



Trabajo de Fin de Grado

# Visión de Sistemas Empotrados

*Vision of Embedded Systems*

Jose Ricardo Pérez Castillo

La Laguna, 1 de septiembre de 2017

D. **Jonay Tomás Toledo Carrillo**, con N.I.F. 78.698.554-Y profesor contratado Doctor del Departamento de Ing. Informática y de Sistemas de la Universidad de La Laguna, como tutor

**C E R T I F I C A (N)**

Que la presente memoria titulada:

*"Visión de Sistemas Empotrados"*

ha sido realizada bajo su dirección por D. **Jose Ricardo Pérez Castillo**, con N.I.F. 54.058.489-W.

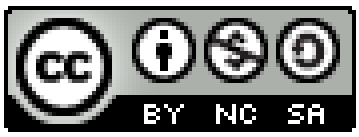
Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 1 de septiembre de 2017.

# Agradecimientos

Especialmente me gustaría agradecer el apoyo y paciencia de mi tutor Jonay Tomás Toledo Carrillo.

También me gustaría agradecer a mi familia y amigos por el apoyo que me han ofrecido.

# Licencia



## Resumen

El objetivo de este trabajo es conseguir obtener a partir de las imágenes captadas por una cámara estéreo información de los obstáculos del entorno.

Para ello se ha usado ROS, un framework para el desarrollo de software para robots que provee la funcionalidad de un sistema operativo, y OpenCV, una biblioteca libre de visión artificial originalmente desarrollada por Intel.

Explicaremos los diversos pasos necesarios para conseguir dicho objetivo y se utilizarán los algoritmos StereoBM y StereoSGBM para conseguir nuestro objetivo. Finalmente las pruebas serán realizadas con la cámara estéreo PlayStation Camera y una placa Up board, donde ejecutar los algoritmos.



***Palabras clave:*** *ROS, OpenCv, PlayStation Camera, Up Board.*

## Abstract

The objective of this work is to obtain from the images captured by a stereo camera information of the obstacles of the environment.

For this, ROS has been used, a software development framework for robots that provides the functionality of an operating system, and OpenCV, a free artificial vision library originally developed by Intel.

We will explain the various steps necessary to achieve this goal and will use the StereoBM and StereoSGBM algorithms to achieve our goal. Finally the tests will be carried out with the stereo camera PlayStation Camera and a board Up board, where to execute the algorithms.



***Keywords:*** *ROS, OpenCv, PlayStation Camera, Up Board.*

# Índice General

# Índice de figuras

# Índice de tablas

# Capítulo 1. Antecedentes y Objetivos

En este capítulo se hablara sobre el estado en el que se encuentra el tema sobre el que trata el Trabajo de Fin de Grado, así mismo, se comentaran los objetivos de éste.

## 1.1 Estado del Arte

Estamos viviendo la época dorada de la robótica, cada vez es más frecuente encontrarnos robots en nuestro día a día, desde aquellos que facilitan las labores del hogar, hasta vehículos capaces de conducir sin intervención humana.

Dentro de este vasto mundo de la robótica, la visión artificial va cogiendo cada vez más fuerza con el objetivo de conseguir que los robots sean cada vez mas autónomos. Para ello se han creado gran variedad de tecnologías que ayudan a este objetivo, desde sistemas laser, hasta sensores de ultrasonido.

Grandes compañías como Google, Sony o Microsoft, han invertido esta tecnología. Google por ejemplo dispone de un coche completamente autónomo el Google Self-Driving Car, el cual desde 2011 es legal que circule por algunos estados de Estados Unidos. Así mismo Microsoft y Sony han desarrollado dispositivos para sus respectivas consolas, con el propósito de controlarlas, mediante el uso de gestos y movimientos, sin tener que utilizar sus mandos, lanzando de esta manera Kinect y la PlayStation Camera respectivamente.

## 1.2 Antecedentes

Este proyecto nace de la necesidad de conseguir utilizando tecnologías de bajo coste un resultado bastante cercano al conseguido con otras tecnologías más costosas.

En este caso nuestro objetivo es el de intenta crear un mapa de costes, para ello tenemos acceso a tecnologías como Kinect, está nos proporcionaría un gran resultado en nuestro proyecto, pero tiene el gran problema de tener dificultades en exteriores ya que está no detecta correctamente los objetos si hay demasiada claridad, lo que limitaría bastante su efectividad.

Por ello, para este proyecto se ha pensado utilizar una cámara estéreo, en concreto una PlayStation Camera, está a pesar de no tener la misma calidad que una kinect, si nos dejaría trabajar en exteriores sin limitaciones por la luminosidad.

## 1.3 Objetivos

En este apartado trataremos brevemente, tanto los objetivos generales, como los específicos que tiene este proyecto.

### 1.3.1 Objetivo General

El objetivo principal de este Trabajo de Fin de Grado es poder crear un mapa de costes a partir de los datos recogidos mediante un sistema estereoscópico de dos cámaras, en nuestro caso, una PlayStation Camera. Para conseguir nuestros objetivos, tendremos que hacer uso de técnicas de tratado de imágenes, dentro de éstas podemos encontrar el cálculo de disparidad, gracias al cual, podremos obtener distancias al comparar las imágenes estéreo, o el uso de imágenes de profundidad o depth image, que nos serán necesarias, para poder conseguir nuestra finalidad.

### 1.3.2 Objetivo Específico

Los objetivos Específicos que forman este proyecto son:

- Prototipo Instalado y funcionando en la UpBoard.
- Integración de un mapa de costes.
- Detección dinámica de obstáculos.

# Capítulo 2. Conceptos

En este capítulo se abordaran todos aquellos conceptos teóricos que han surgido a lo largo del proyecto.

## 2.1 Sistema Empotrado

Hablamos de un sistema empotrado o embebido, cuando nos referimos a un sistema de computación diseñado para realizar una o algunas pocas funciones dedicadas, frecuentemente en un sistema de computación en tiempo real. Al contrario de lo que ocurre con los ordenadores de propósito general que están diseñados para cubrir un amplio rango de necesidades, los sistemas empotrados se diseñan para cubrir necesidades específicas. En un sistema embebido la mayoría de los componentes se encuentran incluidos en la placa base (tarjeta de vídeo, audio, módem, etc.) y muchas veces los dispositivos resultantes no tienen el aspecto de lo que se suele asociar a una computadora.

## 2.2 Visión Estéreo

La visión estereoscópica o visión estéreo, es la técnica capaz de extraer información tridimensional (profundidad) a partir de la posición relativa de un objeto en imágenes bidimensionales al ser observado desde distintos ángulos por dos o más cámaras a una cierta distancia.

### 2.2.1 Adquisición

El procedimiento a seguir para la adquisición del entorno es capturar dos imágenes de una misma escena, desde dos cámaras separadas ligeramente. De esta forma, las imágenes obtenidas, también tendrán un pequeño desplazamiento entre sí.

De manera más formal, se obtiene que para cada imagen capturada por las cámaras, un objeto está en puntos diferentes del plano. Esta triangulación entre el punto P y Q y el origen de referencia, provocan una sensación de profundidad. Mediante el sistema tradicional de una sola cámara, este punto estaría en las mismas coordenadas.

### 2.2.2 Geometría de las cámaras

En función de la posición relativa de las cámaras entre sí, se pueden apreciar dos métodos principales:

- **Visión paralela**: las cámaras están paralelas entre sí y están separadas por una línea horizontal (baseline). El objetivo que visualiza cada cámara es perpendicular respecto a la baseline, mientras que las líneas de correspondencia que unen los puntos de una imagen respecto a la otra son horizontales.

