comparativa FPS y tiempo de respuesta obtenidos en la GPU

FPS/tiempo	FPS de la GPU	Tiempo de subida en ms	Tiempo de descarga en ms
Máximo	8.73	1413.43	2985.82
Mínimo	2.65	95.11	233.45
Media	5.28	885.20	350.03

Debido a que los resultados de disparidad obtenidos no son los adecuados, se añaden nuevas etapas con el objetivo de mejorarlos. En concreto, se evalúan dos alternativas: la umbralización y posterior erosión de las imágenes de disparidad y el algoritmo *watersheed*. En la figura 4.3 se muestra un esquema de estas diferentes etapas, en color verde se presentan las fases de la primera alternativa, y en amarillo las de la segunda. Se puede observar como las primeras etapas son comunes a ambas alternativas.

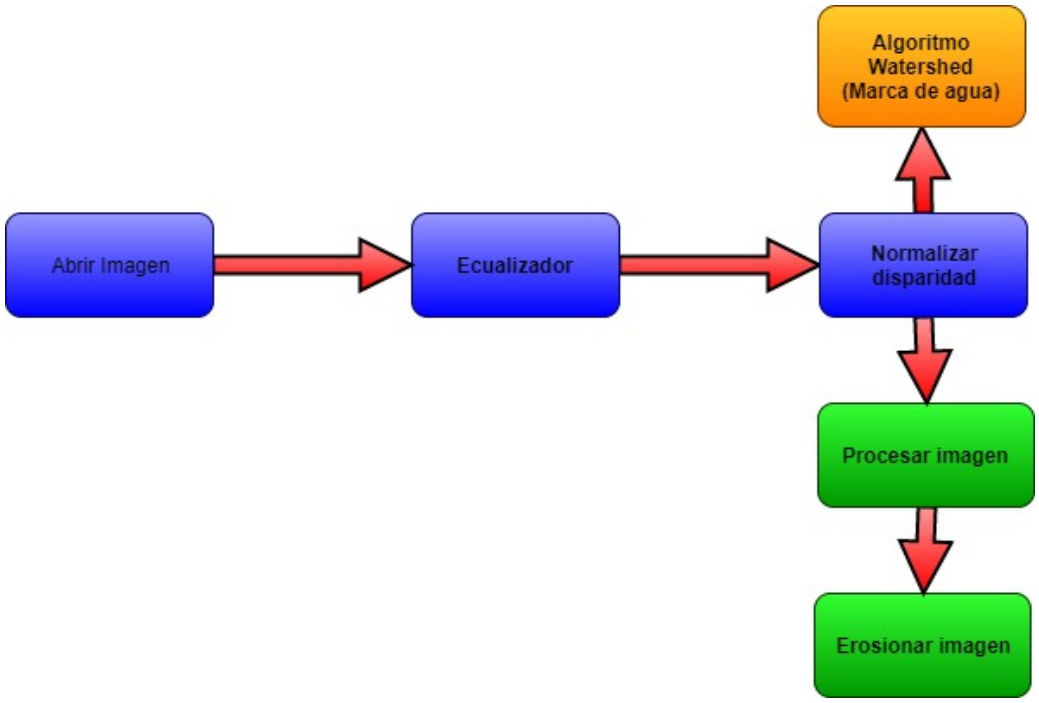


Figura 4.3: Esquema de los algoritmos posibles para el programa

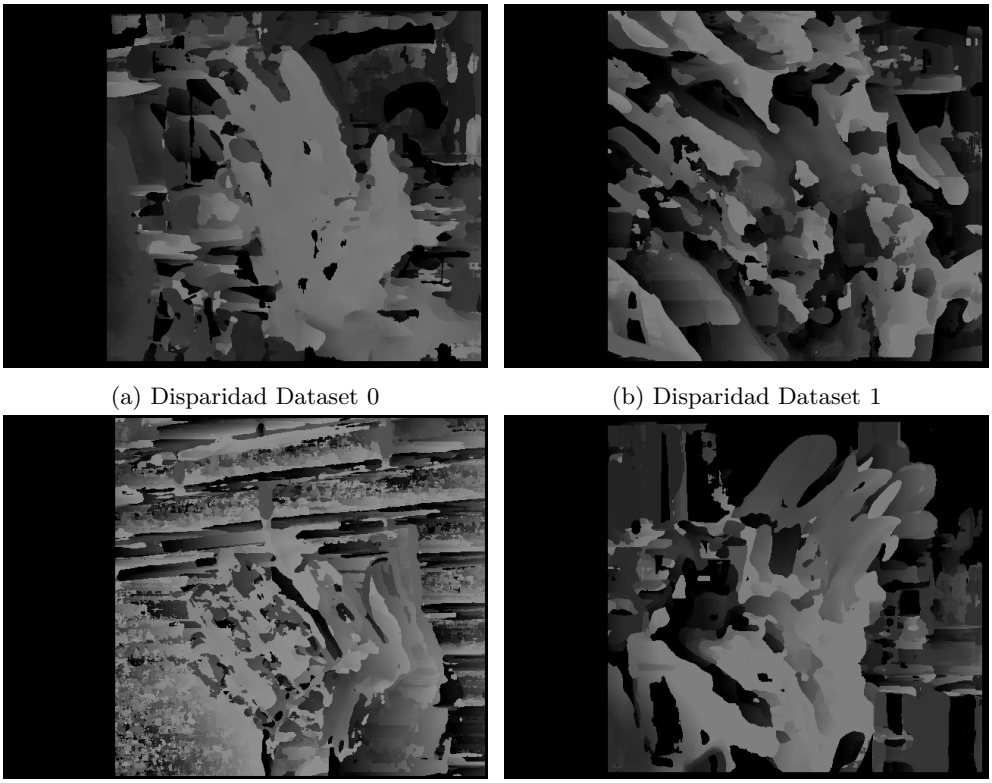
El primer proceso que se hace con las imágenes antes de hacer ningún tipo de cálculo sobre ellas, y que es común a las dos alternativas, es la ecualización de cada imagen (izquierda y derecha) para mejorar la calidad de la imagen de disparidad. Este algoritmo se ejecuta en la GPU, puesto que OpenCV tiene esta función entre las que se pueden ejecutar en la GPU.

Según se puede observar en la figura 4.4, apenas existen cambios entre las imágenes anteriores en los dataset 0 y 2, y sin embargo en los datasets 1 y 3 hay un cambio notable a la hora de conseguir la imagen de disparidades, que son precisamente las que contenían algún tipo de error por el que no hacía la disparidad de la mano de una forma correcta. En cualquier caso, el dataset 1 hace que la detección de la mano sea prácticamente imposible por el nivel de ruido existente en la imagen de disparidad.

El siguiente procesado se lleva a cabo sobre la imagen de disparidad, y consiste en normalizar los valores de la misma para conseguir una mayor facilidad a la hora de distinguir los objetos de las imágenes. El resultado de esta etapa se observa en la figura 4.5. La normalización es un proceso que se ejecuta en la GPU, ya que OpenCV dispone de una función que puede realizar estos cálculos en GPU.

Se observa que hay una ligera aclaración sobre qué es lo que está más cercano al haber adquirido mayor tono de brillo con el normalizado de la imagen, pero sigue siendo un problema en algunos conjuntos de datos en los que el ruido generado por los objetos del fondo de la imagen es elevado.

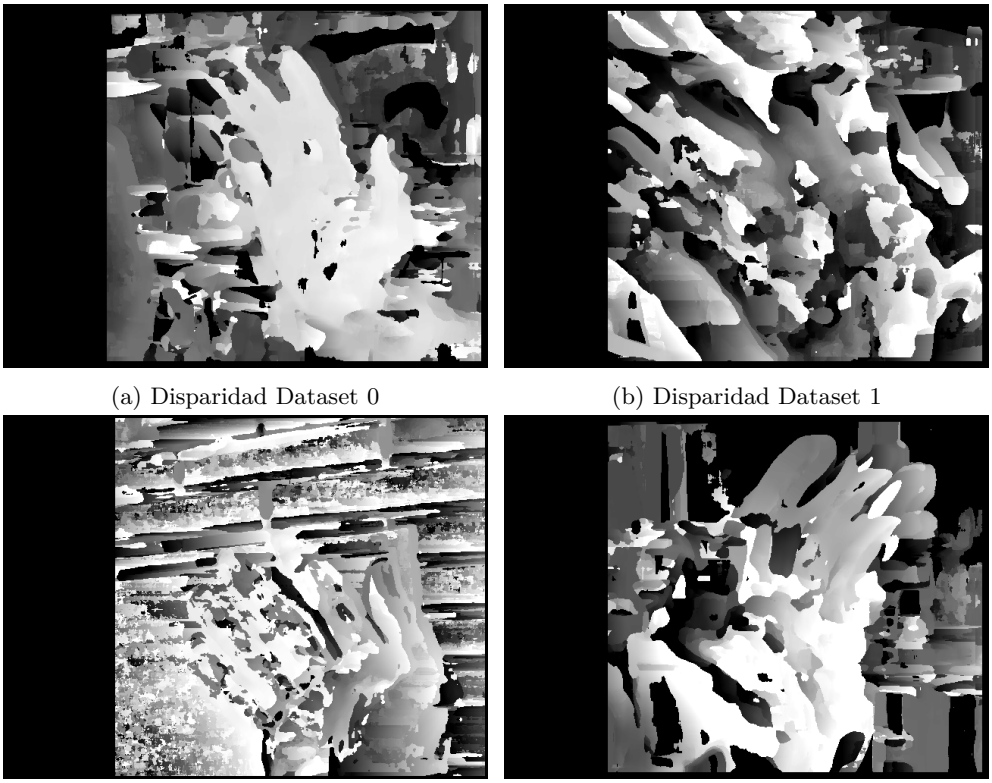
OpenCV ofrece además una opción adicional en la que las imágenes se pueden mostrar en color, con la que obtendríamos unos resultados como los mostrados en la imagen 4.6.



(a) Disparidad Dataset 0 (b) Disparidad Dataset 1

(c) Disparidad Dataset 2 (d) Disparidad Dataset 3

Figura 4.4: Disparidades para cada conjunto de datos con ecualizado



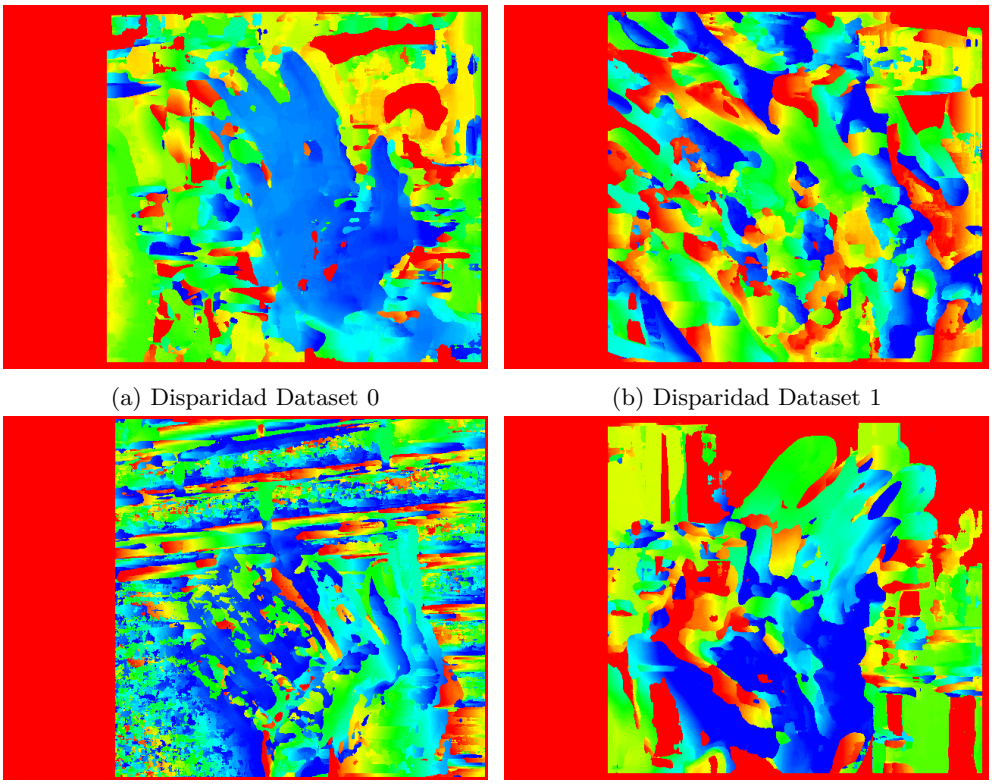
(a) Disparidad Dataset 0 (b) Disparidad Dataset 1

(c) Disparidad Dataset 2 (d) Disparidad Dataset 3

Figura 4.5: Disparidades normalizadas para cada conjunto de datos

Estas imágenes a color representan los mismos datos de distancia, pero los píxeles toman valores en rgb, donde el rojo significa que el objeto está más lejos, y el azul que el objeto está más cerca. Estas imágenes en color nos aportan una ligera idea más clara de cómo y dónde están las manos en las imágenes, pero sigue siendo difícil diferenciarlas en algunos conjuntos de datos. En cualquier caso, es más fácil diferenciar aquí el ruido del fondo de la imagen, ya que existe más contraste por los colores que el que se puede observar en las imágenes con tonos de grises.

Con esto se deduce que el dataset 2 es inviable para poder distinguir la mano del resto de la imagen, o al menos no es posible hacerlo con este algoritmo que se ejecuta para hallar la disparidad. Lo mismo ocurre con el dataset 1, ya que la imagen se encuentra demasiado distorsionada aunque no sea por culpa del ruido que produce el fondo de la imagen.



(a) Disparidad Dataset 0

(b) Disparidad Dataset 1

(c) Disparidad Dataset 2

(d) Disparidad Dataset 3

Figura 4.6: Disparidades en color para cada conjunto de datos

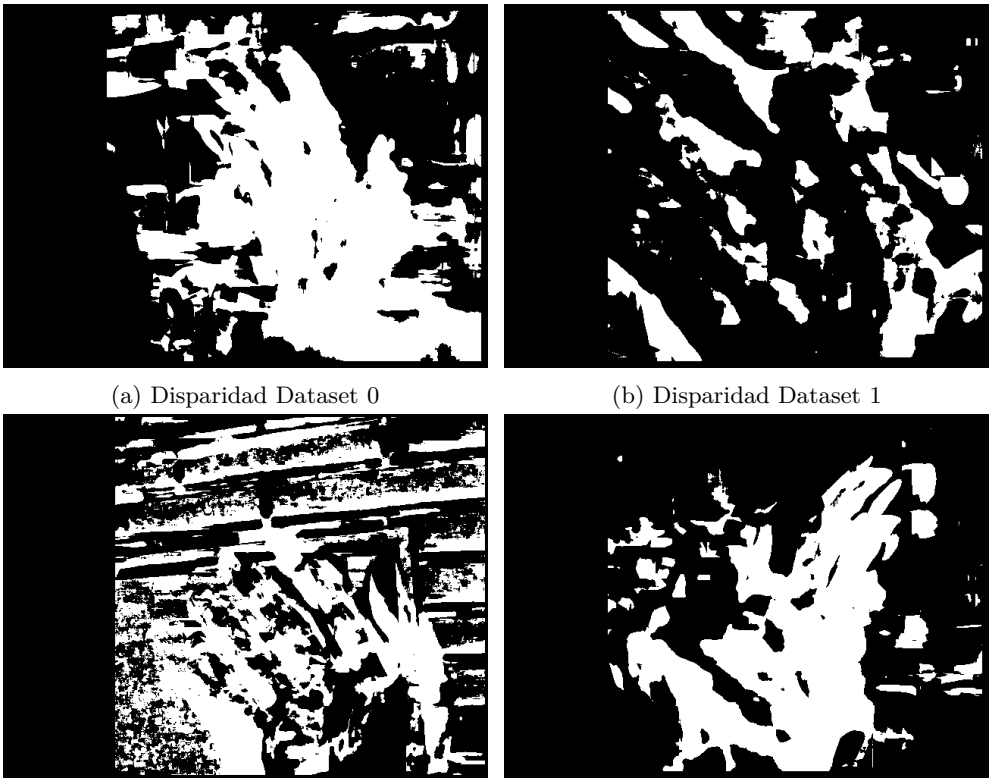
4.3.1 Alternativa 1: umbralización y erosión de la imagen de disparidad

Volviendo a la imagen en escala de grises, el siguiente paso fue la utilización de un umbral para elegir sólo los valores de la imagen a utilizar, tratando así de reducir el efecto del ruido del fondo de la imagen y las disparidades erróneas. Los resultados se pueden ver en la figura 4.7. La umbralización se ejecuta en la GPU de la tarjeta a través de las funciones de las librerías de OpenCV.

Para cada uno de los conjuntos de datos se utilizaron unos umbrales diferentes, a partir del cual los valores menores que ese umbral se tornan negros en la imagen, y los mayores blancos obteniendo una imagen binaria. El objetivo de esta umbralización era resaltar la mano en la imagen en una única área lo más grande posible para poder detectarla posteriormente.

Como se puede observar en las imágenes, el Dataset 0 y el 3 resultan mucho más clarificadoras ahora sobre la posición de la mano, y los Datasets 1 y 2 siguen siendo completamente inviables para la detección

de la mano. Pese a los aparentes buenos resultados en estos conjuntos de datos que sí funcionan bien, en el momento en el que la mano desaparece del centro de la pantalla, todo el ruido del fondo de la imagen hace que sea prácticamente imposible distinguir qué es mano y qué no lo es, por lo que en última instancia se intenta aplicar el siguiente procesado sobre la imagen de disparidad, que consiste en hacer una erosión de las áreas para minimizar las pequeñas manchas de ruido del fondo de la imagen.