Figuras 1. Visión Paralela

- **Visión cruzada**: las cámaras no están paralelas entre sí, tienen una inclinación de tal forma que el objetivo de cada cámara apunta hacia

el lado contrario de una imagen. Por lo que los ejes ópticos se cruzan entre sí. Las líneas de correspondencia, también sufren una inclinación.

### 2.2.3 Disparidad

La disparidad de dos imágenes establece la correspondencia entre los pixeles o características que existen entre ambas en el eje x para obtener la profundidad de la escena. Con esto se consigue estimar la profundidad de cada uno de los puntos de la escena.

El objetivo final es poder construir una imagen que representa la disparidad entre las dos cámaras. Habitualmente el mapa de disparidad se representa como una imagen monocroma donde los objetos con mayor disparidad (más cercanos) son representados con un tono más claro y los objetos con menor disparidad (más lejanos) son representados con un tono más oscuro. Aunque no es infrecuente encontrar una imagen térmica en vez de monocroma.

El algoritmo para calcular la disparidad consiste, de forma general, en buscar cada punto singular de la imagen izquierda en la imagen derecha, estando dichas imágenes rectificadas en todo caso, para observar en qué nuevo píxel se encuentra. A partir de esta información, es posible calcular la profundidad buscando para cada punto de las imágenes la pareja de puntos correspondiente que representa la proyección del mismo punto en el espacio.

El mapa de disparidad es uno de los elementos básicos para la reconstrucción tridimensional de los objetos de una escena a partir de varias capturas.

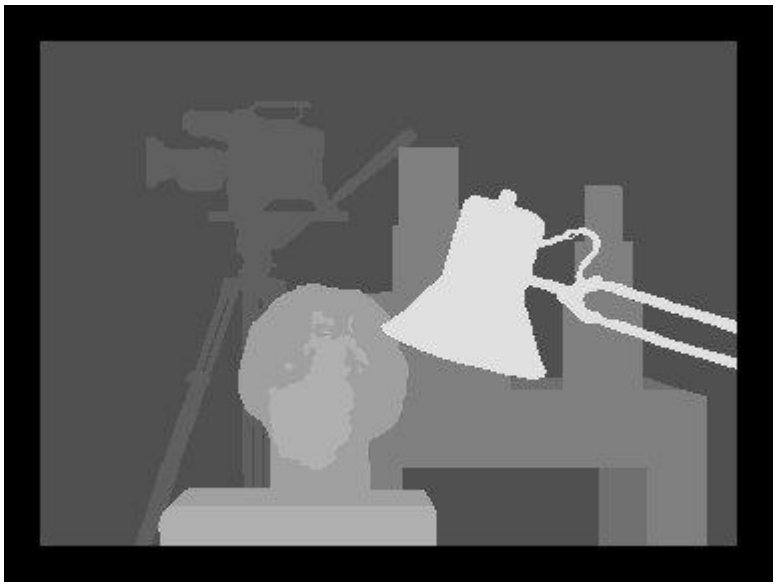


Figura 2. Imagen de disparidad

# Capítulo 3. Herramientas y Recursos

En este capítulo se abordaran todos aquellos recursos necesarios, tanto hardware como software, para el desarrollo de este proyecto.

## 3.1 ROS

Este proyecto se creara teniendo como base ROS, cuando hablamos de ROS estamos haciendo referencia a un framework para el desarrollo de software para robots que provee la funcionalidad de un sistema operativo. Este framework provee servicios estándar del sistema operativo como abstracción de hardware, control de dispositivos de bajo nivel, implementación de las funciones utilizadas normalmente. Es de código libre y se basa en un modelo de paso de suscripción y envio de mensajes. Esto lo hace realmente flexible y adaptable para las necesidades del usuario.

Las partes principales son: el núcleo de framework y ros-pkg, que es un conjunto de paquetes aportados por la contribución de los usuarios que

implementan funcionalidades como localización y mapeo, planificación, simulación, etc.

Además, con ROS se puede organizar el software en una arquitectura de grafos, dividida en nodos, en la que estos son porciones ejecutables de código que se comunican entre sí a través de 'topics'. Todos estos nodos son capaces de encontrarse y comunicarse gracias al nodo ROS master que permite la intercomunicarse entre ellos.

## 3.2 OpenCV

Se trata de una biblioteca libre de visión artificial originalmente desarrollada por Intel. Desde que apareció su primera versión alfa en el mes de enero de 1999, se ha utilizado en infinidad de aplicaciones. Desde sistemas de seguridad con detección de movimiento, hasta aplicaciones de control de procesos donde se requiere reconocimiento de objetos. Esto se debe a que su publicación se da bajo licencia BSD, que permite que sea usada libremente para propósitos comerciales y de investigación con las condiciones en ella expresadas.

OpenCV es multiplataforma, existiendo versiones para GNU/Linux, Mac OS X y Windows. Contiene más de 500 funciones que abarcan una gran gama de áreas en el proceso de visión, como reconocimiento de objetos (reconocimiento facial), calibración de cámaras, visión en estéreo y visión robótica.

El proyecto pretende proporcionar un entorno de desarrollo fácil de utilizar y altamente eficiente. Esto se ha conseguido llevando a cabo su programación en código C y C++ optimizados, aprovechando además las capacidades que proveen los procesadores multinúcleo. OpenCV puede además utilizar el sistema de primitivas de rendimiento integradas de Intel, un conjunto de rutinas de bajo nivel específicas para procesadores Intel (IPP).

### 3.2.1 Algoritmos

En nuestro proyecto utilizaremos varias librerías de OpenCV, pudiendo destacar calib3d, esta nos da acceso a dos clases que nos serán necesarias para este proyecto:

- **StereoBM:** Esta clase se encarga de calcular, en base a las imágenes captadas mediante la cámara estéreo, una imagen de disparidad. Esto es posible mediante el uso del algoritmo block matching sobre el que está basado esta clase. Éste funciona dividiendo las imágenes de la secuencia de video en bloques rectangulares denominados macrobloques. Éstos son utilizados posteriormente para detectar el movimiento en las imágenes cotejándolos con los macrobloques del fotograma de destino, descubriendo así si se produjo un desplazamiento(1).
- **StereoSGBM:** Esta clase al igual que el StereoBM se encarga de calcular la imagen de disparidad, con la diferencia que este, se basa en un algoritmo modificado del block matching, en concreto el algoritmo de H. Hirschmuller, lo que nos proporciona una mejor calidad, aunque conlleva un mayor consumo de recursos(2).

## 3.3 PlayStation Camera

PlayStation Camera es una cámara utilizada por PlayStation 4 como accesorio. Fue lanzada al mercado en 2013 con el lanzamiento de la consola, aunque vendiéndose como un accesorio a parte a un precio recomendado de 59.99€.

Esta cámara está pensada para poder controlar la consola sin necesidad de utilizar el mando tradicional. Los usuarios de PS4 pueden iniciar sesión haciendo uso de las características de reconocimiento facial. También permite utilizar los movimientos del propio cuerpo para moverse por los menús del sistema y utilizar la voz para realizar acciones gracias al micrófono

incorporado. PlayStation camera también es compatible con otros accesorios de PlayStation4 como PlayStation Move (sistema de control por movimiento) para navegación más precisa.

Aunque el objetivo principal de PlayStation camera es su integración en los videojuegos, para conseguir mayor inmersión: control de movimiento, comandos de voz y tecnologías de realidad aumentada.

### 3.3.1 Historia

Los orígenes de PlayStation Camera vienen desde 1999 cuando Sony empezó a investigar sobre la visión artificial y el uso de reconocimiento de gestos mediante una cámara para incorporar esta tecnología a los videojuegos. Finalmente en 2003 se lanzaría EyeToy, una cámara que tuvo notable éxito para PlayStation 2. En 2007, tras la salida de PlayStation 3, se lanzó al mercado una actualización de esta cámara con el nombre de PlayStation Eye, la cual permitía capturar imágenes a una mayor resolución y una tasa de fotogramas por segundo superior.

Mientras que EyeToy y PlayStation Eye compartían el mismo concepto, no fue hasta el lanzamiento de Kinect por parte de Microsoft en 2010, cuando se pudo ver una evolución real en este tipo de accesorios de entretenimiento. Kinect se basó en una cámara RGB-D que contaba con sensores de profundidad, múltiples micrófonos y un procesador independiente de la consola. Kinect permitía captura de movimiento en 3D, reconocimiento facial y reconocimiento de voz.

Kinect fue un éxito de ventas y obtuvo muchos elogios por parte de la crítica, pero no en el plano de los videojuegos, sino de la investigación. Microsoft decidió lanar un SDK oficial para el desarrollo en Windows en base al interés que existía, para proyectos de todo tipo.

PlayStation camera ha tomado nota de Kinect para ofrecer un producto con las mismas ventajas de Kinect, aunque utilizando tecnología

estereoscópica y características más modestas para ser una alternativa económica a Kinect.

### 3.3.2 Especificaciones técnicas

PlayStation Camera cuenta con un chip OV580 encargado de sincronizar las dos cámaras. Cada cámara cuenta con un sensor de imagen OV9713, mientras que el chip que controla el sonido es el AK5703. La configuración inicial de la cámara se almacena en una memoria EEPROM 4g51A. Así mismo esta cámara cuenta con 3 configuraciones de resolución distintos, el primero de ellos nos dejará realizar grabaciones a 1280x800 pixels a una frecuencia de 60 hz, también podremos grabar 640x400 que nos proporciona una tasa de refresco de 120 hz y finalmente, la menor resolución permitida seria a 320x192 y una frecuencia de 240 hz. También podemos destacar de esta cámara, su distancia focal que es de 30 cm, proporcionándonos también un campo de visión de 85º.

Figura 3. PlayStation Camera

# Capítulo 4. Desarrollo

En este capítulo se explicará como configurar el entorno de trabajo de ROS y se hablará del trabajo realizado en este proyecto.

## 4.1 Preparación del entorno

Como se pudo observar en el capítulo 3 Herramientas y Recursos, el desarrollo de este trabajo se realiza en base a ROS, en este caso la aplicación final será ejecutada en la versión Indigo, aunque el portátil donde se ha desarrollado dispone de la versión Kinetic. Es necesario instalar y configurar un directorio de trabajo para ROS donde estará alojado el paquete con el código fuente del proyecto.

El sistema operativo utilizado en la plataforma ha sido Ubuntu 16.04 LTS (64-bits), una distribución GNU/Linux muy sencilla de utilizar para la que se distribuyen los paquetes que confirman ROS, mientras que la utilizada en la upboard será Ubuntu 14.04 LTS (64-bits). A continuación explicaremos paso a paso, como instalar ROS, así como configurar el directorio de trabajo. Como pequeño apunte, indicar que este tutorial podrá ser usado para la instalación de cualquier versión de ROS, únicamente se tendría que cambiar el nombre de la distribución en los diversos comandos.

Así mismo, también se ha instalado OpenCV, para poder usar las diversas librerías que esta nos ofrece.

### 4.1.1 Instalación y configuración de ROS

Para empezar, hay que indicar que ROS no se encuentra dentro de los paquetes oficiales de Ubuntu, por lo que es necesario añadirlo desde un repositorio externo, el oficial de ROS. Para ello se introduce en una terminal lo siguiente:

- *$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'*

Se añaden las claves del repositorio externo:

- *$ sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net --recv-key 0xB01FA116*

Con esto solamente es necesario actualizar la base de datos de los paquetes disponibles en los diferentes repositorios:

- *$ sudo apt-get update*

El gestor de paquetes ya puede encontrar los paquetes de ROS entre sus repositorios. Los paquetes se pueden instalar de forma individual, pero es recomendable realizar una instalación completa para no echar en falta ningún paquete en el futuro. Para ello en la terminal instalamos el siguiente paquete, donde sustituiremos la palabra versión, por la que hayamos decidido usar:

- *$ sudo apt-get install ros-"version"-desktop-full*

El paquete 'ros-"versión"-desktop-full' viene con muchas de las herramientas que se han usado en el proyecto: Rviz, Rqt, etc. Aunque no incluye la herramienta 'rosinstall' la cual permite instalar paquetes de ROS independientes del sistema:

- *$ sudo apt-get install python-rosinstall*

Con esto, ya tendríamos instalado ROS, y procederíamos a su configuración. Antes de poder utilizar ROS es necesario configurar '*rosdep*'. Esta herramienta permite instalar fácilmente dependencias del sistema que

surgen cuando se requiere compilar el código fuente de un paquete ROS. Es necesario introducir en una terminal lo siguiente:

- *$ sudo rosdep init*
- *$ rosdep update*

Por último, también es conveniente cargar las variables del entorno de ROS. Si se utiliza *'Bash'* como shell basta con introducir lo siguiente:

- *$ echo "source /opt/ros/"versión"/setup.bash" >> ~/.bashrc*
- *$ source ~/.bashrc*

En caso de utilizar otra shell por defecto, en /opt/ros/"versión" también se cuenta con el script equivalente, cuyo contenido es necesario volcar en el archivo de configuración de la shell. Con esto último, todas las variables de entorno se cargan automáticamente al abrir una nueva shell.

Una vez ya configurado ROS, es hora de crear un workspace. Un workspace es un directorio donde se puede trabajar con paquetes catkin(los paquetes oficiales utilizados por ROS), pudiendo modificar, compilar o añadir nuevos paquetes.

Al igual que en el entorno real, se puede crear un directorio en el directorio personal del usuario que va a contener el workspace:

- *$ mkdir -p ~/catkin_ws/src*
- *$ cd ~/catkin_ws/src*
- *$ catkin_init_workspace*

Con este último comando, se inicia el workspace, creando un archivo de instrucciones de Cmake. Por último, es necesario compilar estas instrucciones:

- *$ cd ~/catkin_ws*
- *$ catkin_make*

Tras finalizar la compilación, el directorio del worksapace estará estructurado como un workspace valido. Al igual que es necesario añadir a los archivos de configuración de la shell las variables de ROS, también es recomendable añadir los del workspace:

- *$ echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc*
- *$ source ~/.bashrc*

Una vez creado el workspace, es necesario que creemos un nuevo paquete dentro del mismo, sobre el cual trabajaremos. Para ello es necesario escribir en una terminal lo siguiente:

- *$ cd ~/catkin_ws/src*
- *$ catkin_create_pkg tfg*

En este caso se ha creado un paquete llamado "tfg", pero este podría ser sustituido por cualquier nombre que se desee. Con este último comando se creara una carpeta nueva con los archivos de configuración necesarios para compilar el paquete. Si se modifica el código fuente, en un paquete del workspace, es necesario compilarlo de la siguiente forma:

- *$ cd ~/catkin_ws*
- *$ catkin_make*

### 4.1.2 Instalación de OpenCV

A diferencia de ROS, OpenCV si se encuentra dentro de los repositorios que vienen por defecto en Ubuntu, lo cual nos agilizara las cosas, teniendo que usar únicamente los siguientes comandos desde la terminal.