(a) Disparidad Dataset 0 (b) Disparidad Dataset 1

(c) Disparidad Dataset 2 (d) Disparidad Dataset 3

Figura 4.7: Disparidades procesadas para cada conjunto de datos

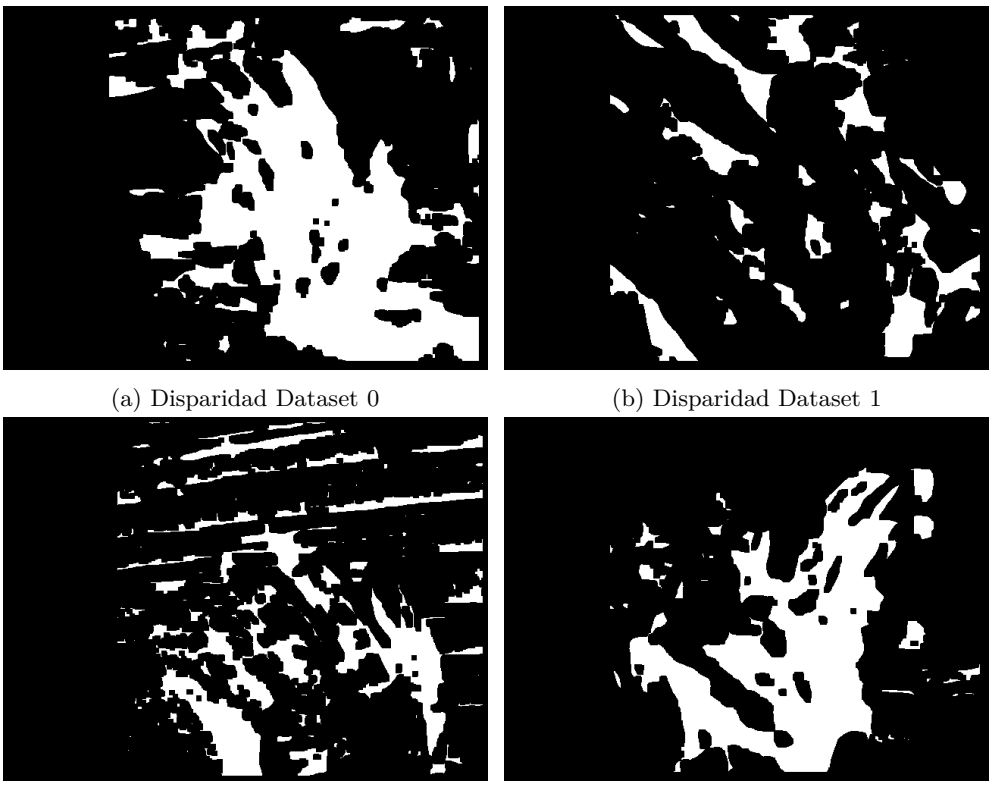
El proceso de erosión que se puede observar en las imágenes de la figura 4.8 consigue en gran medida el objetivo de hacer que el ruido del fondo de la imagen no influya tanto a la hora de ver más clara la mano en la imagen, pero al aplicar la erosión, también se erosiona la mano, confundiendo en algunas capturas más aún su posible posición. Esto es debido principalmente al tipo de fondo que aparece en la imagen, como por ejemplo en el conjunto de datos 2 no hay suficiente diferencia de texturas en el fondo de la imagen, al ser una pared de ladrillos y una pantalla de ordenador, lo cual introduce errores de correspondencia que afectan a la disparidad calculada. Al tratarse de un filtro, OpenCV vuelve a ofrecer soporte para GPU en esta función, ayudando a la mejora del rendimiento del programa.

En cualquier caso, los resultados obtenidos ahora para los conjuntos de datos 0 y 3 serían lo suficientemente buenos como para seguir investigando desde aquí una manera para hallar y procesar dónde está la mano en la imagen. Sin embargo, dado que el objetivo de este trabajo era evaluar la tarjeta de desarrollo jetson TK1, no se añadieron nuevas etapas para la detección de la mano.

En última instancia, se volvieron a calcular los FPS para poder comparar cuánto ha reducido la capacidad de procesamiento del programa el haber añadido estos algoritmos al cómputo (Ecualizado, normalizado, procesado y erosionado). Como se puede observar en la tabla 4.3, el rendimiento del programa se ha reducido, como era de esperar, y lo ha hecho aproximadamente en medio FPS para el guardado, y a la mitad para cuando no se escriben las imágenes en disco. Se puede observar que los FPS rondan

entre los 2.5 para cuando se guardan las imágenes, y unos 2.8 en caso contrario. Esta diferencia es mucho menor que en el caso inicial, lo que nos hace pensar que la principal carga e importancia de la velocidad de procesamiento de nuestro programa depende más de la cantidad de transformaciones sobre la imagen que de las lecturas y escrituras en disco.

Sin embargo, el hecho de no tener conectadas directamente las cámaras a la tarjeta, afecta negativamente al tiempo de procesamiento al requerir una lectura de disco en vez de poder leerlas directamente de memoria, lo que mejoraría el rendimiento del programa.



(a) Disparidad Dataset 0 (b) Disparidad Dataset 1

(c) Disparidad Dataset 2 (d) Disparidad Dataset 3

Figura 4.8: Disparidades erosionadas para cada conjunto de datos

Tabla 4.3: Comparativa FPS obtenidos

FPS	FPS Con guardado	FPS Sin guardado
Máximo	3.37	3.41
Mínimo	0.82	1.58
Media	2.47	2.76

4.3.2 Alternativa 2: algoritmo watershed

La segunda alternativa se basa en el algoritmo conocido como *watershed*, que consta de diferentes etapas cuyos resultados se irán mostrando a lo largo de esta sección. Dichas etapas se presentan en la figura 4.9:

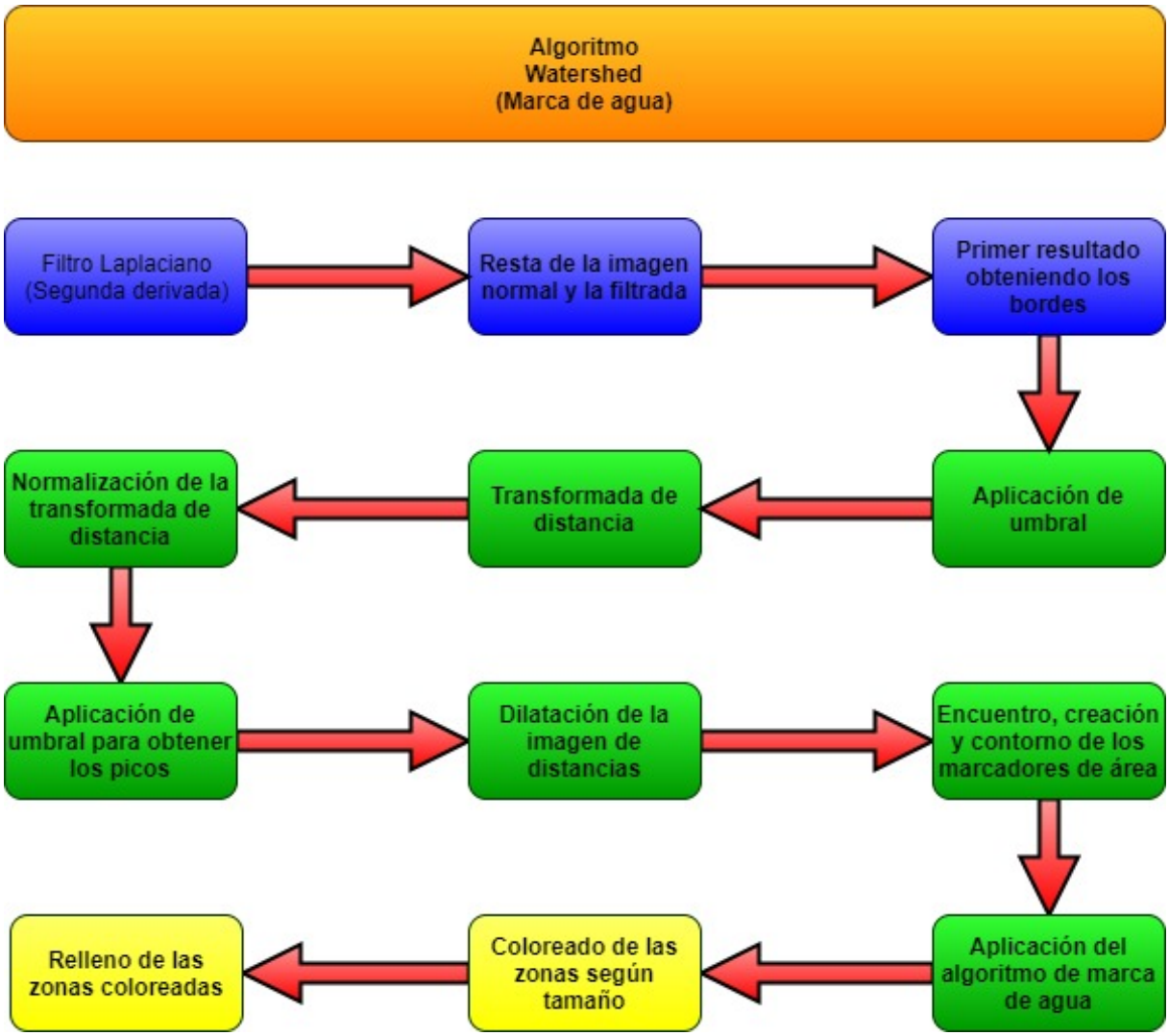
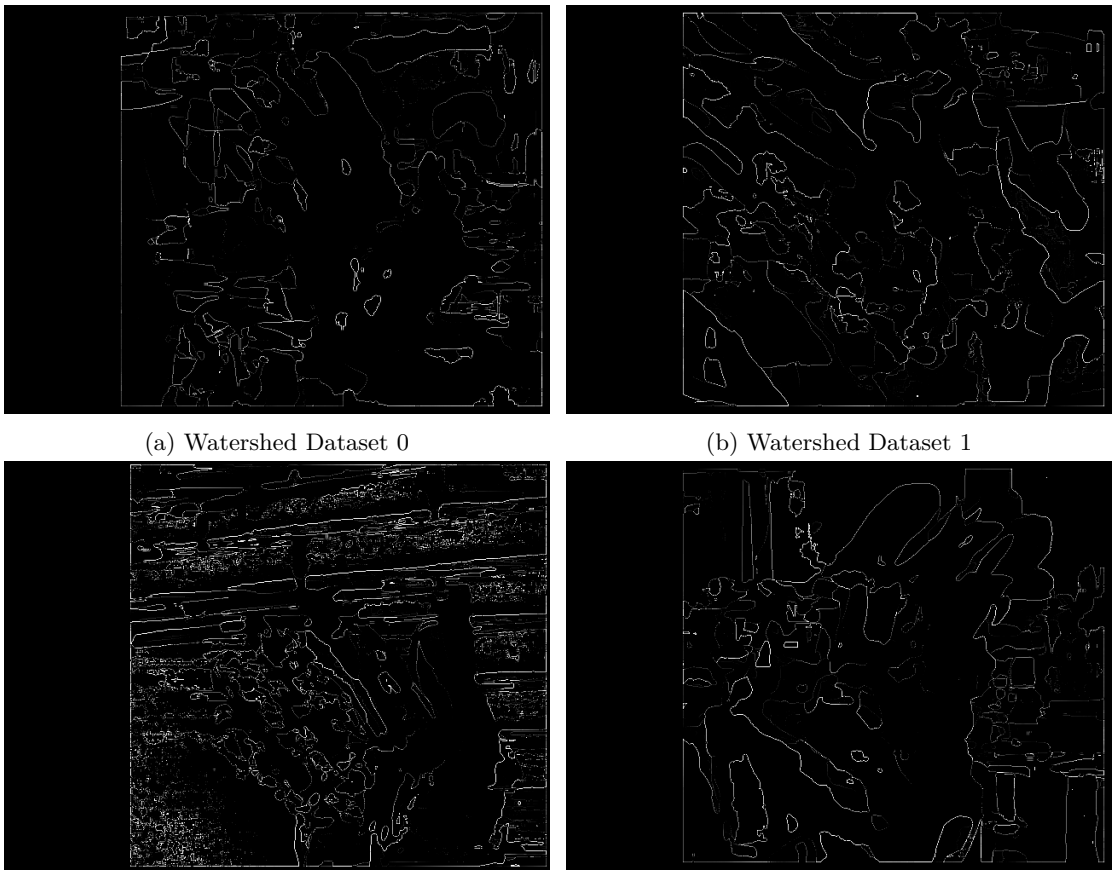


Figura 4.9: Esquema del algoritmo de marca de agua

Se observa en primer lugar la zona azul del esquema de la figura 4.9. Lo primero que se aplica en el algoritmo de marca de agua es un filtro laplaciano para hacer una aproximación a la segunda derivada de la imagen. Con estose consigue que los bordes que haya en la imagen se marquen de una manera clara, con lo que permite distinguir cada objeto que existe dentro de la imagen. A continuación, se resta a la imagen original esta imagen filtrada, para obtener una imagen que contenga sólamente los bordes de los objetos de la imagen, ya que serán los valores más grandes al haber aplicado el filtro. Esto lo podemos observar en la figura 4.10 para cada conjunto de datos. Desgraciadamente, estas operaciones de resta de imágenes y futuros cálculos de este algoritmo (salvo nuevas umbralizaciones) se ejecutan en la CPU de la tarjeta, que elemento por elemento de la matrix debe restar cada píxel de cada imagen, lo que no acelera en absoluto el programa.



(a) Watershed Dataset 0

(b) Watershed Dataset 1

(c) Watershed Dataset 2

(d) Watershed Dataset 3

Figura 4.10: Imagenes del borde para el algoritmo de marca de agua para cada conjunto de datos

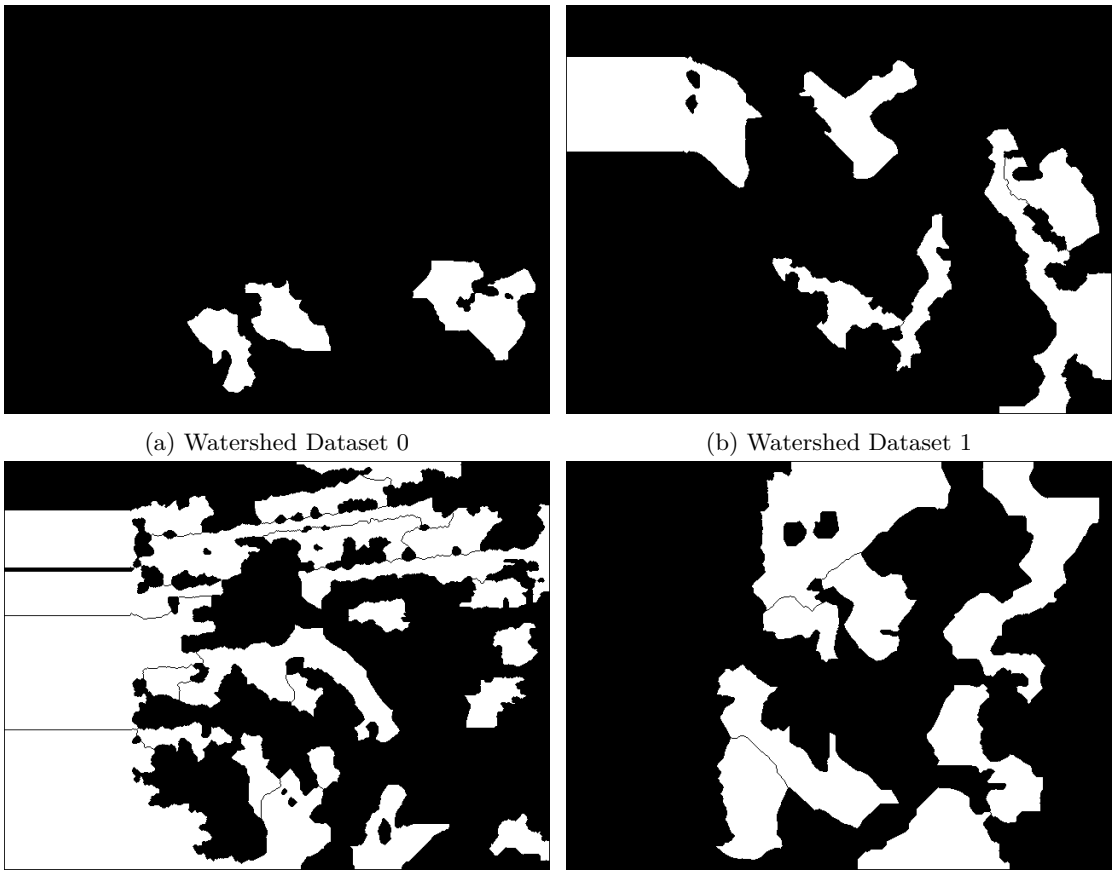
Una vez se han obtenido los bordes de los objetos de la imagen, hay que obtener las áreas que contengan estos bordes. Para ello, se comienza aplicando un umbral para descartar el poco ruido que se ha generado en la imagen, y quedarnos así únicamente con los bordes de los objetos de la imagen.

A continuación, se aplica un algoritmo de la transformada de distancia sobre la imagen original, en el que se obtienen datos de las distancias de los objetos de la imagen. Tras ello, se normaliza como se hizo en el caso anterior para poder tratar mejor con los valores y tener más claras las cosas que están cerca y cuáles están lejos.

Después de esto, se aplica nuevamente un umbral para quedarnos sólo con los objetos más cercanos en la imagen, y poder obtener así los valores más altos de los objetos que están más cerca para hallar las áreas de nuestro interés.

Sin embargo, al contrario que en el algoritmo anterior donde se buscaba erosionar las áreas de la imagen para reducir el ruido, ahora se busca dilatar lo que se encuentra en estos momentos en la imagen, que son los bordes de los objetos, para poder determinar mejor hasta dónde llegan.

Tras todos estos procesamientos, ya sólo queda hallar los contornos, determinando cuáles son sus áreas, centroides, tamaños, y puntos que los contienen. Con ellos hallados, se aplica el algoritmo de la marca de agua, pero ésto no produce un resultado visible, sino que simplemente coloca los marcadores en la imagen. Como podemos observar en la figura 4.11 para cada conjunto de datos, después de determinar qué color deben tener las áreas que están marcadas según su tamaño, se rellenan en blanco o negro pudiendo observarse los resultados.



(a) Watershed Dataset 0

(b) Watershed Dataset 1

(c) Watershed Dataset 2

(d) Watershed Dataset 3

Figura 4.11: Imagenes del algoritmo de marca de agua aplicado para cada conjunto de datos

Sin embargo, podemos ver que la aplicación del algoritmo produce un resultado completamente indeseado, por lo que se descarta su utilización para la detección de las manos en las imágenes estéreo.

Por último, una vez más se calculan los FPS que se alcanzan con este algoritmo, y se obtiene un valor más bajo, debido a que la mayoría de los procesos ejecutados en este algoritmo se llevan a cabo en la CPU en lugar de la GPU. Se ve en la tabla 4.4 que se alcanzan unos FPS de al rededor de 1 para el guardado de las imágenes, y de 1.5 cuando no se guardan las imágenes.