Como al instalar ROS ya tuvimos que actualizar los repositorios, nos podremos saltar ese paso y pasar directamente a la instalación del mismo:

- *$ sudo apt-get install libopencv-dv*

Además si tenemos pensado usar codigo en Python, ya que ROS permite utilizar tanto C++ como Python, instalaremos el siguiente paquete:

- *$ sudo apt-get install python-opencv*

Para finalizar, nos bajaremos toda la documentación, así como algunos ejemplos de aplicaciones ya hechas:

- *$ sudo apt-get install opencv-doc*

## 4.2 Desarrollando el código.

Una vez hemos terminado de instalar y configurar ROS y OpenCV, es hora de empezar a desarrollar el código necesario para nuestro proyecto.

### 4.2.1 Calculando la imagen de disparidad.

El primer paso consiste es transformar las imágenes estéreo obtenidas mediante la PlayStation Camera en una imagen de disparidad.

Para ello tendremos que utilizar una de las librerías de OpenCV, en concreto utilizaremos la librería "calib3d", haciendo uso de las clases StereoBM y StereoSGBM, como se ha comentado en apartados anteriores.

### 4.2.2 StereoBM

Nuestro primer código lo hemos realizado utilizando la clase StereoBM. Para la correcta declaración de la clase, StereoBM nos pide una serie de datos obligatorios, estos se tratan de:

- **preset:** Este valor especifica el conjunto de parámetros del algoritmo, pudiendo ser cualquiera de los siguientes:
  - BASIC_PRESET - Parámetros adecuados para cámaras generales.
  - FISH_EYE_PRESET - Parámetros adecuados para cámaras con un ángulo amplio.
  - NARROW_PRESET - Parámetros adecuados para cámaras de ángulo estrecho.
- **ndisparities:** Este valor representa el rango de búsqueda máximo de disparidad, es decir que se buscara la mejor disparidad entre 0 y el valor máximo de disparidad.
- **SADWindowSize:** Este valor indica el tamaño de los macrobloques sobre los que trabajara el algoritmo, teniendo en cuenta, que a mayor sea el número, más impreciso será la disparidad, además este valor tiene que ser impar.

Una vez declarados estos dos atributos nos quedaría declarar la clase, para ello podremos utilizar una de las siguientes opciones, las cuales varían dependiendo de la versión de openCV instalada, 2.4 o 3.

Para OpenCV 2.4

- *cv::StereoBM sbm = cv::StereoBM(CV_STEREO_BM_BASIC, ndisparity, SADWindowSize);*

Para OpenCV 3

- *cv::Ptr<cv::StereoBM> sbm = cv::StereoBM::create(ndisparity, SADWindowSize);*

Finalmente, para calcular la disparidad nos haría falta la siguiente función, que al igual que con la declaración de la clase, varía respecto a la versión utilizada.

Para OpenCv 2.4

- *sbm.operator()(left_image, right_image, imgDisparity16S);*

Para OpenCv 3

- *sbm->compute(left_image, right_image, imgDisparity16S);*

Los valores en ambos casos se tratan de:

- **left_image:** se correspondería con la imagen izquierda captada por la cámara.
- **right_image:** se correspondería con la imagen derecha captada por la cámara.
- **imgDisparity16S:** se trata de la variable donde se guardará la imagen de disparidad creada por el algoritmo.

Una vez creado esto, el resultado sería el siguiente.

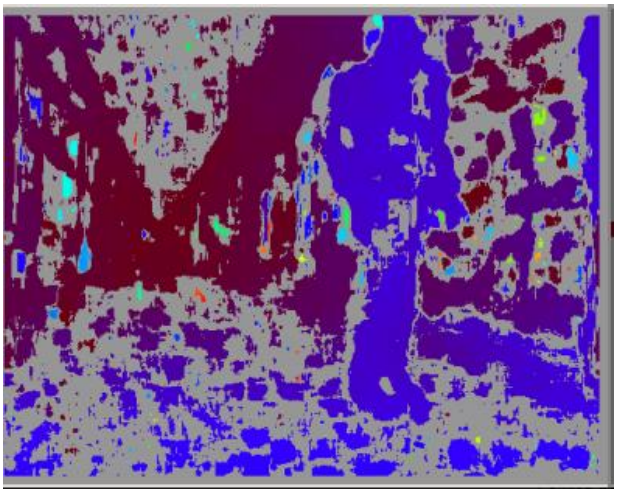


Figura 4. Resultado StereoBM

### 4.2.3 StereoSGBM

Una vez se ha logrado crear la imagen de disparidad con la clase StereoBM, es hora de realizar lo mismo con la clase StereoSGBM, para ello al igual que pasaba en el StereoBM, nos son necesarios una serie de variables, para su correcta declaración:

- **minDisparity:** Este valor indica el rango mínimo de búsqueda de disparidad, al igual que en StereoBM por defecto es cero, pero este algoritmo te da la posibilidad de cambiarlo.
- **nDisparities**: Este valor indica el resultado al restar la disparidad máxima menos la mínima. Siempre tiene que ser mayor de 0, y su valor deberá ser múltiplo de 16.
- **SADWindowSize**: Indica el tamaño de los macrobloques, su valor debe impar mayor de uno, dentro del rango de tres a once.
- **P1**: Primer parámetro que controla la suavidad de la disparidad. Indica la penalización en el cambio de disparidad por más menos uno, entre los pixel vecinos.
- **P2**: Segundo parámetro que controla la suavidad de la disparidad. A más grande sean estos parámetros, mayor será la suavidad de la disparidad. Indica la penalización en el cambio de disparidad por más de uno entre los pixel vecinos, este valor siempre deberá ser mayor que P1.
- **disp12MaxDiff**: Este valor indica la diferencia máxima permitida en la comprobación de disparidad, si se establece un valor negativo, se desactivara esta comprobación.
- **preFilterCap**: Este parámetro indica el valor de truncamiento para los pixeles de imagen prefiltrados.

- **uniquenessRatio**: Se trata del margen de porcentaje por el cual el mejor valor de la función calculada debería ganar el segundo mejor valor para considerar el resultado correcto.
- **speckleWindowSize**: Tamaño máximo del suavizado de las regiones de disparidad para ser consideradas ruido y por tanto invalidarlas.
- **speckleRange**: Variación de disparidad máxima dentro de cada componente conectado.
- **fullDP**: Booleano que si se pone a true ejecutara el algoritmo de programación dinámica de dos pasos de escala completa.

Una vez creadas todas las variables, es hora de declarar la clase, para ello, tal como pasaba con StereoBM, dependiendo de la versión de OpenCV utilizada, utilizaremos una declaración diferente.

Para OpenCV 2.4

➢ *cv::StereoSGBM ssgbm = cv::StereoSGBM(minDisparity, ndisparity, blockSize, P1, P2, disp12MaxDiff, preFilterCap, uniquenessRatio, speckleWindowSize, speckleRange, fullDP);*

Para OpenCV 3

➢ cv::Ptr<cv::StereoSGBM> ssgbm = cv::StereoSGBM::create(minDisparity, ndisparity, blockSize, P1, P2, disp12MaxDiff, preFilterCap, uniquenessRatio, speckleWindowSize, speckleRange, fullDP);

Para finalizar solo necesitaremos ejecutar la función, para que se calcule la imagen de disparidad

Para OpenCv 2.4

➢ *ssgbm.operator()(left_image, right_image, imgDisparity16S);*

Para OpenCv 3

- *ssgbm->compute(left_image, right_image, imgDisparity16S);*

Los parámetros para ejecutar la función de disparidad son los siguientes:

- **left_image:** se correspondería con la imagen izquierda captada por la cámara.
- **right_image:** se correspondería con la imagen derecha captada por la cámara.
- **imgDisparity16S:** se trata de la variable donde se guardara la imagen de disparidad creada por el algoritmo.