Tabla 4.4: Comparativa FPS obtenidos

FPS	FPS Con guardado	FPS Sin guardado
Máximo	1.83	2.18
Mínimo	0.67	0.65
Media	1.02	1.51

Se puede observar que la incorporación de etapas adicionales para la mejora del cálculo de la disparidad reduce en gran medida el rendimiento del programa. Esto se debe a que las nuevas etapas se han desarrollado haciendo uso de las librerías OpenCV que no están optimizadas para la paralalelización de los algoritmos usando la GPU, por lo que la mayor parte del procesamiento añadido se realiza en la CPU. Dadas las características de la tarjeta, que incluye una CPU con una potencia reducida, esto se traduce en una reducción importante del número de frames que pueden procesarse cada segundo.

Como ya se ha comentado, aunque los resultados de disparidad no han sido adecuados para la aplicación planteada, el estudio realizado ha permitido lograr los objetivos del TFG al haber realizado una evaluación exhaustiva de las características de la tarjeta de desarrollo NVidia Jetson TK1, tanto a nivel de hardware, como del software disponible.

4.4 Resultados finales

Estos algoritmos se ejecutan principalmente desde la CPU de la tarjeta, haciendo que el rendimiento del programa se reduzca notablemente. Las partes que se ejecutan en la GPU son la disparidad, que se castea explícitamente desde el programa desarrollado en este proyecto, y alguna función específica de OpenCV, entre otras las funciones para convertir el formato de colores de las imágenes (nivel de grises a RGB por ejemplo), las funciones de umbralización, de ecualizado, y funciones que aplican filtros a las imágenes (como el utilizado en el algoritmo de marca de agua).

Con todo esto, y observando la cantidad de funciones que se ejecutan entre un algoritmo y otro, y cuántas de ellas se ejecutan en GPU, podemos hacer una comparativa en la que observamos que prácticamente todas las funciones de la primera alternativa se ejecutan en GPU, lo que justifica que se obtenga una mayor cantidad de fps. Sin embargo, el algoritmo de marca de agua tiene apenas dos funciones que se ejecutan en la GPU, que son un filtrado para hallar los bordes de los objetos de la imagen, y una umbralización para poder distinguir los objetos en la imagen. Las diferencias halladas entre estas dos alternativas las podemos observar en la tabla 4.5 para el caso que no incluye el guardado de las imágenes de disparidad.

Tabla 4.5: Comparativa FPS obtenidos

FPS	FPS umbralización	FPS Watershed
Máximo	3.41	2.18
Mínimo	1.58	0.65
Media	2.76	1.51

Como se puede observar, se consigue casi el doble de FPS en el algoritmo que más funciones ejecuta en GPU respecto al que realiza prácticamente todos los cálculos en CPU.

Capítulo 5

Conclusiones y líneas futuras

En este apartado se resumen las conclusiones obtenidas y se proponen futuras líneas de investigación que se derivan del presente trabajo.

5.1 Conclusiones

En este trabajo se ha llevado a cabo un estudio a fondo del funcionamiento de las unidades gráficas de procesamiento (GPU), especialmente las de NVidia que utilizan CUDA, mediante la evaluación de la tarjeta de desarrollo NVidia Jetson TK1. Durante el desarrollo del trabajo se han analizado las características de la tarjeta, determinando sus posibilidades y limitaciones para el desarrollo de aplicaciones.

Para lograr los objetivos planteados, se han implementado y evaluado diferentes alternativas para la extracción de información de profundidad a partir de imágenes estéreo, analizando en cada caso los resultados obtenidos y la carga computacional.

En relación a la tarjeta evaluada, tras la realización del trabajo se destacan los siguientes aspectos:

- Rendimiento elevado: la tarjeta es en general un elemento optimizado para el bajo consumo con una aceptable capacidad de procesamiento. No ha defraudado en este sentido, teniendo en cuenta para lo que está preparada y lo que es capaz de hacer. Se calculaban medias de 10 FPS para que la tarjeta realizara cálculos relacionados con la visión en estéreo continuada.

- Bajo consumo: a pesar de contar con una unidad de procesamiento gráfico.

- Problemas de compatibilidad con hardware externo: la tarjeta tiene muchos problemas de compatibilidad en lo referente a añadirle algún tipo de hardware externo, en nuestro caso, cámaras que funcionen por FireWire. Tras probar varios métodos, entre los que se incluía utilizar algún tipo de conversor entre FireWire y los puertos que tiene la tarjeta, como el PCIe, fue imposible encontrar unos drivers capaces de manejar las cámaras desde la tarjeta. Adicionalmente, la limitación de tener un único puerto USB 3.0 hace que sin un elemento externo, como pueda ser un HUB USB, la tarjeta sea prácticamente inutilizable, ya que los teclados y ratones estándar comparten ese mismo tipo de conector, y no se pueden conectar a la vez. Sin embargo, se espera que con las sucesivas actualizaciones se reduzcan estos problemas.

- Baja optimización de las librerías OpenCV: las librerías de OpenCV isponibles precompiladas para esta tarjeta tienen una funcionalidad más reducida que en el caso general. En cualquier caso, muchas

de las funciones que contienen estas librerías de OpenCV que podrían (Y deberían) ejecutarse en la GPU, se seguían ejecutando en la CPU, ralentizando la ejecución del programa.

Respecto a los resultados de disparidad obtenidos, se han encontrado diversos problemas debido al nivel de ruido y la calidad de las imágenes de entrada que han impedido obtener resultados adecuados para la aplicación planteada.

Se ha comprobado además que las librerías OpenCV disponibles no se encuentran optimizadas para el trabajo en GPU, por ello, al añadir etapas de procesamiento se reduce la velocidad de respuesta del sistema (permitiendo procesar un número menor de frames por segundo). Para resolver esto, se plantea implementar funciones específicas que hagan uso de la GPU de NVidia disponible utilizando CUDA, y descartando las funciones de OpenCV.

A pesar de lo anterior, cabe destacar que se han alcanzado con éxito los objetivos planteados en el TFG en relación a la implementación y ejecución de algoritmos de visión artificial para la evaluación de la tarjeta de desarrollo NVidia Jetson TK1, quedando como trabajo futuro la búsqueda de alternativas que presenten un mejor funcionamiento dentro del entorno de desarrollo disponible.

5.2 Líneas futuras

Este TFG abre una línea de trabajo en relación al procesamiento de imágenes en paralelo utilizando la GPU.

Entre los trabajos futuros se encuentra el uso de CUDA para la implementación de funciones con mayor grado de paralelización, con el objetivo de reducir el tiempo de cómputo, permitiendo trabajar en tiempo real. También se plantea combinar las funciones de OpenCV con otras en CUDA con el mismo objetivo.

Por otro lado, es interesante comprobar la compatibilidad de la tarjeta con nuevas cámaras FireWire con el objetivo de poder desarrollar un sistema autónomo que incluya tanto la grabación como el procesamiento de las imágenes adquiridas.

Adicionalmente, se plantea añadir una interfaz sobrescrita sobre las imágenes, en la que se podrían mostrar los parámetros que calcula el programa, como pudieran ser la posición de la mano, los FPS que se generan, o la velocidad a la que deba ir el robot.

Y por último, se podría estudiar porqué no están completas las librerías de OpenCV, completarlas y escribir las funciones que faltan para así poder utilizar las librerías en dispositivos que usen NVidia con una funcionalidad completa, y no encontrar así las limitaciones que se han encontrado al realizar este trabajo.

Capítulo 6

Pliego de condiciones

6.1 Introducción

En este apartado se evaluaran las condiciones para poner en marcha el programa que se ha detallado en los capítulos anteriores. Solamente afectan las condiciones de configuración *hardware* o *software* donde se quiera aplicar.

6.2 Requisitos de *hardware*

6.2.1 Requisitos mínimos

- Utilización de una tarjeta de desarrollo NVidia Jetson TK1.
- 2 cámaras *firewire* para tomar las imágenes.
- Un PC compatible con la tecnología firewire para poder realizar las capturas.
- Adaptador de *HDMI* a *VGA* para poder visualizar la tarjeta Jetson TK1.

6.2.2 Requisitos de hardware recomendados

Estos requisitos son altamente recomendables para tener más facilidades a la hora de trabajar.

- PC con varios puertos *firewire* para conectar las dos cámaras, o bien un *HUB Firewire* para conectar varias cámaras a un sólo puerto.
- *HUB USB* para poder conectar varios dispositivos a la tarjeta de desarrollo NVidia Jetson TK1.
- Pantalla con entrada de *HDMI*

6.3 Condiciones *hardware*

El sistema de adquisición de imágenes que se propone hace uso de 2 hilos independientes en la adquisición en tiempo real. Es por esta razón que se recomienda sistemas multiprocesador, que permitan realizar las tareas de cada hilo de manera independiente.

Se precisan dos cámaras para capturar las imágenes con las que se harán las disparidades. En este trabajo sehan utilizado cámaras con conexión *firewire* por tener una alta velocidad de comunicación de datos con el procesador. Además, la forma física de las cámaras que se han utilizado ha permitido facilitar la colocación y calibración de las mismas para poder tomar unas capturas de imágenes lo más parecido posible a la vista humana.

Se recomienda tener al menos 5 GB de disco duro libre para poder hacer manejar la gran cantidad de imágenes, desde las de los diferentes conjuntos de datos, hasta las creadas por el propio programa como imágenes de disparidad. Es altamente recomendable disponer de 10 GB libres si se van a realizar más capturas con diferentes conjuntos de datos adicionales.

6.4 Requisitos de *software*

6.4.1 Requisitos mínimos

- Utilización de un sistema operativo Ubuntu 14.04.
- Librería `OpenCV`.
- Librería `OpenGL`.
- Para el desarrollo y correcta ejecución de los programas se recomienda instalar `CUDA 6.0`.

6.4.2 Requisitos de *software* recomendados

- No actualizar (bajo responsabilidad propia) el sistema operativo. La versión precompilada que viene por defecto tiene unos drivers que se sobreescriben con la actualización, y deja de funcionar el entorno gráfico de la tarjeta hasta que se restaura este archivo.

6.5 Condiciones *software*

En este apartado software se recomienda utilizar el sistema operativo Ubuntu 14.04 precompilado que proporciona NVidia para esta tarjeta, ya que contiene drivers específicos para su correcto funcionamiento y algunas funciones están optimizadas para funcionar mejor en ella.

6.6 Condiciones generales

La utilización de estas librerías supone la posibilidad de ejecutar cualquier programa de procesamiento de imagen en GPU sobre ella que tenga en cuenta las siguientes condiciones:

- Las limitaciones de la GPU están condicionadas a la cantidad de núcleos que tiene, por lo que no se podrán realizar más cálculos de los que es capaz de soportar.
- La velocidad de procesamiento de la tarjeta está principalmente limitada por su CPU, por lo que en la mayoría de los casos la cantidad de imágenes por segundo que se puedan generar dependerá del procesador.
- Ante cualquier uso de estas librerías deberá tenerse en cuenta el reconocimiento (BY), que no es comercial (NC), y que las obras derivadas se compartan de igual manera (SA).

Capítulo 7

Presupuesto

7.1 Equipo de trabajo

Para la realización del proyecto se va a necesitar un Ingeniero de Electrónica y automática industrial con conocimientos de visión artificial.

7.2 Hardware necesario

Los materiales básicos para desarrollar el presente trabajo se pueden ver en la tabla 7.1.

Tabla 7.1: Material Hardware necesario

Material	Precio en €	Unidades	Subtotal
Tarjeta NVidia Jetson TK1	160.00 €	1	160.00 €
Cámaras FireWire	350.00 €	2	700.00 €
HUB USB 3.0	20.00 €	1	20.00 €
Mini Mac	559.00 €	1	559.00 €
TOTAL	1439.00 €		

7.3 Software necesario

Los programas y librerías básicas para desarrollar el presente trabajo se presentan en la tabla 7.2.

Tabla 7.2: Tabla de costes del software necesario

Software	Precio en €	Versión recomendada
Ubuntu	Gratuito	14.04
CUDA	Gratuito	6.0
OpenCV	Gratuito	3.0
OpenGL	Gratuito	4.0
TOTAL	0 €	

7.4 Costes de tiempo

Las fases necesarias para la realización del desarrollo se resumen en la tabla 7.3:

Tabla 7.3: Tabla de costes de tiempo

Concepto	Precio en €	Cantidad	Subtotal
Estudio previo y configuración	60€/h	40h	2400.00 €
Evaluación de capacidades de la tarjeta	60€/h	40h	2400.00 €
Calibración y captura de imágenes	60€/h	60h	3600.00 €
Implementación y evaluación de algoritmos	60€/h	100h	6000.00 €
Documentación	25€/h	80h	2000.00 €
TOTAL	16400.00 €		

7.5 Presupuesto total

Tabla 7.4: Coste total del proyecto

Concepto	Subtotal €
Hardware	1439.00 €
Software	0.00 €
Horas de trabajo	16400.00 €
TOTAL	**17839.00 €**

El coste total del proyecto asciende a la cantidad de **Diecisiete mil ochocientos treinta y nueve euros.**

En Alcalá de Henares, a 27 de junio de 2017.

Bibliografía

[1] F. Tombari, S. Mattoccia, and L. Di Stefano, *Segmentation-Based Adaptive Support for Accurate Stereo Correspondence.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 427–438. [Online]. Available: "http://dx.doi.org/10.1007/978-3-540-77129-6_38"

[2] S. Mattoccia, F. Tombari, and L. Di Stefano, *Stereo Vision Enabling Precise Border Localization Within a Scanline Optimization Framework.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 517–527. [Online]. Available: "http://dx.doi.org/10.1007/978-3-540-76390-1_51"

[3] "Wiki de la tarjeta de desarrollo Jetson TK1," http://elinux.org/Jetson_TK1l, online; Ultimo acceso: 09/06/2017.

[4] "Tarjeta de desarrollo Jetson TK1," http://http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html, online; Ultimo acceso: 06/06/2017.

[5] "Dispositivo arvision 3d," https://est-kl.com/es/manufacturer/trivisio/arvision-3d-hmd.html [Último acceso 26/junio/2017].

[6] R. Lange and P. Seitz, "Solid-state time-of-flight range camera," *IEEE Journal of Quantum Electronics*, vol. 37, no. 3, pp. 390–397, mar 2001. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=910448http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=910448http://ieeexplore.ieee.org/document/910448/

[7] Y. Ukidave, D. Kaeli, U. Gupta, and K. Keville., "Performance of the nvidia jetson tk1 in hpc," in *2015 IEEE International Conference on Cluster Computing*, Sept 2015, pp. 533–534.

[8] G. Li, P. Ren, X. Lyu, and H. Zhang, "Real-time top-view people counting based on a kinect and nvidia jetson tk1 integrated platform," in *2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*, Dec 2016, pp. 468–473.

[9] G. Reinisch, C. Arth, and D. Schmalstieg, "Panoramic mapping on a mobile phone gpu," in *2013 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, Oct 2013, pp. 291–292.

[10] U. Cekmez and M. Ozsifinan, "Parallel solution for uav route planning problem using ant colony optimisation on gpu with cuda," in *2014 22nd Signal Processing and Communications Applications Conference (SIU)*, April 2014, pp. 1122–1125.

[11] A. Benini, M. J. Rutherford, and K. P. Valavanis, "Real-time, gpu-based pose estimation of a uav for autonomous takeoff and landing," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, May 2016, pp. 3463–3470.

[12] M. T. Satria, S. Gurumani, W. Zheng, K. P. Tee, A. Koh, P. Yu, K. Rupnow, and D. Chen, "Real-time system-level implementation of a telepresence robot using an embedded gpu platform," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2016, pp. 1445–1448.

[13] B. Chrétien, A. Escande, and A. Kheddar, "Gpu robot motion planning using semi-infinite nonlinear programming," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 10, pp. 2926–2939, Oct 2016.

[14] "Lenguaje y herramientas de desarrollo cuda," http://www.nvidia.es/object/cuda-parallel-computing-es.html [Último acceso 23/junio/2017].

[15] "Epipolar Geometry," http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/OWENS/LECT10/node3.html, online; Ultimo acceso: 12/06/2017.

[16] "Tarjeta Raspberry pi," https://www.raspberrypi.org/, online; Ultimo acceso: 06/06/2017.

[17] "Camaras firewire," http://www.theimagingsource.com/products/industrial-cameras/firewire-400-monochrome/dmk21bf04h/l, online; Ultimo acceso: 10/06/2017.

Apéndice A

Manual de usuario

A.1 Introducción

En este capítulo se incluye una pequeña explicación de cómo manejar los archivos del proyecto, y cómo ejecutar los programas utilizados. También se incluye el código fuente de los mismos.

A.2 Manual

Para crear los ejecutables de cada programa, se debe acceder desde una ventana de terminal a la carpeta donde se encuentre el proyecto y ejecutar el comando de make para crear el ejecutable del programa en cuetión. Una vez hecho esto, se ejecuta el programa sin ningún parámetro adicional, ya que todo se modifica previo a la compilación si procede.

Respecto a los programas que no se ejecutan en la tarjeta, como son los de calibración de las cámaras en estéreo y la captura de imágenes, lo único que hace falta para ejecutarlos es conectar las dos cámaras para capturar las imágenes, y tener unas pocas imágenes capturadas con un damero para poder realizar la calibración.

En el programa de calibración de las cámaras en estéreo, solamente hay que seguir los pasos que se indican en consola, para que el programa vaya avanzando hasta haber terminado la calibración por completo con todas las imágenes que encontró, y finalmente anotar los resultados de las matrices que determinan los parámetros del sistema y que corregirán las imágenes capturadas por las cámaras.

En el programa de captura de imágenes, una vez se han inicializado las cámaras, se pueden realizar dos acciones pulsando sus teclas correspondientes; La más obvia es salir del programa pulsando la tecla de escape (ESC), y la segunda es pulsando "s"para guardar las imágenes que haya en ese momento en las cámaras. Éste programa y método se utilizó para guardar las imágenes de calibración con los dameros, y posteriormente todas las imágenes manteniendo pulsada la tecla para formar las bases de datos con las manos.

Por último, el programa que se ejecutaba en la tarjeta, en el que se realizan todos los cálculos de GPU, se controla enteramente desde el archivo common.h. En el encabezado de este fichero, se encuentran todas las definiciones que habilitan o deshabilitan las funciones del programa. Para activar alguna de estas funcionalidades, lo único que hay que hacer es descomentar la línea correspondiente, y esa funcionalidad quedará activada. De igual manera, el conjunto de datos que se elige para hacer la visión en estéreo se

elije a través de este archivo, descomentando su línea correspondiente, y poniendo como comentario las de los datasets que no vayan a utilizarse.

Las definiciones posibles para cambiar con libertad son las siguientes:

- #define SHOW_FPS

  Muestra o no los FPS generados por el programa

- #define GPU_INFO

  Muestra o no la información relacionada con los FPS de la GPU

- #define GPU_MEMORY_INFO

  Muestra o no la información de los tiempos de respuesta de la GPU

- #define GENERAL_INFO

  Muestra o no información general sobre el programa, como los parámetros utilizados o el conjunto de datos seleccionado.

- #define SAVE

  Activa o desactiva el guardado de las imágenes de disparidad

- #define DISPLAY

  Muestra o no por pantalla las imágenes de disparidad

A un nivel más específico del algoritmo:

- #define EQUALIZE

  Activa o desactiva la ecualización de las imágenes antes de realizar la disparidad

- #define DISPARITY_COLOR

  Activa o desactiva el modo de color de las disparidades

- #define NORMALIZE_DISPARITY

  Activa o desactiva la normalización de las imágenes de disparidad

- #define PROCESS_IMAGE

  Activa o desactiva el procesado de las imágenes de disparidad. Se comprueba que la imágen está normalizada

- #define WATERSHED_ALGORITHM

  Activa o desactiva el algoritmo de marca de agua. Si se activa este algoritmo, se desactivan los demás procesamientos de la imágen

- #define DISPARITIES_NUMBER #define BLOCK_SIZE

  Número de disparidades y tamaño de bloque por defecto para el algoritmo de estéreo

- #define DATASET_0 #define DATASET_1 #define DATASET_2 #define DATASET_3

  Descomentar sólamente uno de ellos, el resto deben estar comentados. Se selecciona de esta manera el conjunto de datos que se va a utilizar.

  De esta manera se cargan por defecto los valores de disparidad mínima y máxima para el procesado de la imagen, y el número de disparidades y el tamaño del bloque para realizar los cálculos de la imágen de disparidad

- #define FULL_SCREEN

  Activa o desactiva el modo de pantalla completa, para poder observar mejor los detalles de las imágenes de disparidad, o poder tenerlo en una ventana aparte mientras se ven los resultados de consola

A.3 Anexo 1

Programa para la calibración de las cámaras en estéreo.

```
/*
* Name        : main.cpp
* Author      : Mario Luis ÁLvarez Pastor
* Version     : 0.0.1
* Copyright   : Free code
* Description : OpenCV use (with pthreads) for capturing images with 2 cameras to
*               do stereo vision
*/



/* -------- */
/* INCLUDES */
/* -------- */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <time.h>
#include <sys/time.h>

// Most important includes:
#include <pthread.h>
#include <semaphore.h>
#include <opencv2/opencv.hpp>



/* ---------- */
/* NAMESPACES */
/* ---------- */

using namespace cv;
using namespace std;



/* ------- */
/* DEFINES */
/* ------- */

//#define VERBOSE        // Uncoment this to get a verbose terminal program
//#define SHOW_RESULTS   // Uncoment this to see the results of the calibration



/* -------------- */
/* FUNCTION CALLS */
/* -------------- */



/* --------- */
```

```
/* VARIABLES */
/* --------- */

// Directory of the image to load
char directory[128];

Size imageSize;

// Intrinsics
Mat cameraMatrix [2];
// Extrinsics
Mat distorsion_Coefficients [2];

// Stereo matrices
Mat Rotation, Translation, Essential, Fundamental;

// Vectors of matrices for rotation and translation parameters
vector <Mat> rotation_vectors, translation_vectors;

// Size of the board used to calibrate the cameras
const Size pattern_size ( 8, 5 );
const float side_x = 29.5;
const float side_y = 29.5;

// Vectors with corners of the board
vector < Point2f > corners_2D;    // Image points
vector < Point3f > corners_3D;    // Object points


// Vectors of vectors with all the coordinates of the corners from the boards
vector < vector < Point2f > > R_coordinates_2D;    // Image points
vector < vector < Point2f > > L_coordinates_2D;    // Image points
vector < vector < Point3f > > coordinates_3D;    // Object points



/* ---- */
/* MAIN */
/* ---- */

int main ( int argc, char * argv[] )
{
// error number
int error = 0;

try   // Always use a try - catch structure to get errors
{
// We need the images on gray color
Mat gray;
Mat image;         // Image to read
Mat image_copy;    // Copy of that image

int R_count = 0, L_count = 0;

double rms_error = 0.0;

// First, we will calibrate Right camera:
cout << "\tRight camera calibration" << endl;
```

```
#ifdef SHOW_RESULTS
// Create a window for display
namedWindow ( "Display window", WINDOW_AUTOSIZE );
#endif

while ( 1 )
{
// Set the directory of the next image to load
sprintf ( directory, "./Calibracion_damero/Right_Capture%.04d.png", R_count++ );

// Load it
image = imread ( directory, CV_LOAD_IMAGE_COLOR );

// Check if it has loaded an image
if ( !image.data )
{
cout << endl << --R_count << " right images found" << endl;   // We need to reduce once the
     count

if ( R_count == 0 )   // No images found
{
error = 1;
throw error;
}
else
break;
}

// We will work with this copy, because some functions can overwrite the original
// one
image_copy = image.clone();

#ifdef VERBOSE
cout << R_count << " right images found" << endl;
#endif

#ifdef SHOW_RESULTS
cout << "Showing image... wait or press a key to continue" << endl;

imshow ( "Display window", image_copy );   // Show the image

waitKey ( 5000 );   // Wait up to 5 seconds for a pressed key to continue
#endif

// We convert the image to gray
cvtColor ( image_copy, gray, CV_RGB2GRAY );   // CV_BGR2GRAY or CV_RGB2GRAY

// We founded a Chessboard?
bool found = findChessboardCorners ( gray, pattern_size, corners_2D );

if ( found )
{
// We call this function to get a better aproximation to the corners of the board
cornerSubPix ( gray,
                           corners_2D,
                           Size ( 11, 11 ),   // Size of the window to search
                           Size ( -1, -1 ),   // No size for singularities in matrix
TermCriteria ( CV_TERMCRIT_EPS + CV_TERMCRIT_ITER, 30, 0.1 ) );
```

```
#ifdef SHOW_RESULTS
#ifdef VERBOSE
cout << "Showing founded Chessboard... wait or press a key to continue" << endl;
#endif

drawChessboardCorners ( image_copy, pattern_size, corners_2D, found);

imshow ( "Display window", image_copy );   // Show the image

waitKey ( 5000 );   // Wait up to 5 seconds for a pressed key to continue
#endif

// Save the corners
R_coordinates_2D.push_back ( corners_2D );

}
else
cout << "No Chessboard found on right image " << R_count << endl;

}

cout << "There are no more right images" << endl << endl;

// Create the 3D pattern
for( int i = 0; i < pattern_size.height; i++ )
{
for( int j = 0; j < pattern_size.width; j++ )
{
corners_3D.push_back ( Point3f( j * side_x, i * side_y, 0 ) );
}
}

// Once we have finished the corners searching, we resize them
R_coordinates_2D.resize ( R_count );
coordinates_3D.resize ( R_count, corners_3D );

imageSize = image_copy.size();



// Initialize the matrix
cameraMatrix[0] = Mat::eye ( 3, 3, CV_64F );

//cameraMatrix = initCameraMatrix2D ( coordinates_3D, coordinates_2D, imageSize );

rms_error = calibrateCamera ( coordinates_3D, R_coordinates_2D, imageSize,
cameraMatrix[0],   // Intrinsics
distorsion_Coefficients[0],   // Extrinsics
rotation_vectors, translation_vectors,
//CV_CALIB_USE_INTRINSIC_GUESS | CV_CALIB_RATIONAL_MODEL);
CV_CALIB_RATIONAL_MODEL);

// Show the results:
cout << "\tRight camera matrix: INTRINSIC" << endl << endl;

cout << cameraMatrix[0] << endl << endl;

cout << "\tRight camera matrix: EXTRINSIC" << endl << endl;
```

```
cout << distorsion_Coefficients[0] << endl << endl;

cout << "\tRight camera rms error: " << endl;

cout << rms_error << endl << endl;



/* ---------------------------------- */
/* Right camera calibration finished */
/* ---------------------------------- */



// Now, we will calibrate Left camera:
cout << "\tLeft camera calibration" << endl;

// We need to reset some variables
corners_2D.clear();

while ( 1 )
{
// Set the directory of the next image to load
sprintf ( directory, "./Calibracion_damero/Left_Capture%.04d.png", L_count++ );

// Load it
image = imread ( directory, CV_LOAD_IMAGE_COLOR );

// Check if it has loaded an image
if ( !image.data )
{
cout << endl << --L_count << " left images found" << endl;   // We need to reduce once the
    count

if ( L_count == 0 )   // No images found
{
error = 2;
throw error;
}
else
break;
}

// We will work with this copy, because some functions can overwrite the original
// one
image_copy = image.clone();

#ifdef VERBOSE
cout << L_count << " left images found" << endl;
#endif

#ifdef SHOW_RESULTS
cout << "Showing image... wait or press a key to continue" << endl;

imshow ( "Display window", image_copy );   // Show the image

waitKey ( 5000 );   // Wait up to 5 seconds for a pressed key to continue
#endif
```

```
// We convert the image to gray
cvtColor ( image_copy, gray, CV_RGB2GRAY );    // CV_BGR2GRAY or CV_RGB2GRAY

// We founded a Chessboard?
bool found = findChessboardCorners ( gray, pattern_size, corners_2D );

if ( found )
{
// We call this function to get a better aproximation to the corners of the board
cornerSubPix ( gray,
                          corners_2D,
                          Size ( 11, 11 ),    // Size of the window to search
                          Size ( -1, -1 ),    // No size for singularities in matrix
TermCriteria ( CV_TERMCRIT_EPS + CV_TERMCRIT_ITER, 30, 0.1 ) );

#ifdef SHOW_RESULTS
#ifdef VERBOSE
cout << "Showing founded Chessboard... wait or press a key to continue" << endl;
#endif

drawChessboardCorners ( image_copy, pattern_size, corners_2D, found);

imshow ( "Display window", image_copy );    // Show the image

waitKey ( 5000 );    // Wait up to 5 seconds for a pressed key to continue
#endif

// Save the corners
L_coordinates_2D.push_back ( corners_2D );

}
else
cout << "No Chessboard found on left image " << L_count << endl;

}

cout << "There are no more left images" << endl << endl;

// Different number of images will fail stereo calibration
if ( R_count != L_count )
{
error = 3;
throw error;
}

// 3D pattern has been already created and the image has the same size

// Once we have finished the corners searching, we resize them
L_coordinates_2D.resize ( L_count );

// Initialize the matrix
cameraMatrix[1] = Mat::eye ( 3, 3, CV_64F );

rms_error = calibrateCamera ( coordinates_3D, L_coordinates_2D, imageSize,
cameraMatrix[1],    // Intrinsics
distorsion_Coefficients[1],    // Extrinsics
rotation_vectors, translation_vectors,
//CV_CALIB_USE_INTRINSIC_GUESS | CV_CALIB_RATIONAL_MODEL);
CV_CALIB_RATIONAL_MODEL);
```

```
// Show the results:
cout << "\tLeft camera matrix: INTRINSIC" << endl << endl;

cout << cameraMatrix[1] << endl << endl;

cout << "\tLeft camera matrix: EXTRINSIC" << endl << endl;

cout << distorsion_Coefficients[1] << endl << endl;

cout << "\tLeft camera rms error: " << endl;

cout << rms_error << endl << endl;



/* ------------------------------- */
/* Left camera calibration finished */
/* ------------------------------- */



/* ------------------ */
/* Stereo Calibration */
/* ------------------ */

cout << "Calibrating stereo" << endl;

// Matrices used now: Rotation, Translation, Essential, Fundamental;

rms_error = stereoCalibrate ( coordinates_3D, R_coordinates_2D, L_coordinates_2D,
cameraMatrix[0], distorsion_Coefficients[0],
cameraMatrix[1], distorsion_Coefficients[1],
imageSize, Rotation, Translation, Essential, Fundamental,
TermCriteria(CV_TERMCRIT_ITER+CV_TERMCRIT_EPS, 100, 1e-5),
CV_CALIB_FIX_ASPECT_RATIO +
CV_CALIB_SAME_FOCAL_LENGTH +
CV_CALIB_RATIONAL_MODEL +
CV_CALIB_FIX_K3 + CV_CALIB_FIX_K4 + CV_CALIB_FIX_K5);

// Show the results:
cout << "\tCameras ROTATION matrix:" << endl << endl;

cout << Rotation << endl << endl;

cout << "\tCameras TRANSLATION:" << endl << endl;

cout << Translation << endl << endl;

cout << "\tCameras ESSENTIAL matrix:" << endl << endl;

cout << Essential << endl << endl;

cout << "\tCameras FUNDAMENTAL:" << endl << endl;

cout << Fundamental << endl << endl;

cout << "\tStereo rms error: " << endl;
```

```
cout << rms_error << endl << endl;



/* ---------------------------- */
/* Stereo Calibration finished */
/* ---------------------------- */



cout << "Exit program" << endl;

// end OK
return 0;

}   // end try

catch ( int error )   // get error code
{
// Show that error
cout << "\t\t ERROR" << endl;

cout << endl;
cout << "\t+------------------------------------------------------------+" << endl;

switch ( error )
{
case 1:

cout << "\tThere are not Right images!";
break;

case 2:

cout << "\tThere are not Left images!";
break;

case 3:

cout << "\tThere are not the same number of images!";
break;

default:

break;

}

cout << " Exiting program..." << endl;

cout << "\t+------------------------------------------------------------+" << endl;
cout << endl;

cout << "\t\tEND ERROR" << endl;

// end error
return -1;

}   // end catch
```

```
return 0;

}   // end main



}
```