Una vez creada esta función, solo nos faltaría ver los resultados.

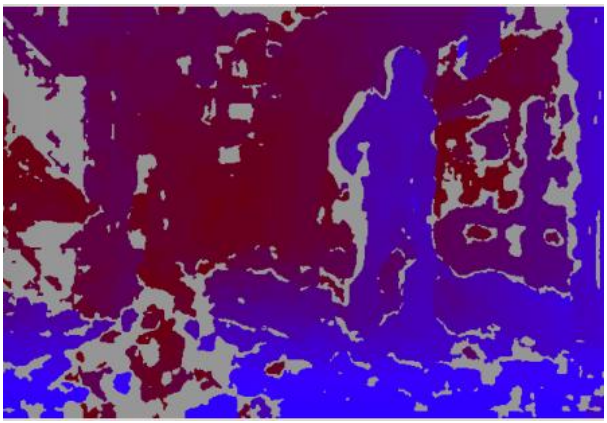


Figura 5. Resultado StereoSGBM

### 4.2.4 Comparativa StereoBM y StereoSGBM

Como se puede apreciar en las figuras 4.1 y 5.1, StereoSGBM nos proporciona más calidad en el resultado final, lo que indica que nos consume más recursos, en la siguiente tabla, se mostrara la cantidad de CPU consumida con cada uno de los algoritmos.

| | StereoBM | StereoSGBM |
|---|---|---|
| **Cantidad de CPU utilizada(media)** | 9% | 12% |
| **Cantidad de Memoria Utilizada(media)** | 10.3MiB | 136.7MiB |

Tabla 1. Consumo de CPU con los algoritmos

### 4.2.5 Calcular la imagen de profundidad(depthImage)

Una vez hemos conseguido la imagen de disparidad, es hora de avanzar más en el proyecto. Al comprobar los valores que necesita costmap para funcionar, nos damos cuenta que este necesita conseguir la información desde un PointCloud o de un LaserScans. Inicialmente tomamos la opción de crear un PointCloud a partir de la imagen de disparidad que ya disponíamos, pero tras su creación, vimos que este consumía demasiados recursos, por lo que al final optamos por crear un LaserScans.

Para ello utilizaremos el paquete de ROS depthImage_to_laserScan, por lo que nuestro primer paso, será crear una imagen de profundidad, o depthImage.

Teóricamente sabemos que la fórmula para calcular el depthImage seria la siguiente:

$$Z = \frac{b \cdot f}{x_R - x_T} = \frac{b \cdot f}{d}$$

Figura 6. Formula depthImage

Donde las variables serian:

- **Z:** seria la depthImage
- **b:** baseline, o la distancia entre ambas cámaras
- **f:** la distancia focal de las cámaras
- **d:** el valor de disparidad

Mediante el uso de esta fórmula, y guiándonos con el paquete de ROS depth_image_proc, dentro del cual existe una función que transforma un depthImage en un DispartityImage, hemos realizado el siguiente codigo, mediante el cual, conseguimos crear un DepthImage, bastante acertado.

```
➢ const float* disp_row = reinterpret_cast<const float*>(&dimage.data[0]);
  int row_step = dimage.step / sizeof(float);
  float* depth_data = reinterpret_cast<float*>(&depthImage.data[0]);
  for (int v = 0; v < (int)dimage.height; ++v)
  {
    for (int u = 0; u < (int)dimage.width; ++u)
    {
      float disp = disp_row[u];
      if (disp > 0)
      *depth_data = constant / disp;
      ++depth_data;
    }
    disp_row += row_step;
```

*}*

Esta función se encarga de entrar pixel a pixel de la imagen de disparidad, y transformarlo correctamente en su equivalente para la imagen de profundidad, lo que nos da el siguiente resultado

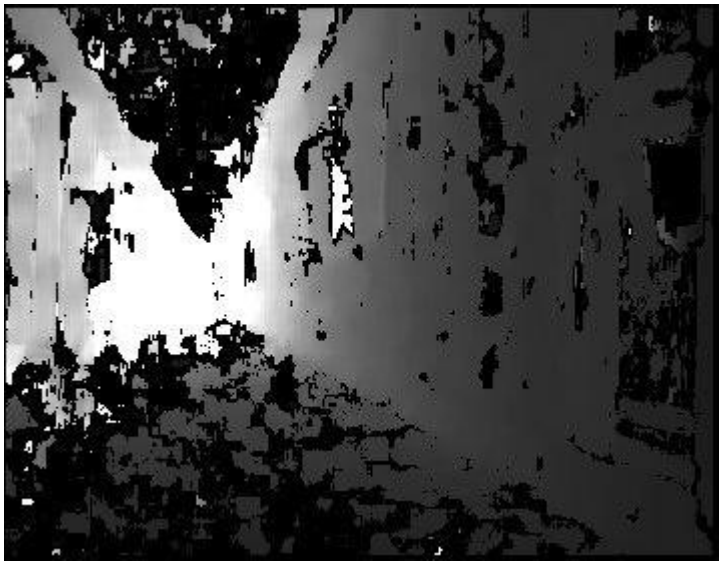


Figura 7. DepthImage con StereoBM

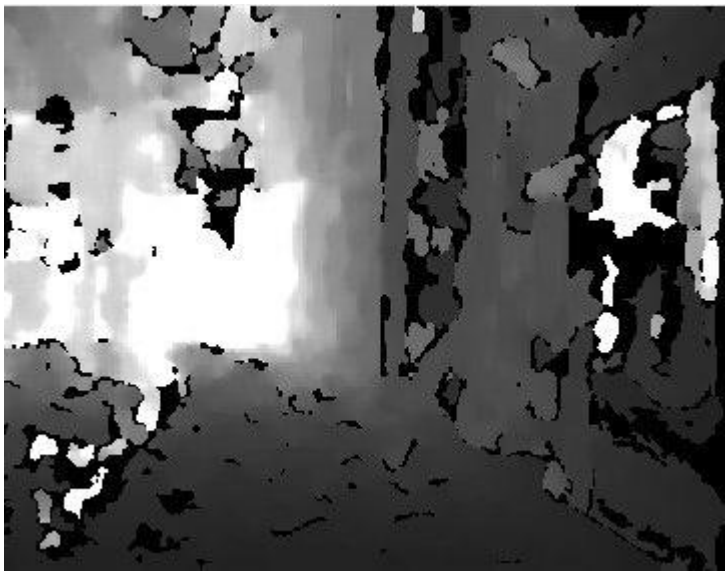


Figura 8. DepthImage con StereoSGBM

### 4.2.6 Creación del LaserScans

Como comentamos en el apartado anterior, para poder crear el costmap, es necesario proporcionarle al mismo un LaserScans, por lo tanto utilizaremos un

paquete de ROS que realiza esa función, necesitando únicamente un depthImage, que calculamos ya en el apartado anterior. Este paquete ya se ejecuta automáticamente al lanzar uno de los launcher que hemos utilizado en el proyecto, dando los siguientes resultados.

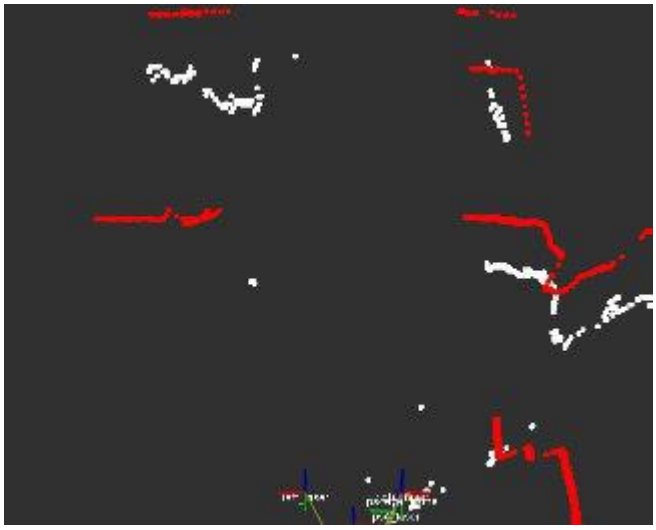


Figura 9. LaserScan con StereoBM

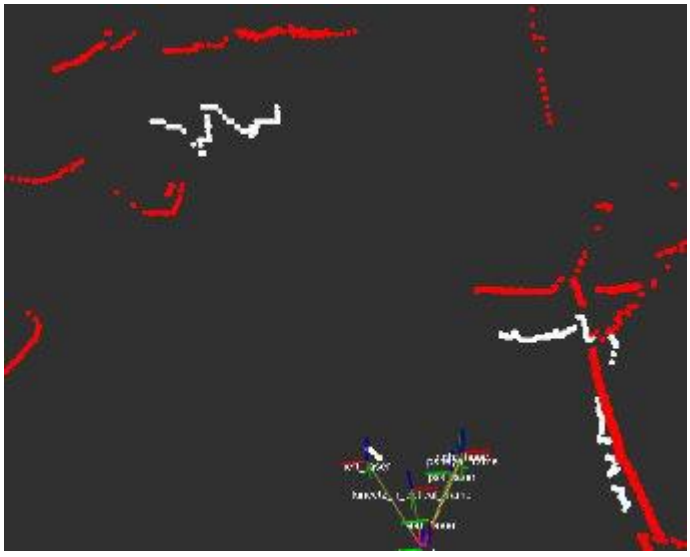


Figura 10. LaserScan con StereoSGBM

En ambas imágenes, las líneas rojas, corresponden a los laser que disponía el robot, al realizar la bolsa de imágenes con las que se ha trabajado durante

el proyecto, mientras que la línea blanca, corresponde a la que se genera mediante la función depthImage_to_laserScan.

# Capítulo 5. Conclusiones y Trabajos Futuros

En conclusión, durante la realización de este proyecto, hemos visto como ha sido posible crear un mapa de costes, con una precisión bastante aceptable, utilizando cámaras estéreo, en vez de cámaras laser, lo que consigue que se abaraten bastante los costes.

Finalmente indicar que este proyecto también podría ser usado como base, para seguirle añadiendo funcionalidades a la cámara, como el cálculo de pointCloud que se comento anteriormente, u otras tecnologías necesarias.

.

# Capítulo 6. Conclusions and future work lines

In conclusion, during the realization of this project, we have seen how it has been possible to create a cost map, with a quite acceptable precision, using stereo cameras, instead of laser cameras, which results in a low cost.

Finally indicate that this project could also be used as a base, to continue adding functionalities to the camera, such as the calculation of pointCloud discussed above, or other necessary technologies.

# Capítulo 7. Presupuesto

En relación al presupuesto del software, este sería de 0, ya que todas las tecnologías utilizadas son basadas en software libre, por lo que todo el coste del mismo caería directamente sobre el apartado hardware:

| Producto | Coste |
|---|---|
| **UpBoard** | 84€ |
| **PlayStation Camera** | 59.99€ |
| **Presupuesto** | 143.99€ |

Tabla 2. Presupuesto

# Apéndice

## A.1. StereoBM.cpp

```
/*********************************************************************************
*
* StereoBM.cpp
*
*********************************************************************************
```

```
   *
   * Jose Ricardo Pérez Castillo
   *
   * 21/09/2017
   *
   * Este fichero contiene toda la configuración necesaria para la creación de una
   * imagen de profundidad a partir de imágenes estéreo, mediante el uso del algoritmo
   * BM.
   ************************************************************************************/
   #include "ros/ros.h"
   #include "sensor_msgs/Image.h"
   #include "image_transport/image_transport.h"
   #include "image_transport/subscriber_filter.h"
   #include "cv_bridge/cv_bridge.h"
   #include "sensor_msgs/image_encodings.h"
   #include "stereo_msgs/DisparityImage.h"
   #include "opencv2/imgproc/imgproc.hpp"
   #include "opencv2/highgui/highgui.hpp"
   #include "opencv2/calib3d/calib3d.hpp"
   #include "message_filters/synchronizer.h"
   #include "message_filters/sync_policies/approximate_time.h"
   #include "message_filters/sync_policies/exact_time.h"
   #include "message_filters/subscriber.h"
   #include <image_geometry/stereo_camera_model.h>
   #include <stereo_msgs/DisparityImage.h>
   #include "std_msgs/Float64.h"
   #include "stdlib.h"

   typedef message_filters::sync_policies::ExactTime<
         sensor_msgs::Image,        sensor_msgs::CameraInfo,        sensor_msgs::Image,
sensor_msgs::CameraInfo
         > MySyncPolicy;

   class DisparityBM
   {
   private:
      ros::NodeHandle nh_;
```

```
    image_transport::ImageTransport it_;
    image_transport::SubscriberFilter image_left_sub_;
    image_transport::SubscriberFilter image_right_sub_;

    message_filters::Subscriber<sensor_msgs::CameraInfo> cameraInfoR_;
    message_filters::Subscriber<sensor_msgs::CameraInfo> cameraInfoL_;
    ros::Publisher camera_Info_Pub_;
    ros::Publisher disparity_bm_pub_;
    ros::Publisher depth_image_pub_;
    message_filters::Synchronizer<MySyncPolicy> sync_;

    mutable cv::Mat_<int16_t> imgDisparity16S; // se define fuera de la función para
mejorar la gestion de la memoria
    stereo_msgs::DisparityImage dispImage;
    const int ndisparity = 16*8;
    const int SADWindowSize = 15;
    const int min_Disparity = 0;

    #if CV_MAJOR_VERSION == 3
        cv::Ptr<cv::StereoBM> sbm = cv::StereoBM::create(ndisparity, SADWindowSize);
    #else
        cv::StereoBM sbm = cv::StereoBM(CV_STEREO_BM_BASIC,ndisparity, SADWindowSize);
    #endif

  public:
    DisparityBM()  : it_(nh_),
       image_left_sub_(it_, "/stereo/left/image_rect", 1),
       cameraInfoL_(nh_, "/stereo/left/camera_info", 1),
       image_right_sub_(it_, "/stereo/right/image_rect", 1),
       cameraInfoR_(nh_, "/stereo/right/camera_info", 1),
       sync_(MySyncPolicy(10),image_left_sub_,     cameraInfoL_,     image_right_sub_,
cameraInfoR_)
    {

       sync_.registerCallback(boost::bind(&DisparityBM::disparity_callback, this, _1,
_2, _3, _4));
       camera_Info_Pub_  =  nh_.advertise<sensor_msgs::CameraInfo>("/tfg/camera_info",
1);
```

```
        depth_image_pub_ = nh_.advertise<sensor_msgs::Image>("/tfg/depthImage", 1);
        disparity_bm_pub_                                                    =
nh_.advertise<stereo_msgs::DisparityImage>("/tfg/disparityImage/bm", 1);

     }

     ~DisparityBM()
     {

     }

    void disparity_callback(const sensor_msgs::ImageConstPtr& image_left_msg,
                            const sensor_msgs::CameraInfoConstPtr& camera_info_L,
                            const sensor_msgs::ImageConstPtr& image_right_msg,
                            const sensor_msgs::CameraInfoConstPtr& camera_info_R)
     {
        static const int DPP = 16; // disparities per pixel
        static const double inv_dpp = 1.0 / DPP;

        cv_bridge::CvImageConstPtr  cv_ptr_left  =  cv_bridge::toCvShare(image_left_msg,
sensor_msgs::image_encodings::TYPE_8UC1);
        cv_bridge::CvImageConstPtr cv_ptr_right = cv_bridge::toCvShare(image_right_msg,
sensor_msgs::image_encodings::TYPE_8UC1);
        const cv::Mat left_image = cv_ptr_left->image;
        const cv::Mat right_image = cv_ptr_right->image;

        if (imgDisparity16S.empty())
               imgDisparity16S = cv::Mat(left_image.rows, left_image.cols, CV_16S);

        #if CV_MAJOR_VERSION == 3
           sbm->compute(left_image, right_image, imgDisparity16S);
        #else
           sbm.operator()(left_image, right_image, imgDisparity16S);
        #endif

         image_geometry::StereoCameraModel model_;
        model_.fromCameraInfo(camera_info_L,camera_info_R);
```

```
        float focal = model_.right().fx();
        float baseline = model_.baseline();

        //Creando mi propio camera_info
        sensor_msgs::CameraInfo myCameraInfo;
        myCameraInfo = *camera_info_R;

            sensor_msgs::Image& dimage = dispImage.image;
      dispImage.header = camera_info_L->header;
            dimage.height = imgDisparity16S.rows;
            dimage.width = imgDisparity16S.cols;
            dimage.encoding = sensor_msgs::image_encodings::TYPE_32FC1;
            dimage.step = dimage.width * sizeof(float);
            dimage.data.resize(dimage.step * dimage.height);
            cv::Mat_<float> dmat(dimage.height, dimage.width, (float*)&dimage.data[0],
dimage.step);

            imgDisparity16S.convertTo(dmat, dmat.type(), inv_dpp, -(model_.left().cx()
- model_.right().cx()));
            ROS_ASSERT(dmat.data == &dimage.data[0]);

       // Stereo parameters
            dispImage.f = model_.right().fx();
            dispImage.T = model_.baseline();

       // Disparity search range
            dispImage.min_disparity = min_Disparity;
            dispImage.max_disparity = min_Disparity + ndisparity - 1;
            dispImage.delta_d = inv_dpp;

        //calcular depthImage
        sensor_msgs::Image depthImage;
        depthImage.header = dispImage.header;
        depthImage.encoding = sensor_msgs::image_encodings::TYPE_32FC1;
        depthImage.height = dimage.height;
        depthImage.width = dimage.width;
```

```
      depthImage.step = depthImage.width * sizeof(float);
      depthImage.data.resize(depthImage.height * depthImage.step, 0.0f);

      float constant = dispImage.f * dispImage.T;

      const float* disp_row = reinterpret_cast<const float*>(&dimage.data[0]);
      int row_step = dimage.step / sizeof(float);
      float* depth_data = reinterpret_cast<float*>(&depthImage.data[0]);
      for (int v = 0; v < (int)dimage.height; ++v)
      {
         for (int u = 0; u < (int)dimage.width; ++u)
         {
            float disp = disp_row[u];
            if (disp > 0)
            *depth_data = constant / disp;
            ++depth_data;
         }
         disp_row += row_step;
      }

      //publicar
      camera_Info_Pub_.publish(myCameraInfo);
      disparity_bm_pub_.publish(dispImage);
      depth_image_pub_.publish(depthImage);
   }
};

int main(int argc, char **argv)
{
   ros::init(argc, argv, "DisparityBM");
   DisparityBM dbm;
   ros::spin();
   return 0;
}
```