A.4 Anexo 2

Programa de captura de imágenes en estéreo.

```
/*
* Name        : main.cpp
* Author      : Mario Luis ÁLvarez Pastor
* Version     : 0.0.1
* Copyright   : Free code
* Description : OpenCV use (with pthreads) for capturing images with 2 cameras
*               to do stereo vision
*/



/* -------- */
/* INCLUDES */
/* -------- */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <time.h>
#include <sys/time.h>

// Most important includes:
#include <pthread.h>
#include <semaphore.h>
#include <opencv2/opencv.hpp>



/* ---------- */
/* NAMESPACES */
/* ---------- */

using namespace cv;
using namespace std;



/* ------- */
/* DEFINES */
/* ------- */

#define KEY_ESC   27

#define DEFAULT_R   0
#define DEFAULT_L   1

// Uncoment this to get a verbose terminal program
//#define VERBOSE
// Uncoment this to get a verbose terminal program on threads
//#define VERBOSE_THREADS
// Uncoment this to get a verbose terminal program on savings
//#define VERBOSE_SAVINGS
```

```
/* -------------- */
/* FUNCTION CALLS */
/* -------------- */

void *captureLEFT ( void * );
void *captureRIGHT ( void * );



/* --------- */
/* VARIABLES */
/* --------- */

// Pressed key
int key = 0;
int frame_count = 0;

// finish flag
bool finish = false;

// Threads
pthread_t thread_1, thread_2;

// Mutex
pthread_mutex_t captureR = PTHREAD_MUTEX_INITIALIZER,
                                captureL = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t SaveR = PTHREAD_MUTEX_INITIALIZER,
                                SaveL = PTHREAD_MUTEX_INITIALIZER;

// Semaphores
sem_t Capture_synchronitation, Sem_captureR, Sem_captureL;
sem_t Show_Rsynchronitation, Show_Lsynchronitation;
sem_t Sem_SaveR, Sem_SaveL;

// Window names
char wndname_right[] = "rigth_cam";
char wndname_left[] = "left_cam";

// Camera structures
VideoCapture Right_cam, Left_cam;

// Frames
Mat Rframe, Lframe;

// Saves
Mat Rsave, Lsave;



/* ---- */
/* MAIN */
/* ---- */

int main ( int argc, char * argv[] )
{
// error number
int error = 0;
```

```
try   // Always use a try - catch structure to get errors
{
// Open the cameras
Left_cam.open  ( DEFAULT_L );
Right_cam.open ( DEFAULT_R );
/*
// Now config both cameras
// cv::CAP_PROP_MODE = 9
// cv::CAP_PROP_SETTINGS = 37
// cv::CAP_PROP_DC1394_MODE_MANUAL = -3
Right_cam.set ( 9, -3 );
Left_cam.set  ( 9, -3 );

cout << endl;

cout << "Capture mode R: " << Right_cam.get ( -3 ) << endl;
cout << "Capture mode L: " << Left_cam.get  ( -3 ) << endl;

cout << endl;
*/
/*
cv::CAP_PROP_POS_MSEC =0,
cv::CAP_PROP_BRIGHTNESS =10,
cv::CAP_PROP_CONTRAST =11,
cv::CAP_PROP_SATURATION =12,
cv::CAP_PROP_HUE =13,
cv::CAP_PROP_GAIN =14,
cv::CAP_PROP_EXPOSURE =15,
*/

Right_cam.set ( 10, 128 );   // Brightness
Left_cam.set  ( 10, 128 );

Right_cam.set ( 11, 50 );   // Contrast
Left_cam.set  ( 11, 50 );

Right_cam.set ( 12, 100 );   // Saturation
Left_cam.set  ( 12, 100 );

Right_cam.set ( 13, 180 );   // Hue
Left_cam.set  ( 13, 180 );

Right_cam.set ( 14, 750 );   // Gain
Left_cam.set  ( 14, 750 );

Right_cam.set ( 15, 30 );   // Shutter
Left_cam.set  ( 15, 30 );

// Initialize the semaphores
sem_init ( &Capture_synchronitation, 0, 2 );
sem_init ( &Sem_captureR,            0, 1 );
sem_init ( &Sem_captureL,            0, 1 );
sem_init ( &Show_Rsynchronitation,   0, 1 );
sem_init ( &Show_Lsynchronitation,   0, 1 );
sem_init ( &Sem_SaveR,               0, 0 );
sem_init ( &Sem_SaveL,               0, 0 );

// First of all, check if the camera is avaliable
```

```
if ( !Left_cam.isOpened ( ) )
{
// end with error
error = 3;
throw error;
}
else
{
cout << "\tLeft camera operative!"   << endl;
// Open and give a name to left window
namedWindow ( wndname_left, CV_WINDOW_AUTOSIZE );

// Create the thread
error = pthread_create ( &thread_1,
NULL,   // &attribute
captureLEFT,
NULL );

if ( error )
{
error = 1;
throw error;
}
}



// Secondly, check if the camera is avaliable
if ( !Right_cam.isOpened ( ) )
{
// end with error
error = 4;
throw error;
}
else
{
cout << "\tRight camera operative!"   << endl;
// Open and give a name to right window
namedWindow ( wndname_right, CV_WINDOW_AUTOSIZE );

// Create the thread
error = pthread_create ( &thread_2,
NULL,   // &attribute
captureRIGHT,
NULL );

if ( error )
{
error = 2;
throw error;
}

}

// Small wait to avoid overload cout
waitKey ( 5 );

cout << "Program ready to capture" << endl;
```

```
// main loop
while ( !finish )
{
// Wait 5ms and get pressed key
key = waitKey ( 5 );

key = tolower ( key );

// We will wait until both threads are waiting synchronitation
sem_wait ( &Sem_captureR );
sem_wait ( &Sem_captureL );

#ifdef VERBOSE
cout << "syncronized" << endl;
#endif

// Then, we release the capture
sem_post ( &Capture_synchronitation );
sem_post ( &Capture_synchronitation );

// Finish if we press e
if ( ( key == 'e' ) || ( key == KEY_ESC ) )
{
finish = true;

#ifdef VERBOSE
cout << "finish" << endl;
#endif

break;
}

// Save the captures after joining threads
else if ( key == 's' )
{
// Temporal names for images to save
char temp_name[80];
// Temporal parameters for the images
vector<int> compression_params;
compression_params.push_back( CV_IMWRITE_PNG_COMPRESSION );
compression_params.push_back( 3 );

#ifdef VERBOSE_SAVINGS
cout << "save" << endl;
#endif

// Wait until both captures are done
sem_wait ( &Sem_SaveR );
sem_wait ( &Sem_SaveL );

// Make sure not to read at the same time of the writting
pthread_mutex_lock ( &SaveR );
pthread_mutex_lock ( &SaveL );

#ifdef VERBOSE_SAVINGS
cout << "saving" << endl;
#endif

sprintf ( temp_name, "./saved_images/Right_Capture%.04d.png", frame_count );
```

```
imwrite ( temp_name, Rsave, compression_params );

sprintf ( temp_name, "./saved_images/Left_Capture%.04d.png", frame_count );
imwrite ( temp_name, Lsave, compression_params );

frame_count++;

cout << frame_count << " captures token" << endl;

// Release mutex
pthread_mutex_unlock ( &SaveR );
pthread_mutex_unlock ( &SaveL );

#ifdef VERBOSE_SAVINGS
cout << "saved" << endl;
#endif
}

#ifdef VERBOSE
cout << "show" << endl;
#endif

// Then, we wait the showing
sem_wait ( &Show_Rsynchronitation );
sem_wait ( &Show_Lsynchronitation );

// Make sure not to read at the same time of the writting
pthread_mutex_lock ( &captureR );
pthread_mutex_lock ( &captureL );

#ifdef VERBOSE
cout << "showing" << endl;
#endif

// Show the images
imshow ( wndname_left, Lframe );
imshow ( wndname_right, Rframe );

// Release the mutex
pthread_mutex_unlock ( &captureR );
pthread_mutex_unlock ( &captureL );

#ifdef VERBOSE
cout << "showed" << endl;
#endif

}   // end while

error = pthread_join ( thread_1, NULL );

if ( error )
{
error = 5;
throw error;
}

error = pthread_join ( thread_2, NULL );

if ( error )
```

```
{
error = 6;
throw error;
}

// Destroy semaphores
sem_destroy ( &Capture_synchronitation );
sem_destroy ( &Sem_captureR );
sem_destroy ( &Sem_captureL );

cout << "Exit program" << endl;

cout << "\tReleasing cameras"   << endl;

// Release cameras
Left_cam.release ( );
Right_cam.release( );

cout << "\tCameras released"   << endl;

// end OK
return 0;

}   // end try

catch ( int error )   // get error code
{
// Show that error
cout << "\t\t ERROR" << endl;

cout << endl;
cout << "\t+-------------------------------------------------------------+" << endl;

switch ( error )
{
case 1:

cout << "\tThe program could not create thread 1"   << endl;
break;

case 2:

cout << "\tThe program could not create thread 2"   << endl;
break;

case 3:

cout << "\tThe program could not open left camera"   << endl;
break;

case 4:

cout << "\tThe program could not open Right camera"   << endl;
break;

case 5:

cout << "\tThe program could not join thread 1"   << endl;
break;
```

```
case 6:

cout << "\tThe program could not join thread 2"    << endl;
break;

default:

break;

}

cout << "\t+-------------------------------------------------------------+" << endl;
cout << endl;

// Destroy semaphores
sem_destroy ( &Capture_synchronitation );
sem_destroy ( &Sem_captureR );
sem_destroy ( &Sem_captureL );

// Release cameras
Left_cam.release  ( );
Right_cam.release ( );

cout << "\tCameras released" << endl;

cout << "\t\tEND ERROR" << endl;

// end error
return -1;

}   // end catch

return 0;

}   // end main



/* --------- */
/* FUNCTIONS */
/* --------- */

// Function to capture an image with a thread
void *captureRIGHT ( void * )
{
//int Rcount = 0;

cout << "initialized right thread" << endl;

// Capture images until finish
while ( !finish )
{
#ifdef VERBOSE_THREADS
cout << "write R" << endl;
#endif

#ifdef VERBOSE_THREADS
cout << "sync R" << endl;
```

```
#endif

// Thread ready to syncronize
sem_post ( &Sem_captureR );

#ifdef VERBOSE_THREADS
cout << "w8ing sync R" << endl;
#endif

// Wait until synchronitation
sem_wait ( &Capture_synchronitation );

// Make sure not to write at the same time of the reading
pthread_mutex_lock ( &captureR );

switch ( key )
{
case 's':

#ifdef VERBOSE_SAVINGS
cout << "Taking right image" << endl;
#endif

// Make sure not to write at the same time of the reading
pthread_mutex_lock ( &SaveR );

// Get a new frame from camera
Right_cam >> Rsave;

// Thread ready to save
sem_post ( &Sem_SaveR );

// Release mutex
pthread_mutex_unlock ( &SaveR );

break;

default:

break;
}

#ifdef VERBOSE_THREADS
cout << "writting R" << endl;
#endif

// Get a new frame from camera
Right_cam >> Rframe;

//cout << "R counts: " << Rcount++ << endl;

// Release the mutex
pthread_mutex_unlock ( &captureR );

// Then, we release the showing
sem_post ( &Show_Rsynchronitation );

#ifdef VERBOSE_THREADS
cout << "writed R" << endl;
```

```
#endif
}

cout << "exiting right thread" << endl;

pthread_exit ( NULL );
}



void *captureLEFT ( void * )
{
//int Lcount = 0;

cout << "initialized left thread" << endl;

// Capture images until finish
while ( !finish )
{
#ifdef VERBOSE_THREADS
cout << "write L" << endl;
#endif

#ifdef VERBOSE_THREADS
cout << "sync L" << endl;
#endif

// Thread ready to syncronize
sem_post ( &Sem_captureL );

#ifdef VERBOSE_THREADS
cout << "w8ing sync L" << endl;
#endif

// Wait until synchronitation
sem_wait ( &Capture_synchronitation );

// Make sure not to write at the same time of the reading
pthread_mutex_lock ( &captureL );

switch ( key )
{
case 's':

#ifdef VERBOSE_SAVINGS
cout << "Taking left image" << endl;
#endif

// Make sure not to write at the same time of the reading
pthread_mutex_lock ( &SaveL );

// Get a new frame from camera
Left_cam >> Lsave;

// Thread ready to save
sem_post ( &Sem_SaveL );

// Release mutex
pthread_mutex_unlock ( &SaveL );
```

```
break;

default:

break;
}

#ifdef VERBOSE_THREADS
cout << "writting L" << endl;
#endif

// Get a new frame from camera
Left_cam >> Lframe;

//cout << "L counts: " << Lcount++ << endl;

// Release the mutex
pthread_mutex_unlock ( &captureL );

// Then, we release the showing
sem_post ( &Show_Lsynchronitation );

#ifdef VERBOSE_THREADS
cout << "writed L" << endl;
#endif
}

cout << "exiting left thread" << endl;

pthread_exit ( NULL );
}
```

A.5 Anexo 3

Programa principal de la extracción de información de profundidad a partir de visión en estéreo, con OpenCV y OpenGL.

main.cpp:

```
/*
* Name        : main.cpp
* Author      : Mario Luis ÁLvarez Pastor
* Version     : 0.0.2
* Copyright   : Free code
* Description : OpenGL use for creating a window and stereo vision with CUDA
*/



/*
* Includes
*/

#include "../include/common.h"
#include "../include/graphics.h"



/*
* Main function
*/
int main(int argc, char * argv[])
{
// namespace is for being allocated for the entire duration of the program
APPLICATION::WindowProps windowProps;   // Window properties

try
{
// Fill the structure
windowProps.position.x  = 0;
windowProps.position.y  = 0;
windowProps.width       = WINDOW_WIDTH;
windowProps.height      = WINDOW_HEIGHT;
windowProps.bpp         = BPP;
windowProps.bFullScreen = FULL_SCREEN;
strcpy ( windowProps.title, WINDOW_TITLE );

// Initialize GLUT
glutInit ( &argc, argv );

// Initialize the window properties
windowProps = APPLICATION::initWindow ( windowProps );

APPLICATION::registerFunctions ( );

// Register the exit function
// This function is not part of GLUT, is part of ANSI (stdlib.h)
// After have called exit() in the code, this function is called before ending
atexit ( APPLICATION::OnEnd );

// Main loop where events are processed. The only way to go out is with exit()
```

```
glutMainLoop ( );

// end OK
return 0;

}   // end try

catch ( GLenum glErrorCode )
{
std::cout << std::endl;
std::cout << "\t+--------------------------------------------------------------+"  << std::
    endl;
std::cout << "\t" << gluErrorString ( glErrorCode )                               << std::
    endl;
std::cout << "\t+--------------------------------------------------------------+"  << std::
    endl;
std::cout << std::endl;

// end Error
return -1;

}   // end catch

return 0;

}   // end main
```

A.6 Anexo 4

Archivo que se encarga de gestionar parámetros del sistema.

system.cpp:

```
/*
* Name        : system.cpp
* Author      : Mario Luis ÁLvarez Pastor
* Version     : 0.0.1
* Copyright   : Free code
* Description : OpenGL use for creating a window
*/



/*
* Includes
*/

#include "../include/system.h"



/*
* Functions
*/

namespace APPLICATION
{
// Calculates the elapsed milliseconds since application started
//
// return:
//    long: Elapsed milliseconds
//
long System::GetMilliseconds()
{
static time_t startingTime = time(NULL);
struct timeval tv;
struct timezone tz;

gettimeofday(&tv, &tz);

float milliseconds = 1000.0f * (tv.tv_sec - startingTime) + (float)tv.tv_usec / 1000;

return (long)milliseconds;
}



// Returns last error code
//
// return:
//    int: Error code
//
int System::GetSystemError()
{
return errno;
}
```

```
// Shows an error message
//
// param:
//    int errorCode: error code to show
//
void System::ShowError(int errorCode)
{
if ( errorCode != 0 )
{
printf("ERROR: %s", strerror(errorCode));
}
}



// Shows a message on screen or console
//
// param:
//    char *title: Message title
//
//    char *message: Message text
//
//    bool bErrorMsg:
//                                              - true:  error message
//                                              - false: informative message
//
void System::ShowMessage(char *title, char *message, bool bErrorMsg)
{
if ( bErrorMsg )
printf("INFO  # %s: %s\n", title, message);
else
printf("ERROR # %s: %s\n", title, message);
}


};   // namespace APPLICATION
```

A.7 Anexo 5

Encabezado del archivo system.cpp.

system.h:

```
/*
* Name        : system.h
* Author      : Mario Luis ÁLvarez Pastor
* Version     : 0.0.1
* Copyright   : Free code
* Description : Header for the OpenGL proyect
*/



#ifndef SYSTEM_H
#define SYSTEM_H



/*
* Includes
*/

#include "common.h"

#include <errno.h>
#include <time.h>
#include <sys/time.h>



/*
* References
*/

namespace APPLICATION
{

class System
{
public:

// Calculates the elapsed milliseconds since application started
static long GetMilliseconds();

// Returns last error code
static int GetSystemError();

// Shows an error message
static void ShowError(int errorCode);

// Shows a message on screen or console
static void ShowMessage(char *title, char *message, bool bErrorMsg);

};

};   // namespace APPLICATION
```

```
#endif   // SYSTEM_H
```

A.8 Anexo 6

Código del archivo graphics.cpp, encargado de inicializar y gestionar OpenGL en nuestra aplicación.

graphics.cpp:

```
/*
* Name        : graphics.cpp
* Author      : Mario Luis ÁLvarez Pastor
* Version     : 0.0.2
* Copyright   : Free code
* Description : OpenGL use for creating a window and stereo vision with CUDA
*/



/*
* Includes
*/

#include "../include/graphics.h"



/*
* Variables
*/

// Create the application class as a global, for being accessible from
// the message function of the window
APPLICATION::Application g_Application;



/*
* Functions
*/

namespace APPLICATION
{
// Function to initialize window properties
//
// param:
//    WindowProps windowProperties:   struct with all window properties
//
WindowProps initWindow ( WindowProps windowProperties )
{
int screenWidth, screenHeight;   // Window size

// Initialize the application
g_Application.OnInit( windowProperties );

// Get the screen resolution
screenWidth  = glutGet( GLUT_SCREEN_WIDTH  );
screenHeight = glutGet( GLUT_SCREEN_HEIGHT );

if ( windowProperties.bFullScreen )
{
windowProperties.position.x = 0;
windowProperties.position.y = 0;
```

```
}
else
{
windowProperties.position.x = ( screenWidth  - windowProperties.width  ) / 2;
windowProperties.position.y = ( screenHeight - windowProperties.height ) / 2;
}

// Window position
glutInitWindowPosition( windowProperties.position.x, windowProperties.position.y );

// Window dimensions
glutInitWindowSize( windowProperties.width, windowProperties.height );

// Initialize video mode
glutInitDisplayMode( GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH );

// Create the main window
glutCreateWindow( windowProperties.title );

if ( windowProperties.bFullScreen )
glutFullScreen();       // Full screen mode

// Initialize OpenGL
g_Application.OnCreateWindow( windowProperties );

return windowProperties;

}   // initWindow



// Function which register all events
void registerFunctions(void)
{
// Register some functions to handle some events
glutReshapeFunc   ( OnResize             );
glutKeyboardFunc  ( OnKeyDown            );
glutKeyboardUpFunc( OnKeyUp              );
glutSpecialFunc   ( OnSpecialKeyDown     );
glutSpecialUpFunc ( OnSpecialKeyUp       );
glutIdleFunc      ( OnIdle               );
glutDisplayFunc   ( OnRender             );
}



// Function called when a key is pressed
//
// param:
//    int keyCode: Key pressed
//
//    int x: X coordinate of the mouse
//
//    int y: Y coordinate of the mouse
//
void OnKeyDown( unsigned char key, int x, int y )
{
g_Application.OnKeyDown( key );
}
```

```
// Function called when a key is freed
//
// param:
//    int keyCode: Key freed
//
//    int x: X coordinate of the mouse
//
//    int y: Y coordinate of the mouse
//
void OnKeyUp( unsigned char key, int x, int y )
{
g_Application.OnKeyUp( key );
}



// Function called when a special key is pressed
//
// param:
//    int keyCode: Special key pressed
//
//    int x: X coordinate of the mouse
//
//    int y: Y coordinate of the mouse
//
void OnSpecialKeyDown( int key, int x, int y )
{
switch ( key )
{
case GLUT_KEY_LEFT:

g_Application.OnKeyDown( 0x25 );

break;

case GLUT_KEY_UP:

g_Application.OnKeyDown( 0x26 );

break;

case GLUT_KEY_RIGHT:

g_Application.OnKeyDown( 0x27 );

break;

case GLUT_KEY_DOWN:

g_Application.OnKeyDown( 0x28 );

break;

default:

g_Application.OnKeyDown( key );
```

```
break;
}
}



// Function called when a special key is freed
//
// param:
//    int keyCode: Special key freed
//
//    int x: X coordinate of the mouse
//
//    int y: Y coordinate of the mouse
//
void OnSpecialKeyUp( int key, int x, int y )
{
switch ( key )
{
case GLUT_KEY_LEFT:

g_Application.OnKeyUp( 0x25 );

break;

case GLUT_KEY_UP:

g_Application.OnKeyUp( 0x26 );

break;

case GLUT_KEY_RIGHT:

g_Application.OnKeyUp( 0x27 );

break;

case GLUT_KEY_DOWN:

g_Application.OnKeyUp( 0x28 );

break;

default:

g_Application.OnKeyUp( key );

break;
}
}



// Function called when changing the window size
//
// param:
//    int width: width on pixels for the window
//
```

```
//    int height: height on pixels for the window
//
void OnResize( int width, int height )
{
g_Application.OnResize( width, height );
}



// Function called when nothing happens
void OnIdle()
{
g_Application.OnIdle();   // Main process

if ( g_Application.IsEnded() )
exit ( 0 );                                    // Only can exit glutMainLoop() with exit
else
glutPostRedisplay();            // Force repaint the window. Invokes OnRender() function
}



// Function called when system is ready to render
void OnRender()
{
g_Application.OnRender();

//glutSetWindowTitle( g_Application.GetMsgTitle() );

glutSwapBuffers();   // Dumps to the current window the rendered image
}



// Function called on program ending
void OnEnd()
{
g_Application.OnEnd();
}

}   // namespace APPLICATON
```

A.9 Anexo 7

Encabezado del archivo graphics.cpp.

graphics.h:

```
/*
* Name        : graphics.h
* Author      : Mario Luis ÁLvarez Pastor
* Version     : 0.0.1
* Copyright   : Free code
* Description : Header for the OpenGL proyect
*/



#ifndef GRAPHICS_H
#define GRAPHICS_H

/*
* Includes
*/

#include "graphicApplication.h"



/*
* References
*/

namespace APPLICATION
{
// Function to initialize window properties
WindowProps initWindow(WindowProps windowProperties);

// Function which register all events
void registerFunctions(void);

// Function called when a key is pressed
void OnKeyDown( unsigned char key, int x, int y );
// Function called when a key is freed
void OnKeyUp( unsigned char key, int x, int y );
// Function called when a special key is pressed
void OnSpecialKeyDown( int key, int x, int y );
// Function called when a special key is freed
void OnSpecialKeyUp( int key, int x, int y );

// Function called when changing the window size
void OnResize( int width, int height );

// Function called when nothing happens
void OnIdle();

// Function called when system is ready to render
void OnRender();

// Function called on program ending
void OnEnd();
```

```
}   // namespace APPLICATION


#endif   // GRAPHICS_H
```