## A.2. StereoSGBM.cpp

```
/*********************************************************************************
*
* StereoSGBM.cpp
*
*********************************************************************************
*
* Jose Ricardo Pérez Castillo
*
* 21/09/2017
*
* Este fichero contiene toda la configuración necesaria para la creación de una
* imagen de profundidad a partir de imágenes estéreo, mediante el uso del algoritmo
* SGBM.
*********************************************************************************/
#include "ros/ros.h"
#include "sensor_msgs/Image.h"
#include "image_transport/image_transport.h"
#include "image_transport/subscriber_filter.h"
#include "cv_bridge/cv_bridge.h"
#include "sensor_msgs/image_encodings.h"
#include "stereo_msgs/DisparityImage.h"
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/calib3d/calib3d.hpp"
#include "message_filters/synchronizer.h"
#include "message_filters/sync_policies/approximate_time.h"
#include "message_filters/sync_policies/exact_time.h"
#include "message_filters/subscriber.h"
#include <image_geometry/stereo_camera_model.h>
#include <stereo_msgs/DisparityImage.h>
#include "std_msgs/Float64.h"
#include "stdlib.h"

typedef message_filters::sync_policies::ApproximateTime<
```

```
        sensor_msgs::Image,        sensor_msgs::CameraInfo,        sensor_msgs::Image,
sensor_msgs::CameraInfo
        > MySyncPolicy;

   class DisparitySGBM
   {
   private:
      ros::NodeHandle nh_;
      image_transport::ImageTransport it_;
      image_transport::SubscriberFilter image_left_sub_;
      image_transport::SubscriberFilter image_right_sub_;

      message_filters::Subscriber<sensor_msgs::CameraInfo> cameraInfoR_;
      message_filters::Subscriber<sensor_msgs::CameraInfo> cameraInfoL_;
      ros::Publisher camera_info_pub_;
      ros::Publisher disparity_sgbm_pub_;
      ros::Publisher depth_image_pub_;
      message_filters::Synchronizer<MySyncPolicy> sync_;

      mutable cv::Mat_<int16_t> imgDisparity16S; // se define fuera de la función para
mejorar la gestion de la memoria
      stereo_msgs::DisparityImage dispImage;
      const int minDisparity = 0;
      const int ndisparity = 16*8;
      const int blockSize = 15;
      const int P1 = 200;
      const int P2 = 400;
      const int disp12MaxDiff = 0;
      const int preFilterCap = 31;
      const int uniquenessRatio = 15;
      const int speckleWindowSize = 100;
      const int speckleRange = 4;
      const bool fullDP = true;

      #if CV_MAJOR_VERSION == 3
         cv::Ptr<cv::StereoSGBM>     ssgbm     =     cv::StereoSGBM::create(minDisparity,
ndisparity, blockSize,
```

```
                                                                    P1, P2, disp12MaxDiff,
preFilterCap,
                                                                    uniquenessRatio,
speckleWindowSize,
                                                                    speckleRange, fullDP);


     #else
        cv::StereoSGBM ssgbm = cv::StereoSGBM(minDisparity, ndisparity, blockSize,
                                              P1, P2, disp12MaxDiff, preFilterCap,
                                              uniquenessRatio, speckleWindowSize,
                                              speckleRange, fullDP)
     #endif

   public:
     DisparitySGBM()   : it_(nh_),
        image_left_sub_(it_, "/stereo/left/image_rect", 1),
        cameraInfoL_(nh_, "/stereo/left/camera_info", 1),
        image_right_sub_(it_, "/stereo/right/image_rect", 1),
        cameraInfoR_(nh_, "/stereo/right/camera_info", 1),
        sync_(MySyncPolicy(100),image_left_sub_,    cameraInfoL_,    image_right_sub_,
cameraInfoR_)
     {
        sync_.registerCallback(boost::bind(&DisparitySGBM::disparity_callback,    this,
_1, _2, _3, _4));
        camera_info_pub_  =  nh_.advertise<sensor_msgs::CameraInfo>("/tfg/camera_info",
1);
        depth_image_pub_ = nh_.advertise<sensor_msgs::Image>("/tfg/depthImage", 1);
        disparity_sgbm_pub_                                                          =
nh_.advertise<stereo_msgs::DisparityImage>("/tfg/disparityImage/sgbm", 1);
     }

     ~DisparitySGBM()
     {

     }
```

```
void disparity_callback(const sensor_msgs::ImageConstPtr& image_left_msg,
                        const sensor_msgs::CameraInfoConstPtr& camera_info_L,
                        const sensor_msgs::ImageConstPtr& image_right_msg,
                        const sensor_msgs::CameraInfoConstPtr& camera_info_R)
{
   static const int DPP = 16; // disparities per pixel
   static const double inv_dpp = 1.0 / DPP;

   cv_bridge::CvImageConstPtr cv_ptr_left = cv_bridge::toCvShare(image_left_msg,
sensor_msgs::image_encodings::TYPE_8UC1);
   cv_bridge::CvImageConstPtr cv_ptr_right = cv_bridge::toCvShare(image_right_msg,
sensor_msgs::image_encodings::TYPE_8UC1);
   const cv::Mat left_image = cv_ptr_left->image;
   const cv::Mat right_image = cv_ptr_right->image;

   if (imgDisparity16S.empty())
      imgDisparity16S = cv::Mat(left_image.rows, left_image.cols, CV_16S);

   #if CV_MAJOR_VERSION == 3

      ssgbm->compute(left_image, right_image, imgDisparity16S);

   #else

      ssgbm.operator()(left_image, right_image, imgDisparity16S);

   #endif

   image_geometry::StereoCameraModel model_;
   model_.fromCameraInfo(camera_info_L, camera_info_R);

   float focal = model_.right().fx();
   float baseline = model_.baseline();

   //Creando mi propio camera_info
   sensor_msgs::CameraInfo myCameraInfo;
   myCameraInfo = *camera_info_R;
```

```
        sensor_msgs::Image& dimage = dispImage.image;
        dispImage.header = camera_info_L->header;
        dimage.height = imgDisparity16S.rows;
        dimage.width = imgDisparity16S.cols;
        dimage.encoding = sensor_msgs::image_encodings::TYPE_32FC1;
        dimage.step = dimage.width * sizeof(float);
        dimage.data.resize(dimage.step * dimage.height);
        cv::Mat_<float>  dmat(dimage.height,  dimage.width,  (float*)&dimage.data[0],
dimage.step);

        imgDisparity16S.convertTo(dmat,  dmat.type(),  inv_dpp,  -(model_.left().cx()  -
model_.right().cx()));
        ROS_ASSERT(dmat.data == &dimage.data[0]);

       // Stereo parameters
        dispImage.f = model_.right().fx();
        dispImage.T = model_.baseline();

       // Disparity search range
        dispImage.min_disparity = minDisparity;
        dispImage.max_disparity = minDisparity + ndisparity - 1;
        dispImage.delta_d = inv_dpp;

        //calcular depthImage
        sensor_msgs::Image depthImage;
        depthImage.header = dispImage.header;
        depthImage.encoding = sensor_msgs::image_encodings::TYPE_32FC1;
        depthImage.height = dimage.height;
        depthImage.width = dimage.width;
        depthImage.step = depthImage.width * sizeof(float);
        depthImage.data.resize(depthImage.height * depthImage.step, 0.0f);

        float constant = dispImage.f * dispImage.T;

        const float* disp_row = reinterpret_cast<const float*>(&dimage.data[0]);
        int row_step = dimage.step / sizeof(float);
```

```
      float* depth_data = reinterpret_cast<float*>(&depthImage.data[0]);
      for (int v = 0; v < (int)dimage.height; ++v)
      {
         for (int u = 0; u < (int)dimage.width; ++u)
         {
            float disp = disp_row[u];
            if (disp > 0)
               *depth_data = constant / disp;
            ++depth_data;
         }
         disp_row += row_step;
      }

      //publicar
      camera_info_pub_.publish(myCameraInfo);
      disparity_sgbm_pub_.publish(dispImage);
      depth_image_pub_.publish(depthImage);
   }
};

int main(int argc, char **argv)
{
   ros::init(argc, argv, "DisparitySGBM");
   DisparitySGBM dsgbm;
   ros::spin();
   return 0;
}
```