A.10 Anexo 8

Código del archivo graphicApplication.cpp, encargado de manejar las funciones y los cálculos realizados en el bucle del programa, así como de gestionar la respuesta a los diferentes estímulos que puede recibir el programa, como son la pulsación de las teclas.

graphicApplication.cpp:

```
/*
* Name        : graphicApplication.cpp
* Author      : Mario Luis ÁLvarez Pastor
* Version     : 0.0.1
* Copyright   : Free code
* Description : OpenGL use for creating a window and stereo vision with CUDA
*/



/*
* Includes
*/

#include <memory.h>

#include "../include/graphicApplication.h"
#include "../include/system.h"
#include "../include/stereo.h"



/*
* Namespaces
*/

using namespace std;



/*
* Functions
*/

namespace APPLICATION
{
// Create the stereo class as a global
#ifdef SAVE
#ifdef DISPLAY
STEREO_VISION::Stereo g_Stereo ( true, true );   // Display and save
#else
STEREO_VISION::Stereo g_Stereo ( false, true );   // Only save
#endif
#else
#ifdef DISPLAY
STEREO_VISION::Stereo g_Stereo ( true, false );   // Only display
#else
STEREO_VISION::Stereo g_Stereo ( false, false );   // No save and no display
#endif
#endif
```

```
// Constructor
Application::Application ( )
{
memset ( &m_WindowProps, 0, sizeof ( WindowProps ) );
m_MinDepthBuffer = MIN_DEPTH_BUFFER;
m_MaxDepthBuffer = MAX_DEPTH_BUFFER;
m_bEnd = false;
}



// Destructor
Application::~Application ( )
{
#ifdef __WIN_OPEN_GL__
ReleaseRenderContext ( );
#endif
}



// Function called on program initialization and before window creation
//
// param:
//    WindowProps windowProps: Window properties
//
void Application::OnInit ( WindowProps windowPropert )
{
m_WindowProps = windowPropert;

std::cout << std::endl;
std::cout << "\t+-------------------------------------------------------------+"  << std::
    endl;
std::cout << "\t|                      START APPLICATION                      |"  << std::
    endl;
std::cout << "\t+-------------------------------------------------------------+"  << std::
    endl;
std::cout << std::endl;

std::cout << "\tSELECTED CONFIG:" << std::endl;
std::cout << std::endl;

std::cout << "Selected dataset: " << g_Stereo.get_dataset ( ) << std::endl;

std::cout << std::endl;

#ifdef SHOW_FPS
std::cout << "fps will be shown" << std::endl;
#endif

#ifdef GPU_INFO
std::cout << "GPU fps will be shown" << std::endl;
#endif

#ifdef GPU_MEMORY_INFO
std::cout << "GPU memory fps will be shown" << std::endl;
#endif

#ifdef SAVE
```

```
std::cout << "Disparities will be saved" << std::endl;
#endif

#ifdef DISPLAY
std::cout << "Disparities will be shown" << std::endl;
#endif

#ifdef NORMALIZE_DISPARITY
std::cout << "Disparities will be normalized" << std::endl;
#endif

#ifdef DISPARITY_COLOR
std::cout << "Disparity color mode enabled" << std::endl;
#endif

std::cout << "Number of disparities: " << DISPARITIES_NUMBER << std::endl;
std::cout << "Block size for disparities: " << BLOCK_SIZE << std::endl;

std::cout << std::endl;

// Get calibration for the cameras
g_Stereo.calibrate ( );

// Initialize time
GetMsElapsed ( );
}



// Function called on program ending
void Application::OnEnd ( )
{
std::cout << std::endl << "\tFinal parameters used:" << std::endl;

g_Stereo.printParameters ( );

std::cout << "Prefilter sobel: " << g_Stereo.Stereo_algorithm->getPreFilterType ( ) << std
    ::endl;

std::cout << std::endl;
std::cout << "\t+-------------------------------------------------------------+"  << std::
    endl;
std::cout << "\t|                      END APPLICATION                        |"  << std::
    endl;
std::cout << "\t+-------------------------------------------------------------+"  << std::
    endl;
std::cout << std::endl;

}



// Returns if application should end
//
// return:
//    bool:
//                        - true:  The application should end
//                                                - false: The application continues
//
```

```
bool Application::IsEnded()
{
return m_bEnd;
}



// Return if application is on full screen mode
//
// return:
//    bool:
//                                          - true:  The application is on full screen
    mode
//                                          - false: The application is not
//
bool Application::IsFullScreen()
{
return m_WindowProps.bFullScreen;
}



// Function called on program initialization and AFTER window creation
void Application::OnCreateWindow( WindowProps windowProps )
{
m_WindowProps = windowProps;

#ifdef __WIN_OPEN_GL__
CreateRenderContext ( );
#endif

glHint ( GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST );   // Perspective calculus
glHint ( GL_POLYGON_SMOOTH_HINT        , GL_NICEST );

m_MinDepthBuffer = MIN_DEPTH_BUFFER;
m_MaxDepthBuffer = MAX_DEPTH_BUFFER;

glClearDepth ( 1.0f );
}



// Function called when changing the window size
//
// param:
//    int width: width on pixels for the window
//
//    int height: height on pixels for the window
//
void Application::OnResize( int width, int height )
{
m_WindowProps.width =  width;
m_WindowProps.height = height;

glViewport ( 0, 0, width, height );   // Visible area

glMatrixMode ( GL_PROJECTION );   // Select Proyection Matrix
glLoadIdentity ( );                               // Reset Proyection Matrix
```

```
gluPerspective ( (GLfloat)FOV, (GLfloat)width / (GLfloat)height, m_MinDepthBuffer,
    m_MaxDepthBuffer );
}



// Function called when a key is pressed
//
// param:
//    int keyCode: Key pressed
//
void Application::OnKeyDown( int keyCode )
{
switch ( keyCode )
{
case KEY_ESC:

m_bEnd = true;
break;

case 'p': case 'P':

g_Stereo.printParameters();

break;

case 'h': case 'H':

cout << "\tCommand info: " << endl;
cout << "\tCapital letters will increase the value, otherwise it will decrease" << endl;
cout << "\t\tS/s: Switch on or off the Sobel filter" << endl;
cout << "\t\tC/c: Increase or decrease prefilterCap" << endl;
cout << "\t\tZ/z: Increase or decrease prefilter size" << endl;
cout << "\t\tB/b: Increase or decrease block size" << endl;
cout << "\t\tM/m: Increase or decrease min disparity" << endl;
cout << "\t\tD/d: Increase or decrease number of disparities" << endl;
cout << "\t\tT/t: Increase or decrease texture threshold" << endl;
cout << "\t\tU/u: Increase or decrease uniqueness ratio" << endl;
cout << "\t\tW/w: Increase or decrease Speckle window size" << endl;
cout << "\t\tR/r: Increase or decrease Speckle range" << endl;
cout << "\t\t1/2: Increase or decrease disparity 12 max diff" << endl;
cout << "\t\t3/4: Increase or decrease min porcess value" << endl;
cout << "\t\t5/6: Increase or decrease max porcess value" << endl;

break;

case 's': case 'S':

// Switch sobel filter
switch ( g_Stereo.Stereo_algorithm->getPreFilterType ( ) )
{
case 0:

g_Stereo.Stereo_algorithm->setPreFilterType ( cv::StereoBM::PREFILTER_XSOBEL );
break;

case cv::StereoBM::PREFILTER_XSOBEL:

g_Stereo.Stereo_algorithm->setPreFilterType ( 0 );
```

```
break;
}

cout << "Prefilter Sobel: " << g_Stereo.Stereo_algorithm->getPreFilterType ( ) << endl;

break;

case 'C':

g_Stereo.Stereo_algorithm->setPreFilterCap ( g_Stereo.Stereo_algorithm->getPreFilterCap ( )
     + 1 );

cout << "Prefilter Cap: " << g_Stereo.Stereo_algorithm->getPreFilterCap ( ) << endl;

break;

case 'c':

if ( g_Stereo.Stereo_algorithm->getPreFilterCap ( ) > 1 )
g_Stereo.Stereo_algorithm->setPreFilterCap ( g_Stereo.Stereo_algorithm->getPreFilterCap ( )
     - 1 );

cout << "Prefilter Cap: " << g_Stereo.Stereo_algorithm->getPreFilterCap ( ) << endl;

break;

case 'Z':

g_Stereo.Stereo_algorithm->setPreFilterSize ( g_Stereo.Stereo_algorithm->getPreFilterSize (
     ) + 1 );

cout << "Prefilter size: " << g_Stereo.Stereo_algorithm->getPreFilterSize ( ) << endl;

break;

case 'z':

g_Stereo.Stereo_algorithm->setPreFilterSize ( g_Stereo.Stereo_algorithm->getPreFilterSize (
     ) - 1 );

cout << "Prefilter size: " << g_Stereo.Stereo_algorithm->getPreFilterSize ( ) << endl;

break;

case 'B':

if ( g_Stereo.SADWindowSize == 0 )
g_Stereo.SADWindowSize++;
else if ( g_Stereo.SADWindowSize < 51 )
g_Stereo.SADWindowSize += 2;

g_Stereo.Stereo_algorithm->setBlockSize ( g_Stereo.SADWindowSize );

cout << "Block size: " << g_Stereo.SADWindowSize << endl;

break;

case 'b':
```

```
if ( g_Stereo.SADWindowSize < 5 ) {}
// Do nothing
else
g_Stereo.SADWindowSize -= 2;

g_Stereo.Stereo_algorithm->setBlockSize ( g_Stereo.SADWindowSize );

cout << "Block size: " << g_Stereo.SADWindowSize << endl;

break;

case 'M':

g_Stereo.Stereo_algorithm->setMinDisparity ( g_Stereo.Stereo_algorithm->getMinDisparity ( )
     + 1 );

cout << "Min disparity: " << g_Stereo.Stereo_algorithm->getMinDisparity ( ) << endl;

break;

case 'm':

g_Stereo.Stereo_algorithm->setMinDisparity ( g_Stereo.Stereo_algorithm->getMinDisparity ( )
     - 1 );

cout << "Min disparity: " << g_Stereo.Stereo_algorithm->getMinDisparity ( ) << endl;

break;

case 'D':

if ( g_Stereo.disparities_number < 256 )
g_Stereo.disparities_number += 8;

g_Stereo.Stereo_algorithm->setNumDisparities ( g_Stereo.disparities_number );

cout << "Number of disparities: " << g_Stereo.disparities_number << endl;

break;

case 'd':

if ( g_Stereo.disparities_number < 16 ) {}
// Do nothing
else
g_Stereo.disparities_number -= 8;

g_Stereo.Stereo_algorithm->setNumDisparities ( g_Stereo.disparities_number );

cout << "Number of disparities: " << g_Stereo.disparities_number << endl;

break;

case 'T':

g_Stereo.Stereo_algorithm->setTextureThreshold ( g_Stereo.Stereo_algorithm->
    getTextureThreshold ( ) + 1 );

cout << "Texture threshold: " << g_Stereo.Stereo_algorithm->getTextureThreshold ( ) << endl
```

```
    ;

break;

case 't':

if ( g_Stereo.Stereo_algorithm->getTextureThreshold ( ) > 0)
g_Stereo.Stereo_algorithm->setTextureThreshold ( g_Stereo.Stereo_algorithm->
    getTextureThreshold ( ) - 1 );

cout << "Texture threshold: " << g_Stereo.Stereo_algorithm->getTextureThreshold ( ) << endl
    ;

break;

case 'U':

g_Stereo.Stereo_algorithm->setUniquenessRatio ( g_Stereo.Stereo_algorithm->
    getUniquenessRatio ( ) + 1 );

cout << "Uniqueness ratio: " << g_Stereo.Stereo_algorithm->getUniquenessRatio ( ) << endl;

break;

case 'u':

//if ( g_Stereo.Stereo_algorithm->getUniquenessRatio ( ) > 0 )
g_Stereo.Stereo_algorithm->setUniquenessRatio ( g_Stereo.Stereo_algorithm->
    getUniquenessRatio ( ) - 1 );

cout << "Uniqueness ratio: " << g_Stereo.Stereo_algorithm->getUniquenessRatio ( ) << endl;

break;

case 'W':

g_Stereo.Stereo_algorithm->setSpeckleWindowSize ( g_Stereo.Stereo_algorithm->
    getSpeckleWindowSize ( ) + 10 );

cout << "Speckle window size: " << g_Stereo.Stereo_algorithm->getSpeckleWindowSize ( ) <<
    endl;

break;

case 'w':

//if ( g_Stereo.Stereo_algorithm->getSpeckleWindowSize ( ) > 0 )
g_Stereo.Stereo_algorithm->setSpeckleWindowSize ( g_Stereo.Stereo_algorithm->
    getSpeckleWindowSize ( ) - 10 );

cout << "Speckle window size: " << g_Stereo.Stereo_algorithm->getSpeckleWindowSize ( ) <<
    endl;

break;

case 'R':

g_Stereo.Stereo_algorithm->setSpeckleRange ( g_Stereo.Stereo_algorithm->getSpeckleRange ( )
     + 2 );
```

```
cout << "Speckle range: " << g_Stereo.Stereo_algorithm->getSpeckleRange ( ) << endl;

break;

case 'r':

//if ( g_Stereo.Stereo_algorithm->getSpeckleRange ( ) > 0 )
g_Stereo.Stereo_algorithm->setSpeckleRange ( g_Stereo.Stereo_algorithm->getSpeckleRange ( )
    - 2 );

cout << "Speckle range: " << g_Stereo.Stereo_algorithm->getSpeckleRange ( ) << endl;

break;

case '1':

g_Stereo.Stereo_algorithm->setDisp12MaxDiff ( g_Stereo.Stereo_algorithm->getDisp12MaxDiff (
    ) + 1 );

cout << "Disp 12 max diff: " << g_Stereo.Stereo_algorithm->getSpeckleRange ( ) << endl;

break;

case '2':

//if ( g_Stereo.Stereo_algorithm->getDisp12MaxDiff ( ) > 0 )
g_Stereo.Stereo_algorithm->setDisp12MaxDiff ( g_Stereo.Stereo_algorithm->getDisp12MaxDiff (
    ) - 1 );

cout << "Disp 12 max diff: " << g_Stereo.Stereo_algorithm->getDisp12MaxDiff ( ) << endl;

break;

case '3':

if ( g_Stereo.min_disparity < 255 && g_Stereo.max_disparity > g_Stereo.min_disparity )
g_Stereo.min_disparity++;

cout << "Min process value: " << g_Stereo.min_disparity << endl;

break;

case '4':

if ( g_Stereo.min_disparity > 0 )
g_Stereo.min_disparity--;

cout << "Min process value: " << g_Stereo.min_disparity << endl;

break;

case '5':

if ( g_Stereo.max_disparity < 255 )
g_Stereo.max_disparity++;

cout << "Max process value: " << g_Stereo.max_disparity << endl;
```

```
break;

case '6':

if ( g_Stereo.max_disparity > 0 && g_Stereo.max_disparity > g_Stereo.min_disparity )
g_Stereo.max_disparity--;

cout << "Max process value: " << g_Stereo.max_disparity << endl;

break;

}

m_KeysPressed[ keyCode ] = true;
}



// Function called when a key is freed
//
// param:
//    int keyCode: key freed
//
void Application::OnKeyUp( int keyCode )
{
m_KeysPressed[ keyCode ] = false;
}



// Function called when nothing happens
void Application::OnIdle()
{
static int  numFps = 0;
static long msAcumulated = 0;
long        msElapsed = GetMsElapsed();

#ifdef SHOW_FPS
numFps++;
msAcumulated += msElapsed;

// Each MS_SHOW_FPS milliseconds we show fps
if ( msAcumulated >= MS_SHOW_FPS )
{
std::cout << "fps: "
<< 1000.0f * (float)numFps / (float)msAcumulated
<< std::endl;

numFps = 0;
msAcumulated -= MS_SHOW_FPS;
}
#endif
}



// Function called when system is ready to render
void Application::OnRender()
{
```

```
fColorRGBA color = { 0.0f, 0.0f, 0.0f, 1.0f };   // Background color

glClearColor ( color.r, color.g, color.b, color.a );     // Background color
glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );   // Clear screen and z-buffer

glMatrixMode ( GL_PROJECTION );
glLoadIdentity ( );

// fov, aspect, near, far
gluPerspective ( 60, 1, 1, 5 );
gluLookAt ( 0, 0, -2,   // eye
0, 0,  0,   // center
0, 1,  0 ); // up

glMatrixMode ( GL_MODELVIEW );
glLoadIdentity ( );

/*
    ------------------------------------------------------------------------------------------------------------
    */
// Render the scene

// Set next image
g_Stereo.next_index ( );
// Read next image
g_Stereo.next_image ( );
// Calculate the image to show
g_Stereo.stereo_vision ( );

// Show them
if ( g_Stereo.get_display ( ) )
{
// Create texture
static cv::Mat flipped_image;

GLuint texture;

// We need to flip the data to be used with openGL
#ifdef WATERSHED_ALGORITHM
g_Stereo.watershed_algorithm ( g_Stereo.disparity );
cv::flip ( g_Stereo.disparity, flipped_image, -1 );
#else
#ifdef NORMALIZE_DISPARITY
#ifdef PROCESS_IMAGE
g_Stereo.process_image ( g_Stereo.disparity_Normalized );
#endif
cv::flip ( g_Stereo.disparity_Normalized, flipped_image, -1 );
#ifdef DISPARITY_COLOR
#ifdef PROCESS_IMAGE
g_Stereo.process_image ( g_Stereo.disparity_color );
#endif
cv::flip ( g_Stereo.disparity_color, flipped_image, -1 );
#endif
#else
#ifdef PROCESS_IMAGE
g_Stereo.process_image ( g_Stereo.disparity );
#endif
cv::flip ( g_Stereo.disparity, flipped_image, -1 );
#endif
```

```
#endif

//putText ( flipped_image, text(), Point ( 5, 25 ), FONT_HERSHEY_SIMPLEX, 1.0, Scalar::all
    ( 255 ) );

if ( FULL_SCREEN )
{
cv::Mat temp_image;
cv::Size image_size = cv::Size ( WINDOW_WIDTH, WINDOW_HEIGHT );

cv::resize ( flipped_image, temp_image, image_size, 1, 1, cv::INTER_LINEAR );

flipped_image = temp_image;
}

glGenTextures ( 1, &texture );

glBindTexture ( GL_TEXTURE_2D, texture );

glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST );
glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,     GL_CLAMP_TO_BORDER );
glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,     GL_CLAMP );

// Load the image
if ( FULL_SCREEN )
{
#ifdef DISPARITY_COLOR
glTexImage2D ( GL_TEXTURE_2D, 0, GL_RGB, WINDOW_WIDTH, WINDOW_HEIGHT,
0, GL_RGB, GL_UNSIGNED_BYTE, flipped_image.data );
#else
glTexImage2D ( GL_TEXTURE_2D, 0, GL_LUMINANCE, WINDOW_WIDTH, WINDOW_HEIGHT,
0, GL_LUMINANCE, GL_UNSIGNED_BYTE, flipped_image.data );
#endif
}
else
{
#ifdef DISPARITY_COLOR
glTexImage2D ( GL_TEXTURE_2D, 0, GL_RGB, IMAGE_W, IMAGE_H,
0, GL_RGB, GL_UNSIGNED_BYTE, flipped_image.data );
#else
glTexImage2D ( GL_TEXTURE_2D, 0, GL_LUMINANCE, IMAGE_W, IMAGE_H,
0, GL_LUMINANCE, GL_UNSIGNED_BYTE, flipped_image.data );
#endif
}

//https://open.gl/content/code/c3_multitexture.txt

glPushAttrib ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
glEnable ( GL_TEXTURE_2D );

glMatrixMode ( GL_MODELVIEW );
glLoadIdentity ( );

// Create a square on the XY
glBegin ( GL_QUADS );

glNormal3f ( 0.0, 0.0, 1.0 );
glTexCoord2d ( 0, 0 ); glVertex3f ( -1.0, -1.0,  1.0 );
```

```
glTexCoord2d ( 1, 0 ); glVertex3f (  1.0, -1.0,  1.0 );
glTexCoord2d ( 1, 1 ); glVertex3f (  1.0,  1.0,  1.0 );
glTexCoord2d ( 0, 1 ); glVertex3f ( -1.0,  1.0,  1.0 );

glEnd ( );

glDisable ( GL_TEXTURE_2D );
glPopAttrib ( );

}

// Save them
if ( g_Stereo.get_save ( ) )
{
// Write disparity
#ifdef NORMALIZE_DISPARITY
#ifdef DISPARITY_COLOR
imwrite ( g_Stereo.m_disparity_filename, g_Stereo.disparity_color );
#else
imwrite ( g_Stereo.m_disparity_filename, g_Stereo.disparity_Normalized );
#endif
#else
imwrite ( g_Stereo.m_disparity_filename, g_Stereo.disparity );
#endif

}

/*
    ----------------------------------------------------------------------------------------------------
    */

// End render
glFlush ( );
}



// Sets window position on the screen
//
// param:
//    int x: horizontal coordinate of the upper left corner of the window
//
//    int y: vertical coordinate of the upper left corner of the window
//
void Application::SetScreenPosition( int x, int y )
{
m_WindowProps.position.x = x;
m_WindowProps.position.y = y;
}



// Calculates the milliseconds elapsed between frames
//
// return:
//    long: milliseconds elapsed
//
long Application::GetMsElapsed ( )
{
```

```
static float msLastTime = System::GetMilliseconds ( );
float msCurrent, msElapsed;

msCurrent =  System::GetMilliseconds ( );
msElapsed =  msCurrent - msLastTime;
msLastTime = msCurrent;

return (long)msElapsed;
}

#ifdef __WIN_OPEN_GL__

// Creates a render context associated to a window
//
// param:
//    HWND hWnd: Handler of the window
//
//    PIXELFORMATDESCRIPTOR pixelFormatDescription: pixel format
//
// return:
//    int:
//           - -1  : error
//                        - others: fine
//
int Application::CreateRenderContext( )
{
unsigned int pixelFormat;
HDC hdc;

try
{
hdc = GetDC( m_WindowProps.hWnd );
if (!hdc)
throw System::GetSystemError();

// We define pixel format
PIXELFORMATDESCRIPTOR pfd;
memset( &pfd, 0, sizeof(PIXELFORMATDESCRIPTOR) );
pfd.nSize      = sizeof(PIXELFORMATDESCRIPTOR);
pfd.nVersion   = 1;
pfd.dwFlags    = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL | PFD_DOUBLEBUFFER;
pfd.iPixelType = PFD_TYPE_RGBA;
pfd.cColorBits = m_WindowProps.bpp;
pfd.cDepthBits = m_WindowProps.bpp;

// We check if windows allows our pixel format
pixelFormat = ChoosePixelFormat(hdc,&pfd);
if (!pixelFormat)
throw System::GetSystemError();

// If windows allowed that, we enable it
if(!SetPixelFormat(hdc,pixelFormat,&pfd))
throw System::GetSystemError();

// We create a render context device for OpenGL
m_hRC = wglCreateContext(hdc);
if (!m_hRC)
throw System::GetSystemError();
```

```
// We associate the render device to the context device
if( !wglMakeCurrent(hdc,m_hRC) )
throw System::GetSystemError();

return 0;
}

catch (DWORD errorCode)
{
System::ShowError(errorCode);
return -1;
}
}



// Releases the render context
void Application::ReleaseRenderContext()
{
HDC hdc;

try
{
if (m_hRC)
{
wglMakeCurrent(NULL,NULL);
wglDeleteContext(m_hRC);
m_hRC= NULL;
}

hdc = GetDC( m_WindowProps.hWnd );
if (hdc)
{
ReleaseDC(m_WindowProps.hWnd,hdc);
hdc= NULL;
}
}
catch (int errorCode)
{
System::ShowError(errorCode);
}
}

#endif



}   // namespace APPLICATION
```