## A.3. remap_images.launch

```
/*********************************************************************************
*
* remap_images.launch
*
```

```
  ********************************************************************************
  *
  * Jose Ricardo Pérez Castillo
  *
  * 21/09/2017
  *
  * Este fichero contiene toda la configuración necesaria para obtener las imágenes
  * desde la cámara, para poder trabajar sobre ellas de costes a partir de imágenes
  * estéreo, mediante el uso del algoritmo BM.
  ********************************************************************************/
  <launch>
     <param name="use_sim_time" type="bool" value="True"/>
     <!-- Just to uncompress images for stereo_image_rect -->
     <node     name="republish_left"          type="republish"     pkg="image_transport"
args="compressed in:=/stereo/left/image_raw raw out:=/stereo/left/image_raw_relay" />
     <node      name="republish_right"      type="republish"      pkg="image_transport"
args="compressed in:=/stereo/right/image_raw raw out:=/stereo/right/image_raw_relay" />
     <!-- Run the ROS package stereo_image_proc for image rectification -->
     <group ns="/stereo" >
        <node pkg="nodelet" type="nodelet" name="stereo_nodelet"  args="manager"/>

        <node              pkg="stereo_image_proc"              type="stereo_image_proc"
name="stereo_image_proc">
           <remap from="left/image_raw"    to="left/image_raw_relay"/>
           <remap from="left/camera_info"  to="left/camera_info"/>
           <remap from="right/image_raw"   to="right/image_raw_relay"/>
           <remap from="right/camera_info" to="right/camera_info"/>
           <param name="disparity_range" value="128"/>
        </node>
     </group>
  </launch>
```

## A.4. mitf.launch

```
  /********************************************************************************
  *
```

```
   * mitf.launch
   *
   ********************************************************************************
   *
   * Jose Ricardo Pérez Castillo
   *
   * 21/09/2017
   *
   * Este fichero contiene toda la configuración necesaria para la creación de un
   * laserScans a partir de una imagen de profundidad.
   ********************************************************************************/
   <launch>
   <node                    pkg="tf"                   type="static_transform_publisher"
name="left_laser_to_base_link_static_transform" args="0.6 -0.32 0.22 -0.1 0 0 base_link
ps4_laser 100" />
   <node           name="depthimage_to_laserscan"          pkg="depthimage_to_laserscan"
type="depthimage_to_laserscan" >
     <remap from="image" to="/tfg/depthImage"/>
     <remap from="camera_info" to="/tfg/camera_info"/>
     <param name="scan_height" value="10"/>
     <param name="range_min" value="0.1" />
     <param name="range_max" value="15" />
     <param name="output_frame_id" value="ps4_laser" />
   </node>
   </launch>
```

# Bibliografía


[1] ROS.

http://www.ros.org/.

[2] Repositorio ROS versión Indigo.

http://repositories.ros.org/status_page/ros_indigo_default.html?s=2.

[3] Repositorio ROS versión Kinetic.

http://repositories.ros.org/status_page/ros_kinetic_default.html?s=2.

[4] OpenCV.

http://opencv.org/

[5] Libreria OpenCV calib3d.

http://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html

[6] Depth_image_proc.

https://github.com/ros-perception/image_pipeline/blob/indigo/depth_image_proc/src/nodelets/disparity.cpp

[7] LaserScan.

http://wiki.ros.org/depthimage_to_laserscan

[8] Navigation.

http://wiki.ros.org/navigation

[9] Costmap_2d.

http://wiki.ros.org/costmap_2d