A.11 Anexo 9

Encabezado del archivo graphicApplication.cpp

graphicsApplication.h:

```
/*
* Name        : graphicApplication.h
* Author      : Mario Luis ÁLvarez Pastor
* Version     : 0.0.0
* Copyright   : Free code
* Description : Header for the OpenGL proyect
*/



#ifndef GRAPHICAPPLICATION_H
#define GRAPHICAPPLICATION_H

/*
* Includes
*/

#include "common.h"



/*
* References
*/

namespace APPLICATION
{
// Application class
class Application
{
// Variable members:
protected:

float   m_MinDepthBuffer;   // Minimum deep buffer size
float   m_MaxDepthBuffer;   // Maximum deep buffer size

private:

#ifdef __WIN_OPEN_GL__
HGLRC   m_hRC;   // Render device. Memory zone where render will occur
#endif

WindowProps   m_WindowProps;        // Window properties
bool          m_KeysPressed[256];   // Keys pressed
bool          m_bEnd;               // Program should end?

// Function members:
public:

// Constructor
Application ( );

// Destructor
virtual ~Application ( );
```

```
// Function called on program initialization and before window creation
void OnInit ( WindowProps windowPropert );

// Function called on program ending
void OnEnd ( );

// Returns if application should end
bool IsEnded ( );

// Return if application is on full screen mode
bool IsFullScreen ( );

// Function called on program initialization and AFTER window creation
void OnCreateWindow ( WindowProps windowProps );

// Function called when changing the window size
void OnResize ( int width, int height );

// Function called when a key is pressed
void OnKeyDown ( int keyCode );

// Function called when a key is freed
void OnKeyUp ( int keyCode );

// Function called when nothing happens
void OnIdle ( );

// Function called when system is ready to render
void OnRender ( );

// Returns the title of the window
//
// return:
//    char *: Window title
//
inline char * GetMsgTitle ( )
{
return m_WindowProps.title;
}

// Sets window position on the screen
void SetScreenPosition ( int x, int y );

// Returns the position of the window in the screen
//
// return:
//    Point coordinate of the upper left corner of the window
//
inline Point GetScreenPosition ( )
{
return m_WindowProps.position;
}

private:

// Calculates the milliseconds elapsed between frames
long GetMsElapsed ( );
```

```
#ifdef __WIN_OPEN_GL__
// Creates a render context associated to a window
int CreateRenderContext ( );

// Releases the render context
void ReleaseRenderContext ( );

#endif   // __WIN_OPEN_GL__

};   // class Application

}   // namespace APPLICATION



#endif   // GRAPHICAPPLICATION_H
```

A.12 Anexo 10

Código del archivo stereo.cpp, encargado de cargar los datos en la GPU, mandar las órdenes para ejecutar las operaciones, y descargar los resultados de las mismas. También contiene los algoritmos ejecutados para hallar las imágenes de disparidad.

stereo.cpp:

```
/*
* Name        : stereo.cpp
* Author      : Mario Luis ÁLvarez Pastor
* Version     : 0.0.0
* Copyright   : Free code
* Description : OpenGL use for creating a window and stereo vision with CUDA
*/



/*
* Includes
*/

#include "../include/stereo.h"



/*
* Namespaces
*/

using namespace std;
using namespace cv;



/*
* Variables
*/

double camR[3][3] = { { 1123.252225105901, 0                , 339.4456657680444 },
{ 0                , 1123.971343979718, 226.5911398268449 },
{ 0                , 0                , 1                 } };

double camL[3][3] = { { 1115.488680426108, 0                , 316.4635984037795 },
{ 0                , 1114.75805126877 , 223.8783082757508 },
{ 0                , 0                , 1                 } };

double distR[8][1] = { -0.1242607410556098  , 3.115517586917172, 0.0002284733690985072,
-0.003829718740541797, 14.83251027386601, 0.1406434860041577,
0.3991843519067836  , 37.86442045489201 };

double distL[8][1] = { -0.1234396176236101   ,  0.8704171312319118, -0.001391213663301998,
9.345524996964323e-05, -125.8131946723568 ,  0.1374422112325338,
-0.7906605634785033   , -114.0148492230641 };

double rot[3][3] = { { 0.9990613255725089  , -0.002165551523724455, -0.04326405126544833
    },
{ 0.002384738940593295,  0.9999845797601961  ,  0.005015302783402157 },
{ 0.04325252322681838 , -0.005113768514713795,  0.9990510840823363   } };
```

```
double trans[3][1] = { 49.21567666544566,
0.745348709965439,
5.11424373573004 };



/*
* Functions
*/

namespace STEREO_VISION
{
// Constructor
Stereo::Stereo ( bool display, bool save ) : m_display ( display ), m_save ( save )
{
m_index = 0;

SADWindowSize = BLOCK_SIZE;
disparities_number = DISPARITIES_NUMBER;

#ifdef DATASET_0
m_dataset = 0;
min_disparity = MIN_DISPARITY;
max_disparity = MAX_DISPARITY;
#endif
#ifdef DATASET_1
m_dataset = 1;
min_disparity = MIN_DISPARITY;
max_disparity = MAX_DISPARITY;
#endif
#ifdef DATASET_2
m_dataset = 2;
min_disparity = MIN_DISPARITY;
max_disparity = MAX_DISPARITY;
#endif
#ifdef DATASET_3
m_dataset = 3;
min_disparity = MIN_DISPARITY;
max_disparity = MAX_DISPARITY;
#endif

// Build the classes and allocate memory
m_imageR_filename = new char[64];
m_imageL_filename = new char[64];
m_disparity_filename = new char[64];

imageR_color = Mat ( IMAGE_H, IMAGE_W, CV_8UC4 );
imageL_color = Mat ( IMAGE_H, IMAGE_W, CV_8UC4 );
imageR = Mat ( IMAGE_H, IMAGE_W, CV_8UC1 );
imageL = Mat ( IMAGE_H, IMAGE_W, CV_8UC1 );
disparity = Mat ( IMAGE_H, IMAGE_W, CV_32F );
#ifdef NORMALIZE_DISPARITY
disparity_Normalized = Mat ( IMAGE_H, IMAGE_W, CV_8U );
#endif
#ifdef DISPARITY_COLOR
disparity_color = Mat ( IMAGE_H, IMAGE_W, CV_8UC4 );
#endif
```

```
GPU_imageR = new cuda::GpuMat ( IMAGE_H, IMAGE_W, CV_8UC1 );
GPU_imageL = new cuda::GpuMat ( IMAGE_H, IMAGE_W, CV_8UC1 );
GPU_disparity = new cuda::GpuMat ( IMAGE_H, IMAGE_W, CV_32F );
#ifdef DISPARITY_COLOR
GPU_disparity_color = new cuda::GpuMat ( IMAGE_H, IMAGE_W, CV_8UC4 );
#endif

// Calibration matrices
camera_Matrix_1 = Mat ( 3, 3, CV_64F, camR );
camera_Matrix_2 = Mat ( 3, 3, CV_64F, camL );
distortion_Matrix_1 = Mat ( 8, 1, CV_64F, distR );
distortion_Matrix_2 = Mat ( 8, 1, CV_64F, distL );
rotation_system = Mat ( 3, 3, CV_64F, rot );
translation_system = Mat ( 3, 1, CV_64F, trans );

// Calibrated matrices
rectification_camera_1 = Mat ( 3, 3, CV_64F );
rectification_camera_2 = Mat ( 3, 3, CV_64F );
reprojection_camera_1 = Mat ( 3, 4, CV_64F );
reprojection_camera_2 = Mat ( 3, 4, CV_64F );
disparity_to_depth_mapping_matrix = Mat ( 4, 4, CV_64F );

rectangles_inside_rectified_image_1 = Rect();
rectangles_inside_rectified_image_2 = Rect();

// Matrices for remap the images
map11 = Mat ( 3, 3, CV_64F );
map12 = Mat ( 3, 3, CV_64F );
map21 = Mat ( 3, 3, CV_64F );
map22 = Mat ( 3, 3, CV_64F );

// Create the Stereo with defined disparities
Stereo_algorithm = cuda::createStereoBM ( disparities_number, SADWindowSize );
}



// Destructor
Stereo::~Stereo()
{
delete GPU_imageR;
delete GPU_imageL;
delete GPU_disparity;
#ifdef DISPARITY_COLOR
delete GPU_disparity_color;
#endif

delete m_imageR_filename;
delete m_imageL_filename;
delete m_disparity_filename;
}



void Stereo::stereo_vision ( void )
{
#ifdef GPU_MEMORY_INFO
workBegin ( );
#endif
```

```
// Upload to GPU the images
GPU_imageL->upload ( imageL );
GPU_imageR->upload ( imageR );

#ifdef GPU_MEMORY_INFO
workEnd ( );

cout << "GPU upload images: " << work_fps << endl;
#endif

#ifdef GPU_INFO
workBegin ( );
#endif

// Compute the algorithm
Stereo_algorithm->compute ( *GPU_imageL, *GPU_imageR, *GPU_disparity );

#ifdef GPU_INFO
workEnd ( );

cout << "GPU fps: " << work_fps << endl;
#endif

#ifdef GPU_MEMORY_INFO
workBegin ( );
#endif

#ifdef DISPARITY_COLOR
drawColorDisp ( *GPU_disparity, *GPU_disparity_color, disparities_number );

// Download the result from the GPU
GPU_disparity_color->download ( disparity_color );

cvtColor ( disparity_color, disparity_color, COLOR_BGR2RGB );
#endif

// Download the result from the GPU
GPU_disparity->download ( disparity );

#ifdef NORMALIZE_DISPARITY
normalize ( disparity, disparity_Normalized, 0, 255, NORM_MINMAX, CV_8U );
#endif

#ifdef GPU_MEMORY_INFO
workEnd ( );

cout << "GPU download disparity: " << work_fps << endl;
#endif
}



#ifdef PROCESS_IMAGE
Mat& Stereo::process_image ( Mat& image )
{
int i,j;
#ifndef NORMALIZE_DISPARITY
double* p;
```

```
#else
unsigned char *p;
#endif

//threshold ( image, image, min_disparity, max_disparity, cv::THRESH_BINARY | cv::
    THRESH_OTSU );

for ( i = 0; i < image.rows; ++i )
{
#ifndef NORMALIZE_DISPARITY
p = image.ptr < double > ( i );
#else
p = image.ptr < unsigned char > ( i );
#endif

for ( j = 0; j < image.cols; ++j )
{
if ( ( p[ j ] < min_disparity ) || ( p[j] > max_disparity ) )
p[ j ] = 0;
else
p[ j ] = 255;
}
}

return image;
}
#endif



#ifdef WATERSHED_ALGORITHM
Mat& Stereo::watershed_algorithm ( Mat& image )
{
//http://docs.opencv.org/master/d2/dbd/tutorial_distance_transform.html#gsc.tab=0

Mat processing_image_color, processing_image;

cvtColor ( image, processing_image_color, COLOR_GRAY2RGB );

/*              // Create a kernel that we will use for accuting/sharpening our image
Mat kernel = ( Mat_ < float > ( 3, 3 ) << 1,  1,  1,
1, -8,  1,
1,  1,  1); // an approximation of second derivative, a quite strong kernel
*/

///*
Mat kernel = ( Mat_ < float > ( 3, 3 ) <<  0,  1,  0,
1, -4,  1,
0,  1,  0); // an approximation of second derivative, a quite strong kernel
//*/
// do the laplacian filtering as it is
// well, we need to convert everything in something more deeper then CV_8U
// because the kernel has some negative values,
// and we can expect in general to have a Laplacian image with negative values
// BUT a 8bits unsigned int (the one we are working with) can contain values from 0 to 255
// so the possible negative number will be truncated
Mat imgLaplacian;
Mat sharp = processing_image_color; // copy source image to another temporary one
```

```
filter2D ( sharp, imgLaplacian, CV_32F, kernel );

processing_image_color.convertTo ( sharp, CV_32F );
Mat imgResult = sharp - imgLaplacian;

// convert back to 8bits gray scale
imgResult.convertTo ( imgResult, CV_8UC3 );
imgLaplacian.convertTo ( imgLaplacian, CV_8UC3 );

// Copy back
image = imgResult;

// Create binary image from source image
cvtColor ( image, processing_image, COLOR_RGB2GRAY );

// Aply threshold
threshold ( processing_image, processing_image, min_disparity, max_disparity, cv::
    THRESH_BINARY | cv::THRESH_OTSU );

// Perform the distance transform algorithm
Mat dist;
distanceTransform ( processing_image, dist, cv::DIST_L2, 5 );

// Normalize the distance image for range = {0.0, 1.0}
// so we can visualize and threshold it
normalize ( dist, dist, 0.0, 1.0, NORM_MINMAX );

// Threshold to obtain the peaks
// This will be the markers for the foreground objects
threshold ( dist, dist, min_disparity / max_disparity, 1.0, cv::THRESH_BINARY );

// Dilate a bit the dist image
Mat kernel1 = Mat::ones ( 3, 3, CV_8UC1 );
dilate ( dist, dist, kernel1 );

// Create the CV_8U version of the distance image
// It is needed for findContours()
Mat dist_8u;
dist.convertTo ( dist_8u, CV_8U );

// Find total markers
vector < vector < Point > > contours;
findContours ( dist_8u, contours, cv::RETR_EXTERNAL, cv::CHAIN_APPROX_SIMPLE );

// Create the marker image for the watershed algorithm
Mat markers = Mat::zeros ( dist.size(), CV_32SC1 );

// Draw the foreground markers
for ( size_t i = 0; i < contours.size ( ); i++ )
drawContours ( markers, contours, static_cast < int > ( i ), Scalar::all ( static_cast <
    int > ( i ) + 1 ), -1 );

// Perform the watershed algorithm
watershed ( image, markers );

//Mat mark = Mat::zeros ( markers.size ( ), CV_8UC1 );
//markers.convertTo ( mark, CV_8UC1 );
//bitwise_not ( mark, mark );
```

```
// Create the result image
Mat dst = Mat::zeros ( markers.size ( ), CV_8UC3 );

// Generate random colors
vector < Vec3b > colors;
for ( size_t i = 0; i < contours.size ( ); i++ )
{
/*
int b = theRNG ( ).uniform ( 0, 255 );
int g = theRNG ( ).uniform ( 0, 255 );
int r = theRNG ( ).uniform ( 0, 255 );
colors.push_back ( Vec3b ( (unsigned char)b, (unsigned char)g, (unsigned char)r ) );
*/

// Emprty the smallest contours, color the biggest
if ( contours[ i ].size ( ) > 480 )
colors.push_back ( Vec3b ( 255, 255, 255 ) );   // White
else
colors.push_back ( Vec3b ( 0, 0, 0 ) );   // Black

}

// Fill labeled objects with colors
for ( int i = 0; i < markers.rows; i++ )
{
for ( int j = 0; j < markers.cols; j++ )
{
int index = markers.at < int > ( i, j );

if ( index > 0 && index <= static_cast < int > ( contours.size ( ) ) )
dst.at < Vec3b > ( i, j ) = colors [ index - 1 ];
else
dst.at < Vec3b > ( i, j ) = Vec3b ( 0, 0, 0 );
}
}

cvtColor ( dst, image, COLOR_RGB2GRAY );

return image;
}
#endif



void Stereo::next_index ( void )
{
int u, d, c, m;

char index_string[5] = "0000";
string directory = "";

// Clear directory
directory.clear();

m = m_index / 1000;
c = m_index / 100 - m * 10;
d = m_index / 10  - m * 100  - c * 10;
u = m_index       - m * 1000 - c * 100 - d * 10;
```

```
// Fill index_srting
index_string[0] = 0x30 + m;
index_string[1] = 0x30 + c;
index_string[2] = 0x30 + d;
index_string[3] = 0x30 + u;

// Set left image to read
directory.append    ( "./captures" );
directory.push_back ( 0x30 + m_dataset );
directory.append    ( "/Left_Capture" );
directory.push_back ( index_string[0] );
directory.push_back ( index_string[1] );
directory.push_back ( index_string[2] );
directory.push_back ( index_string[3] );
directory.append    ( ".png" );

// Read the image
strcpy ( m_imageL_filename, directory.c_str ( ) );

// Clear directory
directory.clear();

// Set the image to read
directory.append    ( "./captures" );
directory.push_back ( 0x30 + m_dataset );
directory.append    ( "/Right_Capture" );
directory.push_back ( index_string[0] );
directory.push_back ( index_string[1] );
directory.push_back ( index_string[2] );
directory.push_back ( index_string[3] );
directory.append    ( ".png" );

// Read the image
strcpy ( m_imageR_filename, directory.c_str ( ) );

// Clear directory
directory.clear();

// Set the image to write
directory.append    ( "./disparities/Disparity" );
directory.push_back ( index_string[0] );
directory.push_back ( index_string[1] );
directory.push_back ( index_string[2] );
directory.push_back ( index_string[3] );
directory.append    ( ".png" );

// Write the image
strcpy ( m_disparity_filename, directory.c_str() );

// Go to next index
m_index++;

}



void Stereo::next_image ( void )
{
// Temporal images
```

```
Mat auxiliar_image_1, auxiliar_image_2;

// Read current images

// Read the image, if unable, then reset index and read again
imageR_color = imread ( m_imageR_filename );
imageL_color = imread ( m_imageL_filename );

if ( imageL_color.empty ( ) || imageR_color.empty ( ) )
{
#ifdef GENERAL_INFO
cout << "finished a round with " << m_index - 1 << " captures" << endl;
#endif

m_index = 0;

next_index ( );

imageR_color = imread ( m_imageR_filename );
imageL_color = imread ( m_imageL_filename );

}

// We need to undistort it
remap ( imageR_color, auxiliar_image_1, map11, map12, INTER_LINEAR );
remap ( imageL_color, auxiliar_image_2, map21, map22, INTER_LINEAR );

imageR_color = auxiliar_image_1;
imageL_color = auxiliar_image_2;

// We will use gray ones
cvtColor ( imageR_color, imageR, COLOR_RGB2GRAY );
cvtColor ( imageL_color, imageL, COLOR_RGB2GRAY );

#ifdef EQUALIZE
// Filter the image: (adaptive) histogram equalization
equalizeHist ( imageR, auxiliar_image_1 );
equalizeHist ( imageL, auxiliar_image_2 );

imageR = auxiliar_image_1;
imageL = auxiliar_image_2;
#endif

}



void Stereo::calibrate ( void )
{
// Check from here:
//https://github.com/Itseez/opencv/blob/master/samples/cpp/stereo_match.cpp

Size image_size = Size ( IMAGE_W, IMAGE_H );

cout << "Setting up calibration..." << endl;

// Get rectification and reprojection matrices
stereoRectify ( camera_Matrix_1, distortion_Matrix_1, camera_Matrix_2, distortion_Matrix_2,
image_size, rotation_system, translation_system,
```

```
rectification_camera_1, rectification_camera_2,
reprojection_camera_1, reprojection_camera_2,
disparity_to_depth_mapping_matrix,
CALIB_ZERO_DISPARITY, -1, image_size,   // Flags and scale (-1 for default scale)
&rectangles_inside_rectified_image_1, &rectangles_inside_rectified_image_2 );

cout << "Calibrating..." << endl;

// Set maps for undistort images
initUndistortRectifyMap ( camera_Matrix_1, distortion_Matrix_1,
rectification_camera_1, reprojection_camera_1,
image_size, CV_32FC1, map11, map12 );

initUndistortRectifyMap ( camera_Matrix_2, distortion_Matrix_2,
rectification_camera_2, reprojection_camera_2,
image_size, CV_32FC1, map21, map22 );

//#define DATASET_0
//---> Parameters:
//                  Block size: 11
//                  Number of disparities: 128
//                  Texture threshold: 1
//#define DATASET_1
//---> Parameters:
//                  Block size: 9
//                  Number of disparities: 144
//                  Texture threshold: 1
//#define DATASET_2
//---> Parameters:
//                  Block size: 15
//                  Number of disparities: 144
//                  Texture threshold: 0
//#define DATASET_3
//---> Parameters:
//                  Block size: 15
//                  Number of disparities: 128
//                  Texture threshold: 1

// Set algorithm parameters
#ifdef DATASET_0
disparities_number = 128;
SADWindowSize = 11;
Stereo_algorithm->setTextureThreshold  ( 1 );
#endif
#ifdef DATASET_1
disparities_number = 144;
SADWindowSize = 9;
Stereo_algorithm->setTextureThreshold  ( 1 );
#endif
#ifdef DATASET_2
disparities_number = 144;
SADWindowSize = 15;
Stereo_algorithm->setTextureThreshold  ( 0 );
#endif
#ifdef DATASET_3
disparities_number = 128;
SADWindowSize = 15;
Stereo_algorithm->setTextureThreshold  ( 1 );
#endif
```

```
disparities_number = disparities_number > 0 ? disparities_number : ( ( image_size.width / 8
     ) + 15) & -16;

Stereo_algorithm->setROI1              ( rectangles_inside_rectified_image_1 );
Stereo_algorithm->setROI2              ( rectangles_inside_rectified_image_2 );
Stereo_algorithm->setPreFilterCap      ( 32 );
Stereo_algorithm->setPreFilterSize     ( 5 );
Stereo_algorithm->setPreFilterType     ( 0 );
Stereo_algorithm->setBlockSize         ( SADWindowSize > 0 ? SADWindowSize : 9 );
Stereo_algorithm->setNumDisparities    ( disparities_number );
Stereo_algorithm->setMinDisparity      ( -42 );
Stereo_algorithm->setUniquenessRatio   ( 8 );
Stereo_algorithm->setSpeckleWindowSize ( 16 );
Stereo_algorithm->setSpeckleRange      ( 8 );
Stereo_algorithm->setDisp12MaxDiff     ( 1 );

/*
minDisparity ? Minimum possible disparity value.
numDisparities ? Maximum disparity minus minimum disparity. This parameter must be
    divisible by 16.
SADWindowSize ? Matched block size. It must be an odd number >=1 .
disp12MaxDiff ? Maximum allowed difference (in integer pixel units) in the left-right
    disparity check.
preFilterCap ? Truncation value for the prefiltered image pixels.
uniquenessRatio ? Margin in percentage by which the best (minimum) computed cost function
    value should ?win? the second best value to consider the found match correct. Normally,
     a value within the 5-15 range is good enough.
speckleWindowSize ? Maximum size of smooth disparity regions to consider their noise
    speckles and invalidate.
speckleRange ? Maximum disparity variation within each connected component.
*/
printParameters ( );

cout << "Calibrated!" << endl;

}



void Stereo::printParameters ( void )
{
cout << endl;

cout << "Prefiltercap: "          << Stereo_algorithm->getPreFilterCap ( )      << endl;
cout << "Prefilter size: "        << Stereo_algorithm->getPreFilterSize ( )     << endl;
cout << "Block size: "            << Stereo_algorithm->getBlockSize ( )         << endl;
cout << "Min disparity: "         << Stereo_algorithm->getMinDisparity ( )      << endl;
cout << "Number of disparities: " << disparities_number                         << endl;
cout << "Texture threshold: "     << Stereo_algorithm->getTextureThreshold ( )  << endl;
cout << "Uniqueness ratio: "      << Stereo_algorithm->getUniquenessRatio ( )   << endl;
cout << "Speckle window size: "   << Stereo_algorithm->getSpeckleWindowSize ( ) << endl;
cout << "Speckle Range: "         << Stereo_algorithm->getSpeckleRange ( )      << endl;
cout << "Disp 12 max diff: "      << Stereo_algorithm->getDisp12MaxDiff ( )     << endl;
cout << "Min process value: "     << min_disparity                              << endl;
cout << "Max process value: "     << max_disparity                              << endl;

cout << endl;
```

```
}



void Stereo::workBegin ( void )
{
work_begin = getTickCount ( );
}



void Stereo::workEnd ( void )
{
int64 d = getTickCount ( ) - work_begin;
double f = getTickFrequency ( );
work_fps = f / d;
}


}   // namespace STEREO_VISION
```

A.13 Anexo 11

Encabezado del archivo stereo.cpp.

stereo.h:

```
/*
* Name        : stereo.h
* Author      : Mario Luis ÁLvarez Pastor
* Version     : 0.0.1
* Copyright   : Free code
* Description : Header for the OpenGL proyect
*/



#ifndef STEREO_H
#define STEREO_H



/*
* Includes
*/

#include "common.h"



/*
* Namespaces
*/



/*
* References
*/

namespace STEREO_VISION
{
using namespace cv;

// Stereo class
class Stereo
{
public:

// Constructor
Stereo ( bool display, bool save );

// Destructor
~Stereo ( );

// Public members

char* m_disparity_filename;

// Stereo parameters
int SADWindowSize;
```

```
int disparities_number;

unsigned int min_disparity;
unsigned int max_disparity;

Ptr < cuda::StereoBM >   Stereo_algorithm;

Mat imageR_color;
Mat imageL_color;

Mat disparity;
#ifdef NORMALIZE_DISPARITY
Mat disparity_Normalized;
#endif
#ifdef DISPARITY_COLOR
Mat disparity_color;
#endif

// Public functions

#ifdef PROCESS_IMAGE
Mat& process_image ( Mat& image );
#endif
#ifdef WATERSHED_ALGORITHM
Mat& watershed_algorithm ( Mat& image );
#endif

void stereo_vision ( void );
void next_index ( void );
void next_image ( void );
void calibrate ( void );
void printParameters ( void );

private:

// Private members

char* m_imageR_filename;
char* m_imageL_filename;

Mat imageR;
Mat imageL;

cuda::GpuMat *GPU_imageL, *GPU_imageR;
cuda::GpuMat *GPU_disparity;
#ifdef DISPARITY_COLOR
cuda::GpuMat *GPU_disparity_color;
#endif

// Calibration stuff
Mat camera_Matrix_1, camera_Matrix_2;                 // M1, M2
Mat distortion_Matrix_1, distortion_Matrix_2;         // D1, D2
Mat rotation_system, translation_system;              // R, T
Mat rectification_camera_1, rectification_camera_2;   // R1, R2
Mat reprojection_camera_1, reprojection_camera_2;     // P1, P2
Mat disparity_to_depth_mapping_matrix;                // Q

Rect rectangles_inside_rectified_image_1;   // roi1 and roi2
Rect rectangles_inside_rectified_image_2;
```

```
// Matrices for remap the images
Mat map11, map12, map21, map22;

int m_index;
int m_dataset;

bool m_display;
bool m_save;

int64 work_begin;
double work_fps;

// Private functions
void workBegin ( void );
void workEnd ( void );

public:

// Setters and getters

inline void set_display ( bool display )
{
m_display = display;
}

inline bool get_display ( void )
{
return m_display;
}

inline void set_save ( bool save )
{
m_save = save;
}

inline bool get_save ( void )
{
return m_save;
}

inline int get_dataset ( void )
{
return m_dataset;
}

};   // Class Stereo

}   // namespace STEREO_VISION



#endif   // STEREO_H
```

A.14 Anexo 12

Archivo básico del programa que contiene la configuración y definiciones de los segmentos de código que se utilizarán en la ejecución del programa.

common.h:

```
/*
* Name        : common.h
* Author      : Mario Luis ÁLvarez Pastor
* Version     : 0.0.1
* Copyright   : Free code
* Description : Header for the OpenGL proyect
*/



#ifndef COMMON_H
#define COMMON_H



/*
* Includes
*/

#ifdef WIN32
#include <windows.h>
#endif

#include <GL/glut.h>   // Include GLUT, OpenGL and GLU

// OpenCV headers
//#include <opencv2/opencv.hpp>
#include <opencv2/core/utility.hpp>
#include <opencv2/cudastereo.hpp>
//#include <opencv2/cudaarithm.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/imgproc.hpp>

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include <iostream>
#include <fstream>
#include <float.h>



/*
* Defines
*/

// Info defines and modes
//#define SHOW_FPS
//#define GPU_INFO
//#define GPU_MEMORY_INFO
#define GENERAL_INFO
```

```
#define SAVE
#define DISPLAY
#define EQUALIZE

// Disparity defines

//#define DISPARITY_COLOR   // Color will overwritte normalized display
#define NORMALIZE_DISPARITY

#define PROCESS_IMAGE
// IMAGE MUST BE NORMALIZED
#ifdef PROCESS_IMAGE
#ifndef NORMALIZE_DISPARITY
#define NORMALIZE_DISPARITY
#endif
#endif

#define WATERSHED_ALGORITHM
// Undefine all the other processes with this algorithm
#ifdef WATERSHED_ALGORITHM
#ifdef NORMALIZE_DISPARITY
#undef NORMALIZE_DISPARITY
#endif
#ifdef PROCESS_IMAGE
#undef PROCESS_IMAGE
#endif
#ifdef DISPARITY_COLOR
#undef DISPARITY_COLOR
#endif
#endif



// Number of disparities and block size for stereo algorithm (by default)
//#define DISPARITIES_NUMBER   128
//#define BLOCK_SIZE           11   // MUST BE AN EVEN NUMBER!
// 16 and 5 works fast but really dark, 64 and 19 is default setting
// 128 and 3 seems to be a better aproach, faster than 64 and 19, but still slow
// 128 and 11 looks better but slower



// Uncomment only one:

//#define DATASET_0
//---> Parameters:
//                 Block size: 11
//                 Number of disparities: 128
//                 Texture threshold: 1
#ifdef DATASET_0
// Max and min disparity values to process the image
#define MIN_DISPARITY   180
#define MAX_DISPARITY   255

#define DISPARITIES_NUMBER   128
#define BLOCK_SIZE           11
#endif
```

```
//#define DATASET_1
//---> Parameters:
//                 Block size: 9
//                 Number of disparities: 144
//                 Texture threshold: 1
#ifdef DATASET_1
// Max and min disparity values to process the image
#define MIN_DISPARITY   180
#define MAX_DISPARITY   255

#define DISPARITIES_NUMBER   144
#define BLOCK_SIZE           9
#endif

#define DATASET_2
//---> Parameters:
//                 Block size: 15
//                 Number of disparities: 144
//                 Texture threshold: 0
#ifdef DATASET_2
// Max and min disparity values to process the image
#define MIN_DISPARITY   120
#define MAX_DISPARITY   255

#define DISPARITIES_NUMBER   144
#define BLOCK_SIZE           15
#endif

//#define DATASET_3
//---> Parameters:
//                 Block size: 15
//                 Number of disparities: 128
//                 Texture threshold: 1
#ifdef DATASET_3
// Max and min disparity values to process the image
#define MIN_DISPARITY   220
#define MAX_DISPARITY   255

#define DISPARITIES_NUMBER   128
#define BLOCK_SIZE           9
#endif



// Graphic application related
#define WINDOW_WIDTH    800
#define WINDOW_HEIGHT   600
#define BPP             32          // Bits per pixel
//#define FULL_SCREEN     true
#define FULL_SCREEN     false

#define WINDOW_TITLE    "Stereo Vision"



#define MIN_DEPTH_BUFFER    0.001f
#define MAX_DEPTH_BUFFER   1000.0f

#define FOV   45
```

```
#define KEY_ESC   0x1B

#define MS_SHOW_FPS   1000

// Stereo vision related

#define IMAGE_W     640
#define IMAGE_H     480



/*
* References
*/

namespace APPLICATION
{
// Struct for a 2D point
struct Point
{
int x, y;
};



// Struct for window properties
struct WindowProps
{
#ifdef WIN32
HWND    hWnd;
#endif

char    title[256];
Point   position;
int     width;
int     height;
int     bpp;
bool    bFullScreen;
};



// Color in real components, from 0.0f to 1.0f
union fColorRGBA
{
float vrgba[4];
struct
{
float r, g, b, a;
};
};

}   // namespace APPLICATION



#endif   // COMMON_H
```

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá